



图灵程序设计丛书

Apress®

Practical API Design Confessions of a Java Framework Architect

软件框架设计的艺术

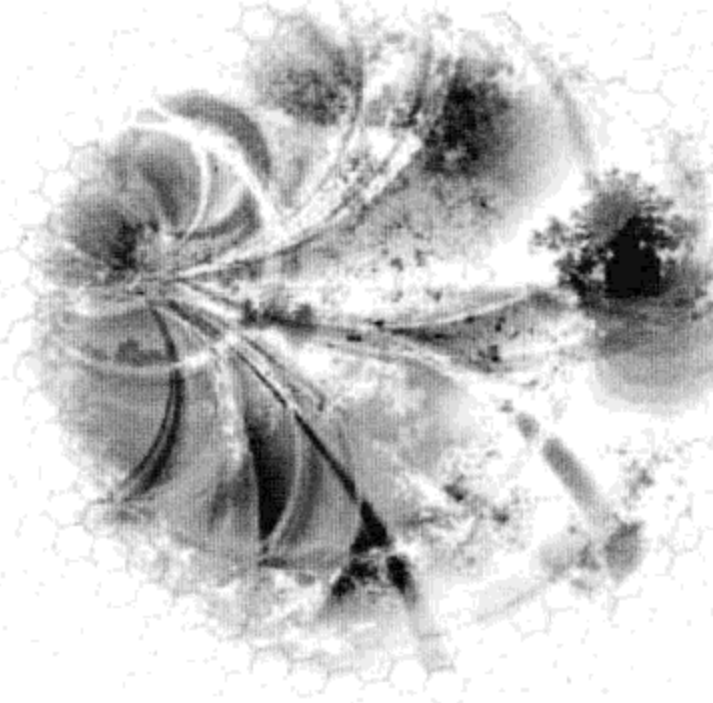
[捷] Jaroslav Tulach 著
王磊 朱兴 译

- NetBeans创始人力作
- 揭开API设计的神秘面纱
- 深入解析，追本溯源



人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书



Practical API Design Confessions of a Java Framework Architect

软件框架设计的艺术

[捷] jaroslav tulach 著
王磊 朱兴 译



人民邮电出版社
北京

图书在版编目 (C I P) 数据

软件框架设计的艺术 / (捷克) 图拉赫 (Tulach, J.)
著 ; 王磊, 朱兴译. -- 北京 : 人民邮电出版社,
2011. 4

(图灵程序设计丛书)

书名原文: Practical API Design : Confessions
of a Java Framework Architect
ISBN 978-7-115-24849-7

I. ①软… II. ①图… ②王… ③朱… III. ①软件设计 IV. ①TP311.5

中国版本图书馆CIP数据核字(2011)第025425号

内 容 提 要

本书帮助你解决 API 设计方面的问题, 共分 3 个部分, 分别指出学习 API 设计是需要进行科学的训练的、Java 语言在设计方面的理论及设计和维护 API 时的常见情况, 并提供了各种技巧来解决相应的问题。本书作者是 NetBeans 的创始人, 也是 NetBeans 项目最初的架构师。相信在 API 设计中遇到问题时, 本书将不可或缺。

本书适用于软件设计人员阅读。

图灵程序设计丛书 软件框架设计的艺术

- ◆ 著 [捷] Jaroslav Tulach
译 王 磊 朱 兴
责任编辑 卢秀丽
- ◆ 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>
中国铁道出版社印刷厂印刷
- ◆ 开本: 800×1000 1/16
印张: 24.25
字数: 587千字 2011年4月第1版
印数: 1~3 000册 2011年4月北京第1次印刷
著作权合同登记号 图字: 01-2009-2894号
ISBN 978-7-115-24849-7

定价: 75.00元

读者服务热线: (010)51095186 印装质量热线: (010)67129223

反盗版热线: (010)67171154

序言：仅仅是又多了一本设计书吗

读者也许会想：“在程序开发领域中，讲述软件设计的技术图书是不是太多了？”，的确如此，因而你有理由来质疑，为什么我还要写一本这样的书而你又凭什么还要再读这样一本书？说起软件设计的经典图书，那本由 GoF 执笔的《设计模式》，对每一个想要掌握面向对象技术的开发人员来说，已经成为案头必备之书。此外，对于不同类型的应用开发，也存在大量专业的软件设计模式图书。还有 *Effective Java*，这本传世之作已经成为众口相传的 Java 程序开发圣经。基于以上事实，还有必要再多一本关于软件设计的图书吗？

我相信这自有其必要性。从 1997 年开始，我一直从事 NetBeans 的 API 设计工作。在此期间，与其他设计框架或者通用功能库的人一样，我也经历了各种酸甜苦辣。刚开始时，我一边尝试将其他语言的一些好的代码风格照搬到 Java 语言上，一边慢慢地找感觉。随后我对 Java 语言使用渐趋熟练，将已知的模式应用到我写的 Java 语言代码上，看起来也就容易多了，当然后来，我又发现事情远不是想象中的那么简单。我认识到，要设计 NetBeans 这样一种面向对象的应用框架，直接套用传统的模式并不恰当，不管那些传统的模式有多成功，它所需要的是一门全新的技艺。

NetBeans 中最早的 API 要追溯到 1997 年了，距今已有 10 多年的历史，至今仍然有不少用户在使用这些 API，而且程序运行一切正常，当然坦率地说其中有很多内容与开始时的设计已经不尽相同。在这 10 多年中，我们也经常根据新的需求来调整和扩展类库的功能，同时对开始时犯下的一些错误进行修正。尽管如此，那些使用了这些 API 的客户仍然可以使用最新版本的类库来编译并运行他们早先的程序。这一切并非偶然，因为我们一直在尽最大的努力来保证类库的向后兼容性。即使客户使用我们 10 多年前提提供的类库来编写程序，然后用最新版本进行编译和运行，这些程序仍然可以平稳地工作。这种有效地保护客户原有的软件投资的理念非常重要，但在常见的设计图书中却无法找到，至少在我读过的那些书中无人提及。当然，在 NetBeans 平台的开发过程中，并不是所有的 API 的演化之路都是一帆风顺的，但我相信，NetBeans 的团队成员已经炉火纯青地掌握了这方面的 API 设计技巧，而其他组织的开发人员也同样需要了解这些技巧。基于这个原因，向后兼容性这个话题在本书中占用了大量篇幅，书中还大量介绍了特殊的 API 设计模式，有助于编写适合向后兼容的代码。

NetBeans 团队工作的扩展能力是当时所面临的另外一个挑战。1997 年项目刚刚开始的时候，由我个人负责 API 的开发，其他工程师的工作则只是写代码，也就是说他们负责完成用户界面并

实现其他 NetBeans IDE，这些工作都需要频繁用到我所设计的 API。毫无疑问，当时我就变成了整个项目的瓶颈。我开始认识到，实现 NetBeans IDE 功能的开发人员越来越多，我一个“架构师”不可能完成所有的 API 需求。随着时间的流逝，急需做出改变。NetBeans 的开发团队中的大部分开发人员都应该能够设计他们自己的 API。而且不管是哪个开发人员设计的 API，我们希望保持一致性。但一致性在这里却成为最大的一个问题，原因不是出在开发人员身上，他们其实也想保持一致性，只是因为我当时无法给他们解释清楚我所指的一致性到底是什么东西。相信很多人也有过与我类似的感觉：自己知道如何去做一件事，但就是无法清楚地解释给别人听。我当时就处于这种状态：我觉得自己知道如何设计 API，但却花了好几个月的时间才整理出想让大家遵守的最重要的规范。

铭记于心的 API 讲座

一直以来，我对设计和发布 API 都有着浓厚的兴趣。在 Sun 公司内部以及为 NetBeans 合作伙伴我也做过多次关于该议题的讲座。但直到 2005 年在旧金山举行 JavaOne 会议时，我才第一次就这个议题进行了公开讲座。当时我和我的朋友 Tim Boudreau 向大会提交了一份议题，名为：如何设计历久弥新的 API。也许是因为在议题摘要中没有写上 Ajax 和 Web 2.0 这种吸引眼球的字样，我们的议题被安排在晚上 10 点 30 分。对于讲座而言，这个时间点实在算不上理想，各种聚会、免费的啤酒宵夜，以及午夜的各种娱乐活动都会抢走我们的听众。我们深受打击，只能期望讲座期间会有一两个朋友来露上一小脸，问上几个问题。当晚我们抵达会场，看到隔壁将要进行一场 JDBC 的讲座时，我们的情绪更是一下子跌到了谷底。走廊里挤满了人，我们以为这些人都是对数据库感兴趣，准备来参加隔壁的 JDBC 技术讲座的。但让我们吃惊的是，大部分听众其实是冲着我们的讲座来的！举办讲座的房间很快就水泄不通了，座无虚席，还有些听众干脆坐在地板上或者倚墙而立，甚至门都不能关上，因为还有部分人只能站在外面的走廊听。最终，我做了一场从未有过的激动人心的讲座。

自那以后，我确信有很多人想要了解如何设计 API，这种需求并非凭空臆造，而是真实存在的。因此在我撰写此书的过程中，每当快要失去动力的时候，便会想起那场激情澎湃的讲座，就又重燃热情。这场讲座时刻提醒着我，那些源于实践并指导我们正确设计 API 的原则应当被记录下来。这些原则虽然基于大量的设计图书介绍的设计知识，但在本书中得到强调和扩展，因为 API 设计有其特殊性。

API 设计的特殊所在

为什么说市场上现有的设计图书还不够用呢？这是因为设计框架或者通用类库是一件非常复杂的事情，其复杂度与自行设计内部系统不可同日而语。打个比方，在一台服务器上基于一个小型数据库来搭建一个 Web 应用之类的小系统，就和盖一间房子差不多。当然，有些房子可能很小，也有些房子会很大，有时也许是一座摩天大楼。这些建筑通常情况下都只有一个主人，只有他才会改造这个房子。如果需要的话，改一下屋顶，换上新窗户，也许会砌面墙多隔出一间来，

再拆旧墙打通两个房间，等等。当然，有些改变算不上什么麻烦事，如换个屋顶不会对地板造成什么大的破坏，换个窗户，只要大小规格不变，也不会影响别的部位。但是，如果想把窗户从小变大，就不那么简单了，而想换个两倍大的新电梯更是不可能完成的任务。再如，罕有人会疯狂到盖个新的一楼，然后把原来的楼层都往上移一层。这样做实在是问题多多，弊大于利。当然，从技术角度来看，上面这些变化仍然是可行的。只要这个房间的主人确有这个需要，铁了心也能做到。

内部软件系统也有些相似。通常也只有一个所有者，而且它还具有完全的控制权。如果现在需要对系统中的部分功能进行升级，那么尽管放手去做就是了！如果要改变一下数据库的模式，也可以悉听尊便。当然，有些改变会比较复杂。拿以下两种情况作个比较：修改一行代码来修复一个 `NullPointerException` 的 bug 和调整数据库模式，显然，后者的影响会大得多。但对于内部系统来说，一切改变都是可行的。因为其所有者有着绝对的控制权，只要系统确实需要大的升级，甚至可以暂时关闭这个应用系统，等完成升级以后再重新运行。此外，我们已经有了不少相关的设计原则帮助，我们更好地管控一个内部系统的变化。有设计模式的书籍帮助开发人员更好地组织代码，还有不少关于设计系统和测试系统的方法论，还有大量的图书介绍如何组织和领导员工进行团队合作。可以说，维护一个内部系统的方方面面，都是非常清楚明了的，也有很多详细文档可供参考。

但是编写 API 则有所不同。可以拿宇宙来打个比方，尽管宇宙不像先前说的房子那么直观，但还算得上比较形象。先来回忆一下我们已知的宇宙，我说“已知”的宇宙，是因为没有人能洞悉宇宙的一切，所有那些恒星、银河，以及其他天体，还包括无形的内容，如所有物理规律。当然，人类现在对宇宙所了解的其实只算得上沧海一粟。我们的视野有多宽，就限定了宇宙在我们眼中的样子，也就是说，我们所说的宇宙，只是在我们自己眼中的宇宙，是真实宇宙在我们眼中的一个缩影。它包含了无数的天体和状态，然而，我们的经验和想象告诉我们，在我们视野范围以外，还有其他的星星和银河，但我们对它们一无所知。千百年来，人类通过打造更先进的设备及不断地认识和理解自然的规律，不停地扩展自己的视野，不断地发现新事物或者新规律，人类关于宇宙的认识和经验就是基于上述实践而逐渐形成的。

宇宙并非亘古不变，而是时刻都在变化，但其变化却有规律可循。这些规律告诉我们，行星、恒星以及其他天体之间是如何相互影响的。举例来说，假设某个人通过自己的观察，发现了一颗新的恒星，那么不管明天、后天还是大后天，都可以观察到这颗恒星，这并不奇怪。虽然现有的自然规律告诉我们：恒星不仅会移动，而且会旋转，甚至还会爆炸。但这些变化都会遵从自然规律。不会有人隔一两周就发现一件宇宙大事，说有某个恒星出现了，消失了，或者是进行随机性的移动。如果宇宙真有这么疯狂，那么就完全颠覆了我们今天对于宇宙的认识。我们通常认为，一旦一颗恒星被发现，它就会长久存在，甚至相信即使无人观察到它，它依然存在。这颗恒星可以被地球上的一个人观察到，也可以被太阳系其他地方的另外一个人观察到，甚至还有宇宙中其他智慧生物观察到，当然也可能无人注意它。但恒星本身并不知道是否被他人所观察，它只会默默遵照自然规律存在和运行。因此一旦被发现，它便一直陪伴着我们。

好的 API 也是如此。一旦某个通用的类库在某个版本引入了一个新方法，就好像发现了一颗

新的恒星。所有使用该类库的人都可以看到并使用该方法。至于是否在自己的程序中使用，则要视乎程序员自己的需要了。有可能，多数 API 的使用者对你添加的新方法完全不在意。但你不能把希望寄托在这种并无根据的猜测之上。多年的开发经验告诉我，API 的用户太有创意了。有时候，他们涉及的领域甚至超过了 API 设计者。换句话说，只要 API 有可能被误用，就一定会有人去误用这个 API。随之而来的结果就是，不管是方法本身还是其设计者，都不知道该方法是否被使用及其使用频率有多大。也许会有很多用户使用它，也许一个都没有，但除非你想破坏优秀 API 的设计法则，即打破其向后兼容性，否则就必须假定有人在观察，必须保证这些 API 得到良好的维护和保留。“API 就如同恒星，一旦出现，便与我们永恒共存。”

宇宙与 API 的设计还有一个相似之处。我们对宇宙的认识不断加深，正如我们的类库在不断演进。古希腊人已经可以识别和观察远至土星和木星的所有行星的运行路线，这就是他们眼中的宇宙的样子。他们尽力去解释行星运行背后的原因，但以今天的标准来看，他们显然并不成功。他们还不能揭示出行星运行的规律。到了文艺复兴时期，哥白尼提出了日心说，而开普勒则用行星三大定律来诠释行星相对于太阳的运行轨迹和速度。这些探索丰富了人类对宇宙的发现，从而能够对“宇宙是什么”进行精确的解释。但没有人知道“宇宙为什么是这个样子”。直到 1687 年，牛顿才对这个问题给出了诠释，他引入了万有引力的概念。万有引力不仅可以用来解释开普勒定律，还极大地扩展了我们对宇宙的认识，因为对于已知宇宙中多个天体间发生的所有事情，基于牛顿定律的物理学^①几乎都可以给出合理的解释。

一切看来都很完美，直到 19 世纪末。很多实验都发现了一些无法用牛顿定律来解释的现象，特别是对于一些高速运动的天体。这些现象促使爱因斯坦创立了相对论，帮助我们更深入地理解宇宙及其各种现象包括对高速运动的天体的理解。事实上，爱因斯坦的理论是牛顿理论的延伸，只要天体慢到一个合理的速度，那么使用这两个理论可以得到相同的结果，此时可谓殊途同归。

前文啰哩啰嗦地说了很多物理和历史背景的内容，这些东西与设计 API 有半点关系吗？接下来先让我们来假设有一个上帝，万能的他通过一个 API 库与人类进行沟通。这个库是人类通向这个已知的宇宙的桥梁。古希腊人使用这个库的 0.1 版本，其功能很简单，只能用来列举不同的行星以及它们的名字。这个库提供的 API 显然不够丰富，但对于那时候的人来说，已经足够了。借助于这个简单的 API 库，古希腊人已经可以分辨几颗行星。在这个库的使用过程中会有不少人经常提出新的需求，希望对这个 API 库加以改进。当开普勒需要进一步了解行星的运行规律时，就发现这个库的功能已经不能满足他的需要了。因此，这个万能的上帝就给了他一个升级版本，姑且称之为 1.0 版吧。这个版本的库能够为每个行星在某个时间点上提供空间坐标，以确定其位置。同时 1.0 版本很好地兼容了 0.1 版本，也就是说，原来那些古希腊人所使用的功能仍然可以正常使用。

只不过，用户从来没有知足的时候，物理学家们亦如是。为了帮助牛顿，那位万能的上帝接着提供了宇宙 2.0 这个重要的版本。该版本不仅描述了太阳和其他行星之间存在的万有引力，还提供了一堆公式用以计算空间物体的引力、加速度及速度，也不再只局限于行星了。不用说，这个新版本仍然兼容以前的版本，古希腊人和开普勒使用的那些功能仍然可以在新版本中正常使用。

^① 指的是经典物理学。——译者注

至此，所有的变化都很直接。一直以来，那位万能的上帝只是为新版本添加了一些功能。在经典物理学成熟以后，物理学家声称，宇宙的所有规律都已经被物理学家发现了，在物理学的领域已经没有什么未解之谜了！这个声明真是一个天大的讽刺，上帝抛出了迈克尔逊实验^①证明了这个臆断的荒谬，进而导致爱因斯坦提出他的相对论。这时候，物理学家们发现，最新版本的宇宙 API 库出现了无法简单向后兼容的问题，因为新的理论指出，以前所有的物理学家，包括牛顿，都存在少许的错误！尽管出现了如此大的一个变动，但新的 API 库仍然可以按向后兼容方式来处理。那是因为只有以特别高速运动的物体，根据原有方式来计算得到的结果才有可能不正确。而在牛顿及其先行者的那个时代，受限于技术等原因，是无法对这样高的运动速度进行测量的。因此，尽管不兼容的问题早就存在，但用以前的测量技术却发现不了，从而也无法证实宇宙 API 库的功能发生了改变。

上述这个荒诞的故事旨在说明我们对宇宙的认识一直在不停地进步。我们编写的 API 库也同样如此。或许乐观的人并不认同，但我真的感觉人类永远都无法了解整个宇宙的奥秘。当然，我认为我们对宇宙的了解会越来越多。虽然程序员的观点各有不同，但我相信所有已经在使用的 API 库都会永无止境，它们会继续演化。对此，我们必须做好准备。我们必须准备好随时改进我们的 API 库，就像我们必须准备好随时修正我们对宇宙的认识。

与建造一所房子或一个内部的软件系统不同，编写 API 库需要开发人员放眼未来，看到今后潜在的需求。但在我看来，现在人们设计 API 的做法往往不是这样。目前市面上的图书也并没有促使人们这样思考。书中的设计模式大多只能用在特定版本，使用者也只是在特定上下文的环境中去考虑问题，他们极少参考老的版本，也不太考虑未来的需求，所举的例子以及相关的上下文都具有很大的片面性。当然并非说这些书全无裨益，在编写通用功能库和框架时还是需要这些技巧。现在，我们必须停止学习如何来设计内部系统，而要开始学习如何来设计一套 API 库。在学习中一定要坚持一个观点：“API 一旦发布，便与我们永恒共存。”

读者对象

如果此时你正在书店面对这本书，在买与不买之间犹豫不决，那是因为你无法判断这本书对你是否有用。老实说，这点我帮不了你，因为我不是你。但我可以告诉你我自己为什么需要这本书，以及我写作该书的缘由，这样也许可以帮助你决定是否应该购买此书。我在设计 NetBeans 框架的 API 时，就像在一片迷茫中寻找光明，总是摸不清方向。最开始，我完全凭直觉，而且认为写 API 是一种艺术。我知道对于艺术来讲，需要创造力，但设计 API，与艺术是不同的范畴。时间慢慢过去，我从已经完成的工作中汲取经验，逐渐整理出一整套思路和度量标准，借助于这些可量化的标准，可以将一个普通的 API 优化成一个优秀的 API。

^① 迈克尔逊干涉仪是 1880 年美国物理学家迈克尔逊为研究“以太”漂移速度实验设计制造出来的。1887 年，他和美国物理学家莫雷合作进一步用实验结果否定了“以太”的存在，为爱因斯坦建立狭义相对论开辟了道路。由于发明了精密的光学仪器和借助这些仪器所做的基本度量学研究，迈克尔逊于 1907 年获得了诺贝尔物理学奖。

本书介绍了 NetBeans 团队中一直以何种标准来评价 API 的质量，并清楚地说明我们团队为什么一直坚持使用这个标准。事实上，这些标准都是我们经过多年的尝试，并从错误中吸取教训，才最终得到的。地球人都知道，重新发明轮子并不是一个好主意，这是在浪费时间和金钱，所以对于那些把 API 设计更多地看作是一种工程而非艺术的架构师们，我郑重推荐此书。在 NetBeans 的初创阶段，只有我一个人来设计 API。当时，我们有一个比较极端的观点：“一群代码开发人员是不可能设计出一个好的 API 的。”其实对于一个单兵作战设计人员来说，即使没有任何规则来约束，他所设计的 API 也具有 consistency。但像 NetBeans 这样的一个大团队，是不能只有一个设计人员的。所以，我的首要任务就是要去寻找一种方式，让更多的人能够设计 API，同时还能保持整体设计上的一致性。那个时候我已经开始撰写本书，希望能告诉大家 API 设计方面的相关理论，以及我们编写 API 的原因和目标等，同时还根据我们过去的经验总结出一些规则，方便大家来量化一个 API 的设计质量。接下来，我将我的经验与 NetBeans 团队中的成员分享。从此，我开始放手让他们也来编写 API，在开始和结束阶段花些时间来评审和指导他们的设计。以我的标准来衡量，可以说这么做很棒。这 10 年来，他们一直在努力地学习和进步，现在看来，我们设计的 API 具有了相当高的一致性，并满足了我们的大部分需求。如果你也处于相应的职位，需要评审或者指导他人来设计 API，你将发现本书中的建议会对你有所裨益。

当我想为 API 给出一个定义时，却发现这个定义的范围非常广泛。要知道，不是说只有一个框架或者是通用库才算是 API。即使只是写了一个普通的类，只要有同事使用了这个类，那么也算是写了一个 API。为什么这样说呢？假设你删除了这个类中的一些方法，或者是改变了这些方法的名称，哪怕是改变了这个类的一些功能，这个类的使用者都会火冒三丈。为共享库写 API 也面临同样的问题。如果你所写的类有多个人使用了，那么你对该类的修改也许会强制要求所有用户都进行相应调整，这无异于一场噩梦。这种噩梦其实是可以避免的。如果在开始写代码时，就把一个类当成一个 API 来认真对待，你会少许多麻烦。换个角度来说，其实这事也不难，只需要设计类的时候再小心一些，在调整的时候多多关注它的兼容性，再参考和借鉴一些好的经验，其实都搞得定。如果根据前面所说的这个定义来分析一下，几乎所有的开发人员都在编写和设计 API。

API 的一个本质特性就是它的工作机制。API 的测试是非常重要的，借助于它，可以更加清楚地说明 API 的原理。没有合适的测试，就不可能编写一个好的 API。本书有几个章节会列出一些测试方式，告诉读者如何来有效地测试一个库的公开接口，而且即使是要处理该库的多个版本，也无碍于测试的正常运行。我会详细地说明测试时要注意的各项内容，包括签名^①、单元测试和兼容性工具。所以，对于那些需要检查 API 兼容性的人来说，本书非常有价值。

最后要说的是，一个得到广泛使用的类库将会是该库作者的财富。如果这个类库能够很好地满足现有用户的需求，就会吸引更多的用户来使用，财富将会不停地增加。要知道，只有在类库拥有了大量的用户以后，才能在这个基础上获得经济利益，从而继续开发和维护这个类库。本书

① 所谓的签名就是英文中的 *signature*，表示一个类或一个方法的标识，它与序列化等多个方面都有关系，通常来说，一个类的签名是由其父类及其所有成员，包括字段和方法决定的，一个方法的签名则由参数或返回值确定。

——译者注

会就这一点展开讨论，那些喜欢从商业角度来审视软件开发的人会对这一个话题很感兴趣。

这本书只适用于 Java

NetBeans 是一个使用 Java 语言开发的 IDE 框架，我的大部分与 API 有关的知识都是从这个项目中学到的。如果由我来回答这个问题：“这本书是否有益于那些非 Java 的开发？”那么答案仍然是肯定的。对于如何评价 API 设计是否良好，本书给出了很多准则，这些准则同样适用于其他的编程语言。本书中的一些议题，例如：开发 API 的原因，编写具有良好结构的文档的规则和动机，还有那些关于向后兼容的原则。这些内容都可以适用于多种编程语言，包括 C、FORTRAN、Perl、Python 和 Haskell^①。

当然，涉及细节的时候，就不得不提到 Java 语言的具体特性。要知道，Java 首先是一种面向对象的语言。为面向对象的语言设计 API 的时候，像继承、虚方法^②和封装这些特性都必须加以考虑。因此，本书给出的一些原则更适用于一些特定的面向对象语言，如 C++、Python 或 Java，至于 C 或 FORTRAN 这些面向过程的语言，虽然不错，但已经有点古老了，不再适用本书中的原则。

Java 是一种非常新的面向对象语言，它引入了垃圾收集器。当前，业界已经普遍接受了 Java，说明即使在产品的正式运行环境下，垃圾收集器也是可用的且有益的。但在 Java 语言出现之前，业界更倾向那种自行管理内存的传统方式，如 C、C++ 都是采用这种方式来管理内存，开发人员需要明确地声明如何去申请和释放内存。当时也有一些语言，像 Smalltalk 或 Ada，它们使用了垃圾收集器，但它们都被当作实验品，没有几个软件开发商敢冒着这样大的风险去使用它们来开发软件。Java 却从根本上扭转了这个现象。目前，可以说，一个基于内存自动管理系统的语言可以用来编写高性能的程序。大多数的软件工程师都已经普遍接受了这个观点，而不像以往只会取笑或者害怕使用这种基于内存自动管理的语言。一个能够自动管理内存的语言，会对你写的 API 有所要求。比如说，Java 只能通过 malloc 一样的构造函数来分配对象，而不是像 C 一样，需要对应地释放 API。而且 Java 中，其内存的释放无须程序员关注。所以本书中给出的一些意见或者做法更适用于带有垃圾收集器的语言，就是那种与 Java 相似、支持内存自动管理的语言。

Java 还推广了虚拟机和动态编译技术的使用。Java 的静态编译技术会将源代码编译成多个类文件。在代码真正运行的时候才去部署和连接这些文件。而这些编译好的类文件格式是不依赖于具体的处理器架构的，而应用程序最终运行于这个架构上。

以上所说的内容通过运行时环境实现，它不仅将分散的类文件连接^③起来，还将指令转换成处理器可以识别的指令。从 Java 诞生之初，这一点就是 Java 与传统语言的相异之处。大家都知

① Haskell 是一种纯函数式编程语言，它的命名源自美国数学家 Haskell Brooks Curry，他在数学逻辑方面的工作使得函数式编程语言有了广泛的基础。Haskell 语言是 1990 年在编程语言 Miranda 的基础上标准化的，并且以 λ 演算为基础发展而来。这也是为什么 Haskell 语言以希腊字母 λ (Lambda) 作为自己的标志。——译者注

② 不同的程序语言中，对于虚方法的定义有所不同，最通用的定义是：能够在运用时才确定的方法就可以称为虚方法，如 Java 中的非 final 非 static 方法，又如 C++ 中的 virtual 都算是，甚至 C 语言中的函数指针也可以算作虚方法。

——译者注

③ 原文中用的是 Link，翻译为连接。——译者注

道，高性能程序不能通过虚拟机的解释来获得，像 Fortran 语言就要在不同的操作系统上分别进行编译，根据实际环境来生成相应的可执行汇编，这样才能够更好地利用硬件的各种特性来提高程序的运行性能。当时很多人，包括我自己在内，都认为只有使用 C 或 C++ 才能编写出高性能的程序，而 Java 则不可能做到这一点。

然而，时间证明，基于虚拟机的编程语言具有一定的优势。例如跨平台，不管在什么硬件平台上，所有的数字类型具有相同的长度，不需要程序员去了解这些基础的硬件架构内容。此外，Java 程序不会因为段错误而崩溃。虚拟机对内存的自动化管理，避免了 C 指针误用而引起的内存泄漏崩溃的问题^①，而且能够保证使用的变量始终都有正确的类型。尽管具有以上所说的多项优势，但对于早期的 Java 虚拟机来说，性能仍然是一个长久困扰人们的问题。随着时间的推移，解释器越来越快，而且用来取代解释器的即时编译器^②能够生成更快的代码。这些新的变化极具吸引力，其他的一些新语言也开始采用虚拟机。目前，虚拟机已经被业界广泛接受，并大量使用。在一定程度上，本书谈论了虚拟机多方面的内容，虽有大量的篇幅用来讲述类文件的格式，但类文件格式正是虚拟机^③的通用语言。

如果想全面掌握 Java 语言结构对于虚拟机的意义，那么必须清楚地理解类文件的格式。在虚拟机的世界里，Java 语言及其格式是非常简单的。其他编程语言，如 C，也有自己对应的 ABI（抽象二进制接口）模型，但 Java 的类文件非常有特色，体现在两个方面。首先，它天生就是面向对象的。其次，它使用动态编译，这就意味着，它所包含的信息远远超过了简单的 C 对象文件。因此，学习虚拟机获得的知识几乎不能用于那些虽然优秀但已经很古老的非面向对象语言。但对于那些与 Java 一样使用虚拟机的新程序语言，这些知识就会非常有用。

Java 是第一个能够将实际代码和 API 文档紧密结合在一起的编程语言。通过 JavaDoc 可以将代码中的注释变成公开的文档，Java 提倡开发人员使用这种代码即文档的方式，这样可以保证文档随时都是最新的。尽管其他的编程语言也允许开发人员在代码中加入注释，但只有在 Java 语言中，才支持通过 JavaDoc 将相应的注释转成可供程序员使用的文档，从而保持文档和代码的高度一致。另一方面，这也不再是 Java 特有的功能了。自从这种从代码生成文档的功能被证明了其实用性以后，几乎每一种在 Java 语言之后创建的新编程语言都支持类似于 JavaDoc 的文档生成功能。而且在 Java 语言出现之前就有的语言也提供了额外的工具用来通过代码生成文档。因此，本书虽然只分析了 JavaDoc 在帮助用户理解 API 方面的有效性，以及文档格式的利弊，但是，给出的结论几乎可以适用于任何编程语言。

Java 5 中开始支持泛型，为 Java 语言带来多方面的改变。虽然本书并不想成为一本全面讲述 Java 语言构造的图书，但泛型这样的一个重要特性是不可忽视的。泛型是 API 设计中的一个重要内容。其新颖之处体现在哪里呢？要知道传统的面向对象语言通过继承来鼓励重用，而其实组合也是一种常见的代码重用形式。只不过大家往往都把注意力集中在继承，而忽略了组合。造成这

① 在 C 中，不正确使用指针可能会内存泄露或者是因为缓冲区溢出而导致程序崩溃。——译者注

② JIT (just-in-time, 即时编译) 也被称为动态翻译 (dynamic translation), 是一种通过在运行时将字节码翻译为机器码, 从而改善字节码编译语言性能的技术。——译者注

③ 如无特殊说明, 书中所指的虚拟机都暗指 Java 虚拟机。——译者注

一问题的根本原因是：继承是面向对象语言内置的特性，而组合则只能由程序员来手工编码，并非常容易出现类型错误。同时，现代已经有大量的语言将组合作为首要的重用方式，其次才是继承，尤其是在 Haskell 这种函数语言中。有些人认为这两种方式（即继承和组合）各有所长，不可偏颇，所以他们花了大量时间尝试将面向对象语言与多态类型函数语言结合起来。

将继承和组合进行有效结合，这正是 Java5^①中引入泛型的原因。一些人认为泛型过于复杂，对其大肆批评，但我自己在 1997 年的研究经验表明，几乎无法找到比泛型更加合适的方式了。在这点上，我欣赏 Java 语言的设计团队，他们在继承和组合这两者之间尽量地保持相对均衡。这也是本书讲述泛型的原因所在。这样做使得本书的部分内容更加贴近于如 Haskell 这种函数式语言。

本书之所以适合于其他语言，恰恰是因为使用了 Java 语言。它不是去发明一种特定的新编程语言来处理 API 问题。整本书都使用我们熟悉的 Java 语言。书中所有的原则和建议都使用 Java 固有的编码风格，没有引入任何新的关键字，也不会对前置和后置条件或者对常量的检查进行一些特殊的支持。对于开发一个通用类库的软件工程项目来说，一旦确定了一种开发语言和实现目标，都会设定一个相应的编码风格，约束开发人员使用合适而且统一的编码风格。要知道，学习新 API 所需要的工作量，与新学习一门编程语言相比，可谓小巫见大巫。

由于具体项目要使用的程序语言是确定的，那么 API 的设计原则也必然使用该语言来描述。我们相信，如果能使用 C 语言来编写一个好的 API，那么也同样可以使用 Java 语言来写一个好的 API。所以本书中只使用 Java 语言就足够了。总之，书中提供了可以应用到任何编程语言的通用内容。书中还有一部分内容会更多地讲述面向对象的概念，在需要进行深入讲述时，会使用 Java 语言给出合适的例子。

学习编写 API

毫无疑问，肯定有不少人用正确的方式开发了很多 API，否则现在的市场上就不会有这么多非常有用的软件产品。但设计原则、设计 API 的技巧及要点，通常都是在开发过程中下意识积累而得，而这一过程往往并不具有借鉴意义。很多设计师往往没有知其所以然就做一些 API 设计上的决策。结果就是在不停地尝试，犯错，再尝试再犯错，周而复始才逐渐在其潜意识中形成了设计 API 的相关知识，耽误了很多时间。这一过程中会产生很多指导人们正常做事的技巧，虽然这种过程非常有用，但因为它自身存在的问题，导致其产生的大量技巧非常分散，不易收集和管理。首当其冲的问题就是，这些技巧都有其特定的背景和作用域。很多人都知道，有大量的技巧，对于某一个项目或者是特定的人群是非常有效的，但只要换个团队或者换个项目，就完全不能用了。

其次，因为各人的思路都不相同，所以知识的传递就变得非常困难。在解决某个特定问题的时候，你觉得使用 Java 类要比使用 Java 接口更为合适，但一旦换了一个问题，这种解决方案可能就完全不合适了。即使你尝试去说服别人接受这种方案，但如果没有充足的理由去解释，你只

① 作者在原著中同时使用了 Java1.5 和 Java5，以及 JDK1.5 和 JDK5，这两者是相同的，只不过前者是习惯性地沿用了 Java 一向的命名规则，而后者则是 Sun 对外的版本，翻译时，都使用 Java5 和 JDK5 这两种说法。——译者注

能使用以往自己成功的案例来说服他人来采用该方案。肯定有人会赞同该方案，也有人会反对，但这都不是知识传承的本意。

凭感觉的 NetBeans API 设计中

必须承认我们在开发 NetBeans 项目的过程中也经历过这个阶段。在设计 API 时，我们会觉得某种设计可行，而另一种设计不可行，这种判断完全是凭感觉来做出的，而不是自底向上有坚实的基础的。这就是说，进行设计时，我们没有严格的推理和分析，只是凭着一种感觉，依靠我们的潜意识来设计 API。想将知识传授给其他人的时候，就会因为他们根本没有类似于我们的经验，也就很难说服他们接受我们的知识。这迫使我们深入思考这一现象，考虑建立可度量的标准，用来帮助大家设计优秀的 API。这本书就是深入思考后的结晶。我们确信，我们所积累的经验已经清楚地揭示了我们决策时的逻辑思路。现在我们把这些决策时的逻辑清晰地整理出来，传授给每一个愿意倾听的人。

阅读本书的人，首要关注的问题莫过于以下两个：为什么创造 API？API 到底是什么东西？本书会就这些问题展开详细的讨论。

即使不读、不理解甚至不同意本书中给出的建议，软件产品从业人员只要理解书中的基本需求和思路就会从中受益。这有助于更好地认识和理解 API 设计及其复杂性。当开发团队中的所有成员都可以自行设计 API 时，成员间的沟通就会非常简单，决策也不再需要多余的解释，因为他们拥有相同的知识，并在此基础上进行思考。这样做的最大好处，是可以提高开发人员间的合作效率，以及开发团队及其合作伙伴间的合作效率，从而保证了软件产品的高质量。

这本书尽力想帮助每个人都解决一些问题。它尽力为每一位读者解释 API 设计的基本动机，并为开发人员提供了例子和很多技巧，它叙述了良好架构的方方面面，不管是谁来设计 API，都可以使用书中给出的那些可度量的原则来评估 API 的质量。

如果你还在怀疑是否应该阅读本书，那么给你一个最简捷的回答：“你要读这本书。”

这是一本备忘录吗

决定要以何种风格来撰写本书无疑是一件非常困难的事情，我当时在两种完全不同的写作风格之间摇摆不定，无法定夺。一种写作风格是：用非常科学化、公式化的方式来说明 API 设计时的动机、原因及步骤。使用这种方式来撰写的话，书中给出的建议和规则具有通用性，可以应用于任何项目。当然，通用性是本书的一个目的，书中所说的内容必须是普遍适用的，而不是简单地描述 NetBeans 项目的 10 年发展史。而另一方面，我坚信，如果只是在不停地说着一些建议，讲述着这些原则性的内容，而不给出合适的诠释，那么再好的建议也不能起到应有的作用。我不喜欢只说上一堆“是什么”，而不去详细地解释“为什么”。我一直想清楚地分析上下文，并以此来评估各种解决方案，然后再根据具体的环境来选择一个最合适的方案。这就是为什么我先把设计的背景和大家说明，只有这样，才有利于大家接受我们的设计原则。那么最佳的方式，就是把 NetBeans 项目中不同阶段面临的所有问题一五一十地摆出来。因此，可以将本书看作一本

NetBeans 项目的备忘录。

本书日志风格的写法也是一点点形成的。本书的写作，并不是一开始就列好题纲，打好草稿，书的议题是我在几年中陆陆续续地添加的。每当我们解决一个具有普遍性问题的时候，就会先在书中增加一个新的议题，找到相应解决方案后，就会记录下来。所以这种方式有效地记录了我们当时解决问题的思路，以及相应的规则。以这种写作方式来完成本书，使得本书读起来就像是记录实验日志一样。但我们的实验日志不是像写日记一样，每天一份，而是针对每个问题进行记录！

为了从这两种写作风格中获得最佳的解决方案，本书对每个专题的分析都详细说明了 NetBeans 项目中需要解决的问题的真实处境，然后从特有的问题抽象出一般性的建议或者解决方案，可以适用于任何框架或通用库项目。这类似我们采用的如下思路：首先是面对一个问题，然后进行分析，并提出解决方案。按照这样的思路来阅读本书，读者就可以一步步地验证我们给出的建议、方案，并判断我们推广的通用规则是否正确。在任何情况下，读者都可以灵活地调整书中所给出的方案、意见、建议等，从而更好地应用到自己的项目中。最后采用同样的思路、步骤，来看是否能得到与我们一致的意见。

API 设计的技术天地非常美妙，但到目前为止，都还处于探索阶段，需要我们一步步地来积累这些知识。今天的软件系统正在变得越发庞大，我们需要运用最好的工程实践来正确地架构软件系统，并提高它们的可靠性。API 设计就是其中的一种实践。在 21 世纪的软件开发中，希望这本书可以对你的开发进行指导！让我们的 NetBeans API 设计探索成为供你学习的案例，让我们总结的经验帮助你消除类似的错误。回顾 1997 年，我们踏上了崎岖不平的探索之路，走过了峥嵘岁月，今天，希望读者借助于这本书，一帆风顺地通过 API 设计的艰难险阻，而不必重复我们的曲折经历。



目 录

第一部分 理论与理由

第 1 章 软件开发的艺术	4
1.1 理性主义, 经验主义以及无绪	4
1.2 软件的演变过程	6
1.3 大型软件	8
1.4 漂亮, 真理和优雅	9
1.5 更好的无绪	12
第 2 章 设计 API 的动力之源	14
2.1 分布式开发	14
2.2 模块化应用程序	16
2.3 交流互通才是一切	20
2.4 经验主义编程方式	22
2.5 开发第一个版本通常比较容易	24
第 3 章 评价 API 好坏的标准	26
3.1 方法和字段签名	26
3.2 文件及其内容	27
3.3 环境变量和命令行选项	29
3.4 文本信息也是 API	30
3.5 协议	32
3.6 行为	35
3.7 国际化支持和信息国际化	35
3.8 API 的广泛定义	37
3.9 如何检查 API 的质量	37
3.9.1 可理解性	37
3.9.2 一致性	38

3.9.3 可见性	39
3.9.4 简单的任务应该有简单的方案	40
3.9.5 保护投资	40

第 4 章 不断变化的目标	42
4.1 第一个版本远非完美	42
4.2 向后兼容	43
4.2.1 源代码兼容	43
4.2.2 二进制兼容	44
4.2.3 功能兼容——阿米巴变形虫效应	50
4.3 面向用例的重要性	52
4.4 API 设计评审	55
4.5 一个 API 的生命周期	56
4.6 逐步改善	60

第二部分 设计实战

第 5 章 只公开你要公开的内容	67
5.1 方法优于字段	68
5.2 工厂方法优于构造函数	70
5.3 让所有内容都不可更改	71
5.4 避免滥用 setter 方法	72
5.5 尽可能通过友元的方式来公开功能	73
5.6 赋予对象创建者更多权利	77
5.7 避免暴露深层次继承	82
第 6 章 面向接口而非实现进行编程	85
6.1 移除方法或者字段	87

6.2 移除或者添加一个类或者接口	88	10.5 避免 API 的误用	176
6.3 向现有的继承体系中添加一个 接口或者类	88	10.6 不要滥用 JavaBeans 那种监听器 机制	180
6.4 添加方法或者字段	88	第 11 章 API 具体运行时的一些内容	184
6.5 Java 中接口和类的区别	90	11.1 不要冒险	186
6.6 弱点背后的优点	91	11.2 可靠性与无绪	189
6.7 添加方法的另一种方案	92	11.3 同步和死锁	191
6.8 抽象类有没有用呢	94	11.3.1 描述线程模型	192
6.9 要为增加参数做好准备	95	11.3.2 Java Monitors 中的陷阱	193
6.10 接口 VS. 类	97	11.3.3 触发死锁的条件	196
第 7 章 模块化架构	98	11.3.4 测试死锁	201
7.1 模块化设计的类型	100	11.3.5 对条件竞争进行测试	204
7.2 组件定位和交互	103	11.3.6 分析随机故障	206
7.3 编写扩展点	116	11.3.7 日志的高级用途	208
7.4 循环依赖的必要性	117	11.3.8 使用日志记录程序控制 流程	210
7.5 满城尽是 Lookup	121	11.4 循环调用的问题	215
7.6 Lookup 的滥用	126	11.5 内存管理	218
第 8 章 设计 API 时要区分其目标用 户群	129	第 12 章 声明式编程	223
8.1 C 和 Java 语言中如何定义 API 和 SPI	129	12.1 让对象不可变	225
8.2 API 演进不同于 SPI 演进	131	12.2 不可变的行为	229
8.3 java.io.Writer 这个类从 JDK 1.4 到 JDK 5 的演进	131	12.3 文档兼容性	230
8.4 合理分解 API	143	第三部分 日常生活	
第 9 章 牢记可测试性	147	第 13 章 极端的意见有害无益	236
9.1 API 设计和测试	148	13.1 API 必须是漂亮的	237
9.2 规范的光环正在褪去	151	13.2 API 必须是正确的	237
9.3 好工具让 API 设计更简单	153	13.3 API 应该尽量简单	240
9.4 兼容性测试套件	155	13.4 API 必须是高性能的	242
第 10 章 与其他 API 协作	158	13.5 API 必须绝对兼容	242
10.1 谨慎使用第三方 API	158	13.6 API 必须是对称的	245
10.2 只暴露抽象内容	162	第 14 章 API 设计中的矛盾之处	247
10.3 强化 API 的一致性	164	14.1 API 设计中的自相矛盾	248
10.4 代理和组合	168		

14.2 背后隐藏的工作.....	251	17.3.2 第二天的解决方案.....	317
14.3 不要害怕发布一个稳定的 API.....	252	17.4 第三天：评判日.....	320
14.4 降低维护费用.....	255	17.5 也来玩下这个游戏吧.....	327
第 15 章 改进 API	258	第 18 章 可扩展 Visitor 模式的案例	328
15.1 让有问题的类库重新焕发活力.....	259	18.1 抽象类.....	331
15.2 自觉地升级与无意识地被迫 升级.....	265	18.2 为改进做好准备.....	333
15.3 可选的行为.....	268	18.3 默认的遍历.....	334
15.4 相似 API 的桥接和共存.....	274	18.4 清楚地定义每个版本.....	337
第 16 章 团队协作	286	18.5 单向改进.....	339
16.1 在提交代码时进行代码评审.....	286	18.6 使用接口时的数据结构.....	340
16.2 说服开发人员为他们的 API 提 供文档.....	290	18.7 针对用户和开发商的 Visitor 模式.....	341
16.3 尽职尽责的监控者.....	292	18.8 三重调度.....	343
16.4 接受 API 的补丁.....	297	18.9 Visitor 模式的圆满结局.....	345
第 17 章 利用竞赛游戏来提升 API 设计 技巧	300	18.10 语法小技巧.....	346
17.1 概述.....	300	第 19 章 消亡的过程	348
17.2 第一天.....	301	19.1 明确版本的重要性.....	349
17.2.1 非 public 类带来的问题.....	304	19.2 模块依赖的重要性.....	349
17.2.2 不可变性带来的问题.....	304	19.3 被移除的部分需要永久保留吗.....	352
17.2.3 遗漏实现的问题.....	308	19.4 分解庞大的 API.....	352
17.2.4 返回结果可能不正确的 问题.....	309	第 20 章 未来	356
17.2.5 第一天的解决方案.....	310	20.1 原则性内容.....	357
17.3 第二天.....	313	20.2 无绪长存.....	358
17.3.1 我想修正犯下的错误.....	316	20.3 API 设计方法论.....	360
		20.4 编程语言的演变.....	361
		20.5 教育的作用.....	363
		20.6 共享.....	365
		参考书目	366

第一部分 理论与理由

发明、设计与编写 API 的过程，既可以看作是艺术创作，也可以当作是科学实践。因此，你可以把一个 API 架构师看作是一个努力改变世界的艺术家，也可以看作是一个架设桥梁的工程师。只不过在我所认识的人中，倾向于艺术家观点则占了大半，因为他们认为艺术家更具有创造性，设计的内容也更加自然和漂亮。但从纯艺术的角度来看 API，却存在着一个巨大的问题：激情是无法完全传递给他人的。艺术家们太自我，太主观了，即使想把他们的思路想法给其他人解释清楚，也多半是白费唇舌。艺术家在创作一件作品时，会将自己的感情体现在作品中，但对于欣赏的人来说，却很难带着同样的感悟来看待这件作品，当然这也正是艺术的魅力所在，每个人都可以在艺术作品上加入自己的想法。因此，如果架构师写 API，就像艺术家作画一样，则他所设计的内容就有被他人误解的风险。如果出现这种情况，那么开发人员在使用这些 API 的时候，就会背离原先 API 设计者的意图。

当然，这也不全是坏事。但一旦架构师要为团队设计 API 时，就会出现这个问题。要不了多久，这个 API 的各种属性就会暴露一大堆的问题，显得破绽百出。例如，API 最重要的属性之一是一致性，它可以避免 API 用户由于不一致而抓狂。如果要求 API 保持一致，就肯定得要求设计组的各个成员都有着很好的默契，合作无间。假设每个成员都像一个特立独行的艺术家，那么这种默契是很难做到的。而要合作无间，就必须有共同的愿景，还需要在团队成员中建立公共的术语来描述这个愿景。此外，还需要有方法论来指导大家如何来实现最终目标。一旦清楚地认识到了这一点，API 架构师就会祈祷让 API 的设计过程变成工程步骤，而不是艺术创造了。任何曾经管理过 20 个艺术家的人，想想去管理 20 个工程师的工作，必然对此心有戚戚焉。

委员会设计 API

我从一开始，就负责设计并实现 NetBeans 的大部分 API。当时，我们就坚信：好的 API 是不可能由委员会设计出来的。因此，一开始，多数程序员的任务是使用那些由他人设计的 API，或者是在此基础上实现功能，并在 API 的使用过程中提供一些反馈意见以便我们进行改善。但后来情况有所改变，其他开发人员也开始开发自己的 API 了。

我对此非常关注，会经常给相关的程序员提意见，偶尔还会提醒他们去做些别的事，或者从 API 中删除一些看上去怪模怪样的东西。有时候，我甚至还会亲力亲为地根据自己的思路把这些 API 重写一遍，然后再通知他们一定要用我写的版本。但我发现我的处境越来越不妙。虽

然我知道我想要的 API 是什么样子,但始终无法把我的想法很清楚地描述出来。我也无法说清楚他们为什么要采纳我所提的建议,因为我的同事并不总是愿意听从我的建议。我当然可以动用手中的权力强制他们按照我的思路来做,但这并不是问题的解决之道。我觉得一定有些地方存在问题,但无法说清楚错在哪里、为什么。因为我们团队中缺少一套共同的术语,从而束缚了沟通。这是因为一直以来我们设计 API 的步骤和方法更像艺术创作,所以这个问题不可避免。

一年后,我证明了当时的这一猜测。因为 NetBeans 是一个开放源码的项目,有一个 API 吸引了一个外部的开发人员。开始的时候,这位外部代码贡献者主要是做一些修复 bug 的工作。随后,他开始单独负责一个子项目,并设计相关的 API。原先负责设计这个 API 的人说他发现这个子项目的 API 变得越来越差,但他找不出原因,希望我帮一手。他说:非常不喜欢这些新写的 API,更不想把它们集成到他负责的项目中。但他也同样说不清楚这些新的 API 到底有什么问题。最后,他说这些新的 API 看起来和他原来的设想不一致。我曾经也对他说过类似的话,以我的标准来评价的话,他所编写的 API 与 NetBeans 的 API 的设想不一致!

经过多年的 API 设计工作,其间还和其他的 API 设计师同事一同设计我喜欢的 API,我逐渐认识到 API 的设计可以像一个工程项目那样整理出一套方法。API 的设计具有很多客观特性,只不过在开始的时候,不太容易整理出来。也就是说,我们可以把 API 设计过程转变为工程师可以理解的东西,把它变成一种科学。

每一门学科都有相应的理论在背后支持着,这个理论定义了该学科领域的方方面面。这个学科领域必须轮廓分明,定义清晰。轮廓越模糊,学科就越不严谨,它就越像一门艺术。本章定义 API 设计,详细说明 API 设计时要注意的各种问题,分析 API 设计过程中要面临的各种情景。而且,一开始就建立一个通用的术语库,让了解该理论的人更容易识别 API 世界中的各种对象及其相互间的关系。在此基础上,我们再得出 API 设计理论中一些比较深入又不太显而易见的结论。

要面向广大的受众写出好的 API 可谓困难重重,尤其是面对国际受众时就更加困难了。每个人都有自己思考和理解问题的固有思路,要想满足所有的受众,可谓难于上青天!要想立即满足所有的受众更是不可能的。此外,如果你的 API 是针对国际受众开发的,它还要处理各种文化差异。所以说写出广泛适用的好 API 是相当困难的。

写一本能够吸引各国读者的书也同样困难。每位读者个人的喜好,以及所处的文化,都会对他的阅读造成影响。有些读者喜欢先了解图书的写作背景,但有些读者就想直接翻到书中的例子,看一下那些例子是否对他们有用。想要同时满足这两个风格迥异的阅读群体,似乎是不可能的。Edsger W. Dijkstra 在他的文章“On the fact that the Atlantic Ocean has two sides”^①中对这一情况给出了精彩的诠释。有些人认为,纯理论的方法不仅难以应用,而且读起来也是极其乏味的,言下之意是,真实的例子会更加有趣一些。这种说法有可能是正确的,尤其在他们特定的文化背景下。

① Edsger W. Dijkstra 是一位伟大的荷兰计算机科学家,其最著名的理论分别是最短路径算法以及 Goto 有害论,同时他还是第一个 Algol 60 编译器的设计者和实现者,与 D. E. Knuth 并称为这个时代最伟大的计算机科学家。他于 2002 年 8 月 6 日在家中去世,终年 72 岁。文中引用的这篇文章“关于大西洋有东西两岸”写于 1976 年,可以在 <http://www.cs.utexas.edu/~EWD/transcriptions/EWD06xx/EWD611.html> 浏览全文。——译者注

另一方面，还有些人则愿先花时间建立一个共同的术语库，然后对要探索的内容一步步地深入了解。这种做法必须对术语做全面的介绍，否则很多用语就会有双重含义，经常导致混乱。很明显，想同时满足这两种读者的需求无疑是非常困难的，但我仍将尽我所能，希望能使每一位读者满意。

稳定的 API

我们这一行有个基本的术语是稳定的 API (stable API)。在 NetBeans 项目中，我一直使用“稳定的 API”这个词和很多成员进行交流，从没有想过会出任何问题。一直以来，在我的脑海里，这个词有着明确且清楚的定义。

有一次，一个同事给我解释了他对稳定的 API 的认识。他说，稳定的 API 是指它永远不会被改变！而事实上，尽管一个稳定的 API 会有某种程度上的“稳定”，但它有时候仍然需要有所改变。

我总是会碰到这种误解的情形，即使是最基本的 API 术语，在不同人的眼里也是有着不同的含义。当然，这种差异会打乱整个交流过程。如果交流的人们认为他们彼此了解所说的内容，而事实上他们各自的理解却不相同，那么还是立刻中止这种交流吧。因为交流双方形成的具体理解是背道而驰的，这种鸡同鸭讲的交流毫无价值。

所以我想最好还是先用一点时间来定义一些基本的术语。

综上所述，为了避免术语上的混淆，本部分会建立一个术语库，并分析 API 设计的常见内容。首先要建立一个基本的术语库，描述整个 API 设计的需求出现的缘由，并概述各设计过程的主要目标。如果作为读者的你，不喜欢这种写作风格，或者说你认为自己不需要了解这些内容，那么随时可以直接跳到第 2 部分，那里有许多代码示例、具体方案、技巧，以及各种小窍门。如果在阅读第 2 部分的过程中，有一部分内容让你觉得陌生，也不必惊讶，这可能是因为你错过了本部分讲述的一些背景知识。同样，如果你急于了解工具、编译器等相关内容的实用建议，也可以直接从第 3 部分开始阅读。只不过请在你的脑子中绷上一根弦，在阅读的过程中，如果您觉得这些建议不好理解，很可能是由于你缺少对 API 设计理论知识的了解，可以考虑回头再看第一部分。

不再啰嗦，让我们直接进入理论。首先讲述基础的理论知识，这将有益于我们对后面内容的阅读和理解。在我们满含激情开始了解如何设计优秀 API 之前，我们先从最基本的问题开始，即为什么要有 API？什么是 API？如何设计 API？

软件开发的历史很短，从有人编写和执行第一个计算机程序至今，也不足百年。尽管历史并不悠久，但其影响力并不逊于其他任何一种开创性的发明。有人把计算机科学发展史与人类认识真实世界的历史放在一起加以比较，这非常有意思，我们也借此来理解为什么软件开发需要优秀的 API。下面我们来试试这么做。

1.1 理性主义，经验主义以及无绪^①

在文艺复兴时期，现代科学产生了两大重量级理论，表现在哲学方面则为理性主义和经验主义。其中理性主义认为理智是信息的首要来源，并给出一个假设：只需要通过思考就能够理解和描述这个真实的世界。理性主义的支持者包括现代科学的众多先驱，像法国哲学家、数学家勒内·笛卡儿（René Descartes, 1596—1650），德国数学家戈特弗里德·威廉·莱布尼茨（Gottfried Wilhelm Leibniz, 1646—1716），还有泛神论的创始人斯宾诺莎（Benedict Spinoza, 1632—1677）。

理性主义可以说是源起于伽利略^②的自由落体实验，该实验证明了无论物体的重量如何，其下落的速度是相同的。这个自由落体实验的结果与人对该现象的本能认识是完全相反的，比如从高处同时丢下一块砖和一张纸，这两者肯定不会同时落地。伽利略和其他现代科学家的智慧体现在：他们在思考问题的时候，是把各种自然规律结合在一起，而不是孤立地分析一个问题。自由落体只是其中的一个规律。伽利略是如何发现这个规律的呢？他是先在大脑中模拟进行自由落体实验。他首先想象两个球，尺寸和重量完全相同，在高处将它们同时丢下。事实上，它们肯定会同时达到地面。然后，他再设想同样的虚拟实验，这次使用一个与原来相同的球，但将另外一个相同的球从中间分成两半，再合在一起变成一个球。可预见到，这个实验的结果与上一个实验应

^① 原文为 cluelessness，英文原意为“漫无头绪”，本书中将这个词翻译为“无绪”，在某些情况下会翻译成“懵懂”或“透明”。这个词在本书中的意思，很难找到一个合适的中文词来对应，它在本书中的主要意思为：即使某个人对某一样事物的内在本质并不清楚，也很可以很好地使用该事物，如大部分人都不知道电视机的原理，但并不妨碍大家使用电视机观看电视，在本书中作者想表达同样的意思，即开发人员不需要知道某个组件或者模块的内在实现原理，也可以很好地使用该组件或模块对外提供的 API 来完成自己所需要的功能。——译者注

^② 伽利略·伽利莱（Galileo Galilei, 1564年2月15日—1642年1月8日），意大利比萨人。物理学家和天文学家，近代实验科学的奠基者之一，科学革命的先驱。——译者注

该是完全相同的，两个球仍然会同时落地。现在，假设把第二个球给分成两半，然后用一条丝线把这两个半球连在一起。不管这两个半球之间的丝线长度是1厘米、1米还是更长，第二个球仍然会与第一个球同时落地。即使把这根丝线给拿走，还是会得到相同的实验结果。这个实验的结果完全是与人的本能经验相反。生活经验告诉我们，一张纸的下落速度要慢于石头。但纯粹通过思维进行分析却说明：重量不会影响下落对象的速度。

数学和物理学的潜意识

看到这里，读者会发现，我很喜欢用物理学上的一些故事或者典故。的确如此，自从我读了 Petr Vopěnka^①的书^②以后，这些内容就总在我脑子里晃来晃去，他的书描述了现代数学和物理学中潜意识的重要性。但我无法用自己的语言来诠释这些内容，所以只好在书中直接引用他所讲的内容。本书中，我时不时会引用他的一些观点，但说得会比较简练，因为原作是一本足足有800页的巨著，而且对所有的术语都有详尽的解释。很显然，本书不能这么做。深入地解释这本巨著，已经超出了本书的能力，更非本书的目的。所以，请读者原谅我对某些内容作了精简。

人们总说，伽利略把石块从比萨斜塔扔下去，然后发现了最著名自由落体规律^③。他也许真的这样做了，但那个完全基于想象而完成的虚拟实验，也毫无破绽地解释了同样的行为。这是历史上第一次仅仅使用理论知识就证明了：纯粹通过观察而得到经验是错误的。虽然在实际生活中，我们可以观察到较轻物体的下降速度比重物的确要慢一些。但我们现在知道，那是因为其他因素干扰了地球引力才造成下降速度有所差异。该实验证明了纯粹理性主义的可用性，这正是莱布尼茨与笛卡儿认为理性主义胜过经验主义的主要动力，它催生了整个理性主义哲学运动。事实上，这种思想一直确信：研究的主题应该是合理的，也必须是合理的。如果通过推理可以发现，则必然要有一个合理的起点。

在英吉利海峡的另一边诞生了经验主义。几乎在同一时代，那些伟大的英国思想家，如戴维·休姆（David Hume, 1711—1776），约翰·洛克（John Locke, 1632—1704）和乔治·伯克利（George Berkeley, 1685—1753），他们都坚持认为，人类对世界认识的主要来源是经验。如果看不到、听不到也感觉不到我们现有的这个世界，那么人们根本无法对这个世界进行思考。理解万事万物的基本方式就是体验，或者换个科学用语，可以称之为做实验。如果追溯一下，是谁第一个使用科学实验的方式来向大众表述一个想法或者一个假设是正确的，那么据传说，当属伽

① Petr Vopěnka 是一位捷克的数学家，代替集合论是由他和他的学生提出的。——译者注

② Petr Vopěnka *Úhelny kámen evropské vzdělanosti a moci* (Prague: Praha, 1999)。

③ 关于伽利略是否做过自由落体这个实验，一直是有争议的，经历史学家考证，没有任何理由表明伽利略做过该实验，因为在伽利略的著作中没有提过任何该实验的信息。而伽利略做比萨斜塔实验的这一传闻来自伽利略晚年的学生维维安尼，他在伽利略的传记中提到伽利略曾做过此实验，事实上，一位亚里士多德学派的物理学家为了反驳伽利略，真的于1612年在比萨斜塔做了一个实验。结果表明，材料相同但质量不同的物体并不是在同一时刻到达地面的，伽利略在《两门新科学》中对此有一个辩护，意思是说，重量1:10的两个物体下落时只差很小的距离，可亚里士多德却说相差10倍。为什么忽视亚里士多德如此重大的失误却盯住我小小的误差不放呢？这也表明伽利略没有做那个实验。

利略。但从经验主义的观点来看，世界并不一定要是合理的。这个世界不需要追根究底，甚至不需要存在，这一切都无关紧要。如果认为它有意义，它就有意义，不需要理解。

但站在今天的立场来看，如果孤立地基于这两种极端的方式来观察世界，那么就会严重偏离事实。现代科学重视通过各种实验来证实相关理论。此外，笛卡儿也非常清楚地指出，对于科学来说，实验是不可或缺的。所以对我们来说，应该将这两种对立的观察方式组合起来。事实上，人类现在已经在这样做了。大多数人不会思考周围事物的哲学意义，我们更关心的是结果。生命应该是一种享受的过程，而不是乏味的研究和推理。而且，我们日常使用的东西能够正常使用即可，通常不需要关注它的工作原理。例如，我们对汽车和手机的工作原理一无所知，我们只是理所当然地在用它们，我们不关心它们是怎么做到的。我们在生活中不需要琢磨出头绪来。

从理性到无绪

为国际受众设计 API 和创作书籍是很困难的事情。个人喜好和文化差异都会影响我们处理问题的方式。理性主义者更喜欢先谈理论，了解现实对象背后的联系，而后再创建实例将理论应用到现实世界中来。而经验主义者则正好相反，先想尽各种方法来获取实践经验，最终才会对事物间的联系给出判断。

本书从一种针对性无绪的角度来解释如何设计 API。我们将 API 看作一种可以将无绪极大化又能得到可靠结果的完美工具。正确理解无绪的真正含义是十分必要的。书中的内容是建立在理性主义的基础上的，在开始的章节中会大讲理论，而非实例。对很多人来说，这也许不是什么好办法，但我不能同时满足两类读者。不过读者不必失望，在理论部分讲完以后，就能建立一个通用的 API 设计术语库，然后我们就会看到大量的实际应用。

对大多数人来说，懵懂无知是一种生活方式。这是今天理性主义和经验主义结合在一起的结果，它无所不在。今天的程序开发和软件工程方法也是如此。

1.2 软件的演变过程

在 20 世纪 40 年代和 50 年代初时，编写代码是一件非常困难的事。人们不得不学习机器语言，同时还要知道寄存器的大小和数量，有时候，事情不妙，还要拿起螺丝刀亲自上场^①，去连接计算单元的信号线。人们的主要精力不在于思考一个算法，而放在将算法编写成可执行程序上，这是一种枯燥又机械的工作。

FORTRAN 语言的出现，就像天使为人间送来了福音。与经验主义者相同，它允许程序员只关心数学公式的计算，而无须考虑其他的内容。程序员可以完全不用了解汇编语言，也不用再关心计算机内部的技术细节。他们完全可以把这些琐碎的事情丢到一边，更专注于更重要的事情，如如何将数学公式写成相应的算法步骤交给计算机进行运算。FORTRAN 语言简化了软件开发过程，几乎没有什么东西是不能用 FORTRAN 来处理的，这是经验主义者的一个巨大成功。

^① 早期计算机的软硬件并不像今天区分得这么清楚，很多工程师都是软硬通吃的人物，而且那个时候电子管计算机经常出故障，需要进行维修，像 bug 一词就来自于一个硬件故障。——译者注

只不过，编码仍然不是一件简单的事情，它需要变得更加简单。为了解决这种矛盾，由此又产生了一些新语言（如 COBOL^①），大家打出的口号是“新手就能学”、“管理层也能读懂”之类的，进一步简化一些特殊任务的编码工作。虽然到了今天，已经没有人会考虑用 COBOL 编写一个新的系统。然而，在那时候，COBOL 与汇编或者 FORTRAN 语言相比，能大大简化对数据库的操作。经验主义风头更甚。

并非所有人都喜欢经验主义。始终会有人认为，万事万物都应该是合理的。我们身边肯定有这类人，程序员也不例外。在 20 世纪 50 年代，理性主义者约翰·麦卡锡（John McCarthy）在 λ 演算^②的数学模型的基础上发明了 LISP^③语言，λ 演算的数学模型包含了大量的理论知识。数学是一门纯理性学科，因此，LISP 完全由纯粹的推理来支持的。据说，在设计 LISP 语言时，很多人认为，保持数学的纯洁性是最重要的。多么执著的理性主义者！他们认为语言无须是有用的，甚至无须是可实现的，但它必须是纯粹、干净且合理的。

所以，有人认为计算机科学可以分成两大学派，欧洲派和美国派。美国人通常更加务实一些（的确如此，历史上的美国是实用主义的发源地），而欧洲人则更愿探索远景。在计算机工程界也能发现这一特点。很多示例表明欧洲人更倾向于理性主义而轻视实用主义。Edsger W. Dijkstra，这位基于消息的计算和信号量同步模式的发明者，在 *Selected Writings on Computing*^④ 中写道：从他的角度出发，他认为“编程是一门数学味儿很浓的工程学科”。按照他的说法，我们应该坚持理性主义。然而，当我环顾四周，看看程序员编写的那些会计软件、医院所用的病患管理软件等，我就感觉其实写程序和做饭差不多，没有多少数学的内容在里面。当然，编写好的算法需要一定的数学背景。然而，Niklaus Wirth，这位发明了 Pascal、Oberon 和其他软件系统的大师，却认为：简单而且优雅的方案往往更加有效，只不过想找出这样的方案却很困难，需要更多的时间。这个说法当然没有问题，但在当前上市时间决定能否成功的年代，根本没有时间用来探寻最

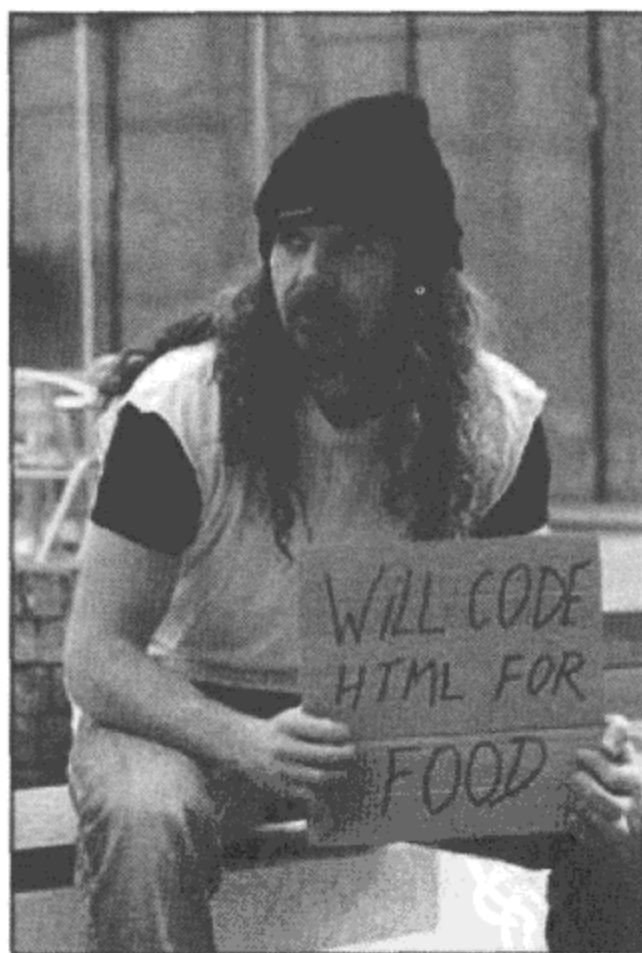


图 1-1 我要编写 HTML 页面谋生

① COBOL 是 Common Business Oriented Language 的缩写，是最早的高级编程语言之一，是世界上第一个商用语言。

——译者注

② λ 演算 (lambda calculus) 是一套用于研究函数定义、函数应用和递归的形式系统。它由阿朗佐·丘奇 (Alonzo Church) 和他的学生斯蒂芬·科尔·克利尼 (Stephen Cole Kleene) 在 20 世纪 30 年代引入。——译者注

③ LISP 是由约翰·麦卡锡在 1960 年左右创造的一种基于 λ 演算的函数式编程语言，它是第一个函数型编程语言，主要应用在人工智能 (AI) 上，包含多字符提取函数，供作自然语言的分析之用。——译者注

④ Edsger Dijkstra, *Selected Writings on Computing: A Personal Perspective* (New York: Springer-Verlag, 1982)。

佳方案。

看来，在今天的软件工程世界中，理性主义已无存身之地。特别是现代的程序员，几乎没有几个推崇理性主义了。图 1-1 表明，我们已经完全丧失了秉持理性主义的程序员了。这并不新鲜，Dijkstra 就曾经说过：“大部分程序员都没有能力编写出好的代码。”真理啊！现代社会不断地对软件业提出新的要求，但我们又可以做些什么呢？

对我们常人来说，基于理性主义还是经验主义去理解这个世界，其实无关紧要，最好的建议莫过于无知一点儿，同样，编写程序也是如此。这个世界，或者说我们现在所处的这个社会并不需要每个人都成为哲学家。这个社会给我们这些懂得少些也即更无绪的人留有空间，而一切也能正常。同样，软件工程也不要求所有的程序员都是受过高等教育的科学家。如果想开发软件，就需要一个系统，程序员可以无绪也能产生可靠的软件系统。

事实上，上文一直在说的无绪并不是说完全不懂编程。很明显，仅仅在键盘上随机地输入字符是不太可能产生一个可编译的程序的。所以对于程序员，知道如何编写代码是一个先决条件（就像人类社会中，人们必须具有观察、理解、讨论电视广告等能力一样）。在软件工程中所谓的无绪是指程序员在不需要深入了解很多内容的情况下，就可以写出好的代码。这里不能简单地说哪些知识是必需的，而哪些则不是。我们的目标是找到一种编码实践方法，让开发人员不用深入了解所有事情，即选择他们所需的知识。我把这称为针对性无绪^①。

1.3 大型软件

在 21 世纪的前 10 年中，大部分软件系统都可以用脏乱差来形容，没有哪个软件的设计配得上用优雅这个词。这主要是因为开发时，大家的目标就是用尽可能少的资源来尽快地开发完项目。为了达到这个目标，开发团队往往直接复用现有的一些软件框架，而完全不顾这些重量级的框架其实是远远超出他们的需要。

发布网页

最近，我想在自己的服务器上放一个动态网页。有两种方法来完成这件事，一是在某个端口开一个套接字，读入数据流，再写点什么作为应答；二是基于现有的一些技术组装个系统。这两种方法我都尝试了一下。

首先尝试的是第一种方法，即“从零开始”。我先阅读 HTTP 协议规范的相关文档，解析读入的头文件，再写了输出。完成这些功能的代码量不大，经过一些调试以后，程序就可以正常运行了。但接下来，还要再添加一些功能，如为网页的访问加上安全验证和处理 POST 请求等。我还需要阅读 RFC^② 文档，并实现文档中的各项内容。最终所花费的工作量比我预计的要大得多。

① 原文为 selective cluelessness，直译为选择性无绪，这里翻译成针对性无绪。——译者注

② RFC 是 Request For Comments 的缩写，是一系列以编号排定的文件。文件收集了因特网的相关信息，以及 UNIX 和因特网社区的软件文件。目前 RFC 文件是由 ISOC (Internet Society) 所赞助发行的。基本的因特网通信协议都在 RFC 文件内详细说明。——译者注

因此我尝试了第二种方法。我选择了 Tomcat 作为 Web 服务器, 然后编写了一个 Servlet, 配置好相关的文件, 短短时间内, 我就完成了所有的开发工作。仅有一个缺点, 那就是我现在的系统大小超过 1 MB, 而不是 50 KB。

现代的软件都是基于大型组件进行组装的。没有人独自从上到下完成所有内容, 而是你会先安装一个可靠又低廉的操作系统, 然后在上面安装 Web 服务器和数据库服务器。在安装了大量的软件以后, 才能动手解决真正的问题, 比如说想生成一个 HTML 页面。在此时写一个 HTML 页面简直易如反掌。但没有人敢说整个系统非常简单。事实上, 这个对外只提供简单页面浏览的系统已经复杂到了极点, 相信没有人敢说自己完全了解整个系统的全部内容。这可以称为无绪状态在软件开发中的绝佳示例: 编写这个 HTML 页面的程序员在对整个系统仅有很少了解的前提下, 仍然可以完成自己的工作。

整个方案看起来就像一台推土机的工作方式一样。开发时, 如果需要数据库, 就安装一个数据库软件, 如果需要一个比较可靠的运行平台, 就安装 Java 平台和一个应用服务器, 完全不会考虑这些框架是否过重了。总能找到一台足够大的推土机, 把这些东西推到应用上去。如果发现应用占用了太多的内存, 从理性主义观点出发, 你会去考虑优化, 而如果继续采用推土机式的处理方式, 只需要再买上几 GB 内存就搞定了。如果系统的运行效率还不高, 甚至可以多花点钱搞个集群, 或者干脆上虚拟化技术, 这样一层层堆上去, 整个程序就会变得越来越大, 永远都不会变小了。

那么使用推土机式开发方式是好还是坏呢? 事实上, 这种处理问题方式的开发效率很高, 相比之下, 试图寻找 Wirth 那种“简洁又美观的解决办法”的效果更好, 但是太耗时。如果你将目光放到 Web 系统上, 会发现大量的系统是采用这种推土机方式建立的。Amazon、Yahoo!, 以及其他的大型站点也采用了这种方式, 而且运行得很好, 没有出现什么严重的问题。一切似乎都表明, 对于现代的软件设计和编码来说, 推土机方式完全可行。我们已经拥有了不可思议的计算能力, 这使得人们更倾向于野蛮的程序开发方式^①。而且这种工作方式行之有效!

1.4 漂亮, 真理^②和优雅

我敢肯定本书的很多读者对于我美化无绪会愤愤不平。现在这样一种重量级的开发方式往往会把一个程序变成一堆垃圾, 怎么能用这种开发方式取代以前那种优雅的软件开发呢? 如此丑陋的应用怎么能够保证其正确性呢? 其实答案是肯定的, 我们只需要去仔细看一下我们大部分人现在所担心的事情。

科学理念仍然深植在我们心中, 并始终影响着我们的思维方式。这些由多个世纪以前的古希

① 作者这句话的意思是指, 早期计算机处理能力比较低, 网络也不好, 硬件配置与现代计算机相差很多, 所以编写代码时要特别注意, 才能保证性能, 但现在的计算机处理能力已经有了若干个数量级的提高, 所以编码时对性能等各方面考虑的就比较少了。——译者注

② 原文此处为 truth, 直译应该为真理、真实, 但考虑到下文所说的内容很多都与几何学有关, 国内对于几何学的规则一般都翻译成公理, 翻译本节时也将部分 truth 翻译成公理。——译者注

希腊人创建的科学，仍然能在今天影响着如何看待真理和美丽之间的关系。对古希腊哲学家来说，最有价值的科学知识其定义是非常简明的，这些知识不会被他人误解，其意义清楚明白，绝不含糊，于是几何学就成为所有科学中最有价值的。这是因为几何学不是一个关于现实世界的科学，它只是几何图形知识，比如说两点间的线是直线，球是一个圆形体，都只是理论知识。几何学的完美是其他任何科学都无法匹敌的，特别是那些与现实世界相关的科学更无法与之相提并论。从远处看时，一个圆状的石头像是一个几何上定义的球。但走近看时，就会发现，这块石头所谓的圆状只是一个概念上的幻想，它其实不是光滑无瑕的。所以岩石方面的科学就不如几何学那样清晰。它必须考虑到真实物件间必然存在的误差。

古希腊人的几何学世界与现实世界的一个重要区别在于稳定性。现实世界中的物体是在不停地变化着的，比如说今天的石头，到了明天，就可能被打成碎片，或者雕刻成某样物品。但几何学中所有的几何物体都会保持一致，直角永远都是 90 度。这样可以保证几何学中关于物体及物体间关系的思路 and 理论会永远有效。几何学中的真实和现实世界中的真实完全不同，前者可以永远为真，而后者则未必。所以古希腊人把几何学看作绝对真理的科学。

几何学的物体会因为其复杂性而产生改变。比如说，定义一个圆就需要定义它的圆心和半径。定义一个椭圆，则需要定义两个半径，一个半径是不够的。所以定义一个圆要比定义一个椭圆简单。同样，定义一个正方形要比定义一个长方形简单。从几何学的角度来说得清楚一些，那就是圆要比椭圆更加纯粹或者更加漂亮，正方形也比长方形纯粹而漂亮。在古希腊的哲学家认识到这一点以后，几何学就不仅仅是一门关于真理的科学，更是一个充满美丽和优雅的科学。从此以后，真理与美丽就成双成对地出现，不仅体现在几何学中，而且在艺术和其他领域也是如此。古希腊的雕塑也体现了几何学的美，雕塑中使用了很多不同的比例（比如说头应该是躯体高度的 $1/8$ ^①，还有黄金分割，等等），这些都体现了几何的真实、漂亮和优雅。

在文艺复兴时期，古希腊风格以及古希腊人对于比例、美丽及和谐的思想，在艺术和科学方面可谓是无所不在。那时的艺术，正如文艺复兴其名，完全是建立在古希腊美学的遗产之上的。但古希腊美学的影响力并不限于艺术，它也进入了哲学以及刚刚诞生的、研究现实世界的物理学领域。伽利略和其他人将几何学引入了现实的世界。他们将现实的世界抽象成理想而且完美的几何学世界，像透过玻璃窗一样看这个现实的世界，他们眼中看到的不仅仅是一个现实世界，更是现实世界背后隐藏的几何世界。比如说，他们把现实世界中的物体看作是点的集合，它们的移动方式是弹道曲线，以圆的方式自转。这一切把几何世界与现实世界紧密地结合起来了。几何学变成了现实世界背后的本质。伴随着几何学的发展，现实世界也迎来了真实与美丽。

文艺复兴时期的物理学取得巨大的成功，牛顿的物理学定律和此前的几何学相结合，使得物理学几乎变成了最完美的科学。它在描述现实世界的同时还能保持几何学的优雅。行星为何按一条椭圆轨道运行，抛出的物体为何按抛物线轨道运行，这一切都可以用科学来加以解释。在此基

① 这种比例在中文，有人称为九头身，意思就是头部的长度与身高的比例为 1:9，也就是头与主干的比例为 1:8。

——译者注

基础上, 人们可以对未来做出一定程度的预测。物理学诠释了这个现实的世界及它的运作方式, 同时揭示了其背后隐藏的规律。从此以后, 在人们的眼中, 这个世界就不再是一团迷雾, 它是如此地美丽。于是, 了解牛顿物理学的人会发现, 真理和美丽又紧密结合在一起, 这次它们不仅存在于几何世界中, 也存在于现实世界中。

牛顿物理学是文艺复兴时期最杰出的理论, 它为文艺复兴画上了完美的句号。经典物理学是如此地优雅、正确和美丽。它基于欧几里得^①的几何学来描述这个现实世界, 事实上, 它是理性主义最完美、最精确的表达形式, 仅仅结合人类的思维而不用借助于经验就创建了这样的物理学世界。然而自牛顿物理学创立以来, 物理学已经发生了变化。爱因斯坦^②让我们认识到, 空间并不像欧几里得模型所描述的, 事实上它是弯曲的。量子理论证明了几何学并不是一种能够描述现实世界的模型。事情变得越来越复杂, 这导致现代物理学与古希腊几何学已经完全没有关系了。当然科学仍然有其用途, 可以用来揭示某些真理, 但它也与几何学的距离也越来越远了, 现实世界似乎是越来越无法用美学来定义了。

另一方面, 大多数成年人(包括软件工程师在内)只知道牛顿所建立的物理学。我们也有很多人听说过相对论, 但却没有几个人可以解释相对论。因此我们有个假想, 认为这个世界仍然可以用牛顿物理学进行完美的诠释, 我们仍然相信, 这个世界的其他部分也是美丽的。事实上, 每一类科学都尝试着能够像几何学和物理学一样, 变得纯粹。只有它结合了真实与美丽, 才被认为是良好的科学。也许美丽只是能够更自然地识别和记忆, 而相反, 混沌理论却是没有什么结构可言的。

电影《密码迷情》

最近我看了一部名为《密码迷情》(Enigma) 的电影, 这部电影讲述的是在第二次世界大战中的一个爱情故事, 与安全加密相关。有人问作为数学家的主人公是否喜欢数学。他的回答是: “我喜欢数字, 因为数字是真实和美丽的结合。当你发现一切在变得更加美好时, 你会知道前行的方向是正确的。数字让你可以更接近所有事物背后的秘密。”我很难想象还有比这句话更好地褒扬希腊人热爱真理、美丽和优雅的话了。一部给非科学家看的浪漫爱情电影中竟然也会出现这样的对白, 可见这三者合一的理念是多么深入人心。

计算机科学和软件工程其实也不例外, 它们也需要美丽和真理。然而, 应该牢记一点, 软件的首要目标是在发布给客户以后能够可靠地运行, 以提供客户需要的功能。在软件正式发布前的冲刺阶段, 所有的软件工程师根本不会想到美丽这个词, 而是把精力集中在修复 bug 上, 严格地说是绕开严重 bug 上, 然后按计划来发布软件产品。事实上, 这个时候, 简单和优雅根本不是工作的目标。虽然大家都会感觉应该做这件事, 但已无暇顾及。现在我们既然认识到这一点, 就可以祭起无绪开发这样的大旗, 作为从今往后的软件开发方法论。

① 欧几里得 (Euclid) 是古希腊著名数学、欧氏几何学的开创者。——译者注

② 全名为阿尔伯特·爱因斯坦 (Albert Einstein, 1879 年 3 月 14 日—1955 年 4 月 18 日), 理论物理学家, 相对论的创立者。——译者注

1.5 更好的无绪

我们已经看到，简单和优雅都不会作为评价软件部署成功的标准。就像哲学，理性主义过于学术化了，不能帮助我们去理解现实世界中的日常问题。而那种推土机式的务实工作方式，却让人感觉很有发展前途：把市场上可用的组件装配成应用程序，不管用了多少功能库，都把它们粘在一起，只要它能用，也不需要了解清楚个所以然出来。虽然很多人不认同这个观点，但它的确是当今大型软件项目不由自主采用的做事风格。然而，既然现在已经认识到了这个问题，我们能否使无绪的方法更好地工作呢？

推土机式的工作方法的优点在于，即使参与者（如程序员）不完全了解系统情况，也能得到不错的结果。第一次听到这种话你可能会吓了一跳。但事实上，我们一直都是这么干的。你开汽车，但不需要去了解汽车的内部设计；你刷牙也不用先学习化学。对于开发人员也是如此，写一个简单的 Win32 应用程序，并不需要了解 Windows 源代码，只需要对 Windows 的 API 有所了解，再找到一些适当的文档就可以了。大部分系统都是如此。像开发 Linux 代码，编写 Java 程序，做一个网页，你都只需要了解冰山一角，就可以完成这些工作了。之所以能这样完成工作，是由于用抽象包裹了每一个库和框架。这种抽象内容也就是 API，它将系统的复杂性隐藏起来了。这也正是本书的主题所在。

在工作时，如果不需要深入了解很多内容时也能完成任务，那么在这种情况下做出来的系统就更可靠。本书会逐步探讨各种方式来帮助大家在不深入了解各种细节的前提下，也能够很好地使用库，并对后续的版本仍然可以这么做。

“无绪”一词的起源

2006 年 10 月，我在俄勒冈州波特兰参加一个 OOPSLA^①会议。在其中一个主题讲座上，我第一次听到了“无绪”这个词。Martin Rinard^②应邀在这次会议上发表演讲，他演讲的题目是“在开发和维护软件系统时，应该避免让开发人员深入了解系统”，这个主题非常有煽动力，同时也引发争议。

他通过了一些证据来说明，人类的大脑功能是有限的，只能处理有限数量的数据。如果要建立日益变大的应用程序，就必须做到这一点：每个人只了解应用程序的一部分，也能完成程序的开发。他的谈话包含了如下 3 个方向的探索。

- 验证程序正确性
- 系统工程
- 容错性

① OOPSLA 是一个年度会议，此年度会议为面向对象编程系统、语言以及应用程序举办，主办方是计算机协会 (ACM) 的 SIGPLAN 以及 SIGSOFT 团体。——译者注

② Martin Rinard 是麻省的一位计算机教授，它在 OOPSLA 所做的讲座为“Minimizing Understanding in the Construction and Maintenance of Software Systems”，可以在 OOPSLA 的维基上得到讲演的 PDF 文稿，地址为 http://www.oopsla.org/oopsla2006/index.php?title=Martin_Rinard's_Talk。——译者注

他在演讲时还探讨了在一个程序存在错误的时候，如何还能让程序正常运行。对于那些热爱真理、漂亮和优雅的人来说，这样一个观点太可怕了，但对于那些开发真实软件系统的人来说，却是可以接受的方案。

虽然我很喜欢 Rinard 先生的演讲，我也同意凡是软件都存在错误这一结论，但这本书下面将集中讨论系统工程，以及验证程序的正确性。不过，在开发高可靠性的软件系统时，还要尽可能地做到无绪，这是一个富有吸引力的目标，而且也是有望达到的目标。

“无绪”这个词并不是一个贬义词。我们只是用它区别两种层次的理解水平。有一种理解可以称为“浅层理解”，它是指对一种事物的了解程度只限于掌握使用方法即可；还有一种理解则可以称为“深层理解”，是指会对某种事物背后的原则、规律有所掌握^①。在日常生活中，我们通常只需要“浅层理解”。我们可以看电视，而并不需要深入了解电视机的工作原理。如果我们需要找出当前自己所在的位置，也并不需要了解全球定位卫星是如何运作的。了解这些事物背后的规律已经不是我们所要关心的内容。对我们来说，能够认识 GPS 上的经度和纬度其实就已经足够了。当然，有些人需要了解更深入的内容。如 GPS 设备、汽车或电视机的维修人员，就需要深入了解相关内容。不过，即便如此，他们所需要的仍然可以算是“浅层理解”^②，他们也并不需要了解事物背后的每个小细节。当然如果说确有其必要性的话，我们可以去学习电视机、汽车等领域的专业知识。然而，这种对事物进行深入学习的必要性通常不大，因此，大多数人在日常生活中只需要做到“浅层理解”就足够了。

其实在软件开发中所说的“无绪”，也表示在大部分情况下，我们只要做到“浅层理解”就可以了。“针对性无绪”这个词用在这里只是提醒大家，有些内容需要深入了解，而有些内容则无需如此。所以“针对性无绪”（在后面的内容基本上只使用“无绪”或者“透明”）是个彻底褒义的词。

① 所谓的“浅层理解”也就是中文常说的了解，而“深层理解”则可以称为精通，但翻译时还是遵守了原文。

——译者注

② 这一个“浅层理解”可以对应到中文常说的熟悉。——译者注

设计 API 并不容易，且代价不菲。相比之下，试图创建一个带有 API 的产品比发布一个丝毫不包含任何 API 的产品工作量要大得多，而且后者更加容易。尽管如此，在“无绪”的原则下，我们认为，基于设计合理的 API，你可以设计出更好的软件系统，同时无需深入地了解系统细节。在开发系统子模块时，良好的设计，加之使用系统各个独立组件的 API 可以有效地改善你设计时所使用的系统工程方法论。改进现有的系统设计技能，是一种将“无绪”利益最大化的有效途径。

2.1 分布式开发

无绪的模型需要利用全世界范围的软件项目中大的组件模块，以装配的方式来开发应用程序。要尽可能多地复用，而不要从零开始编写软件，这样就可以保证产品团队能将重点集中在软件的独特功能上，即应用程序的具体业务逻辑上。这样，开发人员就无需花费时间去创建和编写基础设施，重用现有的框架和由第三方提供的功能库即可。相信现在不会有人因为自用的原因去写一个 SQL 数据库服务器软件，而是会使用一些商业或者开源的数据库。创建一个私用的数据库可以说是一种低效的资源浪费。对于软件技术的其他领域也是如此。Web 服务器、编程语言及其功能库、集成开发环境还有富客户端程序框架^①都已经有了设计良好的组件可供使用了。就像活动板房，只需将其各个组件从架子上拿下来，组装到一起，并根据需要适当抛光即可。这种模式极大地缩短了软件产品的开发周期。

在组装应用程序的时候，需要将各个独立的模块整合在一起。这些模块之间需要相互通信，通常来说，是通过定义好的 API 来完成的。每个组件的 API 都能增加你迈向无绪的步伐。不必了解组件内部机制，只阅读该组件的文档，调用其 API 就能够使用该组件的功能。利用 API 可以避免去了解第三方组件的细节。

抽象也会泄漏天机

API 可以被看作是对功能和组件内部实现的抽象。通常来说，外部人员只要了解抽象即可，但在某些情况下，一个 API 也可能会“泄漏”内部组件实现中的一些信息。来看一个文

^① 即 Rich Client Platform，缩写为 RCP。——译者注

件系统抽象的例子。通用的文件系统将文件和目录以树状的层次结构组织在一起。在获得一个资源以后，可以访问其内容^①，或该资源的子资源。资源的内容通常是使用流方式^②来提供的。现代操作系统几乎都使用了这种方式来支持对文件内容的访问。这样在开发时，程序员就不需要去关注实际的存储和文件系统的类型。在读取文件时，不需要知道该文件是放置在硬盘驱动器、光盘、闪存盘中，还是网络上。不同的存储介质以这种通用的抽象提供了一致的处理方式。

然而有时候，API 却会把这些具体实现的相关信息泄露出来。比如说，在读取闪存盘上的内容时，可能会因为用户拔下了闪存盘而使得正在访问的资源突然消失了。或者在使用网络存储的时候，可能会因为网速很慢或者网络性能差，使得文件资源的访问有很长时间的延迟。在这种情况下，开发人员不仅要了解抽象的信息，可能还要了解具体实现的部分内容。

但从“针对性无绪”的角度来说，这样做也是可以接受的。这种通用的 API 只是一个基本的抽象，它提供服务，只要用户不在意网络延迟，或者移去闪存盘时引起的崩溃。如果你在意的话，还有一些高级的方式来检测处理这些情况。因此文件系统 API 可以让程序员做到无绪，而且也不会影响我们在必要情况下深入了解系统信息。任何一个对外提供功能的 API 都应该做到这一点。

一个典型的应用程序不仅仅是一个或者几个类库组成的。当今开发的应用程序，会用到全世界开发人员提供的很多开源功能库，这些开源贡献始于 Unix 内核、C 语言基础功能库和命令行工具。发展至 Web 服务器和 Web 浏览器，以及 Java 实用程序，如 Ant、Tomcat、JUnit 和 JavaCC 等。事实上，每个类库都有自己的 API，因此每个写这种软件的人都在从事 API 的设计，不论他自己意识到没有。

对于各种 Linux 发布版来说，将模块集合在一起打包，然后发布出去，这是常见的一种组装方式。软件是由不同的人编写的，然后再通过编译、打包，最终整合在一起。通常，不同 Linux 版本的发行商只需要写一个中心管理器的模块，用来提供质量保证，让所有选中的软件都可以正常运行。对于大部分供应商和用户来说，这种方案行之有效，能够降低创建 Linux 分发版本的成本。这种模式还有一个成功的案例，就是 Mac OS X 系统，它是在一般 FreeBSD Unix 版本上添加了一些由 Apple 公司开发的功能模块。

分布式开发自有其特殊之处。最大的不同就是，整个程序的源代码已经不再完全处于开发人员的控制之下，而是散布于世界各地。与完全基于内部代码仓库^③中的源代码来构建一个程序相比，这种构建软件的方式是明显不同的。

我们需要认识到，在这种开发模式下，产品的开发进度是无法全面掌控的。软件的源代码和开发人员都散布在全世界，而且开发人员各有自己的进度安排，所以项目领导者对此无法一一掌控。猛一听，感觉项目完全不可控，但真实情况并不像乍听起来那样危险。任何一位项目领导者，只要其带领的团队超过 50 人，那么他一定明白，所谓的全面掌控往往只是一种理想

① 如果是文件，就访问文件内容，如果是目录，就访问其子资源。——译者注

② 在 Java 中，即常用的 `InputStream` 和 `OutputStream`。——译者注

③ 这里的代码库其实就是暗指常用的版本管理系统。——译者注

状态。在开发过程中，考虑到时间、资源等因素，有时不得不砍掉某个功能或者发布一个老一点的版本^①。对于分布式开发模式来说，每个人在使用功能库的时候都可以自由选择使用老版本还是新版本。

事实上，要设计良好的 API，开源并不是唯一驱动力。商业组织也提供了大量共享的功能库和框架。还有不少公司提供了一些现有标准的具体实现产品，如各大数据库厂商的产品都支持 SQL 标准，还有一些厂商提供了自己的 API^②。带有自由许可证方案（liberal licensing schemes）的开源运动已经成为组件的首要来源，这些组件可以作为可复用构件来使用。最终用户知道很多开源软件方案，不需要支付任何费用即可使用。而且开源软件没有许可权的限制^③，这对于开发人员来说也是很重要的。很容易就可以根据自己的需求来找到一个组件，然后把这个组件用作自己应用程序的一部分。迟早会有人这么去做。这也说明每一个开放源代码的组件迟早都应该有一个 API。这些通用的组件是由各类程序人员开发的，可能是刚开始做项目的大学生，也可能是能够投入大量时间在某个项目的开发人员，还有可能是某些公司专门雇用的开发人员，他们看到了开源软件的商业机会。这些开发者有着不同的开发技巧，不同的开发方式。无论如何，设计优秀的 API 是一件非常重要的事，因为 API 是无绪使用库的开始。类库和框架这类东西自然是越多越好。但要想让无绪重用能够成功，就必须让 API 体现库的内在功能。这就是为什么 API 的设计如此重要，成为本书写作的主要动机。

2.2 模块化应用程序

模块化的应用程序是由分布式团队开发出来的独立组件组成的。这些独立的组件通常都会提供一个自己的 API，当然在具体执行的时候，也需要第三方组件的 API 或者其他功能才能保证正确运行。例如，Tomcat 服务器需要 Java 运行时实现。同样，标准的 C++ 模板库也需要 libc，这样才能调用 printf 方法。如果使用了大量的组件，那么面临的最大问题就是能否看清整个应用的全貌。只有理解了整个系统以后，才能理清楚模块间的交互关系。在上一节中，我们可以看到一个组件的 API 只会把其最重要的功能给暴露出来，大部分情况下，用户无需关注其内部的实现，只需要集中精力了解 API 即可。但如果系统包含成千上万个组件的话，光是组件 API 的信息就会非常多，很难无绪地处理。所以我们来看看能否找到一些可行的方式，在不需要深入了解所有组件的情况下，也能把组件整合在一起来构建可用的系统。

在设计 API 时，第一课也是最重要的是：给自己的组件起个漂亮的名称。这些名称必须是独一无二的，能够根据该名称在整个系统中查找到相应的组件，而且这个名字应该尽量做到让用户闻其名知其意。对 Linux Kernel 来说，其中 kernel 这个词就是一个很好的名字；libc 是 C

① 指到了指定版本发布时间，可能最新的版本尚不能稳定或者目标没有达到，所以不能正式发布，只能将前一个内部的稳定版本发布出来。——译者注

② 如 JDBC 和 ODBC 都需要对外 API。——译者注

③ 原书是这样说的，但事实上，开放源码并不意味着就可以免费使用，很多开源软件都是双协议，对开源软件或者非商业软件都是免费的，但商业用途并不免费，如著名的 Ajax 框架 Ext，所以原书中说的并不是很准确。

——译者注

语言的基础库，这个名字也算不错；org.netbeans.api.projects 是 NetBeans 平台上的一个支持项目结构的组件，这个名字堪称完美。一般来说，现代的所有组件都有一个名称，因此要开发新组件的话，也很自然地应该先给它起个名字。但深入地考虑一下就会发现，所谓的名称，其实只对人有意义，对于机器来说，无论阿猫阿狗哪个名字都无所谓。对于机器来说，任意一个 16 进制的名字都可以，哪怕是 0xFE970A3C429B7D930E。事实上，有些组件的名称还带有人名，这也说明了组件的名称其实主要是针对人而非机器。组件的名字对客户和终端客户很有用，他们可以利用组件的名字了解组件对于供应商和集成商来说，名字也同样重要，有助于他们使用这些组件来构建应用程序。

在知道如何为自己所开发的组件命名以后，还要看一下每一个组件运行的环境。没有哪个组件是在真空中运行的。它需要从周围的环境中取得相应的服务。所以，使用一个组件可能还需要完全了解该组件的环境需求^①，搞不好还要深入了解其内部实现才能明确其环境需求，如果运气好一点，通过运行该组件也可以了解其环境需求。但这都不是我们想要做到的“针对性无绪”，因为这样提高了对集成人员的要求，他们在构建一个应用程序之前，还需要了解每一个库的很多细节才行。如果一个库对其用户提出了如此高的要求，那么会严重地阻碍其用户群的发展。事实上，一个库的大部分用户不需要了解该功能库的内部实现。也应该如此，用户可以在不深入了解一个库内部情况的前提下，就能使用库完成自己的工作。通过正确设计和描述每一个独立组件就可以做到这一点。如果一个组件能够自动处理自己所需要的运行环境，那么相应的集成人员在使用该组件时，就能做到尽量无绪，因为相应的环境不需要人为干涉，不需要使用编译器、链接器和集成工具。

在模块化系统中，每一个组件都会提供一些其他组件所需要的信息。组件的设计者需要将这些信息设置进去，或者由打包工具自动处理。例如，RPM 安装文件的构建就是如此，Fedora、Mandriva 和 SUSE 这些 Linux 的不同版本，都使用该格式来创建它们版本的安装包，这些安装包能够自动检查本地动态链接库以查找该软件运行时需要的动态链接库，并自动调整安装包内容以保证能够使用这些动态链接库。不管这些工作是工具自动完成，还是由人工完成，都只需要处理一次就可以了。完成该工作的开发人员应该就是该独立组件的作者，他们知道组件的内部机制，了解组件运行时所需环境，从而正确地列出其依赖的内容。这也是“无绪”的又一个例子：一个工程师多花点时间和精力来列出组件所依赖的内容，而组件的用户，如应用的集成人员或者其他开发人员，则只需要这个组件提供的信息来让工具自动完成相应的工作即可。

配置类路径时的噩梦

纯粹只依赖 Java 平台来编写 Java 应用程序的好日子已经一去不复返了。可用于开发 Java 应用程序的开源软件类库已经很多了，而且还在每天增加。结果，几乎现在每个 Java 程序都会使用那些已经打包的 JAR 类库，如 Apache Commons、HttpClient、JUnit、Swing 部件等。如果要运行这样一个程序，第一件事就是正确设置程序运行时的类路径。直接把这些库都包含在

^① 作者的意思应该是指在执行某组件时，如果不符合前提环境，可能会抛出某些异常通知开发人员进行修改，如找不到某个类或者某个配置文件。——译者注

类路径中是一件比较简单的事情，但是每一个库都还有自己额外的依赖库，也需要将这些额外的依赖库也加入到类路径中，如此反复，直到所有的依赖库都加入到类路径中，这样做使得类路径的配置变成了一个噩梦。

我最近有机会使用了 FreeMarker，这是一个漂亮的模板引擎，我要把它作为一个类库用在 NetBeans 的开发中。引入一个 freemaker.jar 类库文件很容易，但在我尝试检查它的所有类是否成功连接的时候，却觉得非常痛苦。这个 JAR 包引用了很多别的项目，如 Apache ANT、Jython、JDOM 和 log4j，还有 Apache Commons Logging。对于 FreeMarker 来说，真需要这么多的项目吗？如果没有这些库，那么 FreeMarker 还能运行吗？如果后者答案是肯定的话，那么又是哪些功能用到了这些库，而且这些功能是否会因为库的缺失而引起系统的崩溃？我不知道，其实我也不想知道，我只希望在使用这个库的时候，能做到无绪，但现在我无法达到这个目标。我现在必须深入研究源代码，找到那些我们在使用 FreeMarker 时不会调用到的第三方类。我曾经祈祷 FreeMarker 能够使用某种模块化系统来标识它依赖于哪些内容，如 NetBeans Runtime Container 就提供了类似的功能。

如果从技术角度来对分布式开发给出一个好的解决方案，那么就是将应用程序模块化。那些非模块化的程序由大量代码组成，然后相互之间紧密地耦合在一起，而模块化程序则可以由很多小的独立代码段^①组成。这些小的代码段是独立存在的，且被唯一地标识，通过公开接口的方式供他人使用，每一个代码段还可以正确地描述该代码段所需要的运行环境（比如说第三方组件或部件使用该代码段时，如何准备其环境）。对 Linux 版本的发行商来说，这种开发方式是可行的，而且也证明了其有效性，可以交由分散的团队按照自己的计划来分别开发相应的模块，然后再由一个管控者将所有的模块集成在一起，这样可以有效地规避时间安排和团队分散的风险。而且这样做可以更好地支持无绪的开发模式：如果开发人员能够将自己所负责部分的依赖正确地描述出来，对系统的集成人员来说，在集成该部分时就不需要了解组件的内部信息了，并且仍然能够成功地构建最终的应用程序。

但现在系统开发面临着新的挑战，要知道，软件开发中的组件并不是静态的，而是会随着时间的推移在不停地向前发展，时刻在变化着。不管这种改变是因为要修正现有的 bug，或者增加新的功能，但一个 API 确实是在不停地变化着，因此仅仅通过一个名字是不能够唯一标识一个 API 的。为了让很多组件能够整合在一起运行，就必须能正确地标识该组件提供了何种 API。

例如，如果一个用 Java 编写的类引用了 String.contains(String) 方法，那么这个类可以在 Java 5 中运行，因为这个方法是从 Java 5 这个版本开始在 String 类中提供的。但这个类在 Java 6 中也可以使用，因为这个版本的 Java 也提供了这个方法。考虑到 Java 团队的兼容性处理方式，相信对于最新的 Java 7，这个类也是可以使用的。但老版本的 Java 没有提供这个方法。所以如果使用老版本的 Java 运行使用了该方法的类，就会出现类无法正确连接的问题^②。

① 这里的代码段，在本书中是组件或者模块的另一个说法。——译者注

② 如果要出现作者说的这种情况，在使用 Java 5 或以上版本编译代码时，要将兼容性级别调整为以前的版本，否则执行时，就会先抛出版本号不支持的异常，可以通过 javac -? 了解更多这方面的信息。——译者注

事实上，如果开发人员想明确某个类或者某个应用对运行环境的需求，就必须列举该类调用的所有方法，以及它引用的所有类和字段。当然关于依赖的描述信息会非常长、不具有可读性，有时甚至比实际的源代码本身更大。可以简单设想一下，有一个类声明自己需要某一个版本的 Java，要求这个版本的 Java 所提供的 `java.lang.String`，必须有一个构造函数及 `length` 和 `indexOf` 这两个方法，而且这个类还实现了 `java.io.Serializable` 序列化接口。这样细节的描述已经使得组件超出了无绪的范畴，而本书却一直在强调无绪。尽管一台计算机可以自动检查这些约束关系，但对于人来说，要做到这一点就比较困难了。人们更合适处理一些简单的场景，如使用自然数来标识组件的版本。如果按照这种方式来描述前面的需求，就变成“当前类库要使用 Java 5”。

但仅仅使用数字还可能导致别的问题。设想一下，如果你想将组件 A（它要求使用 Java 5 版本）和组件 B（它要求使用 Java 6 版本）集成到自己的程序中，如图 2-1 所示。在这种情况下，需要有一个环境能同时支持组件 A 和组件 B，但问题在于到底使用哪一个 Java 版本。为了处理这种问题，大部分 API 开发模型都会用到兼容性概念。即如果在某个版本 N 中加入了某个 API，在接下来的版本 $N+1$ 、 $N+2$ 、 $N+3$ … 都会兼容这个 API。所以处理前面问题的方式就是使用 Java 6 版本：因为组件 B 要求使用这个版本，而组件 A 则需要 Java 5，但这也说明组件 A 可以在 Java 6 上运行，因为 Java 6 会对 Java 5 兼容。事实上，引入这种 API 开发模型大大简化了开发人员集成程序时的工作。他们在集成程序时，可以做到非常无绪。每一个类库都有一个“小小”的用户需求：向后兼容。事实上，这个需求绝对不小。它非常复杂，并不是一个可以轻易达到的目标，这也是本书的一个主题所在。本书会强调，如果要把分布式开发团队开发的组件集成到大型程序中，就需要让集成人员做到最大化的无绪，也就意味着这些开发团队不仅要细心开发各自组件和相应 API，还要保证兼容性。

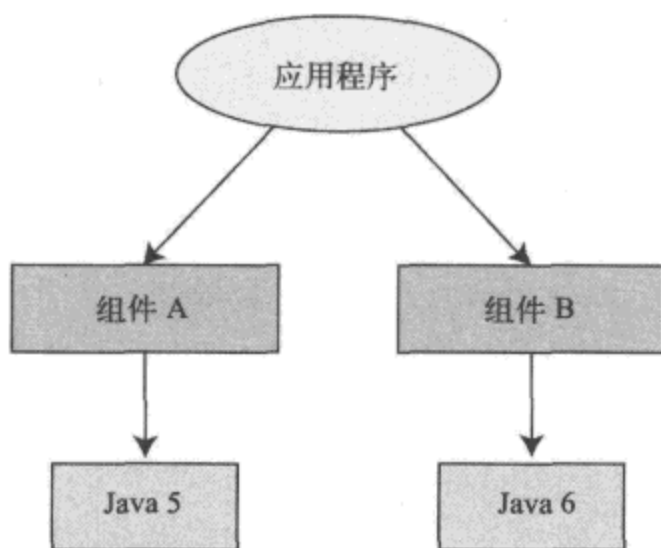


图 2-1 应用程序需要一个组件的两个版本同时运行

软件版本的非线性

给软件的版本进行编号的时候，最常用的编号方式不是基于自然数。相反，更习惯使用一个圆点分隔的十进制。这么做自有其必要性，因为在软件开发中，发布的版本其实是非线性的，因为开发时并不一定是沿着预先定义好的主干进行开发，而是会有很多分支，每个分支都有其特殊含义，比如说一个用来修复 bug 的分支等。如图 2-2 所示，假设有一个 1.1.1 版本，这个版本的编号要比 2 版本小，其



图 2-2 版本树

包含的功能也会比 2 版本要少一些，但事实上该版本的发布可能还落在 2 版本的后面。

模块化应用程序的每一个模块都会有一个版本号，像 1.34.8 这样。如果有新版本发布了，则会有一个新版本号，比原有版本号大（按字典顺序排列），如 1.34.10、1.35.1 或 2.0。

在一个模块化系统中，如果要表示一个组件对其他组件的依赖关系，可以用名称加上需要的最小版本号。不管是 XML 解析器，还是数据库驱动，或者是文本编辑器，乃至 Web 浏览器，都可以通过它们的依赖声明来获取其版本号。比如说，从依赖声明中，可以知道该模块需要版本高于 3.0 的 XML 解析器和版本高于 1.5 的 Web 浏览器等。这些假设都是建立在完全兼容的情况下：也就是说，即使某个模块的版本号比较高，集成的系统也仍然可以正常工作。

想只通过一个版本号就能提示一个功能库从创建至今的所有变化，显然是不可能的。但版本号比较实用，而且更重要的是它可以让相关人员在使用某个功能库的时候，尽量地保持无绪，这样也允许那些功能库的开发人员来通过一个容易识别的版本号来识别其功能上的改变。

只有在开发和设计时遵守一定的规则，前面那种处理依赖关系的方案才能生效。第一个规则是，如果发布了一个新的版本，那么前一个版本提供的功能和约定，对于新版本应该继续有效。当然，这事说着容易，做起来可麻烦多了，否则还要一个质量控制部门做什么。第二个规则是，如果本组件的外部依赖有所改变，那么就需要与其进行同步调整。因此，一个模块化系统需要依赖某个新功能，比如说要使用一个 HTML 编辑器，就要加入一个“html editor>=1.0”这样的依赖关系的声明。如果还要使用 Web 浏览器在 1.7 新版本中引入的新特性，那么还要加入“web browser>=1.7”这样一个依赖关系的声明。

组件的某些版本可能存在特定的 bug，所以为了处理这种情况，还会使用第二个版本号（实现版本号）。与前面标准的版本号不同，这个版本号通常是一个字符串，如 Build20050611，对于这种版本，测试方面就没有什么区别了。如果某个 bug 出现在 3.1 版本中，那么在 3.2 版本中却不一定有。所以出于 bug 修复的考虑，为功能库加入特定的第二个版本号还是非常有用的。

一个系统会有相应的版本和依赖关系，应该由一个管控者来统一管理，以保证系统中每个模块的需求都会被满足。这样一个管控者可以对系统进行检查，保证所有已经安装的模块能够保持一致性。Linux 的分发版本中可以分别使用 dpkg 和 rpm 两种命令来对 Debian 和 RPM 两种安装包进行检查。而且还可以使用这种依赖关系处理程序运行时的一些特殊要求。

例如，基于 NetBeans 平台开发的应用程序，都是由多个模块组成，它们在运行时被加载。NetBeans 模块系统使用声明的依赖关系来查找需要加载哪些相应的模块，不仅如此，在加载每个模块的时候，还会根据依赖关系为其设定相应的父 ClassLoader，这样可以有效地保证各个模块对应的类路径不会产生冲突。这样也可以加强各个模块声明的依赖关系：如果一个模块没有声明自己依赖于某一个指定的外部模块，那么就无法调用该外部模块的代码；而且除非所有的依赖都已经满足，否则当前模块是不会被加载的。

2.3 交流互通才是一切

到这里，我们理想的应用程序的轮廓应该很清楚了。应用程序应该基于无绪原则来开发，尽

量让最终负责集成的相关人员不需要深入了解系统也可以把集成工作做好。所以，我们理想的应用程序应该基于模块化架构来开发，可以由散布在世界各地的独立开发团队分别负责编写相应的模块。他们可以按照自己的日程来安排工作，以达到最终的目标。但这种做法却存在一个重要的问题，那就是模块间的关联关系。

大多数模块并不能孤立存在，它们要依赖于其他模块提供的环境。只有少数模块才可能完全不依赖其他模块而独立对外提供功能。实际上，大部分模块化的组件都需要其他组件为其提供服务。这就意味着这些模块的开发人员需要去发现和了解如何使用外部模块提供的 API。而这样的模块也是由其他的开发人员编写的，所以还会出现一个隐藏的问题，即该模块又依赖于另外的模块。

API 不是给计算机用的

我花了很长的时间才明白，API 的第一受众并不是计算机，其实是人，是我们这些开发人员，这恰恰与我最初的想法相反。可以设想一下，对一个编译好的程序来说，它的第一受众或者说唯一用户就是计算机，编写好的代码是用来编译和执行的，所以可以把程序看作是人类和计算机沟通的一种方式。

但这个想法对 API 来说并不正确。要知道，API 中往往包含了大量的内容，还有相应的说明文档，其实对于要运行程序的计算机来说根本不需要这些东西。在我认识到这一点后，我发现从学校里学到的那些编码技巧其实并不足以教导我来设计 API。API 的设计与这些技巧简直风马牛不相及。从来没有人教过我如何设计 API，我估计其他程序员也没有受过这方面的专业教育。

共享功能库和不同框架的作者需要和他们的用户进行交流。他们还可能需要与那些无意中^①使用了他们库的用户进行沟通。交流的方式有很多，比如说可能通过电子邮件，或者参与电话会议之类的方式，但这些方式都需要面对面或者耳对耳地进行交互才行。所以最常用的方式还是通过 API 自身以及相关的文档。API 的用户在开发过程中研究最多的肯定是 API 的文档。如果想成功且高效地开发模块化应用程序，就需要有清楚、易于理解的说明文档，而且 API 自身也尽量可以自描述。这样做有助于开发人员完成各自的工作，无论他们身在何处。

API 应该建立在普遍适用的基础知识上，这样其用户可以很容易理解接受该 API。如果 API 是可靠的，它就必须能避免用户误用，而且可以在未来被加以改进：要知道很少有哪个 API 在第一个版本就能做到完美。本书中讨论的问题，大部分都与如何避免错误有关，只有避免了相应的错误，才可能让 API 在未来得以改进。

最重要的方式就是分解（separation）。要做到分解，就需要为设计和维护 API 制定相应的规则。如果不能分解，那么整个产品就是紧耦合的，一旦这个程序被开发完成后，也就不需要对外提供一个 API 了。但是，现实世界中的产品开发，往往是以模块化的方式进行的，可以让分散的

^① 这里所说的无意中，是指某个程序员使用了某个模块，然后该模块又使用了另外的模块，所以程序员也必须使用另外的模块，但这个程序员可能没有意识到这一点，这种是不自觉使用，也可以说是被强迫使用。——译者注

团队基于这样的组件架构来开发产品，各个团队完全可以根据独立的时间表来开发自己的项目，当然这样做的前提就是需要有一个稳定的契约，来保证这种沟通的有效性。

并非所有项目的质量都要高得像 API

前段时间我同表哥聊了一些我们工作上的内容，说到我们所做的项目，以及我们如何组织和管理这些项目。我给他介绍了无绪这一概念，并说到如果能够做到针对性无绪的话，就能开发出更可靠的软件系统。虽然他和我的观点还是有少许的不同，但在大部分内容上，我们还是取得了一致。

有时，特别是开发的应用程序是给最终用户^①使用时，代码质量似乎没有必要像 API 的代码质量那么高。对于这样的系统，其用户只有人，而人与自动化系统相比，是可以容忍一定错误的。而且在应用程序部署完成后，都不会有什么大的变化。它会持续运行多年，直到确有必要时，它才会被另外一个程序来替代，而这个替代的程序是完全可以被重写，或者重新组装而成的。

这与使用 API 来编写一个模块是完全不同的。这样的应用程序就像恒星一样，会始终存在，不会轻易地被新软件替代。这也说明，本书中给出的各种原则，并不适用于所有的项目，它们更适用于系统中的那些关键的可复用组件。而对于其他非核心的功能模块来说，则完全不用去担心未来的情况，它们的开发过程完全可以适用“开发一次，然后弃之不管”的软件方法论^②。

然而，这些外围的功能通常都是围绕一个核心组件建立的，这个组件才是整个系统的灵魂所在。这样的组件是本书的关注点，应该基于 API 设计的最佳实践来开发这样的组件。事实上，我的表哥也承认他所开发的系统中同样存在一个核心库，必须正确地设计和开发这个核心库，才能保证程序能正常运行很多年。

2.4 经验主义编程方式

很少有人能够以合理的方式来分析一个新 API。没有人在学习使用一个 API 的时候，去尽力理解该 API 背后的设计思路。现实正是如此：现在的开发人员想的是在无需理解所有内容的情况下，尽快完成他们的工作，如果他们使用的 API 能够完成他们的工作，就不会花时间来深入思考这个 API。

在本书第 1 章中提到的那些经验主义哲学家，如果看到了这种情况，想必会非常自豪。不需要花费大量时间来思考、学习和理解相应内容，开发人员就可以通过尝试调用 API 中的某个方法来测试是否可以完成其目的。如果测试结果正确，开发人员就会很开心。如果测试结果不正确，开发人员就会继续调用别的方法。NetBeans 开发人员把这种开发方式称为经验主义编程方式

^① 这里的最终用户就是指最后使用软件的人。——译者注

^② 这里的意思是指，开发此类复用性不强的软件时（如某个财务系统），只需要将注意力集中在核心组件上，如数据库、算法等，这些内容要考虑复用，而提供交互的用户界面，则不用太关心。——译者注

(empirical programming): 先用一个 API 做个实验, 如果不成功, 就再试一下别的。经验第一, 然后才是深入了解。不过有时候, 并不需要深入了解。

因此这就对 API 的结构提出了要求。API 必须是自描述的, 即用户无须任何文档就可以正确使用该 API。这样的 API 能引导用户利用其提供的各种功能和元素来轻松地完成任务。用户在编写代码的时候, 能很容易找到解决方案。IDE 能够提供各种提示, 但都应该是与主题相关的, 且不会将用户引入歧途。只有这样, 凭经验去使用 API 才能成功。

返回值不能为空

我必须承认, 相比去深入理解某个用到的 API, 我也更喜欢这种“经验式编程”, 特别是调用那些有返回值的方法时。

在 Java 语言中, null 值是一个非常特殊的东西。任何一个变量类型都可以通过 `obj=null` 的方式被赋成 null。当调用这个对象方法或者访问其字段的时候, 就会抛出一个 `NullPointerException`。在一个表现良好的程序中, 应该避免出现这种问题。开发人员可以在调用对象方法或者访问对象字段时之前, 先通过 `obj!=null` 来避免出现这种问题。但这样做会使代码变得非常零乱, 而且难以阅读。

这样导致了某些程序员都把 null 作为一个异常值, 在设计库的时候, 他们都会尽可能地避免使用 null。经常会在 Javadoc 中看到这样的说明文字: “这个方法永远都不会返回 null。” NetBeans 项目组把这个规则作为一个默认的规则: 除非一个方法中声明了它可以接受 null 作为参数, 或者允许方法的返回值为 null, 否则这个方法就不能接受一个为 null 的参数, 也不会返回一个为 null 的值。

虽然, 不管文档中对一个方法承诺了多少这方面的内容, 但还是会有大量的代码对返回值是否为 null 进行检查。这些检查的结果值不能是 false, 但开发人员不能确定, 尤其在没有任何阅读相关文档的时候。

这是一种防御性的经验主义编程方式, 我经常也会使用这种编码方式。这样做可以保证程序运行时的安全而不出乱子。很容易就可以检查出 null 值, 然后结束执行, 而不是等程序发布给了成千的用户, 再莫名其妙地抛出一个 `NullPointerException` 异常。

有时, 创建能够复制和容易修改的示例代码, 对那些第一次试用一个 API 的程序员是很有帮助的。如果这些示例代码能够包含 API 常用的功能, 那么这个试用者会觉得这样的 API 比较容易使用。然后, 要深入 API 编写代码就会复杂一些, 而且可能还需要读上一些文档才行。这可以接受。如果某个人投入了一些时间和精力来试用某个 API, 并完成了任务, 那么他对这个 API 的态度就会变得积极主动。此时, 他会感觉这个 API 很有用, 自然愿意花费一些时间来阅读相关文档, 并发现 API 提供的一些其他功能。

我们总是使用“经验主义编程方式”, 每一个 API 都需要支持这种类型的用户。如果一个 API 的方法名称、类名称以及方法的组合处理都能够帮助开发人员正确地编写代码, 那么即使第一次使用该 API 的开发人员也能正确地使用该 API, 并愿意多知道一些关于此 API 的内容。

2.5 开发第一个版本通常比较容易

如果你想说服其他程序员使用你设计的 API，那么必须先取得他们的信任。你需要让这些程序员们相信，你有能力来设计、编写和维护一个稳定的 API，而且能在后续的多个版本中坚持做到这一点。即使不考虑使用一个 API 有多困难，如果该 API 的新版本发布后，所有的客户端代码都需要重写，这简直就是厄运的开始。想一下，有这么多的开发商^①在微软的 Windows 平台开发程序，如果因为 Windows 用户安装了一个操作系统的新版本，所有的程序都无法正常运行了，估计真是一场大的厄运。因此 API 设计会涉及一个信任问题：通过遵守本书给出的最佳实践，可以有效地减少不兼容问题，从而获得用户的信任。

有时对兼容性的破坏是无法避免的，毕竟我们生活在一个现实世界，绝不可能尽善尽美。但只要存在着相应的可能性，就应该避免破坏兼容性。将一些内容标识为不建议使用，然后明确以文档的形式来告诉用户如何将现有的系统迁移到新组件，这些对于组件的用户来说是可以接受的。但如果某个组件每个新版本都会进行一番大的改动，那么客户可能会直接放弃这个组件，选择另一个组件，以避免这种迁移的痛苦。对组件的用户来说，如果组件不兼容，成本将非常高昂，简直就是对用户的一种惩罚。这将导致失去用户的信任。

有些人会说，如果加把劲，多用点心，即使是第一个版本，也可能实现一个完美的 API。他们说的可能正确，但必须费心劳力才有可能达到这个目标。即使你已经尽了最大的努力，设计 API 时所面对的需求仍然可能会随着时间而调整。如果需求产生了变化，最好的情况是 API 提供的功能不够用了，最差的情况就是，因为需求的改变，使得 API 提供的功能已经完全无用了。无论何种情况下，设计一个 API 就必须提前做好准备，使得在需求改变的情况下，API 也能随之改变，而且不会破坏其用户的现有代码。因此，我后面会用很多篇幅来讨论 API 设计如何考虑后续改动的需求。

没有借口

事实上，第一个版本几乎从来没有完美的，但不能说这是因为第一次设计，所以设计上做得不好，这不应该成为一个借口！对于第一个版本来说，设计起来总是比较容易的，而在后续版本中加以改进则比较困难。所以在第一个版本中，要尽量将其设计做到完美，同时不要忘记，今天看来完美的设计，明天可能就变得非常差。改变迟早都会出现。

开发 API 的第一个版本通常比较容易，问题往往来自于后续版本。要对一个 API 加以改变，并加入一些新功能，但同时还不能影响原来的功能，这是一门精妙的艺术，既要考虑原有的 API 被使用的所有方式，还要保证这些用法适用于新版本，太困难了。

对 API 进行改进是一件不可避免的事情，因此对于用户来说，API 必须能够在后续的时间里得以改进。我们必须努力做到在保证一个组件的可靠性的时候，还能修正其 bug 并处理用户的新

^① 这里所谓的开发商，只是一个统称，任何开发任何的人员或者公司，也不管他们是否为商业，还是开源，都统一称为开发商，也可以称为开发方，但前者更为通用，所以译文沿用了该术语。——译者注

需求。要做到这一点，就要避免“闭门造车”。如果在设计 API 时就考虑到未来改进，而预先留出相应的空间，那么即使出现升级的情况，也不会给用户添加额外的工作。

好比言论自由

除了改进以外，决定 API 好坏的还有其它多方面重要的内容。API 必须有对应的文档，它也应该容易被用户了解和使用。但除了这些内容以外，能够被改进才是最重要的。如果 API 能够被持续改进，那么初始版本中的问题就能在新版本中修正。

如果你拥有言论自由的权利，你就可以说出其他缺失的权利，并要求拥有这些权利。如果将要失去某些其他东西，你可以去努力争取并可能最终得到它。

如果 API 已经为其后续的改进做好了准备，虽然它现在有些丑陋，而且缺乏文档，还很难使用，但这些问题都可以通过发布一个新版本，并且更新 API 来加以修复。这正是我为什么把改进看作 API 最重要的内容，在设计一个功能库和其接口时，应该将其放在首位。

因为开发人员不喜欢那些不必要的工作，特别是那些因为库的开发商考虑不全面，最终使得库无法兼容，进而为开发人员带来一些不必要的工作，所以在开发可供他人使用的组件时，必须要考虑在组件改进的时候还能保持兼容。



对于使用 Java 语言来编写各种类库的开发人员来说，往往对 API 存在着一个误解：认为所谓的 API 不过是类和方法，再加上那些用来对类和方法提供说明信息的 Javadoc。事实上，Javadoc 的功能只不过是以索引的方式帮助人们查找和使用 API，而广义的 API 远超上述所说的范围，它还包括很多其他方面的内容。

在继续下面的话题之前，再来提醒大家一次，我们为什么要来开发一个 API 呢？其根本原因在于：我们希望能够将大块的构建模块“无绪”地集成成应用程序，这些构建模块，包括共享类库、框架、预先定义好的应用程序架构，以及这些内容的组合。我们相信如果每一个程序员都可以很好地完成自己所负责的模块，也就是说他们设计的 API 完全正确，那么程序的集成工作就会变得非常简单，集成人员不需要花费时间进行调试、阅读源代码、打补丁，更不用去考虑他人到底是如何来设计和开发一个模块的。一句话，我们完全可以在“无绪”的状态下来完成集成工作。

使用源代码

从本章开始，将会提供一些源代码来解释一些设计上的问题，所以要额外说一下如何获取和使用这些源代码。本书中所有的示例代码片段，都是来自于真实的项目。可以通过访问 <http://source.apidesign.org> 这个地址，然后按照页面上的提示信息，来下载这些源代码。

如果从前面“无绪”的角度来说，一个 API 包含的内容，远比在 Javadoc 中定义的 API 要多很多。为了让大家对这个词的理解更具体，本章会列举一些 API 类型范例。

3.1 方法和字段签名

对于一个 API 来说，最明显也是最没有争议的内容就是：一个 API，就是一堆类以及类中方法和字段的集合。也许所有使用 Java 语言编写类库的开发人员都会将类文件打成一个 Jar 包文件。这些类及其成员就可以称为这个类库的 API。

当然，不是所有签名都是 API 的一部分。对于类库中所包含的那些非 public 级别的类，就像 private、package 这一类访问级别的字段和方法，通常不建议把这些内容公开给用户使用，所

以这类内容可以不归到 API 类别中。虽然这些内容没有公开,但可以通过反射或者其他的底层技术来访问这些 `private` 字段,或者调用某些 `private` 方法,但这种方式是一种非常规调用方式,类似于黑客入侵,不建议使用。而且对于一个系统来说,内部的多个组件进行交互时还需要使用反射来进行功能调用,这说明必然是有些地方出现了问题,或者说提供的 API 不足。因为存在各种问题,才迫使他人只能通过这种非常规的方式来访问未公开的内容。要注意的是,这种非常规的方式往往受限于具体的版本,某些反射调用也许在老版本上正常运行,但在新版本上就行不通了。因为未公开的内容不是 API,这表示它不对外承诺兼容性。

这并不是说在开发 API 时不能使用反射技术。事实上,即使在 Java 基础平台的代码中,也可以找到一些使用反射技术的例子,这些例子都说明了反射技术存在的合理性,只要适当地使用反射技术就可以很好地简化开发工作或者创建出新的开发模式。比如说,在 JavaBean 规范中,就是通过反射技术才能在运行时获取几个 JavaBean 方法和字段的相关信息,还可以通过反射来修改字段的值。如果没有反射的话,那么这些信息就只能在设计时使用,会大大降低 Java 语言的威力。有些场合,还会要求一个类提供一个默认构造函数,因为有些代码会使用 `Class.newInstance` 来创建类的实例对象。使用反射技术来创建对象时,不需要知道具体类型,只需要以字符串的方式传入类的全名即可创建一个实例对象。因此,反射是一种非常有用的技术,但是开发人员在使用时必须小心再三。如果 API 中使用了反射,那么建议开发人员最好在文档中对其加以说明。

3.2 文件及其内容

开发中有一项内容是经常被人们忽略的,但它却是非常重要的,那就是应用程序执行时要读写的文件以及这些文件的格式。

先来看一个简单的例子,现在有一个 `telnet`^① 程序,分析一下它是如何与一个支持 Kerberos^② 验证的系统进行交互的。这两个组件是由不同人员分别开发的,因为编写加密代码和处理套接字连接属于两个不同技术领域的范畴。但只要一个文件 API 就可以让这两个组件相互协作。

这个 `telnet` 应用程序在运行时会从系统的指定位置读取一个配置文件,诸如 `/etc/telnet.rc`,或者 `$HOME/telnet.rc`,这个文件放置了当前用户的配置信息。Kerberos 也依赖于这个文件,在一开始安装 Kerberos 时,就会根据 `telnet` 的信息来写入相应的内容,从而访问需要验证的系统。而这个 `telnet` 应用程序就会加载配置文件中的内容,验证用户的登录密码,最终以加密的方式连接到系统中。

三明治式的文件配置

与 Unix 应用程序不一样,有些 Java 程序的运行根本不依赖于磁盘^③。但对于 NetBeans 平

① `telnet` 协议是 TCP/IP 协议族中的一员,是因特网远程登陆服务的标准协议和主要方式。它为用户提供了在本地计算机上完成远程主机工作的能力。——译者注

② Kerberos 协议主要用于计算机网络的身份鉴别(Authentication),其特点是用户只需输入一次身份验证信息就可以凭借此验证获得的票据(ticket-granting ticket)访问多个服务,即 SSO (Single Sign On)。由于在每个客户和服务之间建立了共享密钥,使得该协议具有相当的安全性。——译者注

③ 如 Java Applet,是加载远程的文件。——译者注

台来说,特别是它的配置文件,还是建立在文件的基础上。但它并不会直接访问磁盘上的文件,而是通过虚拟文件的方式来处理文件。

基于 NetBeans 平台所开发的程序,其配置方面的内容与 Unix 系统中的/etc^①路径差不多。Unix 系统中的每个功能或 NetBeans 平台上的每个模块都有一个或者多个经过精心设计的位置,每个位置存放不同的配置信息,由程序在运行时读取。比如说,在图 3-1 中,NetBeans 界面上所显示的主菜单和工具栏,都是程序在运行时读取 /Menu 和 /Toolbars 这两个路径下面的配置文件,然后根据配置文件动态生成的界面。事实上,这种配置信息也是一类重要的 API,因为其他模块必须了解这些文件的信息,然后才能将自己模块的配置文件也放置到指定目录中。而且配置文件的名称不能随意更改,否则就表示已经注册了的配置文件不能被系统识别出来。同时配置文件的格式也必须保持稳定,否则就无法加载这些配置文件。

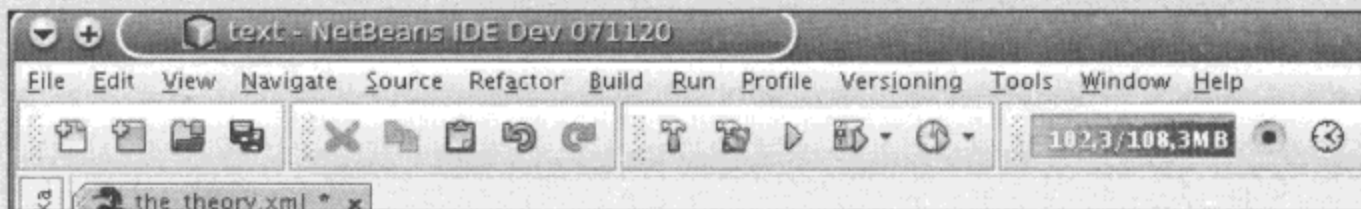


图 3-1 NetBeans 平台通过读取配置文件来生成菜单和工具栏

NetBeans 的这种方式与 Unix 中的/etc 文件配置目录差不多。但有一个最大的不同之处,那就是在 NetBeans 的文件配置目录中,大部分配置文件都是虚拟出来的;也就是说这些配置文件其实是从很多 XML 文件系统中抽取出来的内容。就像一个庞大的三明治,配置文件也是包含了多层内容。每一个组件(也就是 NetBeans 中定义的每一个模块)都要提供一个 XML 文件层,会注册多个配置文件到系统中。下面就是这个 XML 文件的格式:

```
<filesystem>
  <folder name="Menu">
    <file name="Open.instance"/>
  </folder>
</filesystem>
```

接下来,NetBeans 系统会将这些配置文件合并到一个文件系统中。这样做有点像现代 Linux 发行版本的处理方式。合并后会出现一个庞大的文件系统,它可以用来支持 NetBeans 系统运行时所需要处理的配置信息。整体看来,就像一个包含了多层结构的三明治一样。

这是我在 2000 年做出的一个设计,我一直对此颇为得意,因为它避免了在磁盘上存放过多的配置文件,而是先从每一个模块中读取配置文件,然后再通过虚拟文件的方式对外公开。而且这个设计还带有一个突出的优点:对外通过虚拟方式提供的配置文件总能保持一致性,而对于那些模块来说,则很难做到这一点,它们在安装时会把配置文件复制到磁盘上,而卸载时又会删除该配置文件^②。

① 在 Unix 下, /etc 目录下面存放关于系统配置的文件。——译者注

② 这里作者所说的内容可能与相对路径有关系,即安装一个模块时,其配置文件的路径可能是绝对的,每次都不一样,而通过虚拟文件的方式,则可以转成相对路径,保持一致,下文也说到这一点。——译者注

这种“文件类型”的 API，其重要性不言而喻。如果 telnet 软件读取的 Kerberos 共享文件的位置发生了变化，比如把 /etc/telnet.rc 改成 /etc/telnet5.rc，则任何 Kerberos 组件对配置文件的修改都不会对相应的程序有所影响。因为 Kerberos 修改的是 /etc/telnet5.rc 文件，而原来软件读取的仍然是 /etc/telnet.rc 文件。同样的情况，如果文件的格式有所改变，那么 Kerberos 对文件的修改会破坏安装时的配置。其他软件再读取该文件时，因为不认识新的格式，就会失败，造成像 telnet 客户端程序无法正确运行的这种情况。

3.3 环境变量和命令行选项

环境变量和命令行界面作为上下文环境，对于功能库来说，往往不是很重要。但在 Unix 的使用手册中有一个章节却指出，这类接口在特定环境下也是非常重要的。在 Unix 世界里，通常会使用 shell^①脚本来编写应用程序，shell 有很多小工具来提供相应的功能。

在执行这些小工具的时候，需要从环境变量中取得相关的信息。例如，用来查找可执行程序的路径就是通过变量 PATH 来定义的。通过调整这个变量的值，就可以指定所执行的程序。有一种基本的入侵计算机的技术就是重新定义 su^②这个命令的功能。如果某个家伙的电脑没有加锁，这种入侵的方式就很简单，先是编写一段 shell 脚本，这个脚本的功能就是把用户输入的信息，如超级用户密码，发给自己，然后再把这个脚本名称改成 su，最后对 PATH 变量进行修改，保证这个新写的 su 脚本在系统命令 /bin/su 前面被调用。也就是说，当用户调用 su 命令的时候，其实调用的是这个 su 脚本，而非系统的 /bin/su 命令，这样就可以获取当前主机的超级用户密码了。为环境变量提供不同的参数，可以让程序执行操作时其行为完全不同。这里说到配置文件的问题，是要强调配置文件也是一类 API，和类名、方法签名有一样的重要性。对于一个基于模块化架构的程序来说，也存在与 Unix 命令行功能相似的情况。如果从外部可以通过代码来设置输入和输出的字符串，那么这些内容对于一个组件来说，也是其 API。

Java 应用程序中的环境变量

一直以来（从 Java 1.1 到 Java 5 版本），Java API 都缺少一种读取环境变量的方法。这可能是考虑到 Java 是跨平台的开发语言，但并不是所有的操作系统都提供了环境变量方面的功能，而且一旦某个程序需要依赖环境变量，就表示它无法方便地换到任意一台机器上。但有时候，不使用环境变量，开发程序时会存在一定的麻烦。比如说在 NetBeans IDE 中，CVS 和 Subversion 插件都是调用命令行来完成各种版本控制操作。这两个版本控制软件在 Windows 和 Unix 平台上都支持通过命令行来调用相应的功能。但如果想在 NetBeans IDE 中调用它们，就必须对环境变量进行一些正确的配置。在配置环境变量时，通常都会定义一个全局变量。所以在 Java 程序中，就需要读取当前的环境变量值，然后编写脚本调用前面所说的那些命令。还好现在不

① Unix shell 也叫做命令行界面，它是 Unix 操作系统下传统的用户和计算机的交互界面。用户直接输入命令来执行各种各样的任务。——译者注

② 该命令用来将当前账号切换为超级用户。——译者注

用使用那些老套的方法了，因为 Java 5 提供了一个标准的 API 来读取环境变量^①。

对于环境变量，其实 Java 还有一个取代方案，那就是系统属性。Java 中的任何方法都可以调用 `System.getProperty` 方法，通过传入一个字符串名称来读取与属性相关的值。这和 Unix 的环境变量差不多：可以在启动 Java 虚拟机的时候，将 Unix 的系统属性传递给 Java 应用程序。即使在 Java 虚拟机运行的时候，也可以改变属性值。

在 NetBeans 平台中，这种属性配置也被称为 API，通常用来改变某些系统模块的功能行为，这些行为是我们想实现的，但又不想令其变得过于显眼。这样就比较方便对某些操作进行微调，比如说处理拖放操作，限制对粘贴板上内容的访问，以及调整代码以更方便进行测试。在使用 NetBeans IDE 或者为 NetBeans 平台开发功能模块时，都不会用到这些 API，但是对于那些需要在 NetBeans 平台基础上进行产品定制的公司或者个人来说，就非常有用。Java 语言提供了 Javadoc 功能，从而很方便地对类和其方法、字段进行详细地说明，这是最重要的一类文档，虽然这种属性配置的 API 不可能像 Javadoc 一样重要，但我们也尽量保证这类 API 在后续的版本中能够保持一致。对于我们来说，这也是真正意义上的 API，每个人都可以使用这些 API 来达到自己的目标。

当然，Unix 的功能不仅仅会受到环境变量的影响，还与被调用时所传递的参数有关。比如前面提到的 `su` 命令，就能够知道当前是谁登录了，因为它被调用的时候传入了相应的参数。如果改变了参数的意义、顺序，或者原来版本支持的参数在新版本中不再被支持，这样类似的情况也同样会影响程序的正确执行。因为，输入参数以及在文档中提及的能够影响系统功能的变量，也都是 API。

3.4 文本信息也是 API

与它的前辈以及后续的很多操作系统相比，Unix 有一个很大的特点，就是它使用了管道^②技术。很多程序不仅仅可以通过环境变量和命令行参数进行控制，还可以向一个程序输入一段文本作为参数并取得一个返回值。在任何 Unix 系统中，可以用管道技术把很多程序串在一起变成一个程序，这种功能非常强大，而且易于理解。对于很多 Unix 系统用户来说，这已经是 Unix 系统上最常用的一个功能了。同样，如果一个程序有输入值和输出值，那么也应该被看作一个 API。所以必须小心审视，下文将对此加以详细说明。

由他人输入的文本以及由程序输出的文本都可以看作是一类非常重要的 API。在前面已经多次说过一个程序可以接受和输出配置文件、流等内容，事实上还远不止这些。在 Java 中，允许通过调用 `toString` 方法得到每一个对象的字符串表达方式。`java.lang.Object` 提供的默认实现只返回一个类名和一个 16 进制的散列编码，基本上没什么用。为了方便调试，很多开发人员都会

① 应该指的是 `java.lang.System` 的 `getEnv` 方法，这个方法是在 JDK5 才加入的，可以参考它的 Javadoc。——译者注

② 管道是从一个程序进程向另一个程序进程单向传送信息的技术。与其他形式的进程间通信，如 IPC、MESSAGE PASSING、SOCKET 等不同，管道的特点是单向的。通常，管道把一个进程的输出传给另一进程作为输入。在接受进程接收信息前，系统临时保留管道信息。——译者注

覆盖^①toString 这个方法以提供更有逻辑意义的字符串信息。这样做其实是有问题的，因为有时这个输出的字符串要依赖于对象的很多信息，在输出时就会公开内部的很多信息，会使得外部对象依赖于这个 toString 的返回值。像这种无意中形成的 API 可能不利于分布式开发的发展前途，而首先有个 API 却恰恰是基于分布式开发这个主要动机。

toString 方法的误用

在 NetBeans 开发中，也有一次对 toString 方法的误用，当时是 FileObject 这个类覆盖了 toString 方法。NetBeans 中的 FileObject 类其实算是一个对 java.io.File 功能的封装，这样做的目的是统一虚拟文件系统的接口，不管虚拟文件系统背后的实现是 FTP、HTTP 还是 XML^②，都可以使用同样的代码。开始的时候，这个方法返回的是本地文件路径。这样做的后果，就是其他开发人员可以用 `new File(fileObject.toString())` 这样的代码来创建一个 java.io.File 对象，然后直接访问本地文件。但这只是一个特例，如果 FileObject 中具体内容包含的是 CVS 或者其他版本控制的远程文件，或者是一个 JAR 包中的某个压缩文件，这种操作就会出现错误。但开发人员通常是在自己本地磁盘上测试这些代码，所以运行时一切正常，直到产品正式上线的时候才发现问题。当然，只要用到任何其他的虚拟文件系统，都会立即出错。

我们决定要解决这个问题。我们把 toString 返回的字符串变成 URL 格式，试图让用户知道，如果要取得路径应该调用 getPath 方法。但过了几个星期以后，我们发现有些同事又开始使用 toString 返回的那个 URL 来完成某些功能了！幸好，我们在产品发布前发现了这个问题，于是我们决定让 toString 方法返回一个没有意义的字符串。现在 toString 返回的值先是一个前缀，然后加上一个 URL，再接下来就是一个通过调用 `System.identityHashCode()` 方法得到的 16 进制数字。这样终于达到了我们预期的目标，它清楚地告诉开发人员，这个 toString 方法没有用。于是，我们的同事开始使用 getPath 这个方法了，这个方法会返回一个当前虚拟文件的相对路径，这才是开发人员的首选方法。但这个并不是一个皆大欢喜的结局。尽管这个方法在 Javadoc 中清楚地标示着其返回值只是一个相对路径，但是对于 FileObject，这个值仍然是绝对路径。所以还是有很多开发人员会用 `new File(to.getPath())` 来访问文件，如果说我们返回相对路径的规则有所变化，那么这些代码就会出现问题。这种问题仍然和前面差不多，有似曾相识的感觉。

还有一个相似的案例，就是直接把 toString 返回的字符串显示到用户图形界面上，这种对 toString 的使用方式也是一种滥用，会直接影响用户界面上的内容^③。显然，在一个模块化

① 英文中的 override 是指对父类方法的覆盖，本书翻译为覆盖，而 overload 则是指同名方法，但有不同参数，本书翻译为重载。——译者注

② 这里所谓的 XML，其实是指“三明治式的文件配置”中提出的那种虚拟 XML 文件，也是一个虚拟的文件系统，所以与 FTP、HTTP 并列。——译者注

③ 比如说调用 toString 把返回的字符串显示在标签里，那么当 toString 返回的数据不是名称，而是像 FileObject 一样返回无意义的字符串时，对用户来讲，UI 界面上的内容也就毫无意义了。——译者注

的系统中，组件对外提供的所有信息都可能会被滥用。所以一定要明确地标识出哪些内容是 API，哪些内容不是 API。

如果出现下述情况，就一定要非常小心：有时候，用户想得到某些信息，却又没有办法通过公开的 API 来取得，就可能去分析程序执行时输出的那些文字信息，从中得到自己想要的相关信息。比如说 JDK 在 1.4 版本之前，Exception 这个类中只提供了一个 `printStackTrace(OutputStream)`^① 方法，可以将错误文本信息输出到指定的输出流中。如果程序员想知道错误发生的地方，只能得到输出的错误信息，然后再去解析输出的错误文本信息。我们当时在 NetBeans 中就是这样做的，相信还有很多人也和我们一样。结果，这个错误信息的输出格式也变成了一个 API，而且也不能再去修改输出格式了，任何修改都可以导致代码出现问题。如果开发人员很看重自己对外提供的 API，那么就一定要避免这种情况。

在 Java 1.4 中，这种 API 的误用已经得到了修正。Exception 类中添加了一个 `getStackTrace()` 方法^②，用来返回结构化的错误信息。这样在新开发的 Java 程序中，就可以避免开发人员自行通过解析文本信息来分析错误堆栈。但对于 Java 1.4 以前版本所编写的程序，还是得沿用老的方式，通过解析记录错误的文本信息来分析错误堆栈，因为老版本的 Java 没有这个方法^③。所以还是需要保证原有的文本输出格式的兼容性。这又再次验证了前面的说法：API 就像恒星一样，一旦被发现了，就得永远呆在那里。

3.5 协议

协议是针对文本内容的 API。它们用来定义网络传送中的信息格式，所以非常重要。不管开发人员是想读取还是输出网络数据，只要他打开了一个套接字端口，那么就等于是开始了一个 API 的事务。这个事务不是那种类似于数据库的事务，这类事务 API 极具危险性，比那种基于本地 API 调用方式还要危险得多。

首当其冲的问题就是没有访问控制，当然如果从外部添加控制^④则是另外一回事。在 Java 中，可以通过 `public`、`private`、`protected` 和 `package` 等访问声明来进行控制，但对于一个对外开放的套接字来说，其实是没有办法进行控制的。每一台计算机都可以访问你的计算机，通过一个端口进行连接；然后开始交互信息，并变得依赖于你的计算机。这种 API 有一个好处，那就是，如果你不用将你的应用程序分发给其他独立运行服务器的用户，那么你就可以对请求频率和交互的内容进行分析和计数，因为这都发生在“你的”服务器上。另外这样做也有助于分析套接字的访问情况，可以知道都是谁访问了这些功能，可以分析他们的请求，并确保做出明智的反应。随

① `printStackTrace(OutputStream)` 应该是 `printStackTrace(PrintStream)`，而且该方法其实是加在 `java.lang.Throwable` 类上的，作者此处可能有误。——译者注

② 这个 `getStackTrace` 方法其实是加在 `java.lang.Throwable` 类上的，作者此处可能有误。——译者注

③ 这里的意思是指使用 Java 1.3 以前版本开发的程序，即使现在用 Java 1.4 版本运行，但原来代码基于 Java 1.3 开发，所以还是使用老的方式来获取错误堆栈。——译者注

④ 所谓的外部限制，则是指路由器之类的外层系统。通常情况下，网络套接字是对外全部公开的。——译者注

着时间的推移,可以调整回复数据的内容,但是必须保证调整后仍然可以处理任何连接,因为没办法对连接加以限制。计算机的端口,现在变成了你的 API,可以开放给所有人使用。

另一个问题则因使用网络协议而被放大,即每个网络协议会有多种客户端,协议还会有多个版本与你的程序进行交互,且这种多样性还会不断扩散。我曾经读过一篇与 Subversion 开发有关的文章^①,非常有意思。Subversion 是一套版本控制软件,这篇文章讲的是开发人员在程序演变时所面临的挑战,以及所使用的协议。Subversion 的开发人员的确需要处理多方面的内容,因为它所提供的 API 覆盖的范围很广。可以通过命令行来使用 Subversion,所以要能支持参数和环境变量。作为一个版本控制软件,它必须定义自己的元数据文件,并能存取和分析其格式以便管理被检出的文件。最后要说的是一件非常重要的事情,就是用户还可能使用各种第三方工具来访问 Subversion 服务器,以便更新和检出文件。要是只有一种版本的 Subversion 系统,那可就简单了。只不过 Subversion 和其他软件一样,都会有所演进。需要去修复 bug,同时还会有人不断地提出各种新功能需求。于是可能会出现这种情况:几个星期以前,用户使用了某个版本的工具来检出文件到本地,过段时间可能又换了一个该工具的新版本操作原先检出的文件。这些不同的版本必须能够分析相应的数据,至少也要能知道它是否可以对现有数据进行操作。唯一性匹配^②是最重要的一步:一旦 Subversion 确认是另外一个版本所产生的数据,那么可以拒绝对其进行操作,然后要求用户升级到新版本上。对于版本冲突的问题,这其实是一种常用的解决方案。但这种方案,只能用在本地文件操作上。

这种数据不匹配的情况,对于本地文件操作,也许只是可能会出现,但在与远程服务器进行通信时,则是必然会出现。几乎可以肯定,服务器上的 Subversion 软件版本与客户端计算机上所使用的版本不一致。为什么呢?因为客户端计算机非常多,而且每台计算机上都可能运行不同的客户端工具。即使服务器端建议客户端升级到某个特定的版本,也没有办法保证客户端一定会这样做。对于那些 Subversion 的客户端来说,想获得一个特定版本,未必是一件容易的事情,会受到网络、操作系统等一系列的限制。即使做到前面所说的,能够获取特定版本,然后再访问服务器,也并没有解决这个问题。很多项目都会在自己的服务器上安装 Subversion 作为版本控制软件,每一个项目使用的 Subversion 肯定都有所不同。如果每一个服务器都强制要求客户端使用某个特定版本,那么也许需要在本地系统上安装 svn1.0.1、svn1.1 和 svn1.2.3,还得记住到底哪个项目用哪个版本,以保证在文件更新和检出时能够使用正确的客户端来连接服务器。因此,如果 Subversion 的版本有不兼容问题,就会给用户带来很多麻烦,所以最好还是能保证用户在切换版本控制软件时,能够保持兼容。

为什么这种协议类的 API 也要做到向后兼容?这是因为不同版本本来就应其各自独立的生命周期产生兼容性问题,而这类 API 会使这个问题加剧。一旦某个协议被定义出来以后,就开始了其生命周期。会有很多不同的客户端程序使用这个协议。那些客户端程序也会有多个版本,分布在世界各地的不同计算机完全根据各自的喜好,任意选择不升级或者升级到某个版本。所以总

① Garrett Rooney, "Preserving Backward Compatibility" (2005), <http://www.onlamp.com/pub/a/onlamp/2005/02/17/backwardscompatibility.html>。——译者注

② 这种唯一性匹配是指每一个版本都只处理与自己版本所匹配的数据。——译者注

会出现两个不同版本的程序进行交互，而且交互时也可能使用两个略有不同的协议^①。尽管协议有所不同，但仍然要能支持它们之间的交互。

限制本地服务器

尽管大部分基于 NetBeans 平台开发的应用程序都是桌面程序，比如说 NetBeans IDE，但我还是有机会尝试了一把基于协议的 API 设计，而且出现了对 API 加以演进的问题。首先 NetBeans 应用程序都是单例的，也就是说，通常情况下每个用户只能启动一个 NetBeans 的实例。如果这个用户尝试开启 NetBeans 的第二个实例，NetBeans 并不会真正启动，而是把先前的那个 NetBeans 实例显示给用户，并提示用户已经有一个实例在运行了。NetBeans 平台是通过一个文件锁的机制结合套接字来实现这一点的。对于一个基于 NetBeans 平台开发的应用程序，在启动时会生成一个随机端口号，然后将这个端口号写到一个锁文件中。如果这个锁文件已经存在，程序就会读取该文件，然后从文件中取得一个端口号，再根据这个端口号通过套接字的方式进行连接，看是否已经有一个 NetBeans 程序的实例在运行了。如果发现确实有这样一个程序实例，那么这两个进程就会通过命令行参数的方式进行交互，执行某些操作，并进行数据交换。最后，第二个 NetBeans 程序的进程就会自动退出，并将第一个 NetBeans 程序显示给用户。这种情况其实很多，比如说打开文件时，如果程序已经启动，就将要打开的文件信息以这种方式传递给已经启动的程序。要处理这种情况，这算是一个不错的方案。

因为交互的协议是在同一个程序之间，所以非常简单而且实用。当编写实现这些功能的代码时，我觉得不需要去考虑它的扩展性和通用性。我只需要在同一个程序的两个实例中进行交互即可。我觉得可以完全控制这些内容，也可以很放心地为其添加功能，只要我能同时正确地修正发送和接受这两方即可，这根本不是一个对外公开的 API。

但在 NetBeans 发布多年以后，我才发现我当时的想法真是太天真了。我收到了一些 bug 报告，经过调查后，才得知有些人尝试用 NetBeans IDE 6.0 去连接 NetBeans IDE 5.5，从而引发问题。因为这两个版本所使用的协议略有不同，所以无法正常运行。

从那以后，我学会了一件事：每个协议建立连接时，即使这个协议只是在本地系统上用于内部交互，第一件要做的事情就是一个握手协议，明确一些关键信息。

破坏协议这个例子和前面说的把 Subversion 从一个版本换到另外一个版本差不多，都是一个非常痛苦的过程。所以对于协议来说，先做一个握手交互是非常重要的。而且这个握手交互应该是在连接建立时就要做。在握手交互中要先声明连接双方的版本，以便它们选择通用的语言进行交流。另外这也要求对于多个版本的客户端来说，要能够有一个通用的语言进行交流。并不要求这个语言去处理复杂的任务，就像“基础英语”足以支持常见的情况就行了，如 Subversion 的检出操作、当前状态以及文件的差异。

如果要向协议中添加新功能，就需要为协议语言添加新的表达方式，此时协议可以说是有所

^① 这里作者使用的是 slightly，表示略有不同，这是因为有很多协议都是主版本不兼容，而小版本不兼容，如 1.0 和 2.0 不兼容，而 1.0 和 1.1 兼容。——译者注

演进了。但对于 Subversion 客户端工具的最终用户以及 Subversion 库的用户来说,这种改变并不能强求他们去进行调整。不管使用的是 svn1、svn1.01、svn1.1、svn1.2.3 或者其他版本,都是如此。具体的处理细节应该由一个 Subversion 的功能类库来完成,并对使用者隐藏相关操作。比如说,可以在一个功能类库内部放置多个独立版本的协议代码,在对应不同协议的时候,调用相应的代码,但对外公布常用功能 API 的时候,却不要公开这些内容,以简化用户操作。

3.6 行为

提供 API 可避免我们去了解一个组件内部的具体实现,可以让我们在“无绪”的情况下使用其他的黑盒程序模块来组装我们自己的程序。这样说,是不是就表示 API 所定义的内容应该只停留在表面呢?如果真能做到这一点,那就太好了。但通常不是这样的:不论 API 的抽象度有多好,API 对应的内部实现经常会泄露具体的内容,从而使得这些内部实现也变成了 API 的一部分。

我碰到的很多人都认为所谓的 API 不过就是类或者方法。不管这个 API 背后的具体实现内容是什么,这些细节都不属于 API。我为自己曾经也这样想感到非常惭愧。当 NetBeans 的质量管理部门问我软件是否做到了向后兼容,我坚称做到了,因为我们没有修改任何公开 API 中的内容。其实我错了。之所以要开发组件,是因为这样可以方便模块化系统的开发。而兼容性则意味着随时可以用一个模块的新版本来替换系统中的这个模块,而且在替换以后,整个系统可以同以往一样正常运作。判断兼容性时最重要的方式就是如何看所谓的“正常运作”。而这一点却是由内部具体实现来决定的。

如果一个方法在上一个版本中声明该方法的返回值不会为 null,但在下一个版本中却返回了一个 null 值,这种改变是不兼容的,因为这种变化能从外部观察出来,并为组件的用户带来负面影响。如果一个接口的方法开始时可以在 Java 的 AWT 事件调度线程中正确运行,随后修改为组件可以从任何线程调用这个方法,就会破坏这个接口的现有实现。这样就不能正确地创建一个可视控件,如对话框之类的内容,导致最终用户无法对程序进行操作。API 的很多行为操作都是有关联的,如程序流程、调用顺序和锁,这些内容往往都是相互依赖的。

这些例子都表明尽管 API 的具体行为很难控制,但它仍然是 API 契约中非常重要、甚至最重要的一部分。只有组件的行为能够保证稳定,其用户在组装程序时,才能做到用“无绪”的方式来替换一个模块。只有确认模块的版本升级不会影响其功能,开发人员才会相信并使用这样一个模块。这正是本书将组件的对外行为也看作是一类 API 的根本原因所在。

3.7 国际化支持和信息国际化

很多 API 的设计,并不会考虑到所有人的所有需求。如果我是一个 Linux 下的 C 程序员,那么我并不需要去关心 Linux 内核中的那些 API。这些内容太过于底层了,超出了我需要了解的范畴。对于一个 C 程序员来说,这些细节性内容太多了,完全让我无法负担。我希望这些内容能够远离我的开发工作。另一方面,的确有人非常关注这些 API,对于他们而言,这些就是合适的、感兴趣的 API。

通常,不同的 API 会有不同的目标受众。有一个比较极端的例子就是国际化支持,会有不同的团队将独立的模块翻译成不同的语言。在 Java 中,处理国际化的方式通常是使用 `ResourceBundle`^①。为了避免以硬编码的方式将文字信息放在代码中,可以定义一个键值,然后通过 `ResourceBundle` 结合该键值来取得国际化后的文本信息。这个键值和国际化信息放置在 `Bundle.properties`, `Bundle_en.properties` 和 `Bundle_ja.properties` 等文件中,然后在程序执行时根据当前用户的语言来选择合适的国际化文件。

开发人员通常只提供 `Bundle.properties` 作为默认的国际化文件,一般来说,其中存放的文本都是英语。然后由其他专业人士提供其他语言的国际化文件。这些翻译工作都是独立完成的,有自己的进度安排,由分布在世界各地的不同团队来完成,很可能与代码开发人员相距千里。

很多开发人员都是针对一个功能类库进行编码的,所以从这样一个 API 普通用户的角度来看,这类国际化信息中的键值是一个底层的实现。要使用这些信息就像利用反射技术来访问 `private` 类、方法和字段这样非公开的内容一样。依赖于这类内容可谓是违反了开发惯例和通常的看法。把它们记录到 Javadoc 中或者使用它们烦扰普通的 API 用户是没有意义的。所以这类 API 通常不需要其用户去了解其内容。

在日本地区使用 Solaris^②

NetBeans 在 `ResourceBundle` 的基础上扩展了一个类,然后使用这个类进行国际化支持,这个类一直运行得很正常。但有时候我们需要超出这个契约,并定义一些扩展。但最终证明这样做简直是自找麻烦。

最近我定义了一个半在线契约,就是 NetBeans 的用户界面中有一部分内容是通过读取网站上的信息而得到的。比如从一个特定的地址读取一个 HTML 页面,然后解析,将其中的部分内容显示出来,将另一部分用来创建按钮、标签等控件。我在自己的 Linux 系统上很容易就实现了这项功能。但当翻译团队开始工作时,我马上就收到了一些错误报告。

因为当地人员通常都使用 Solaris,而且大多都翻译成日语,所以我们收到的 bug 报告大部分都与 Solaris 和日语有关。重现这些 bug 的过程实在是件痛苦的事情:先启动 Solaris;然后切换到日语环境;再去研究如何调用其终端,启动 NetBeans,还有浏览器等其他的事情。在这种“Solaris+日语”的环境下之所以出现这么多的问题,就是因为常用的 Linux 系统都是使用 UTF-8 编码,而日语系统默认使用的则是 `euc_ja` 编码。在不同编码间进行转换通常会影响程序的一致性。

最好在高压力的环境下来测试应用程序。但对于 NetBeans 和其他与国际化有关的 API,我们知道在“Solaris+日语”这种最复杂的情况下,出现的问题最多。

对于负责国际化翻译的团队来说,国际化文件中定义的键值就是一类重要的 API,而且对他们来说,这是最重要的 API。为了简化程序人员和翻译人员的分布式协作,最好也遵守前面所说

① 即 `java.util.ResourceBundle` 类。——译者注

② Solaris 是 Sun 公司研发的计算机操作系统。它被认为是 UNIX 操作系统的衍生版本之一。——译者注

的那些 API 演进时对应的规则，比如说不允许移除键值或者重命名键值，因为这些调整都不是向后兼容的变化。

3.8 API 的广泛定义

关于“API 定义”这个问题，现在看起来可能已经很清楚了，所谓的 API 远远不止是类、方法、函数和签名这些东西。为了有利于在“无绪”的状态下把一个大的系统以组件集成的方式拼装出来，从这个角度来看，API 的定义就非常广泛，从简单的文本信息到那些复杂或者很难控制的组件行为，都可以算是 API。现在大家可以想像，如果随意地改变数据库、XML 结构，或者调整 WSDL、REST 或者 IDL 服务的定义，这些改变的后果可能会很严重。因为这些内容都是 API，所以对这些内容进行处理时要始终注意演进的问题。

3.9 如何检查 API 的质量

如前所述，人们往往把是否正确这样一个问题与优雅挂上钩。所以讨论优秀 API 的特性时，许多人也会说，优秀的 API 应该是非常漂亮的。我们承认，漂亮是一种优势。在第一次遇到某样东西时，发现这样东西很漂亮，很让人喜欢，就会留下一个良好的印象，它被人接受的可能性就更大。正因如此，大家都愿意努力设计出漂亮的 API，但漂亮并不是衡量优秀 API 的唯一标准。

漂亮的定义以及它的衡量标准都是很主观的。没有哪两个人对漂亮的认同能达到完全一致，每个人都有各自的喜好。如果把这一点套用到 API 设计上来，就意味着每个 API 都有其特点，且对用户的技术要求也各不相同。这样会为 API 的广泛使用造成壁垒。漂亮属于艺术的世界，而软件工程就是工程。工程的首要目标是制作可正常运作的系统。我们在内心深处深受古希腊人的影响，所以总认为正确的事物也应该是漂亮的，但这种思想不应该让我们偏离主要目标：我们应该设计易于使用、广为接受且富有成效的 API。一套工程方法需要一种客观的方式来衡量其产品的质量。我们需要对每个 API 定义详细规则，以保证能满足给定的客观标准。

在下面几节中，我们将探讨 API 用户比较关注的几个方面，并分析其重要性以及遵循这些原则的方式。考虑到本书的主要目标读者是工程师，而工程师一直被教导要测量一切事物，因此，我们将要实施的衡量标准可以看成是对“拒绝漂亮”的一种补偿，因为，虽然很遗憾，但我们也不得不承认，漂亮是无法量化的，因而也不适合作为质量的一个评价标准。

3.9.1 可理解性

那些使用 API 的用户必须能够理解 API。这条标准听起来有点含糊，但也是最重要的。我已经说过多次：API 其实就是设计者与开发人员之间交流的途径，后者要根据相应的 API 提供具体实现。就像人们使用语言进行交流，如果两个人无法沟通，那么肯定是我们的沟通方式出现了问题，而程序员的沟通方式就是 API。

如果要找一种和 API 设计相似的工作，那大概就是写书了：一个作者会对应很多读者。读者

对作者总有一定程度的了解，但作者却对读者知之甚少，甚至一无所知。所以要想设计一个易于理解的 API，对用户的技巧和知识猜测准确也属于一种颇具技术含量的方法。

每个人的世界观都会受限于自己的视野。离我们近的事物，我们可以看得比较清楚，但当相应的事物位于视野边界处时，就看得有些模糊了。如果这个事物超出了我们视野，就会消失，完全不在我们的思维中。对于一个优秀的 API 来说，它所涉及的概念必须处于用户的视野范围内，否则他们就无法理解这些概念。API 设计者需要理解其目标用户普遍具有的知识，并在此类知识的基础上设计 API。当然，有时引入一些新的概念，超出用户的视野范围，这也是可以接受的。但这种新内容的引入应该是渐进式的，因为忽然引入大量超出视野的内容，经常可能会把用户引到未知的领域中，有迷失的危险。

如果使用 Java 编码，那么很自然地就可以给出一个定论：开发人员都使用过大部分 `java.*` 的类库，因此也了解其涉及的概念。如迭代器、枚举、I/O 流、JavaBean 监听器以及可视化控件是几乎所有程序员都会遇到的内容。使用大家都熟悉的技术、类和术语能够有效降低开发人员的学习难度，这同时也降低了对 API 用户的能力要求。

在不是很熟悉要开发的功能时，开发人员通常会有一种常见的做法：很多 API 用户都会去找一个现有的程序，这个程序的功能要与现在要开发的功能比较相似，然后把这个程序的代码复制过来，再加以调整来完成自己现在开发的功能。所以即使某个 API 非常特殊，其实对用户的影响也并不大。因为用户可以根据自己的需要来找到很多相似的例子，这样能够极大地帮助用户理解 API。如果某个 API 比较新，或者比较少见，那么提供大量的例子能够有效地帮助大家使用 API。

3.9.2 一致性

API 有一个很重要的评价标准，而且也很容易进行检查，那就是一致性。如果某个 API 的用户需要花费大量的时间来学习某一个概念，那么对于这个 API 来说，这个概念必须能够保持一致性^①。比如说，假设一个系列的 API 都使用工厂模式来创建对象，那么所有工厂的注册机制应该统一。这样一来，用户在使用 API 时就无需了解多种使用方式。

再比如说，假设有一些方法可以在线程中正常调用，但还有一些方法在线程中调用时有特定的要求，要求一定在某些特定的线程中来调用特定的方法^②，与其在文档中对这些情况加以详细说明，还不如在整个 API 中就贯彻执行一种线程模型。

从 API 生命周期的角度来看，在整个 API 的开发过程中最好保持一致的风格，至少对 API 进行演进时一定要保持一致。如果做不到这一点，用户就很难发现一个新版本的 API 在哪些地方进行了修改。

现在来看一下 `org.w3c.dom` 这个包中的一些接口。这些接口在 JDK 1.3 版本以后，就成为标准 JDK 的内容，尽管有警告说这部分内容会有具体的实现，但用户可以像实现 JDK 的普通接口

① 这里的一致性包括两方面，一是 API 可能有多个版本，版本间保持一致；二是一个 API 对外提供功能一致，比如说像下说的注册工厂都应该用 `registerFactory`，不能有的写 `registerFactory`，有的写 `addFactory`。——译者注

② 基本上所有与 UI 相关的操作，都有一定的线程要求，如 `Swf` 和 `Swing` 都有这方面的限制。——译者注

一样去实现这些接口。实现 `java.lang.Runnable` 是比较安全的，因为有很多人实现了这个接口，所以开发者就不会轻易向 `java.lang.Runnable` 类中添加一个新方法，否则会引发一系列的问题。但在新的 JDK 中，`org.w3c.dom` 这个包中的接口却添加了新方法。这样做引发了一系列问题，对于那些实现了这些接口的用户，发现他们先前编写的代码现在用新版本的 JDK 就无法进行编译了。尽管大部分程序仍然可以正常运行，但总有一定程度的兼容性被抹杀了，因为新方法的添加使得开发人员发现原来在老版本中编译正常的代码现在无法在新版本上进行编译了，是一种源代码级别的不兼容。所以除非相关多方都达成了一致，否则这种类型的演进，简直是一种对信任的赤裸裸背叛。

3.9.3 可见性

如果有人想使用某个 API，但根本找不到这个 API，或者很难清楚地了解这个 API 如何使用，那么这样一个 API 就算再漂亮，也没有有什么用。

假设有一个项目声称自己能够解决你的问题，同时提供了 API 供你使用。当你去看 Javadoc 的时候，发现这个 API 有 5 个包和 30 个类，但是却找不到一个入口或者向导来告诉你如何使用，还有什么情况能比这更让人生气呢？如，`java.awt.Image` 的 Javadoc 就是这样写的：`Image` 这个类用来表示一个图像，可以在显示设备上画图或者把图像以标准的格式保存到磁盘上。`java.awt.Image` 是一个抽象类，但大多数程序员只是想从磁盘上来加载一个图像。而在相应的 Javadoc 中却给大家出了一个猜谜游戏：“加载图像时必须按照各平台特定的方式”^①。如果这时程序员比较聪明，可能会注意到 `Image` 这个抽象类有个子类 `BufferedImage`，然后去看一下这个类如何操作。可以利用该类来创建一个空的图像，但从上读到下，也无法从 Javadoc 中找到如何从磁盘文件中加载一个图像，也不知道怎么在显示设备上画图，更不用说怎么把图像保存到文件中。说到参考文档，它的作用就是应该提供入口来帮助用户使用 API，但是这个例子却完全没有做到这一点，真是一个绝佳的反例。

在大部分情况下，这些类并不是 API 用户的目标所在。这些用户更关心的是如何完成他们的工作。考虑到其目的，用户更愿意去看一些例子，从例子中查找与自己需求关联最紧密的内容。这也从某个方面解释了为什么开源软件如此成功，因为在使用开源软件时，通常只需要把现有的一些源代码复制过来就可以开始工作了。事实上，这些源代码也能起到文档作用，为用户提供指导性的内容。可以看一下 HTML 的发展史：在 Web 发展的早期，那些初出茅庐的 Web 设计者只是把他们在浏览器中看到的示例代码复制过来就算完成工作了。也许互联网自身才是最大的开源项目^②。

不管你提供何种类型的 API，最好都是有一个入口来作为用户开始使用 API 的起点，可以帮助用户找到正确的方式来解决他们的问题。因为用户关心的可能不是具体的类，所以这个入口的组织方式最好能基于实际的或至少是预期的目标和任务，这种组织方式可能更为理想一些。

^① 原文为 *The image must be obtained in a platform-specific manner*。JDK 1.4 和 JDK 5 都是这句说明。——译者注

^② 这里作者指在浏览网上，HTML 就是以源代码的形式存在，相当于开源了。——译者注

3.9.4 简单的任务应该有简单的方案

有时候一个 API 要支持多种目标受众。例如，Java 的 JNDI 就允许开发人员通过名称来查找对象，同时也允许用户通过提供新的解析器来扩展新的名称。这两种行为完全是为两类 API 用户提供的。

JNDI 只是一个代表，实际上这种类似的 API 设计还有很多：不同的用户群共用一个 API。设计 API 时最容易也是最常犯的错误就是把针对不同受众的功能都放在一个 API 中。这样会降低程序的可见性，因为某些开发人员只关心 API 某一方面的功能，但 API 中为其他受众提供的功能可能也会影响这些开发人员学习和使用这些 API 功能。

常用的方案则是把一个 API 分解成两个或者多个组成部分：一部分是用来针对调用功能的开发人员，而另一部分则应该放在独立的包中，或者有其特定的命名空间，用来方便他人扩展当前模块的功能。JNDI 采用的方案非常好，它根据受众分类将功能放在不同的包中。API 的调用者只需要使用 `javax.naming` 和 `javax.naming.event` 这两个包，而扩展系统功能的开发人员则只需要关注 `javax.naming.spi` 这个包即可。

与通过文档来说明 API 能自动分类相比，这种功能上的分包处理甚至要更重要，因为 API 的不同受众很容易将精力集中到他们关注的领域中，在 Javadoc 中逐个查看类时，也不用问：“我需要关注这个功能吗？”

如果没有正确地分解 API，或者说如果出现更坏的情况，将功能分包时，对受众群的分析产生了偏差（这种情况出现的几率可能比较小），这样会严重地降低 API 的可用性。比如说 JavaMail 这个 API 包含了大量的概念以及具体的类。即使开发人员只需要发送邮件或者是从邮件服务器上得到邮件列表这种简单工作，他也必须要花费大量的时间来学习这些 API，但其实大部分的学习都是浪费了。从另外一个角度来说，整个 JavaMail API 在协议扩展方面的设计做得非常好。但问题在于大量的程序员只是想利用这个类库来收发邮件，相比之下，那些需要扩展这种邮件协议的需求却少得多。虽然说起来可能没有依据，但可以肯定，正是这种错误的优化方向造成基于 Java 的邮件应用程序相对极少的局面。

3.9.5 保护投资

任何开发人员在设计 API 时一定要牢记这一点——善待自己的同伴，也就是说要多为 API 的用户考虑一些。API 的用户越多，对这个 API 的使用就会越多，反之亦然。就像使用 Java 编写程序的人越多，那么未来 Java 程序的数量就越来越多。使用 JUnit 框架的人越多，那么 JUnit 框架和它的开发风格也就变得益加重要。如果要使用一个类库，那么就要去了解它，确信它能够减少自己的工作量，而不是带来更多的问題。但最重要的是，为了让 API 用户在使用类库时真正感觉非常舒服，还必须要保证类库的新版本不会破坏现有的代码，更不会让用户辛苦的工作化为乌有。编写一个类库需要投入时间、精力还有金钱。对于 API 设计者来说，首要的任务就是要保护其用户投入的资源。

API 的用户就是它需要珍惜的宝贵财富。这些用户的工作应该得到尊重。用户体验越好，他

们对这个 API 的赞誉就会越高，他们就越发愿意使用它，而这个 API 也会得到更好的提升和更广的传播。最终就会形成一个更愉快的 API 用户群。正是基于这个原因，我们一定要保护用户的投资，争取永远以一种兼容的（或者至少是可预测的）方式来演进 API 契约。类库的每个新版本都应该确保现有的客户模块能够继续以一种合理的方式来执行，就算不能做到这一点，也要确保用户能够轻松地升级现有的源代码以编译并使用新版本的契约。只要有可能出现不兼容的问题，一定要提前调整好对升级需求的期待。

使用一个类库通常有两种模式。如果运行环境对灵活性要求不高或者是存在大量分布式开发的情况，那么每一个最终用户都会有一个二进制的应用程序，该程序会包含类库的某个具体版本。如果该类库发布了新版本，为了让用户满意，在升级到新版本以后，整个程序还必须能做到正常运行。如果确实能够做到，我们就可以认为：对于那些使用该类库进行程序开发的人来说，其投资受到了保护。

还有一种不严格使用类库的模式，则会带来更多的灵活性，比如说获取源代码，然后修正问题，再重新编译。这样做在开源世界里很常见，对于 Linux 来说，其多种发布版本都采用这种方式。在这种情况下，并不见得对二进制兼容性有严格的要求、只要能够很容易地编译出一个新版本即可。如果能做到这一点，而且结果也正如用户所预期，那么也可以认为用户的投资基本受到了保护。

与上述两种保护方式相对的是，想尽力让 API 能够更漂亮一些：比如说把方法的名称改一下使之更清楚，调整一下 API 的结构使之更易理解，等等。在第一个版本发布之前，这样做非常合适。但一旦发布了第一个版本，更重要的问题则是保证所有更改不会破坏现有 API 客户端代码的运行。只有那些不看重用户的开发人员才会去做这种破坏用户投资的事情。



很多人都以为，设计和开发一个软件系统，包括 API 的设计开发，都是等到正式公布的那一天，所有的工作都算完成了。但这种情况实在是罕见，事实上，对于那些比较重要的部分而言，很少有真正“完成”的。通常来说，一开始发布的版本只是整个软件系统生命周期的起点。如果顺利的话，还会有多个版本陆续公布。对于 API 也是如此。判断一个 API 是否优秀，并不是简单地根据第一个版本给出判断的，而是要看多年后，该 API 是否还能存在，是否仍旧保持得不错。

4.1 第一个版本远非完美

如我在 2.5 节中提到的，第一个版本总是来得特别容易，不仅容易开发，而且容易发布。我同时也说了第一个版本不可能是完美的。这两点结合在一起就说明每一个 API 都会随着时间的推移被加以改进。API 的需求会随着时间的变，那些过去有效的 API 可能现在已经不再适用了。而且每个程序中都会存在 bug，需要不断地来修复，这样做带来的副作用人所共知：修复一个 bug 的同时会引入两个新 bug。这些观点普遍适用于所有软件系统，API 也不例外。

现在只能放弃原有的观点，承认“第一个版本远非完美”。无论为这个正式发布的版本付出多少努力，它仍然存在 bug，而用户也总是想用它来完成一些超出你设计范畴的正当任务。我们没有魔法水晶球^①，不能预测未来，只能面对现实。

但我们没必要为这个结论而感到悲观。API 因为需要不断改进的事实算不上什么坏事，只是对现实的一种坦诚。如果你所做的设计需要考虑未来对版本加以改进，那么最好在设计第一个版本时就思考再三，这样在后续版本中进行改进才有可能不会对用户造成问题。每一个 API 的作者都应该为未来的改进做出计划。这种计划是一种比较高层次的，要考虑未来版本会对 API 中哪些内容加以改进。这种计划可能会用到两种方式。一种极端的方式是放弃老的版本，重新开始做一套新系统。还有一种方式则是修正用户提出的问题，并强化现有的 API，保证兼容性，从而使得现有客户端的功能不会有所改变。

在理想情况下，增量改进可以做到修正 bug，提高程序性能，并提供更漂亮的界面，而这一

^① 在西方，水晶球在魔法传说中有着神奇的力量，可以预知未来，本文作者这里指没有办法预测到用户在需求和使用的两方面会如何改变。——译者注

切都不需要客户付出额外的工作。客户端只需凭借对老的 API 的信任, 就可以使用更新也更好的版本。这样做可能还不错, 但是考虑到每修正一个 bug 可能会引入两个新 bug, 每一个新版本的发布都可能为现有的 API 客户端带来新问题, 所以想做到前面所说的理想状态实在是非常困难。

放弃现有的 API, 并从头开始编写一个新的 API 来完成同样的任务, 可以避免不兼容问题。因为用户所沿用的那些旧 API 仍然可以保持原样不变, 这样就不会为客户端带来潜在的问题, 而新的 API 则可以引入很多创新和更好的功能。这样做唯一的问题就在于: 那些使用旧 API 的客户端只能继续沿用老的 API, 除非重新编写他们的代码, 以升级到 API 的新版本上。所以这样做的缺点也是不容忽视的。

完全重新编写 API 的优点在于避免了细微的不兼容问题, 但让客户端被锁定在一个特定的版本中, 即使新的版本提供了大量的改进, 这些客户端也无法从新版本中获益。虽然对 API 进行改进固然是一件重要的事情, 但相比之下, 兼容性更为重要。只有在这两者之间巧妙地取得平衡才能让一个 API 成为可用的 API。

4.2 向后兼容

使用旧版本 API 所开发或者编译的程序, 如果可以在新版本的 API 上正常运行, 那么这种能力被称为向后兼容。对于每一个 API 的设计者来说, 都渴望做到“向后兼容”, 因为不管是现在的 API 用户, 还是潜在的 API 用户, 都只信任那些可兼容的 API。但向后兼容有多个层次上的意义, 而且不同层次的向后兼容, 也意味着不同的重要性和复杂度。

4.2.1 源代码兼容

说到兼容性, 最先要面对的问题, 就是保证源代码编译时的兼容。如果基于 Java 1.3 版本开发程序, 那么可以用 Java 1.4 版本来编译这些程序的源代码吗? 如果能做到这一点, 那么可以说 Java 1.3 和 Java 1.4 这两个版本是源代码兼容的。但源代码兼容是非常难以达到的。之所以出现这种问题, 主要是因为每个新版本的 Java 语言都会添加一些语法上面的新功能, 这种改变往往都会体现在执行文件的格式上, 也就是 Class 文件的格式会有所调整。在 NetBeans IDE 中就出现过这种问题, 假设有一个基于 JDK 1.3 写了下述代码:

```
public class WrappingIOException extends IOException {
    private IOException cause;
    public WrappingIOException(IOException cause) {
        this.cause = cause;
    }
    public IOException getCause() {
        return cause;
    }
}
```

以上代码可以在 Java 1.3 上编译通过, 但在 Java 1.4 就无法编译通过, 因为编译器觉得以上代码试图去覆写一个在 Java 1.4 版本引入的办法^①。Java 1.4 语言规则禁止规定, 覆写拥有相同方法和参数

^① 在 JDK 1.4 版本中, java.lang.IOException 其超类 java.lang.Throwable 中引入了名为 getCause 的方法, 方法返回值类型为 java.lang.Throwable (即 java.lang.IOException 的超类型)。——译者注

但不同返回值类型的方法，否则将会导致编译失败。令人吃惊的是，以上代码在 Java 1.5 上却可以顺利编译通过，因为 Java 语言做了改进，允许在方法覆写时将返回值类型改为更特化的类型。

向类中添加了新的方法以后，如果这些新加的方法允许子类覆盖，那么这样做可能会破坏源代码兼容性，但这并不是我想说的唯一问题。向一个现有包中添加新的类也同样可能引发问题。现在关注下述基于 Java 1.1 开发的代码：

```
import java.awt.*;
import java.util.*;

/** 这段代码无法在 JDK 1.2 上编译通过，因为 JDK 1.2 新添加了 java.util.List 类 */
public class VList extends List {
    Vector v;
}
```

这段源代码使用 Java 1.1 是可以正常编译的，在 Java 1.2 发布以后，这段代码就无法编译通过了。因为 Java 1.2 引入了新的集合类，即 `java.util.List`。这样，前面代码中 `List` 这个类名就产生了歧义，因为它在 Java 1.1 中指向 `java.awt.List`，而在 Java 1.2 中，却可能指向 `java.awt.List`，也可能指向 `java.util.List`。简言之，向一个现有的包中添加新的类做不到源代码兼容。

上面的代码演示了向类中添加新方法会引发源代码不兼容问题，而添加新的类也同样可能引发不兼容问题。相应地，从类中移除方法或者从包中移除类也是不兼容的，那么如何进行修改呢？只能一动不动地大眼瞪小眼吗？还是说干脆放弃源代码兼容？进退两难中，也许放弃对源代码的兼容是个比较好的选择。

当然，如果说能够做到 API 的源代码兼容，那么努力去达到这个目标自然是首选任务，但是正如前面所演示的一样，达到这个目标是非常困难的，像编程语言上一些新加的功能就很容易破坏源代码兼容性，比如 `*` 号代表的包级导入。所以也不必花费过多的精力去做到绝对的源代码兼容。编程语言在设计时也没有在源代码兼容性上花费太多的精力。它的目标有所不同，如果开发人员需要重新编译的话，那么对于编程语言来说，只要修正相应的问题所费代价不大，那就 OK 了。像前面的 `*` 式导入，只要精确地声明要导入哪个类即可。

4.2.2 二进制兼容

如果一个基于老版本类库开发的程序，在不需要重新编译的前提下，可以和新版本类库进行正常连接并执行，那么这种情况可以称作二进制兼容。因为有两种场景需要这种兼容性方面的支持，所以要做到这一点也是非常重要的。首先，用户基于某个版本的类库编写了一个程序以后，原先开发的程序应该都可以一直正常运行，不管用户手中的类库是哪个版本，是否升级到了最新版本，程序的运行都应该是正常的。这样做可以极大地简化程序的维护、打包和发布工作。其次，如果用户只有一个老版本的二进制类库，也同样可以开发程序，随后再移植到新版本上，这样就无须用户来重新编译程序^①。这两种场景都有各自的用途，它们提升了配置方面的灵活性，并赋

^① 这种在模块化的分布式开发中非常普遍，即在开发时 A 团队依赖于 B 团队的 0.4 版本类库，但正式发布时，可以使用 B 团队的 0.5 版本类库。——译者注

予模块开发人员和用户更多的自由。为了达到这种相互调用的灵活性,开发人员至少需要了解一些源代码编译后生成的二进制格式。对于 Java 语言来说,就表示开发人员需要去了解 Class 文件的格式,以及 Java 虚拟机如何加载 Class 文件。下面介绍一下源代码兼容和二进制兼容之间的主要区别。

虽然并不完全相同,但 Class 文件的格式与 Java 源代码的格式非常相似。第一个不同之处在于源代码中的*式导入是不存在的,因为在 Class 文件中,所有的名称必须是全名^①。同样,一个字段或者方法的名称所包含的不仅仅是源代码中的名称,而是使用了真实的类名和方法名称。对于一个方法,就表示所有的参数和返回值都使用全名。因此,在一个类文件中,其实可以有两个同名方法,即使它们的参数名称完全相同,只要返回值不同,那么就是合法的,虽然 Java 源代码中不允许这样写。看下面的代码。

```
/** 这段 Java 代码不能编译通过,但如果用二进制的方式,这种结构却是存在的。*/
public abstract class DoubleReturnType {
    public abstract String getName(int x);
    public abstract StringBuffer getName(int x);
}
```

这段代码在 Java 中是无法编译通过的,据我所知,还没有哪一个 Java 编译器能够正常编译这段代码。另一方面,尽管这段代码无法被编译成 Class 文件,但仍然可以在一个 Class 文件中写入类似的内容。在 Java 语言中,这是源代码与二进制文件一个最大的区别之处。

破坏 Class 文件

如前面所说,尽管可以创建一个 Class 文件,而且在文件中包含两个同名方法,即使参数相同,只要返回值不同,那么这个 Class 也是合法的。但问题在于,怎样编写 Java 源代码才能将它编译成这种字节码呢?

答案是要对 Java 二进制代码做些手脚。比如说,可以手工修改类文件,直接使用 Java 虚拟机指令来调用相应的方法。相比之下,这样做要比编写 Java 源代码要复杂得多,因为要修改的是类文件这种二进制格式。但 Java 中的二进制格式内容在相关规范已经写得很清楚,也很容易理解。而且还有多种工具来帮助开发人员分析和修改二进制内容,如 jasm、BCEL、classfile 等很多工具都有此功能。

有段时间,NetBeans 项目也碰到了一个向后兼容的问题。我们需要两个 getIcon 方法:一个 getIcon 方法用来返回 java.swing.Icon,另一个则要返回 javax.swing.ImageIcon。想完成这个任务最简单的办法就是为这两个方法起两个不同的名称,然后在生成的 Class 文件中进行查找和替换。下面是我们所编写的 Java 代码,可以正常编译。

```
public static ImageIcon getIcon() { return null; }
public static Icon g3tIcon() { return null; }
```

然后我们在字节码中使用 getIcon 名称替换了 g3tIcon。做法简单而有效。这样一个类中就

① 所谓的全名就是带有完整命名空间,如 java.util.List。——译者注

有了两个同名方法，其参数也相同，只有返回值不同。

```
<!-- 在使用了一些后门技术以后，得到期望的类文件。比如说，先编译 Java 代码，然
--> 后再修改生成的 class 文件，替换一些特定的内容。
<target name="-build-and-rename">
  <mkdir dir="build/apimerge/classes"/>
  <javac
    srcdir="src-apimerge" destdir="build/apimerge/classes"
    source="1.4" target="1.4" classpath="{cp}"
  />

  <!-- 这里放置实现替换的代码，因为替换是可以通过文本的方式来完成。
      我们需要使用一些合理的编码方式，能将所有的 byte 值都看作字符。而 UTF-8
      并不支持 Java 文件头 0xCAFEBAE，所以我们选择了西欧编码方式，如下所示。
  -->
  <replace
    dir="build/apimerge/classes" casesensitive="true"
    encoding="iso-8859-1" summary="true"
  >
    <include name="**/*.class"/>
    <replacetoken>g3tIcon</replacetoken>
    <replacevalue>getIcon</replacevalue>
  </replace>
</target>
```

同时程序员还应该知道，在访问一个字段的时候，其实也会使用类的全名^①来对字段进行准确定位。虚拟机会在指定的类中查找该字段，而如果是查找一个方法，那么不仅在指定类中，还会在其所有的父类中查找该方法。所以相比使用一个字段，最好还是使用一个方法，这样方便在未来加以改进。

在使用面向对象语言时，必须要明白地知道声明一个方法到底意味着什么，而覆盖一个方法又意味着什么。尽管现代的虚拟机，如 HotSpot 虚拟机，虽然其具体实现已经有所不同，但传统的“虚方法表”仍然足以说明一些基本的内容。当定义了一个包含有非 final 方法的类时^②，就会自动创建一张“虚方法表”。运行时，方法的名称、参数描述以及返回值，还有代码运行时方法被调用的真实入口地址，这些内容都会通过这张表进行映射。如果有人为某个类创建了一个子类，也就相当于又创建了一张相同的表，只是填入的内容不同，它会用一些新的指针来指向某些被覆盖方法在调用时的地址^③。当方法代码被调用时，会去检查这张虚方法表，以便从中查找到正确的方法入口，然后再跳转到相应的地址，执行相应的代码。这种“虚方法表”的机制简单而强大。基于这种机制，代码执行时就可以做到动态跳转了，真正被执行的代码取决于执行时的对象所声明类中的“虚方法表”。所以开发人员牢记这一点还是非常有用的，因为它可以解释在面向对象语言中，继承这种行为是如何做到的。

① 在标准的 Class 文件中，为了节省空间，所有类的全名都有一个缩写的编码。——译者注

② private 和 static 方法默认为 final，不能覆盖。——译者注

③ 如果没有覆盖父类的任何非 final 方法，那么表就是空的。——译者注

表 4-1 描述了在覆盖了 `java.lang.Object` 的一些方法以后，再调用这些方法，会发生什么事情。比如说现在拿到一个对象实例，要调用该对象实例的 `toString` 方法，假设这个对象实例其实是 `String`，那么调用的其实是 `String` 类实现的 `toString` 方法。每一个类都有一张这样的表。在 `Java` 的类中，只要一个方法没有被声明为 `final`，^①那么就会对应着一个虚方法，一旦有代码调用该方法，就会在相应的虚方法表中查找这个方法。代码调用方法时可以从实际对象处拿到正确的虚方法表。这是一个很正常的操作过程。但通过这个机制，开发人员可以很容易了解面向对象编程背后隐藏的内容。比如说，如果一个方法可以被子类覆盖，那么在虚方法表中，就会有该方法的一个映射。

表4-1 Java基础类的虚方法表示例

方 法	Object	String	Number	Integer
"equals"	Object.equals	String.equals	Object.equals	Integer.equals
"toString"	Object.toString	String.toString	Object.toString	Integer.toString
"finalize"	Object.finalize	Object.finalize	Object.finalize	Object.finalize

4

在 `Java` 中，二进制兼容有其独特的表现。比如说，大家都会以为如果一个程序分别基于两个版本的类库进行编译时，编译的二进制结果应该是相同的。有时候，可能的确如此，但通常情况下，事实并非这样。特别之处就在于可被重载的那些方法。两个同名方法，但参数却有所不同，这样就可以称为方法重载。比如说 `java.util.Arrays` 有多个 `toString` 方法，每一个方法的参数都是不同的数组类型，如 `byte[]`、`short[]` 和 `Object[]`，这些就是方法重载。在编译代码时，编译器会根据参数类型来选择最合适的方法。在 `Java` 语言规范中定义了：如果有多个方法都能匹配到现有参数上，那么选择最“接近”的一个方法^②。

StringBuffer 新功能的不兼容

近几年，`NetBeans` 项目都能支持最新的两个 `Java` 版本^③。这样做，就可以避免强迫用户为了使用 `NetBeans` 而升级到最新的 `Java` 版本。当然，这样做的缺点在于我们被迫只能使用前一个版本所支持的功能和 `API`。

在 `Java 1.4` 发布以后，我们尝试使用了它提供的一些新类库。我们可以使用反射技术来调用那些 `Java 1.3` 中不支持的新方法和类。但用反射来编码，并不方便。无法使用编译器对类型进行强制检查，而且在使用反射时还必须写上一大堆处理异常的代码。所以我们决定修改我们的编译脚本，在使用 `Java 1.3` 编译时，可以忽略所有名称为 `*14.java` 的源代码，而在使用 `Java 1.4` 进行编译时，则不必忽略。这样就可以把 `Java 1.3` 和 `Java 1.4` 的代码分别放置，而且可以同时

① 其实 `private` 和 `static` 方法默认是 `final`，子类是不可能在此虚方法表中对其进行处理，所以这里说的 `public` 和 `protected` 级别方法，但又没有声明为 `final`。——译者注

② 假设有两个方法 `test(Object)` 和 `test(String)`，现在 `String t; test(t)`，此时两个方法都可以接受 `String` 参数，但后者更接近一些，所以会使用后者。——译者注

③ 最新的两个版本，是指最新发布的大版本，现在就是 `JDK 6` 和 `JDK 5`。在 `JDK 6` 发布前，是 `JDK 5` 和 `JDK 1.4`。而本书所写的则是 `JDK 1.4` 和 `JDK 1.3`。——译者注

支持这两个版本。有兴趣的读者，可以参考 conditionaluseofapi 这个项目，看一下，我们是怎么处理这种问题的，下面就是一段示例代码：

```
interface AddString {  
    public void addString(String msg);  
    public String getMessage();  
}
```

这个 AddString 接口有两个实现类，现在需要选择一个合适的实现类，具体的选择则要看使用的是 Java 的老版本，还是新版本：

```
AddString add;  
  
try {  
    Class onlyOn15 = Class.forName("java.lang.StringBuilder");  
    Class codeOn15 = Class.forName(  
        "conditionaluseofapi.StringBuilderAdd15"  
    );  
    add = (AddString)codeOn15.newInstance();  
} catch (ClassNotFoundException ex) {  
    add = new StringBufferAdd();  
}
```

如果我们基于最新的 Java 版本来编译代码，就会生成所有的类，也可以使用所有的功能。如果是使用老版本的 Java 来编译代码，我们得到的类就会较前者少一些，但整个执行过程仍然可以在老版本的 Java 上正常运行。我们曾认为这是一个比较安全的解决方案。事实上，我们比较害怕那种偶然用到的新 API，也就是说新版本的 Java 添加了一个新方法，有程序员无意中调用了，如 JDK 5 中在 Collections 中加了一个 add All 方法，相应代码在 JDK 1.4 就无法编译通过。所以我们会同时在两个 Java 大版本上编译代码，以保证不会出现这种情况。这种同时使用两个 Java 大版本来编译代码的方式，可以避免无意中使用了最新版本引入的新 API，最终才能保证编译结果可以在两个版本上运行。在设定这个方案时，我们的确是这样想的。

但让我吃惊的是，我们还是能时不时地收到 NoSuchMethodError 的错误报告。在经过调查后，发现错误是由下面代码而引发的：

```
StringBuffer sb = new StringBuffer();  
StringBuffer another = new StringBuffer();  
sb.append(another);
```

问题出在 Java 1.4 中为类 StringBuffer 新引入的方法 append(StringBuffer sb)，在 Java 1.3 版本中是没有这个方法的。但一直以来，StringBuffer 都有一个通用的方法 append(Object obj)。所以前面的那段代码不管在 Java 1.3 还是 Java 1.4 中都可以编译通过，但编译的结果是分别指向不同的方法。

这下明白了，所谓的两版本同时编译并没有满足我们的希望。使用 Java 1.4 编译的代码，在 Java 1.3 上是运行不起来的。但我们又很希望使用 Java 1.4 的新功能，所以我们改进了产品编译的方式。不只使用 Java 1.4 进行编译，我们其实将代码编译了两次：第一次使用 Java 1.3，

这次编译会忽略`**/*14.java`文件，但可以正确使用 Java 1.3 所支持的 API。然后再使用 JDK 1.4 来编译代码，但这次不会编译那些已经编译成功的代码了，只是将原来忽略的`**/*14.java`代码文件进行编译。

说到二进制兼容，方法重载并不是唯一出现问题的地方。出于某种考虑，二进制格式中对 String 与原始类型（如整数）有着不同的处理方式。如果定义了一个 static 的 public 常量，而且声明为 final，不管是 String 还是 Int，那么它的值是不能通过引用的方式来使用的，而是在编译期间将其值复制到具体的二进制 Class 文件中^①。这说明，如果在 API 新版本中改变了一个这样常量的值，这种对值的改变却不会影响那些已经编译好的类文件，即使那些类文件的确使用了这些常量。你可以把这一点用于某些特殊需求上，比如说版本控制，因为这些常量与版本相关，像 JBuilder 就干了这事，看下面的代码：

```
public abstract class API {
    public static final int VERSION = 1;

    protected API() {
        System.err.println("Initializing version " + VERSION);
        init(API.VERSION);
        System.err.println("Properly initialized: " + this);
    }
    protected abstract void init(int version) throws IllegalStateException;
}
public class Impl extends API {
    protected void init(int version) throws IllegalStateException {
        if (version != API.VERSION) {
            throw new IllegalStateException("Wrong API version error!");
        }
    }
}
```

看一下前面的代码片段。为什么在内部的 init 方法中对 version 做这样一个奇怪的检查呢？传入的值不就是那个 VERSION 常量吗，为什么还要再进行一次判断呢？事实上这两个值的确是相等的。也肯定是相等的。但这种相等是有一个前提条件的，就是 API 和 Impl 这两个类必须是同时编译。做个假设，如果先写好了 Impl 类。就像为 JBuilder 写了一个扩展功能，然后没有重新编译该类，就安装到 JBuilder 的新版本中，而原来使用 API 中的变量已经从 1 更新为 2，可以参考下面的代码：

```
public static final int VERSION = 2;
```

此时 Impl 类的 init 方法对 VERSION 变量的检查失败。数据的检查其实是 $2 \neq 1$ 了，因为 Impl

① 比如说有一个 A 类声明了 `public static final string a="A"`，然后 B 类使用了这个 A 类中的 a 字段，如 `public static final string b=A.a`。如果对 A 中的 a 进行修改，使之成为 `public static final string a="AA"`，如果不重新编译 B，此时 B 中的 b，其值仍然为“A”，而不是“AA”，这就是在第一次编译 B 的时候，已经将确认好 A.a 的值直接写入到 Bb 中了。这种不是引用，而是直接将值复制过去。——译者注

这个类在原先编译时，将 `API.VERSION` 这个常量记录写入到自己的 `Class` 文件中了，也就是说，除非重新编译，否则在 `Impl` 类中，`API.VERSION` 的值始终是 1。很让人吃惊吧，但对原始类型（包括 `String`）常量的改变的确做不到二进制兼容。

二进制字节码的格式与 `Java` 源代码的格式非常相似，这有好的方面，也有坏的一面。说它好，是因为这样的格式很容易理解。说它坏，是因为它会引发一些误解。但大家应该记住，在编写 `API` 的时候，只有通过二进制格式才能最终判断不同版本的 `API` 是否兼容，也就是说代码执行时的兼容性才是最根本的。所以一定要了解 `Java` 源代码是编译成何种样子的字节码。如果有疑问的话，最好反编译一下 `Class` 文件，检查一下到底是不是自己期望的样子。有可能你会为反编译的结果大吃一惊！

4.2.3 功能兼容——阿米巴变形虫效应

即使一个类库的新版本可以正确地替换老的版本，而且程序也可以正常进行连接，但这仍然不是“无绪”想要的最终目标。请让我来为各位读者再温习一下“无绪”的定义：集成多个模块构建一个程序时，无须深入了解每一个模块。如果我们能做到二进制兼容，只是保证了系统能够被正确连接，但正确的连接并不能保证系统正常运行。二进制兼容仅仅是兼容性开始的第一步，接下来还有更多更重要的兼容性话题需要讨论。

对于集成和升级一个基于组件架构的系统来说，无绪的最终目标其实是希望能够做到分布式开发。不同的开发小组可以基于不同版本的类库来编译自己的程序，而且不管最终用户机器上安装和部署的类库具体是哪个版本，都可以让程序正常运行。要达成这个目标，并不是说仅仅把程序正确地连接起来，就一切没问题。当然，连接是一个首要的前提条件，接下来就是要保证各个模块能够按照设计时的目标正确运作。这种情况，可以称为“功能兼容”。

给“功能兼容”下个定义还是很简单的。前面已经讨论过了，如果所编写的代码可以同时到老版本和新版本上都编译通过，那么这两个版本可以称为源代码兼容。如果编译后的代码在执行时，可以同时在老版本和新版本上都连接通过，那么这两个版本可以称为二进制兼容。如果一个类库在运行时，不管所引用的是老版本还是新版本的类库，其产生的结果完全相同，那么这两个版本可以称为功能兼容。这个定义看起来简单，但背后的含义却不简单。

作为开发人员，你可能会清楚地知道你所开发类库都提供了哪些功能，假设你提供了优秀的规范和完善的文档还有其他的信息，从而能够清楚地对类库的功能加以说明。当然这只是一个假设，从来都没有什么优秀的文档可以做到上面所说的目标。在现实世界中，文档总是比程序慢上一步，而且其描述的信息也只是整个程序的一部分内容。但还是先假设有一些开发人员已经完美地分析了程序，并清楚地知道程序的所有功能。如图 4-1 中所描绘的那样。

但大家都清楚地知道软件开发中的一条金科玉律：每个程序至少都有一个 `bug`。什么意思呢？所谓的 `bug` 其实就是程序的功能不符合预期定义的内容。即使开发人员愿意相信程序的行为如图 4-1 中定义的那样完美，而现实中，程序的功能与其预期定义的内容都是存在出入的。在特定情况下，代码并没有实现预期的功能，而在其他情况下，所完成的功能要超出预期。如图 4-2 所展示的那样。

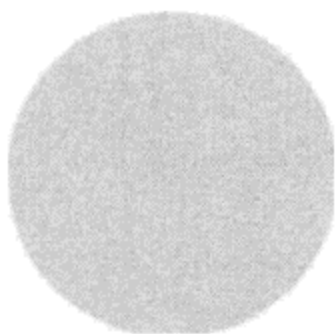


图 4-1 理想世界中程序功能的描述

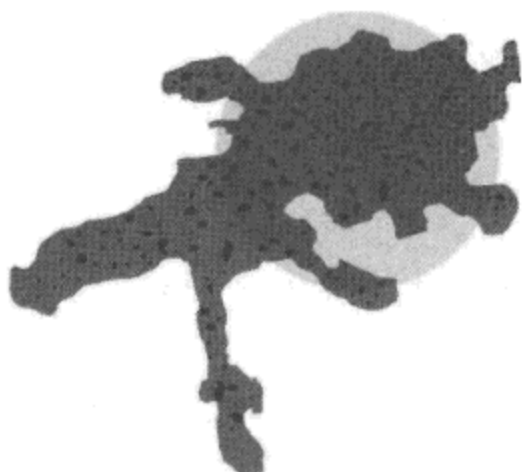


图 4-2 现实世界与理想世界中程序功能的区别

问题就隐藏在这张图的背后！假设这张图同时描述了现实世界与理想世界中的程序功能。那么圆是一个清楚的功能定义，而具体程序则有所出入，有时没有完成规范中定义的某些功能，有时却又超出了规范中定义的某些功能。而程序员在开发代码时，往往不会去阅读相应的规范。事实上，几乎没有哪个开发人员会去读这些 API 的规范，直到出现严重的问题。他们会用“编码/运行”这样的方式来完成自己的工作。比如说开发人员写了一些代码，运行之后发现完成了自己需要的功能。此时，他们关注的是那些 API 背后具体实现所完成的功能，并不关心规范怎么说。这样做，其实是让代码依赖于具体的实现，而这些实现内容并不会写到规范中。显然，不管是少提供了功能，还是多提供了功能，这两种情况都是非常危险的，会影响到未来类库版本的共存，更会影响使用这些类库的程序。一旦该类库有新版本发布，修正了某些 bug，或者再添加一些功能，都有可能会影响你的程序，就像图 4-3 一样，从圆变成了不规则形状。



图 4-3 下一个新版本将程序变得更不规则

再重复一次，功能的具体实现可能与其预期差别很大。而且新版本与旧版本的功能差别可能也会很大。每一个新版本的发布都可能存在这种情况。每个新版本的功能都会有所变化，就像阿米巴^①这种变形虫一样，不断地在改变自己的形状。结果可能会造成，一个类库的某些功能方式无法再正常运行了，于是，想让 API 保持兼容的这个目标就被破坏了。这个 API 无法再支持各开发小组的独立开发了，其分布模块的发布计划会受到影响，也无法让不同版本的模块正常运行了。

前途未卜

NetBeans 的开发经验告诉我，人们更容易接受源代码和二进制兼容，但对功能兼容的理解就要差一些。造成这种情况的原因不止一个，但我认为人们对这方面认识不足的主要原因只

^① 阿米巴，一种变形虫属或相关属的原生动物，存在于淡水和土壤里，或为其他动物的寄生虫。变形虫没有固定形体，身体主要由原生质组成，包括由一个柔软、有韧性的外膜包围的一个或一个以上的细胞核。——译者注

有一个，接下来我会对此进行详述。

如果想影响源代码和二进制的兼容，那么开发人员总得公开做点事才行，比如说创建一个类，然后设置为 public 级别，并添加一个 public 方法，只有这样才有可能让源代码无法编译通过或者二进制代码无法正确连接。但说到“功能兼容”这个话题，其实不需要开发人员特意去公开任何内容。API 的行为就可以自动证明兼容性了。虽然有些人会认为并不是所有公开的类或者方法都是 API，但所有公开的功能或者说行为也都需要考虑兼容性，这一点正是隐藏在深处的“功能兼容”。

就像阿米巴变形虫模型显示的那样，程序的实际行为并不等同于我们的期望值。即使没有特意去破坏，也会出现一些错误的功能行为。令人遗憾的是，API 的用户没办法区分哪种是特意破坏，哪种只是巧合。如果一个外部的开发人员基于某个版本的类库来编写代码，而且在此过程中，依赖了某项功能，这项功能虽然不在 API 的规范中，但可以正确运行，保证程序员可以快速地完成自己的工作任务。但在类库的下一个版本中，可能会因为功能兼容性受到了破坏，从而使得整个程序都无法运行下去。现在出现了问题，但这个问题的责任到底是谁的呢？

任何一个好的老板都知道，责怪自己的客户决不是一个好策略。同样，如果一个 API 的用户使用 API 的方式不正确，最终在升级到 API 新版本时，造成原有的程序无法运行，那么应该责怪的人只能是 API 设计者，如果想设计一个能够被广泛使用的成功 API，那么设计者就必须更加留意这一点。

接受这种责难是一件比较困难的事情。没有人喜欢空头支票，必须要保证当前版本的每一个功能行为都应该是 API 的一部分。即使你用最恶毒的语言去咒骂每一个破坏了兼容性的功能，也无补于事，这样做也不对。因为一旦有一天，大家都普遍认为建立大系统时应该能够做到“无绪”，那么就必须调整自己现在的态度。

API 设计者需要为 API 产生的阿米巴变形虫效应负上责任，而不能把这种问题认为是规范以外的内容，就置之不理。

要为自己的行为负上责任也不是一件容易的事。所以 API 设计者首要的目标就是要减少阿米巴变形虫效应，要让 API 的功能行为尽可能地与规范保持一致。这事做起来决不简单。需要开发人员对 API 要完成的功能有清楚的认识，同样还要有良好的技术水平，才能将自己的意图贯彻到代码上，此外还得评估一下 API 的用户会如何来使用（还要想一下这些用户如何来误用 API）。下节列出了要达到该目标所需要做的几步工作，从而减少阿米巴变形虫效应。

4.3 面向用例的重要性

请记住，如果一个 API 被广泛使用了，那么就不可能了解所有使用该 API 的用户。比如说，Linux 内核的作者不可能知道所有使用该内核的开发人员以及他们使用内核的动机所在，也不清楚地球上有多少人使用了 `ioctl`^①这个方法从内核得到相应的内容。面对这种情况，唯一的办法就

① `ioctl` 是设备驱动程序中对设备的 I/O 通道进行管理的函数。所谓对 I/O 通道进行管理，就是对设备的一些特性进行控制，例如串口的传输波特率、马达的转速等。——译者注

是站在用户的角度来设计内核的接口。同样开发 Java 及其核心类库的人,也不可能知道都是哪些人使用了 Java 语言,更不知道使用的方式为何。所以,如果设计者希望能够设计出像 Linux 和 Java 这样被广泛使用的 API,那么必须站在用户的角度来理解如何设计 API 库,以及如何才能设计出这样的 API 库。

既然不知道自己的客户,自然也无法进行交流,那么就有两种解决方案:一是找一些用户,对其进行研究,还是一种方式就是基于用例。基于用例也就表示站在用户的视角,然后再对部分用例进行针对性的处理。从用户处取得反馈信息,并对可用性进行研究,这是一种很好的工作方式,可以通过反馈来检验设计的用例是否正确。在做设计之前,必须进行分析,明确为什么要写这样的 API,API 应该长什么样,以及如何才能完成目标。

没有真实用户,何来无错的 API

很多人都和我说过,如果一个人在设计 API 时,没有和真实用户进行交流,并收集建议,那么他所设计的 API 是很难做到完美的,或者说这个 API 不可能没有错误。因为衡量一个 API 的最终标准就是用户的满意度。用户只会对一种情况感到满意:可以很容易地使用 API 来完成其工作。

这好像是一个先有鸡还是先有蛋的问题。没有 API,何来用户,更不用说写一个好的 API,不仅要倾听用户的意见,还要在设计时将用户的好建议加到 API 中。你也许会想这种情况是没有令人满意的解决方案的。但我们却需要一个解决方案,因为我们随时都要处理这类问题。事实上,只要设计一个系统,就需要发布第一个版本。这个版本不一定要建立在真实用户的基础上,因为还没有真实用户来使用这个 API 呢,那么到底怎么做呢?需要给自己找一些假的用户,设想一下这些人会使用 API 来做什么事情。需要假想一下用户的用例。

当然这些用例都是编的。当 API 有了真实的用户时,才会发现这些用例与真实情况可能相距甚远,与真实需求也有很多不同之处。从这一点上来说,第一个版本决不可能完美。但可以减少 API 中存在的错误。说到 API 设计错误,并不是指这些 API 不能满足用户需求,这是很正常的,因为在整个系统的生命周期中,会不断地有用户提出新的需求。所谓的错误是指 API 不能在后续的版本中以扩展的方式来满足用户的这些需求。但如果你学习了本书,在设计 API 时,不仅可以通过扩展的方式满足新需求,而且新版本也不会破坏客户基于第一个版本开发的那些代码。

一个用例其实就是对 API 一种用法的描述,它指出了用户可能要面对的问题,而且这个问题不是一个具体的问题,而应该是很多具体问题的抽象。要想提供一个通用的解决方案,首先就是尽力地去收集那些接近用户实际需求的问题。这里所说的解决方案,就是本书的重点,即 API。来看一下 NetBeans 平台上提供的一个 API,它是一个数据库管理器,其功能就是注册 JDBC 驱动,功能说明如下。

一个外部的模块可以将 JDBC 的驱动注册到当前系统中,比如说写一个将数据库服务器功能整合进来的模块。此时该模块肯定要包含一个针对该数据库的 JDBC 驱动,然后基于数据库管理器的 API 将数据库的内容显示在数据库浏览器中。还有一种类似的 API,就是将 J2EE 服务器功能整合到 NetBeans 平台中。有时候,一个 J2EE 服

务器会整合一个数据库管理器，这样有助于提高用户的使用感受。当这个 J2EE 服务器注册到 NetBeans IDE 中，其内部整合的数据库服务器也会注册到数据库管理器中。

当有了这种用例以后，就可以着手考虑这些 API 的详细功能了。我们把 API 要完成的每一个功能称为场景 (scenario)。一个场景其实就是对用例中问题的回答。仿佛用例会向设计人员发问：“我应该怎样做？”而场景则会回答说：“你应该先做这件事，然后再做那件事，就可以了。”现在还是来看一下对于注册 JDBC 驱动这样的一个用例，其定义の詳細场景。

注册 JDBC 驱动的方式可以是直接调用 JDBC DriverManager 或者写一个能够描述驱动信息的 XML。这个 XML 文件的格式由 JDBC 驱动的 DTD 所指定。下面就是一个使用 PostgreSQL 数据库注册格式的写法：

```
<!DOCTYPE driver PUBLIC
    '-//NetBeans//DTD JDBC Driver 1.0//EN'
    'http://www.netbeans.org/dtds/jdbc-driver-1_0.dtd'
>
<driver>
  <name value='postgresql-7' />
  <display-name value='PostgreSQL (v7.0 and later)' />
  <class value='org.postgresql.Driver' />
  <urls>
    <url value='file:/folder1/folder2/drivers/pg74.1jdbc3.jar' />
  </urls>
</driver>
```

这个 XML 文件应该放置在模块的 Databases/JDBCDrivers 目录下。为了安装一个 JAR 包格式的模块到 NetBeans IDE 中，需要使用 NetBeans 指定的安装协议，可以用一个 URL 来表示：nbinst:/modules/ext/bundled-driver.jar。

这个场景将抽象的用例与 API 中的具体实现结合在一起。它清楚地指明需要实现哪些类和接口，并将声明的配置文件放到哪个具体位置以便将相应的类注册到系统中。当然，这个场景中还包含了更多细节性的 API，如 JDBC DriverManager 及其 Javadoc，以及描述 JDBC 驱动的 XML DTD (Document Type Definition, 文件格式定义)，可以通过相应的链接以便更深入地了解其内容。提供这样的一个场景可以方便 API 的用户快速定位到他们感兴趣的内容，比如要实现的类、接口等。这种处理方式，可以帮助开发人员根据场景中的内容分析出要做的基本工作。如果这些内容还不足以帮助用户使用 API，那么用户可以去找相应的文档进一步地了解相关信息，如 Javadoc、DTD，甚至源代码中的某些信息。

NetBeans 的文档结构就是这种从上至下的方式，我们觉得这种组织方式可以更好地帮助我们的用户浏览和查找他们所需要的信息。但我们在内部还是使用“用例”、“场景”和“Javadoc”这样的组织方式，因为我们觉得这样有助于评价 API 编写的质量。

前文展示的阿米巴变形虫模型说明了对外的功能描述和其内在实现之间是存在着差异的，正是这种“差异”引发了 API 维护中的主要问题。所以要尽可能地减少这种不一致。但想要实现这

个目标也要有一个前提，就是能对外提供一个定义清楚而且明确的规范，否则没有规范，拿什么来进行比较呢！

在开发 NetBeans 的 API 时，所谓最新的规范，其实只存在于编写代码和相应 Javadoc 的程序员的脑袋中。根本没有办法检查一份工作做得是好还是坏，也根本不知道是否完成了其设计时所定义的功能。即使其完成的功能已经超出了设计功能，也没有办法确认这一点。换句话说，我们根本无法清楚地知道原有的设计已经变成了什么样子。唯一知道的人应该算是那个具体编码的程序员了。但开发人员往往将精力集中在具体的实现上，完全无视他人的工作。如果说这位开发人员是一位非常优秀的设计者，那么事情可能还好办一些。如果都是这种类型的人来开发代码，那么可以高枕无忧了。但请记住那句话：“第一个版本远非完美”。假设项目中那位能够掌握全局的人离开项目组，那么不管其他的开发人员技巧如何娴熟，代码都会出现无法维护的情况，可谓一团乱麻。如果一个类库，没有说明文档，没有全局的角度来对其开发过程进行管控，这样一个东西有可能就是天才之作，但它也很脆弱：因为每一个小小的改动都有可能使模型产生变化，使得整个系统向着阿米巴虫模型的方向发展。最终使得使用该库的程序可能出现问題。

当然，这样做也有可能产生好的类库。但这样的工作方式更合适于那些艺术家，至于工程师，还是算了吧。不管人们是否乐意承认，要编写一个稳定的软件，需要一种完全不同的方式，必须将过程和质量进行量化，这种工作方式与艺术是迥然相异的。当 NetBeans 团队认识到了这一点以后，就开始坚持使用“用例”、“场景”和“Javadoc”的方式，这样就不需要把开发团队中的所有人员集中在一起才能讨论清楚一个问题。现在可以从更高的角度来看待我们的 API，而且那些非 API 作者的人也可以更好地评价这些 API。现在我们有了更好、更有深度，而且更全局化的文档，我们的 API 变得更加容易维护了。能做到这一点，是因为我们能够更好地理解编写类库背后的动机。

4.4 API 设计评审

在 API 开发方面，NetBeans 经历了几个阶段。刚开始的时候，所有的 API 都由一位架构师^①来开发。随后发现这位架构师成为整个项目开发的瓶颈，拖慢了整个团队的开发步骤。于是我们改变了这种工作方式，改由一个架构师指导一个团队来设计 API。现在，由整个团队的成员来开发 API 了。虽然每种方式都能在软件开发理论中找到自己坚实的理由，但没有哪一个是完美的。

过去，人们一直认为设计工作是不能由一个集体来完成的，它需要一个架构师对所有的设计进行决策。当然，这样做可以简化很多事情，但仍然有一个规模上的限制。就算不考虑模块规模大小这方面的限制，这位首席架构师的压力也是非常庞大的，其责任包括设计、维护 API，还要告诉别人 API 应该怎么使用，这些工作内容都需要占用大量的时间，毕竟这位架构师一天只有 24 小时，不可能无限制地工作。

在 NetBeans 开发的头几年，我就扮演着这样一个唯一架构师的角色，我可以确定地告诉大

^① 这位架构师就是本书的作者，可以参见第一部分中，作者曾经说过这一话题。——译者注

家,这种唯一架构师的方式一定会受到规模的限制。随着团队成员的增加,想满足他们的全部需求会越来越困难。解决方法就是从团队成员中选择一些技术最好的人,指导他们来设计自己所需要的 API。但这样做会造成一致性方面的问题,因为每个人在设计 API 时都有其个人风格。肯定无益于 API 的质量,必须解决这一问题。

还好 NetBeans 团队中对 API 的存在价值已经有所共识,认为 API 就是开发人员与用户间的一种沟通方式。与此同时,我们还发现每一个设计良好的 API,都有着相同的动机。所以我们现在更加开放,更加专注,有一个团队来配合 API 的作者对 API 进行评审工作。事实上,考虑到前言中对 API 所下的广泛定义,其实每一个开发人员都会编写 API。

一旦有一个 API 需要改动,任何人都可以提交一个改变的请求。其他人则需要在代码正式提交前,进行一次评审,检查新调整的内容是否符合一个优秀 API 的基本要求。比如说,我们会按照“优秀 API 规则”进行检查,保证能够满足这些规则。下面详细地列出了这些规则。

用例驱动的 API 设计:设计 API 时,要基于一些具体的场景和对 API 的认识进行抽象分析,最终给出设计。

API 设计的一致性:API 往往是由多位设计者来完成的,但整个团队中必须能够保持“最佳实践”的一些基本原则。一个接口设计得再好,只要它违反整个团队的一致性,就宁愿退而求其次。

简单明了的 API:简单而且常见的任务应该更容易处理。如果基于用例驱动的方式进行设计,就可以很容易地通过那些可以简单实现的场景来验证这些 API 是否可以完成那些重要的用例。

少即是多:一个 API 对外提供的功能应该只包括用例中说明的功能。这样可以避免出现需要的功能与实际提供的功能两者之间出现差异。

支持改进:以后也必须能够维护这个类库。如果出现新的需求,或者原作者离开,都不会出现放弃这个类库的情况。

一个目标明确的小团队可以对 API 设计的各方面进行检查。好处在于,这样一个团队在审核某一个 API 时,并不需要都是该 API 领域的专家。因为评审中要检查的内容都是 API 最通用的地方,每个人都可以根据自己的经验给出自己的评价。所以 NetBeans 中的 API 设计可以由一个小团队来完成,而且即使整个开发队伍非常庞大,也仍然可以保持 API 设计的一致性。负责评审的人总会不停地向团队成员解释为什么要这样设计 API,以及 API 设计的内容,而这样 API 设计的工作就可以分配给不同的人,而且 API 的设计机会也开放给任何人。如果读者有兴趣了解更多这方面的事情,可以通过 <http://openide.netbeans.org/tutorial/reviews> 来更加深入地了解这个团队。

4.5 一个 API 的生命周期

我已经说过很多次了,开发 API 的过程其实就是一个沟通交流的过程。沟通的双方就是 API 用户和 API 设计者。

API 有可能是这样产生的,有些人写了一些代码,而另外的人发现这些代码的价值,就开始

使用这些代码。在这种情况下，API 是以一种自然的方式产生的。随后 API 用户和 API 作者有了相互沟通的渠道，开始交流经验，可能发现这个功能一开始的设计并不是很通用，或者说一开始时，作者并没有把这个功能当成一个 API 来设计。为了让这个功能成为一个 API，他们开始讨论如何进行调整才能改善这个功能。经过几轮的迭代，才会带来一个有用而且稳定的 API。

贡献，而不是破坏

前面所说的这种 API 开发方式需要双方进行相互合作，共同帮助开发人员走向成功。至少要希望那些有意改善 API 的新用户能够将自己的意见传递给 API 设计者，清楚地说明自己对 API 的一些特殊需求。如果做不到这一点，API 就不可能有所改善。使用 API 的方式就会剑走偏锋，更像是那种黑客的方式来使用这些 API^①。

从另一方面来说，提供 API 的人必须准备好接受此类请求，并予以合适的回应。直接粗暴地回答“别这样用，因为这根本不是 API”，是没有任何意义的，应该告诉用户其他的可选方案。双方应该寻求一种合作及维护方式，以便进行 API 开发。大家经常关心的一项内容就是高昂的维护费用。但在 14.4 节中已经指出：维护的成本问题其实有更好的解决方案。

但再换一个角度来看，API 设计者希望在没有对外提供一个 API 之前就能和相关的用户进行沟通。这种 API 的开发方式更接近于基于业务的设计方式。在这种场景下，系统中的两个组件间的协定是已经明确的，至少说也是已经有了需求。收集需求，定义问题域，明确用例，再由指定人员来设计 API。现在，其他人员就可以使用这个 API，并可以给出自己的建议，列出 bug，提出一些功能方面的改进意见。这些建议都有助于改进 API，使其用途更广，也更稳定。

长期投资

API 总有其存在的必要性。没有 API 的话，就不可能使用“无绪”的方式来装配应用程序。但开发人员需要清楚地认识到一点：开发了一个 API，并不意味着这个 API 就能得到广泛的使用。

比如说在 NetBeans 中，我们需要一个处理命令行的 API。我们希望在运行 NetBeans 时，这个 API 能读取传入的文件参数，并能在 NetBeans 中使用相应的编辑器打开该文件。但因为 NetBeans 采用模块化架构，所以在命令行 API 中不能直接使用编辑器来打开参数指定文件，这个模块认为命令行 API 完全不知道文件这个概念，也就是说不能通过命令行的方式来访问文件。经过几个版本的调整以后，我们提供了一个 private 级别 API，而且也没有在文档中对该 API 加以说明，通过这个 API 就可以在两个模块之间建立关联，然后能正确地处理文件。但在 NetBeans IDE 6.0 中，我重写了一个正式的 API，以便大家都能使用这一功能。

在 NetBeans 6.0 IDE 中，没有其他的模块使用这个解析命令行的 API。那是不是意味着创造这样一个 API 没有任何意义呢？我不这样认为。我在邮件组中见过关于这个 API 的用法讨

^① 这里所说的那种黑客方式来使用 API 是指，因为 API 提供的某些功能没有正式对外公开，比如说方法是 private 的，那么就会使用反射的方式来调用这个方法，书中对于这种不遵守公开方式来调用 API 功能的操作，都称为黑客方式。——译者注

论, 既然将这个 API 正式地放入公开的文档中了, 表示其他人可以找到这个 API, 并考虑如何使用该 API。该 API 可以提升 NetBeans 平台的活力, 可以吸引新的开发人员。但在它被广泛使用之前, 还有很长一段路要走。这个 API 公开出来供大家使用, 可能不会有人立刻就使用它, 但这种 API 是一种长期投资。

尽管这些 API 的案例各有其不同的缘由, 但有相同的特点: 每一个都需要时间来让用户进行试用, 并进行反馈, 然后才能宣称这个 API 是可以正常运行的。当然不是说这样做就可以带来稳定的 API, 有时候, 也有可能最终什么都得不到。如果出现这种情况, 最好还是放弃这个 API 吧。有时候, 可能交流的双方无法进行高质量的正式交流。在开始的时候可以简单地聊一下, 交流相应的需求, 但如果新发布的版本也证明了这种简单的沟通方式并不合适, 那么双方也许应该更进一步地进行交流, 使得沟通能更简单更有效一些。

一切皆有可能。但对于那种开发人员之间的沟通问题, 最好还是要描述清楚。如果你要设计一个 API, 那么在这个 API 没有成熟前, 最好能够清楚地告诉其用户: “这个 API 还没有完善, 你可以尝试使用这个 API, 但一定要小心。” 在有了稳定的版本以后, 还可以骄傲地告诉用户: “这是我开发过的最好的 API! 放心使用这些 API, 我可以保证它能一直提供支持。” 这样做可以俘获 API 用户的“芳心”, 让他们“拜倒在你的石榴裙”下。但请一定要注意, 对于 API, 要能够清楚地标识其当前状态, 以使用户了解相应版本是否可以稳定使用。

像 Linux 内核及其外部接口都遵守了这一原则, 它使用小数点后的一位偶数来表示稳定的版本, 而用一位奇数来表示开发版本。如 2.0, 2.2, 2.4 和 2.6 都是稳定版本, 而 2.1, 2.3 和 2.5 则是开发版本, 不是很稳定。这样可以明确地告诉用户每个版本所处的状态, 以使用户根据自己的需要选择合适的版本: 喜欢尝鲜的人可以选择带有新功能、新技术的开发版本, 而对于谨慎的人, 可以选择没有什么新功能, 但很稳定的版本。

如果想告诉类库的用户当前发布的版本还不稳定, 那么最简单的方式就是把其版本标识为 0.x。因为它还没有到达 1.0 版本, 表示还在开发中, 也就可能还会有所变化。不管用哪种版本标识方式, 最重要的是要让 API 的用户清楚地知道当前版本的状态, 以便他们决定如何使用该 API。

对于 NetBeans 来讲, 如何通过版本号来描述 API 稳定性? 要说一句, 不管是 NetBeans 还是其他的大型项目, 都会面对一个问题, 就是: API 并不是一个整体, 而是多个有关联关系的集合, 每一部分的 API 当前所处的状态都有所不同。有一些 API 可能是 0.x 版本, 还不稳定, 而其他的 API 可能已经非常稳定了。

某个 API 是正在进行改进, 还是已经非常稳定, 允许用户使用了呢? 为了清楚地说明这一点, NetBeans 团队决定采用一种系统化的分类方式对 API 的稳定性进行描述。希望 API 的作者能通过这种方式来说明一个类库当前所处的阶段, 这样用户就可以根据这些信息确定是直接使用这个类库, 还是再耐心一点儿等它成熟后再使用。NetBeans 针对 API 提出了以下的分类方式。

- 对于那种仅供内部使用, 不公开给外部开发人员使用的模块或者类库, 可以归为 Private 一类。哪些功能可以归为这一类呢? 我认为如果一个 API 只包括环境配置文件, 以及读

取文件的代码等，那么这一类的 API 可以归为 Private。大部分情况下，如果代码或者类库都只是读取配置文件，那么这些功能都不需要提供给外部使用。比如说，通过一个配置文件来打开日志功能，或者通过一个文件来配置缓存。这些外部因素的内容都会对类库的行为造成影响，但并不可过分依赖于这些内容。因为这些功能可以在特定的版本上正常运行，而每一个版本都可能有所改变，所以依赖于这些 API 的风险是比较大的，应该尽量避免。对于这类 API 来说，如果有所改动，那么进行评审并不是必须的，但还是建议走这样一个流程，可以保证这类 API 始终以 Private 的方式出现，同时也可以尝试找更好的处理方案。

- 对于那种为系统内部其他模块提供相应功能的 API，可以归为 Friend 一类，但这类 API 的限制比较多。对于一个大的系统来说，这种情况是很常见的，有其存在的必要性，因为在一个大的系统中，没有哪个模块可以完全不依赖于其他模块而独立存在。在 NetBeans 中，一个模块如果使用了这个 API，那么就在提供 API 的模块和使用 API 的模块之间建立了一个约定，表示该模块可以接受这个 API 所做的修改，允许出现不兼容性问题。这样的“书面约定”就意味着参与的各方要接受所有的规定。在 NetBeans 架构中，允许每一个模块提供一个 Friend 列表，明确告诉系统哪些模块可以访问它所提供的 API，这样系统可以在运行时对其加以检查。这种处理方式可以限制那些 API 仅被指定的模块所使用。这样做的好处在于可以清楚地知道提供 API 的模块和使用 API 的模块。这两者之间有着紧密的关联关系，因为某一个模块重新编译的时候，有可能破坏关联模块的源代码兼容性和二进制兼容性，通过这种声明，可以重新编译关联模块，有助于维护系统的兼容性。所以 NetBeans 指出：如果是由同一个团队同时来提供多个有关联关系的模块，那么最好使用这种方式。通过一个列表明确地告诉 API，这个模块可以接受 API 的改变，当然这样的列表也可以在不同版本中有所变更。
- 如果一个版本将自己标识为正在开发的版本，就表示它还不是一个稳定的版本。NetBeans 也像 Linux 一样，使用 2.3.x 和 2.5.x 版本号来表示这些都是不稳定的开发版本，正如很多类库都用 0.x 这种版本号来表示自己还不稳定。这种 API 可以公开给客户使用。虽然不兼容的可能性比较小，但是下一个版本还是有可能出现不兼容的情况。当然，出现这种不兼容必须要有一个充分的理由，比如说无法解决的性能问题，或者干脆是一个完全错误的设计方案。另外在任何情况下，如果一项内容只能让 API 看起来漂亮一些，但没有实质上的改进，那么这项内容就决不应该加入到 API 中。我们通常会建议 API 的用户订阅一份邮件列表，对 API 进行改进之前，相关的一些评审内容都会公布在这个邮件列表中。对于每一个开发版本来说，都应该每天公布相应的这种稳定程度的 Javadoc。
- 有一类 API 可以称为稳定的 API，它表示这些 API 已经满足产品级别的质量要求，可以由外部用户来使用这些 API 了。客户在使用这些 API 时，不用担心新版本会带来不兼容性问题。如 Linux 的 2.4.x 和 2.6.x 版本就是这类 API，还有一些类库的 1.0 版本（及其以后的升级版）也表示这些类库已经可以稳定使用了。这类 API 一旦正式发布后，后续的

版本就必须保证其兼容性，直到这类 API 被弃用，或者有充分的理由认为其生命周期已经结束了。稳定意味着要保护 API 客户在使用 API 进行开发时的各种投资，如果要对 API 加以改进，就一定要考虑这一点。对 API 所有的调整在正式提交之前都应该进行恰当的评审。对于正在开发的版本来说，稳定的 API 应该通过 Javadoc 公布出去，像 NetBeans IDE 每一个版本的 API 都会这样公布出去。

- 如果一些 API 已经稳定下来了，而且其命名方式遵守了 NetBeans 的正式命名规则，那么这类 API 可以称为正式 API。NetBeans 的命名方式包括：`org.netbeans.api`、`org.netbeans.spi` 或 `org.openide`。对于以这种方式发布的包，NetBeans 会认为这些 API 已经非常稳定。这样做可以让用户很容易地就知道哪些 API 是稳定的。如果看到这种命名空间的类，就可以认为这是一个非常稳定的 API 了。
- 由第三方提供的接口可能不会遵守 NetBeans 指定的规则，所以就很难归类。所以可以参考第 10 章“小心使用第三方 API”一节中的内容，尽量不要将第三方提供的接口暴露出去。
- 标准 API 与上述第三方接口类似，同样由 NetBeans 之外的第三方提供。但是，此类 API 一般要以兼容的方式来扩展接口，Java Community Process 及其规范请求（specification request）就是一个例子。标准 API 不会频繁变动，也不会存在兼容性问题。
- 最后一类则是弃用的 API。在经过一段时间以后，几乎每一个 API，不管是处于什么阶段，都会变得过时。需要开发新的功能以支持新需求，需要使用新 API 来替换旧的 API。在这种情况下会将这些老的 API 用 `deprecated` 关键字标识为弃用。一个原来很稳定的 API，如果被标识为 `deprecated`，就意味着，这个 API 只能再使用一段时间，再经历若干个版本，等用户完成了代码移植以后，该 API 就会从产品中移走，就是说用户不能再使用这个 API 了。当然为了支持老客户使用这些 API，可以采用其他的替代方案，比如说提供一个包含该 API 但不正式发布的类库，或者通过 NetBeans 的模块注册功能自动从网上下载相关的类。

在本章的开始处，我谈到了开发 API 的两种不同方式。对于自发的 API，往往会变成前面说的那种 `Private` 或 `Friend` 类型的 API。一旦有人发现这种 API，并且感觉比较有用，才会慢慢地演化成稳定的 API。而另一种从宏观设计开始的 API，则在一开始就是处于开发阶段，经过一段时间的开发以后，再正式第一次发布，成为稳定的 API，其中含有所有的承诺与保证。

4.6 逐步改善

我已经提过多次，这里再多重复一次，第一个版本远非完美。事实上，不仅第一个版本，哪个版本都不会是一个完美的版本。不管怎么样，设计的场景不可能完全准确，不可能完全符合之前的方案。对于版本间的变化，也有两种极端的处理方式：一种是逐步改善，还有一种则是完全重写。

什么叫逐步改善呢？比如说，增加了一个方法或一个类，或者是向 DTD 文件中加了一个新的元素，又或者是增加了一个能够影响类库功能的属性。在保证老版本 API 能正常运行的情况下，演化出新版本 API。这种改进是一步步进行的。

而另外一种完全重写的方式，则是完全放弃了现有的 API，提供了一个全新版本。显然，使

用老版本的用户基本上是不可能轻易就移植到新版本上。对于这些用户来说，他们要考虑到使用哪个版本。因为一个组件只能使用一个版本，不可能同时使用两个版本。如果要使用新版本，需要很痛苦地重新编写代码，以移植到新版本上。

但如果乐观地来看这个问题，其实这种情况应该算是写一个新的功能模块，与原来的功能模块完全不同。只不过带来的问题很复杂，就是说以这种方式来重写 API，会使得这两个模块不能共存^①。系统的所有组件，只能在老模块和新模块中二选一，不可能同时使用这两个模块。这样会放大移植问题，因为整个系统中所有使用了该 API 的用户都被要求同时从旧版本迁移到新版本上。对系统进行这样大的调整需要进行良好的协调和规划。这样做意味着要冒一个很大的风险，当然，还是有可能完成这项工作的。但这已经完全违背了分布式开发的本意。

关于“逐步改善”有一个谬论，就是说“因为只有少许的改动，所以以前用户使用老版本 API 编写的程序可以在新版本上继续运行”。每一个改变都有潜在的风险，因为它可能引发一个甚至更多的不兼容问题。每一个不兼容问题（即使看似微不足道）都会反映到客户开发的程序中，可能就会产生非常严重的后果。开始时，要能够把真正的内容与最初的设想保持一致，而且任何一个小的改变都不会引发任何问题，这样才可能避免阿米巴虫模型出现。

如果考虑到向后兼容性问题，那么也许重写一个全新的版本可能是更现实一些，这样可以清楚地告诉类库的用户，如果要移植到一个新版本上，就要花费一些时间进行代码迁移工作。与前面“逐步改善”的方式相比，这样做显得更诚实一些。但这样做也在很多方面都存在问题。首先，如果新旧版本保持兼容，那么迁移的工作量就非常小，否则完全重写一个实现需要投入大量的时间，还需要一个充分的理由来说服用户接受这样的方式。如果没有令人信服的原因来说服用户，相信用户宁愿守着老的版本。要知道，每个项目最重要的问题就是日程计划的安排。如果没有一个充分的理由，没有人愿意花费大量的时间将代码升级到新版本上，他们会去做其他更重要的事情。

如果用这种态度为 API 的客户提供服务，那么就不会带来一个好的合作氛围。但还有更坏的合作方式，就是完全不提供迁移的方案。有时候，对于一个愿意迁移到新版本的 API 客户来说，还是可以接受一个 API 完全重写。但如果说让所有的用户都只使用老的版本或者强迫他们立即都升级到最新版本，对于分布式开发来说，这两种方式都不现实，正如本书一开始所说的那样。如果 API 经常产生重大的变化，而且要强迫用户随之迁移，那么客户就会转向其他的方案，而放弃现有的 API。

JDK 的类库是一个“逐步改善”的例子。在新版本中，JDK 会添加新的包、新的类以及新的方法，但多少都能保证让那些基于老版本 API 开发的程序能够在新版本上正常运行。这样，就可以把那些用旧版本编译的代码拿到新版本的 JDK 上来运行，形成了一种双赢的局面。

在 Java 语言中，也有一个完全重写的例子。Java 5 引入了一些新的语言特性，此时要求其用户做出一个重要的选择：不使用泛型，这样可以让程序跑在 Java 1.4 上；或者使用泛型，这样程

^① 因为这两个模块具有相同的命名空间，所以无法共存，至少对于普通的 Java 平台是这样的，如果引入 OSGi 或者是 .Net，情况就有所不同，所以作者的话比较适用于普通 Java 平台。——译者注

序就只能运行在 Java 5 上。Java 5 这些新加的功能普及得非常慢，这说明用户要做出这样一个选择是非常困难的，很难痛下决心做这种迁移。与“漂亮的编程语言”相比，用户还有很多更重要的事情要做，因此，对这种迁移工作，他们通常是一拖再拖，如果程序已经正常运行，就更不会轻易地进行迁移。

还有一个进退两难的例子出现在 Linux 的内核从 2.4 升级到 2.6 一事上。这两个内核并不是完全兼容的，所以基于 Linux 2.4 开发的程序必须进行移植。而且这两个不兼容的内核也不可能同时运行。所以用户在 2.6 内核上只能选用新的应用程序，而 2.4 的内核用户也只能坚守老的应用程序了。所有使用分布式开发的应用程序都必须在一个版本内就升级到新内核，这事说来甚是不易。还好，Linux 的内核比较小，可以通过逐步分段的方式来完成迁移工作。至少有一些应用程序还是设法做到了这一点，同时支持这两个版本的内核。

软件熵^①

此外，导致软件出现变化的另一个主要因素就是软件开发自身的趋势。我在 NetBeans 团队的同事 Tim Boudreau 将这种情况称为“软件熵”，他用笑脸图片来描述这种情况。与前面所说的那种阿米巴变形虫模型相比，其角度更为全局化，可以说是自始至终地描述了在一个维护人员眼中一个软件项目是如何变化的。

每一个软件项目的第一个版本都很漂亮。新项目从零开始，所有的内容都是新开发的。因为全新开发，就意味着没有历史负担的问题。第一个版本的 bug 非常少，当然，程序员也尽力做到最好。这意味着，在开发人员的眼中，第一个版本可以算是完美：很漂亮、设计良好、架构优秀，就像图 4-4 中描述的一样。整个程序的组织方式像是一个漂亮的星状结构：有一个中心控制模块来提供所有的 API，而其他模块则通过这些 API 进行交互。所有的设计、依赖等一切的一切都非常漂亮、整洁。

当每一个版本发布以后，开始有人发现 bug，并公布出来，这些 bug 都需要被修复。这时第二个版本的起点就要比第一个版本高得多。第二个版本并非从零开始，而是建立在有问题的基础上。而且，对于大部分的软件项目来说，往往没有足够的资源和时间把事情做得非常完善。所以对新版本的调整和修改就没有第一个版本那样漂亮和整洁。就像图 4-5 显示的那样，开发人员走了一些捷径，比如使用了一些非正式的 API 或者其他的一些手段。

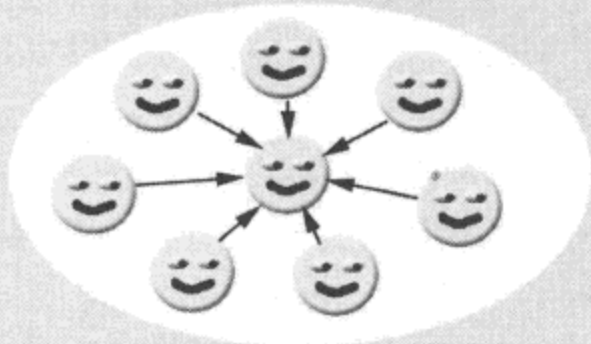


图 4-4 第一个版本往往非常漂亮

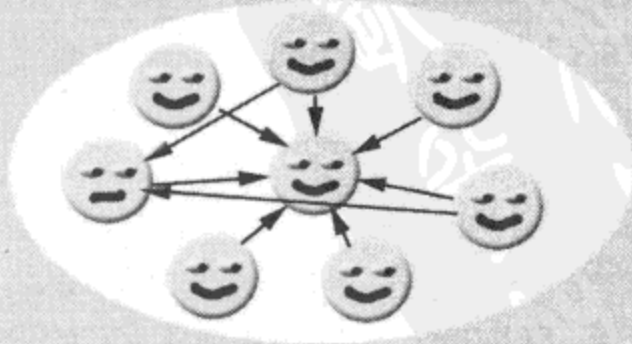


图 4-5 快乐总是短暂的

^① “熵” (entropy) 是德国物理学家克劳修斯 (Rudolf Clausius, 1822—1888) 在 1850 年创造的一个术语，后来该词被用到社会、信息等多种学科中，往往表示混乱的程度，即熵值越高，就越混乱，本书沿用这个意思。——译者注

这样做会在后续版本中不断地增加软件熵，参见图 4-6。代码变得更加难以维护。负责维护的开发人员开始天天抱怨，特别是那些代码原作者已经不在的情况下，抱怨就更加严重。

终于有一天，项目会处于一个特定的阶段，此时，开发人员只能说：“别去动这个软件了！”，如图 4-7 所示。代码犹如一团乱麻，任何改变都有可能让某些功能无法正常运行。问题越来越大，最后管理层得到的信息就是这个软件病入膏肓了，他们也许会决定重新开发一个软件。

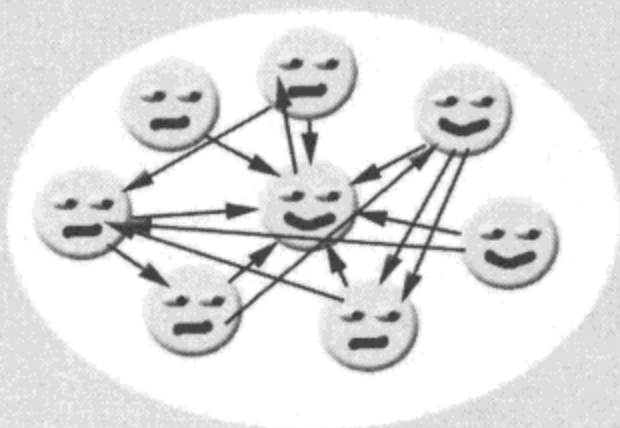


图 4-6 软件熵不断增加

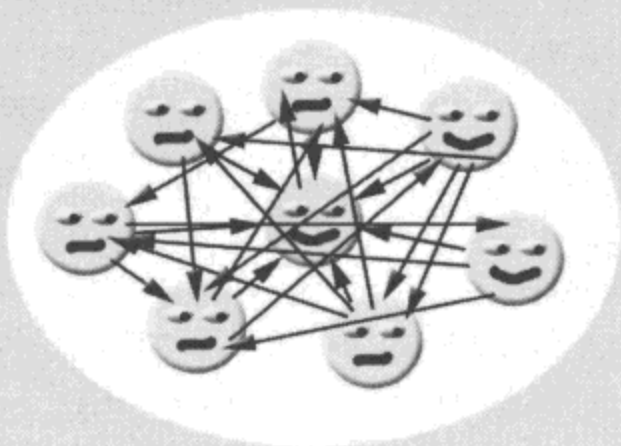


图 4-7 天啊，千万不要动它

很有可能，开发新的版本所用时间会比原计划的时间要多得多。而且功能也可能比老版本要少。但这个新版本将会非常漂亮和整洁，参见图 4-8。很明显，因为这个版本又是从零开始了。从设计角度来说，不会有什么问题。这个版本又像第一个版本一样的优雅、美丽。现在终于达到我们预计的目标了。

事实上，出现在第一个版本中的问题会同样出现在最新重写的这个版本中，如图 4-9 所示。代码会再一次地变成一团乱麻，需要有人去维护，会引入各种修改方法，所有出现在第一个版本的问题都会一一重现。直到有人痛下决心，想从根本上解决问题时，又会出现前面的那种推倒重写的事情。虽然重新开发新的版本的确付出了不少努力，但每一次我们都会重新回到起点，重犯之前的错误。这种努力就是白费。

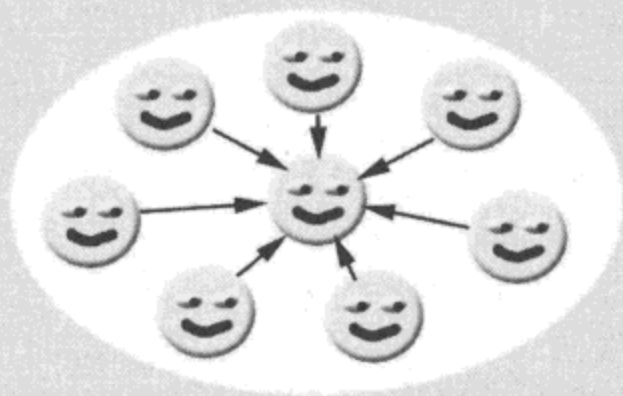


图 4-8 新版本总是那么漂亮

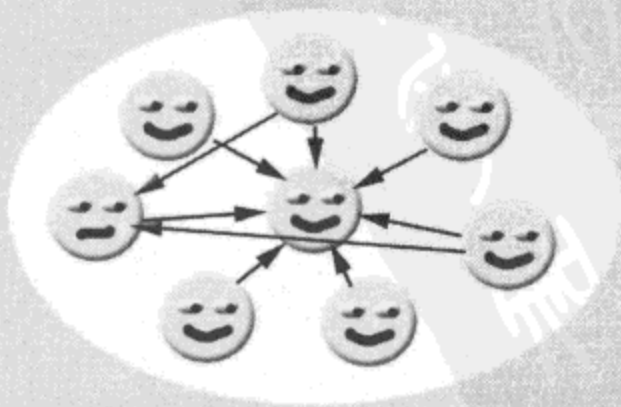


图 4-9 软件熵的回归

对于推倒重来的这种做法，我们往往是抱着会有更好结果的想法去为之，但最终却往往是同样的错误一犯再犯。所以没有任何理由支持我们这样从头开始来做一件事。如果想要一个更好的结果，就需要改变开发的方式。也许是时候改变一下编码的风格了。也许问题出在设计和测试中。只有改变了工作方式，才可能有更好的结果，也只有认识到这一点，才能指望推倒重来的做法最终会成功。但矛盾的是，如果真能够改变工作方式，也许就不需要通过推倒重来的方式对软件进行改善。如前文所说，除非是开发第一个版本，否则就不可能指望能够做到精巧和优雅。但本书 1.4 节中已经明确指出，在软件工程中，漂亮并不是衡量质量的第一标准。相比之下，我们更需要组件的可靠性和设计者的责任感。只有做到这些，才有可能让我们在做程序集成时能够“无绪”。

总而言之，还是要准备使用增量改进的方式！人们需要软件加以改进，但改进时引入的伤害也应该最小化，特别是要避免重新编写的那种大变化。如果因为 API 设计上的问题，使得无法增量改进，那么也许会有充足的理由进行一次重新编写，但这种大的变化应该限定于开发方式上的一些基础性变化。本书的大部分篇幅都会讨论用于 API 增量改进的设计实践。如果出现了很大的变化，我们会同时强调要为一个 API 提供多个大版本类库。只有这种方式才能保证 API 能够变得更好，而且使 API 用户的痛苦最小化。



第二部分 设计实战

前文中关于理论的话题已经说得太多了。现在毫不迟疑,让我来带领读者直接进入设计实战。

第一部分主要是对 API 设计的缘由和目标进行了说明。想来定有一些读者会觉得那部分内容很无聊,不仅没有什么新意,而且这些理论放之四海皆准,没有什么可以争议的东西。然而,在第一部分对 API 设计理论进行总体介绍还是有必要的,因为一旦大家对同一个词条有不同的理解时,就会产生许多误会,所以有必要进行说明。

前面讲述到的理论并不完美,但至少它们明确了 API 设计的目标,并给出了一种可以用来衡量 API 质量的合理方案。那么,接下来要做的事情就是将理论与实践相结合,把前文的理论基础应用到 Java 开发中去。

下载源代码

本书示例中所涉及的源代码都来自真实项目。所有给出的源代码都可以编译、修改并执行和调试。这些代码片段都是经过精心挑选的,每个代码片段都尽量做到可以独立运行,且易于理解。这样在阅读书中的这些例子代码时,读者即使没有通读上下文所有代码,也可清楚地知道每段代码片段的功能。总而言之,在学习 API 设计中,通过编写代码进行实战演练是非常重要的。如果读者有兴趣,可以从 <http://source.apidesign.org> 这个地址得到本书所有的源代码。

与第一部分相比,本部分讨论的话题比较有争议性。因为每位读者都有自己的经验和想法,对于谈论的具体问题,会因为各自环境、立场而有不同的思路,与本书给出的建议也可能不尽相同,所以读者在阅读本部分给出的一些建议时大可持保留态度,尽量做到独立思考。另外书中给出的一些建议可能看起来会比较怪异,还有一些方案也会比较复杂,请读者一定要根据自己的实际情况进行思考,并合理地应用这些建议,以免误用。文中所有的建议都来自 NetBeans 项目这 10 年来的经验总结,并不是随便写出来的。虽然书中给出的建议和方案并不会是最优的,但它们或多或少地解决了我们的实际问题。

在本书中一再提起 API 开发时如何进行交流的话题,这是因为如果使用共通的语言进行交流,就会达到非常好的效果,所以如果 API 设计人员使用大家普遍理解的结构,就能对项目有所帮助。说到交流,就必然涉及软件术语,提到软件术语,就不能不说设计模式,可以把它作为交流中的重要组成内容。“设计模式”一词已经为开发人员所熟知,市场上这方面的图书也是日益

增多，在那些开发培训课程中，设计模式也已经成为常见的教学内容。因此，把它们作为 API 设计人员间进行沟通的语言再合适不过了。

一个设计模式就是“针对一个软件设计问题的可重用方案”。它由四部分组成：模式名称、问题描述、解决方案以及处理效果。当把这些内容牢记在脑海中以后，只要说起这个设计模式的名称，所有的听众对此都会有相同的理解。给复杂的数据结构起一个简单的名字能简化交流。例如，一提到“单例”，你就会想起很多内容，如类之间的关系、对应的实例化管理和生命周期。听到某个词，就能对所描述的内容有一个大概的印象。一个简单的单词就能达到这样的效果，称得上“闻弦歌而知其雅意”。这就是为什么设计模式能简化对类库或者 API 架构的描述。

传统的设计模式主要是针对内部系统的开发。当然，它们对于 API 设计和分布式系统开发也同样有很大的指导意义。为了更好地设计 API，设计者们应该能够做到将设计模式的内容烂熟于心。在序言中，我已经多次说明设计一个宇宙要比建造一栋房子要复杂得多。因此，我们需要对现有设计模式的内容进行扩展。这样才能够借助设计模式来构造一个宇宙，而不是去建造一栋房子。

先来对书中提到的 API 设计模式做一番说明。从某种意义上来说，这些设计模式有助于简化 API 架构定义，不过除此以外，它们还有另一个特征：强调改进。除了前面所说的设计模式四元素——模式名称、问题说明、解决方案和处理效果以外，还包括“改进计划”，用来描述当 API 第一次发布以后，可能会出现哪些问题，后续需要如何加以改进。前文中，我反复地强调过“第一个版本远非完美”。所以更加需要一个改进计划，这样才能对 API 可能有所变化的各方面加以改进，与此同时，这样的改进还要保证对前一个版本的兼容。

几乎每一个 API 都不完美，需要在未来有所改进，因此要特别强调 API 的改进。如果一个设计模式能够为一个不完美的 API 留下修正的机会，就意味着这个设计模式是一个好的 API 设计模式。只有掌握了正确的设计模式才能在 21 世纪的软件开发中立足。现在让我们开始 API 设计之旅吧。



好的 API 有很多特征，本书会逐一讨论。最先开始讨论的内容却是一个老生常谈的建议。该建议已经广为人知，可以说每本设计模式的书都会提到它，但这条建议无论强调多少次都不为过，这就是：API 公开的内容越少越好。

有一些 API 设计者很有助人为乐的精神，所以他们编写的 API 中提供了大量的辅助方法和实用工具类。他们将所有的类都变成外部可访问的 public 级别。还有很多的类中还带有要么是 public 要么是 protected 声明的成员。这些设计者认为：终究会有人用得上这些功能，只要有人用，当然就要公开。对于这种极具贡献精神的利他主义，我也不知道说什么才好，但这些设计者带来的问题往往远远多于其解决的问题。大家都很清楚：向 API 中添加一个新方法是很容易的一件事，但与此相反，想从 API 中移除一个方法却非常困难。进一步说，一个 API 对外提供的功能越多，那么为保持向后兼容性所做的工作也就越多。也就是提供的功能越多，留给具体实现的空间也就越小。如果 API 公开的不必要的内容越多，那么它未来可改进的空间也就越少。

决定哪些功能应该成为 API 的一部分，哪些功能不应该成为 API 的一部分，这无疑是一个两难选择。在 NetBeans 项目的团队中，成员们使用一种基于用例的方式来解决该问题。只有一个有效的用例才能证明一个方法或者一个类应该成为 API 的一部分。如果找不到这样的有效用例，或者说所找到的用例不具有说服力，那么就不应该将这样一个功能作为 API 公开出去。因为这也意味着，该功能可能应该只作为一个内部特性存在，不应该，也不需要对外公开，在 Java 中如果对类、字段或者方法没有声明访问级别，那么就是默认的包访问级别。如果 API 的用户总是抱怨说某一个 API 的内容应该对外公开，此时才可以考虑是否存在这种真实的需求。但决不要听风就是雨，仓促将一个功能公开到 API 中并非好事，而是要先去验证这个请求中的用例是否有足够的说服力。

通过编码技巧提高代码的可读性

经验告诉我们：对于 API 设计者来说，其水平越差，他所编写的 API 越会公开大量不必要的内容。此类情况不胜枚举，这往往是由于设计者对用例的理解不足造成的。由于理解不足，一旦有人需要类库中的一部分功能，作者就将这些功能作为 API 的一部分公开出去了。这种设计方式没有任何规划，最终把相关的 API 给搞成一堆浆糊。用例不足造成处理方式简单化、粗

鲁化，进而把所有的事情搞砸。

对于一个没有多年维护 API 经历的设计者，限于其经验，他们经常不会意识到：他们所写的 API 可能会被他人误用。API 中公开的所有方法和类都可能被误用，换句话说，用户使用这些 API 的方式可能与设计者的初衷完全不同。我认识的 API 设计者，大都有这方面的体会。几乎所有的 API 设计者都有此共识：一个人设计 API 的时间越长，他设计的 API 公开的内容就会越少。

这里给出的第一条建议就是：在发布 API 的第一个版本之前，将不必要的内容从 API 中移走。这当然不是说要把所有看不顺眼的内容都移走。首先要知道，每一个额外的辅助类或者实用工具类，其背后往往隐含着很多潜在的需求，所以设计人员才会提供这些额外的内容以满足其需求或者方便开发人员使用。这些 API 的设计者认为他们提供的内容需要对外公开，应该与其他用户分享。我也曾经有过类似的想法。初始的想法不可谓不好，但一定要明白，公开一个 API 只是万里长征开始的第一步。每公开一个 API，意味着你对该 API 的用户做出了一个未来的兼容性承诺。事实上，当你迈出第一步的时候，要表示你已经选择了一个方向，接下来的工作就是要沿着这个方向一直走下去。如果 API 设计者感到确有必要创建这个 API，那么他走出的这一步也是无可厚非的。但从另一方面来讲，当我们只是想“四处走走”时，就应该知道怎么做才能避免就“方向”给出承诺。请记住，最简单、有效的解决方法就是，一切都以最终用户为中心，但要逐步来满足他们的需求。

5.1 方法优于字段

这里要说的第一个 API 设计技巧就是：不要把类中的字段直接对外公布，而应该放置在内部。最好让外部用户只能通过相应的 `getter/setter` 方法来访问字段，千万不要将字段直接公布出去。因为 `getter/setter` 方法是具体的操作过程，可以在这个操作过程中添加很多额外的处理机制，如计算、转换等。但访问一个字段就只是一个简单数据读与写操作。如果使用 `getter` 方法来访问一个字段，就很容易在相应的 `getter` 方法中进行数据懒加载、同步访问或者使用一个特定的算法来得到数据。而 `setter` 方法，则允许在调用时对参数值是否合法进行检查，或者是发送字段改变的消息来通知第三方。

Java 虚拟机规范也建议开发人员在访问数据时，优先使用方法而非直接访问字段。假设要将一个方法从子类移到其超类中，这种改变在二进制代码的层面是兼容的。先来看一下 `getPreferredSize(Dimension d)` 这个方法，它原先是 `javax.swing.JComponent` 这个类中的一个方法，在 JDK1.2 版本中，它被移到了 `java.awt.Component` 类中了，但使用该方法的代码无须更改，可以正常运行。这是因为 `JComponent` 是 `Component` 的子类，同时 `JComponent` 中原来声明的 `prefSize` 字段没有直接暴露出来，而是通过方法来访问的，所以在移动该方法的同时仍然可以保持兼容。相反，如果直接把 `prefSize` 字段公开，那么就不可能做到这样的二进制兼容了。一旦一个类中声明了一个字段，这个字段就再也不能从这个类中移除，否则就破坏了二进制兼容性，这正是人们建议把字段变成私有的原因之一。

运行速度比你想的快

我曾听一个.NET的架构师说过：也许应该牺牲一定的封装来提高程序的运行性能。他说有一个程序，其中有一个类在开始设计的时候是使用getter/setter方法来访问字段的，随后使用了一个只有字段信息的结构体来代替这个类，结果整个程序的性能提升了很多。他说得对，也许.Net需要这样的处理方式，但我坚信Java完全不需要这种小把戏。

回溯到1998年的秋天，我参加了Sun公司在柏林主办的一个研讨会。在会上，JavaSoft公司的老总语惊四座，他说，解释执行可以比编译执行还要快。那时，我还没有成为Sun公司的一员，但我也清楚地知道Java慢得吓人。我同其他听众一样，听了这话以后，捧腹大笑。但现在回头再细想一下他的言论，却有新的感触。自Sun发布了HotSpot虚拟机以后，Java虚拟机的性能已经大幅提高。HotSpot虚拟机首先解释执行类文件中的内容，然后监控程序的执行情况。待找到性能瓶颈点后，再将这部分内容转成本地代码执行。与静态编译不同的是，它不进行预连接，也不根据通用的环境进行编译，而是由HotSpot编译器根据本地操作系统、硬件等环境在程序执行的时候进行优化处理。

这种处理方式带来的一个好处就是HotSpot编译器可以将虚方法进行内联。对于所有在C中尝试以模拟的方式来使用虚方法的程序员^①，他们都很清楚这样做是需要去访问内存和进行指令跳转才能实现的。但HotSpot编译器可以做得更好，因为它很清楚地知道代码连接的相关信息，比如那些没有被覆盖的虚方法，或者被两个类覆盖了虚方法。如果一个虚方法没有被覆盖，就可以对实际方法中的内容进行内联，所有对该虚方法的调用就变成了直接执行指令，与传统编译生成代码相比，就少了定位、跳转等指令环节，执行速度自然就更快了。如果一个虚方法被覆盖了，而且执行到一定的次数以后，就把对虚方法的调用转成实际方法的入口，而无须先定位再跳转。相对于只能基于虚方法表的静态编译，或者是在C中来模拟虚方法的处理方式，HotSpot在程序执行时所编译出来的代码，由于参考了具体的运行环境，其运行性能要比静态编码高出很多。

尽管解释执行的代码确实不如编译代码的性能高，但只要我们把当时柏林会议上的那句话略作改动，变成“动态编译的代码要比静态编译的代码执行快”，那么这个论断就成立了。可以这样说，在HotSpot编译器的支持下，这个论断已经逐渐成为事实。

请牢记，HotSpot虚拟机不仅可以将普通方法进行动态内联，还可以将调用频度极高的虚方法进行动态内联。所以在频繁调用getter/setter方法的情况下，HotSpot虚拟机会自动进行动态内联，根本不存在前面.Net那种性能降低的情况。因此在Java中，也就不需要牺牲封装特性以换取性能的提高。

除了以static和final声明的基本类型、String常量、枚举和不变对象以外，API的所有字段信息都不应该对外公开。使用方法来访问字段要好于直接访问字段。如果你在编写API的时候遵守了这一建议，那么可以在这个API的下一个版本中对方法的具体实现进行很多改动，如添加数据的验证和校验、覆盖一个方法等，还可以做一些大的改变，如将方法移动到超类中、对模型进行同步处理。但如果直接用字段，那么以上改变就行不通了。

^① 这里是指使用函数指针技术可以使得C语言动态使用相应的方法，与面向对象语言中的虚方法相类似。

5.2 工厂方法优于构造函数

使用工厂方法而不是直接使用构造函数来创建一个对象，那么对 API 的改进可谓好处多多。如果在 API 中公开一个构造函数，那么就表示通过该方式创建的对象一定是类的实例，而不能是该类的子类实例。而且每次调用构造函数一定会创建一个全新的对象。

工厂方法则为开发人员带来了很高的灵活性，它通常是一个静态方法，这个工厂方法的参数与构造函数的参数相同，返回值与构造函数创建的对象也是一致的^①。使用工厂方法的第一个优点就在于工厂方法返回值并不一定是声明类型的实例，可以是它的子类实例。使用工厂方法可以更好地体现面向对象语言中多态优势，代码也会显得更加简洁。另外，每次返回的对象也并不一定都是新创建的对象，完全可以将其缓存。相比之下，每次调用构造函数都会创建一个实例，而一个工厂方法则可以缓存以前创建的对象，并重用它们，从而节省了内存。第三个优势则在于对同步的控制，在工厂方法中可以将创建对象前后的相应代码进行统一处理。构造函数对于这种情况就无能为力了。

因为 JDK 5 引入了泛型，所以我们对 NetBeans 的 API 进行了重写。在此过程中我们发现了工厂方法的另一个益处，那就是工厂方法支持参数化的返回类型，但构造函数是做不到这一点的。参数化类型这个词，猛一听起来不甚好懂，来看下文的这个例子吧。

```
public final class Template extends Object {
    private final Class type;

    public Template(Class type) { this.type = type; }
    public Class getType() { return type; }

    public Template() { this(Object.class); }
}
```

当时要将 NetBeans 源代码迁移到 JDK 5，很自然就对 Template 的类构造函数的参数进行泛化。除了最后一个构造函数以外，其他的代码一切 OK。

```
public final class Template<T> extends Object {
    private final Class<T> type;

    public Template(Class<T> type) { this.type = type; }
    public Class<T> getType() { return type; }

    // now what!?
    public Template() { this(Object.class); }
}
```

最后一个构造函数的本意是创建一个 Template<Object>的对象实例，但在 Java 5 中，这种对象是创建不出来的。它不支持这种灵活的表达方式。其实从另一个角度来说，也可以看作是我们设计上的一个错误。当初我们设计这个类的时候，需要关注的事情很多，至于如何在 Java 5 中把泛型应用到这个类中，根本就不是当时设计所考虑的重点。只不过后来再讨论这个设计问题为时已晚。

^① 这里的一致性是指工厂方法返回的对象是构造函数所表示类的实例，或者是其子类的实例。——译者注

我们所能想到的唯一方案就是先通过 `deprecated` 这个关键字建议用户不要使用这个构造函数，并使用 `SuppressWarnings`^① 避免编译警告，建议用户使用另外一个带类参数的构造函数。如果在一开始设计这个类的时候，就采纳本书的建议，使用一个工厂方法，而不是直接公开这个构造函数，解决这个问题就会容易得多。下文的代码在语法上就不存在问题了。

```
public final class Template<T> extends Object {
    private final Class<T> type;

    public Template(Class<T> type) { this.type = type; }
    public Class<T> getType() { return type; }

    @Deprecated
    @SuppressWarnings("unchecked")
    public Template() { this((Class<T>)Object.class); }

    public static Template<Object> create() {
        return new Template<Object>(Object.class);
    }
}
```

上述例子说明相对于构造函数，工厂方法在泛型方面有更高的灵活性。根本原因在于，一个方法并不会限定返回值实例的类型，可以是类，也可以是接口，但构造函数的返回值却一定是指定类型的实例。这正是工厂方法对于构造函数的另一个优势。

5.3 让所有内容都不可更改

通常情况下，人们在设计一个类的时候，如果不考虑让其拥有子类，就应该让这个类不能被继承。要知道一个 API 一旦发布以后，会有各种使用方式，随之而来的就是大量无法预计的事情，其后果可能会非常危险。下文这个例子就说明了用户使用 API 的多样性：

```
public class Hello {
    public void hello() { System.out.println ("Hello"); }
}
```

直接公开这个可被继承的 `Hello` 类，会有很多问题，比如说下文的代码，是直接从外部调用 `Hello` 的 `hello` 方法的：

```
public static void sayHello() {
    Hello hello = new Hello();
    hello.hello();
}
```

同时，开发人员也可以编写一个继承 `Hello` 的类 `MyHello`，重载 `hello` 方法，代码如下。

```
private static class MyHello extends Hello {
    @Override
    public void hello() { System.out.println ("Hi"); }
}
```

① 后续实例代码中 `@SuppressWarnings` 批注由 Java 自身提供，告诉编译器对被批注的代码元素内部的特定警告保持沉默。相应地，`@SuppressWarnings("unchecked")` 告诉编译器忽略被批注代码元素内部未经检查的类型转换警告。

——译者注

还可以写一个 SuperHello 类来继承 Hello, 不仅重载 hello 方法, 还调用 Hello 父类的 hello 方法, 如下所示。

```
private static class SuperHello extends Hello {
    @Override
    public void hello() {
        super.hello();
        System.out.println("Hello once again");
    }
}
```

用户可以用各种方式来做类似的事情, 可谓花样百出, 远超设计者所想。但解决此类问题的方案却非常简单: 将 Hello 这个类变成 final, 不可继承, 一切问题迎刃而解。

在编写 API 的时候, 如果不想让用户继承某个类, 那么就坚决不允许他们继承这个类。如果不这样做的话, 几乎可以肯定, 总有一些用户会继承 API 中的某些类。就像 Hello 这个例子, 当你公开这个类以后, 至少有 3 种方式来使用这个类, 这样就会要求你对这三种方式都提供支持。考虑到未来对该类的改进, 最好还是让该类无法被继承, 减少外部的使用方式。第 8 章将详细讲述相关内容。

再强调一次, 最简单的解决方案就是让你的类变成 final, 即不可继承。当然还有一些方案, 如不公开构造函数, 转而提供工厂方法, 我们强烈推荐这种方式 (在 5.2 节中对此有精彩论述), 或者把大部分的方法变成 final 或者 private, 避免被覆盖。

以上所说的继承方式只适用于类。从虚拟机的角度来说, 如果是接口, 就无法避免外部来实现该接口了。只能在 Javadoc 中添加相应的说明来要求用户不要实现该接口。虽然 Javadoc 是一种不错的沟通方式, 但只要无法通过强制的手段来约束外部开发人员, 使得他们无法继承指定类或接口, 那么在 Javadoc 中给出再多的建议也难以避免这种情况的发生。如果没有更好的方式来避免外部继承该类的话, 最好还是使用 final 关键字来约束开发人员, 让他们无法继承指定的类。

5.4 避免滥用 setter 方法

在开发了 NetBeans 多年后, 我们才得到一条宝贵的教训, 那就是: 如无必要, 决不要在正式的 API 中声明 setter 方法。所谓“正式的 API”, 则是指提供给外部开发人员来实现的接口。虽然通常情况下, 并不需要在此类接口提供 setter 方法, 但如果确有其必要性, 最好在该接口的基类中声明, 避免直接在接口中声明。

现在仔细看一下 javax.swing.Action 这个类, 可以清楚地看出, 它违反了这个规则。该类提供了一个 `setEnabled(boolean)` 方法, 这个方法本不应该放入该接口中的。它完全可以作为一个 `protected` 方法放在 AbstractAction 这个基类中, 这样即可以方便实现 Action 接口, 也不会带来额外的问题。要知道方法“`boolean isEnabled()`”才是真正的 API, 每一处都会调用它来判断当前 Action 是否可用。看完这个类, 不禁要反问一句, `setEnabled(boolean)` 这个方法到底写给谁用呢? 也许只有编写这个方法的作者用得上, 对于其他人, 这只是一个具体的实现。如果你写了一

个 Action 类，允许其他人通过相应的 Listener 进行监听，倒是可以写上一个 `protected` 级别的 `setEnabled(boolean)` 方法，该方法可以用来手动设置是否允许将事件通知给 Listener。通常来说，`setEnabled(boolean)` 应该作为一个不可覆盖的 `protected` 方法放置在 `AbstractAction` 这样的基类中。Action 类中的 `setEnabled(boolean)` 方法设计上出现了问题，有如下几个原因。

- 公开这个方法表示所有开发人员都可以在外部调用这个方法，事实上没有这个必要。
- 因为这个 Action API 设计时对上下文环境考虑不足，即使多了这个方法，对于那些上下文敏感的 Action 类来说，也没有任何意义。我们一直在尽力找出一些场景，能够更合理地使用 Action，还算有效。但可以肯定的是，原先设计 Action 这个类的人，完全没有考虑到模块化设计方面的内容。
- 如果一个 Action 对象始终可用，那么调用 `setEnabled` 这个方法就没有任何效果。因为这样的 Action 对象，只需要通过 `isEnabled` 方法返回 `true` 值就可以了，完全不用考虑其他事项。
- 如果一个 Action 通过方法 `isEnabled` 来判断当前操作是否可用，那么 `setEnabled` 这个方法是没有意义的。如果要判断相应的 Action 状态，使用 `setter` 方法决不是一个好的方案。如果想用一种推^①的方式来达到某个目标，首先要明白一点：不可能掌控所有的全局信息。所以，应该让系统在需要的时候来对状态发出查询请求，就像在合适的时候调用 `isEnabled` 方法来判断某个操作是否可用，而不像 `setEnabled` 方法一样由外部来控制其状态。

上面对 `setEnabled` 方法给出了这么多意见和建议，但并不是说 API 中的每一个 `setter` 方法都是错误的。有时候，一些 `setter` 方法还是有用的，比如说如果要使用 Spring^② 的注入功能，少了 `setter` 方法就行不通了。只不过，大部分情况下，这些 `setter` 方法都是多余的。至于那些过分使用 `setter` 方法的例子简直是俯拾皆是，这是因为设计者在设计这个 API 的时候，希望给它尽可能多的功能，但往往适得其反。另外加一句，针对上面的那个 Action 例子，它的语义也比较难理解，什么时候应该调用这个方法也不是很清楚。针对此类问题，我所给出的建议很简单：小心 API 中的每一个不必要的 `setter` 方法。

5.5 尽可能通过友元的方式来公开功能

说到如何避免在一个 API 中公开太多的内容，还有一个比较有效的方式就是通过友元的方式来访问 API 的功能。比如说，创建一个对象或者调用一个方法，这些功能都只有满足友元条件的代码才能访问。

在 Java 中，所谓的友元也就是默认的 `package` 访问方式，即只允许同一个包内的代码之间进行互访。如果有一些功能，只想那些处于同一个包中的代码进行互相调用，那么就使用默认的

① 这里所谓的推方式其实是指某些代码主动地去调用 `setEnabled` 方法，与之对应的是拉，如菜单或者工具条按钮在显示的时候，去调用 `isEnabled` 方法来判断是否可用。——译者注

② 虽然 Spring 框架很流行但它并不是某一个规范的开源实现。它主要由 Spring Source 公司开发和控制。Spring 框架结构是基于依赖注入 (Dependency Injection, DI) 的设计模式。它的官方网站是 <http://www.springframework.org/>。

——译者注

package 友元级访问。比如说将构造函数、字段以及方法都声明为 package，这种友元的访问方式可以避免外部类对包内功能的调用。

不要在 API 中公开那些外部不应该调用的方法

在浏览 Javadoc 中各方法的说明时，可能会看到这样的文字“该方法不应该被调用，因为这个方法是具体实现中的内容”，或者“该方法只为内部使用，请勿在外部调用”。我曾多次看到这种说明，认为这是最坏的 API 设计，完全是一种反模式。不仅仅是因为这些方法让人觉得整个 API 非常不专业，更重要的是它分散了读者的注意力，把本应关注 API 的精力，放在那些不必要的实现细节上了。

此类方法通常意味着，位于各个分散包中的部分功能实现需要被赋予访问特权，即友元的权力，以访问 API 包提供的功能^①。这也正是友元访问器模式所要解决的问题。

有些开发人员认为友元的访问方式有些过于复杂了，不想在自己编写的 API 中引入这些复杂的代码。这种说法与事实不符。虽然在 Java 中使用友元方式的确会使编码有一点点麻烦，但是那只是具体实现的细节，对 API 的外部用户并不可见。为了让用户更加简单明了地用好你的 API，就不要在你的 API 中公开太多的方法，不要给用户犯错的机会。

因此，请不要在 API 中包含那些与实现相关的细节性内容，特别是那些不应该被外部调用的内容。

有时候需要在更大的范围使用一些功能，此时必须对一些友元访问的内容进行扩展。例如，定义了两个包，一个包用来定义纯粹对外的 API，而另外一个包则放置具体实现的内容。如果采用这种开发方法，下文给出的这些技巧，对于需要对外提供功能扩展的要求来说非常有效。以 Item 类为例，如下所示：

```
public final class Item {
    private int value;
    private ChangeListener listener;

    static {
        Accessor.setDefault(new AccessorImpl());
    }

    /** 只有友类才能创建对象实例。*/
    Item() {
    }

    /** 任何人都可以修改 Item 对象中的值。
     */
    public void setValue(int newValue) {
        value = newValue;
        ChangeListener l = listener;
    }
}
```

① 此外“API 包”即包含功能抽象定义的包，“各个分散包”可理解为与 API 对应的、包含具体实现细节的包。

```

        if (l != null) {
            l.stateChanged(new ChangeEvent(this));
        }
    }

    /** 任何人都可以拿到 Item 对象中的值。
     */
    public int getValue() {
        return value;
    }

    /** 只有友元类才能对值改变进行监听。
     */
    void addChangeListener(ChangeListener l) {
        assert listener == null;
        listener = l;
    }
}

```

这个类是作为 API 的一部分对外公开的，但外部不可能创建该对象，更不可能通过 `ChangeListener` 对它进行监听，只有那些与 `Accessor` 在同一个 API 包中的类可以做到。在这种情况下，可以通过在一个非 API 包中定义一个 `Accessor` 类达到相应的目的。

```

public abstract class Accessor {
    private static volatile Accessor DEFAULT;
    public static Accessor getDefault() {
        Accessor a = DEFAULT;
        if (a != null) {
            return a;
        }

        try {
            Class.forName(
                Item.class.getName(), true, Item.class.getClassLoader()
            );
        } catch (Exception ex) {
            ex.printStackTrace();
        }

        return DEFAULT;
    }

    public static void setDefault(Accessor accessor) {
        if (DEFAULT != null) {
            throw new IllegalStateException();
        }
        DEFAULT = accessor;
    }

    public Accessor() {

```

```

    }

    protected abstract Item newItem();
    protected abstract void addChangeListener(Item item, ChangeListener l);
}

```

Accessor 类提供了很多抽象方法来调用 Item 类所有的友元构造函数和方法, 该类相当于一个单例, 通过 static 字段获取 Accessor 的具体实例。该技巧就是将 Accessor 公开, 同时在 API 包中使用了一个非公开的 Accessor 的子类, 通过 Accessor 的单例来对外公开 Item 的功能:

```

final class AccessorImpl extends Accessor {
    protected Item newItem() {
        return new Item();
    }

    protected void addChangeListener(Item item, ChangeListener l) {
        item.addChangeListener(l);
    }
}

```

这样一旦有人访问了 Item 这个类, 就会调用其 static 初始化代码, 为 Accessor 设置一个唯一的默认实例, 该实例就可以访问 api.Item 这个类。static 的初始化代码如下所示:

```

static {
    Accessor.setDefault(new AccessorImpl());
}

```

按照上述方式处理以后, 那个放置实现代码的包中的类就可以通过 Accessor 这个类调用原先被隐藏起来的功能。

```

Item item = Accessor.getDefault().newItem();
assertNotNull("Some item is really created", item);

Accessor.getDefault().addChangeListener(item, this);

```

使用运行期容器来保障代码安全

在 NetBeans 平台中, 有一个运行期容器, 称为 NetBeans Runtime Container, 它能够在 JVM 的基础上进一步保障代码安全。该容器在类加载器这一级添加了额外的安全机制, 可以根据相应的配置来禁止一些功能操作, 具体的做法也很简单, 只要在相应 Jar 包的 manifest.mf 文件中声明 `OpenIDE-Modules-Public-Packages:api.**` 即可。这样运行期容器就可以保证其他模块只能访问 api 包中的所有类。这样就不需要采用上面的方案来保护 Accessor 类中的方法了, 因为运行期容器只允许与 impl.Accessor 同一模块中的类访问 Accessor 类。这样在类加载器这一级上就禁止了外部模块对其的访问。

Java 也许会在未来的版本中添加一种新的访问级别^①, 在编译的时候就加入相关控制内容,

① 在 Java 7 中引入了模块, 可能会逐渐取代一直使用的类路径方式, 它是一个新的 JSR 规范, 编号为 294, 可以通过 <http://jcp.org/en/jsr/detail?id=294> 来了解更多的信息。它与 OSGi 比较相似, 但它的依赖是在 module-info.java 中定义的, 会被编译成类文件。而 OSGi 的依赖则是在 MANIFEST.MF 文件中定义的。——译者注

只允许所指定的多个包间的代码进行相互访问。但为什么不在当前发布的 JDK 中加入这种功能，而是要等后续的版本呢？因为新的访问级别也许需要对虚拟机加以改进，这样大的改动也许会使一些老的版本，如 Java 6，就无法运行这些代码。新的访问级别有可能会取代原先内置的构造函数，同时还要保证在任何版本的 JVM 上都能达到与原先几乎完全相同的效果。请注意，这里我用的词是“几乎”，是因为有一类功能可能无法直接使用这些方式，即新的访问级别会允许用户从一个友元的包里来继承 API 包中的类，即使这个类没有对外公开，也能做到这一点。其实这种限制很容易消除：并不是一定要直接继承，基于代理模式也是可以做到的，在第 10 章中对此有详细的说明。

我总是把这种 API 设计模式比作电话接口 (teleinterface)，如图 5-1 所示。其实这种方式可以比作科幻小说中的宇宙飞船，当它在多维空间航行时，在超空间^①中穿梭，从一个入口进入另外一个入口，但这在常规空间是观测不到的。电话接口模式也是这样，让明确知道彼此的双方进行交流，这样其他人无法观测到他们之间的互动，尽管知道这的确存在。

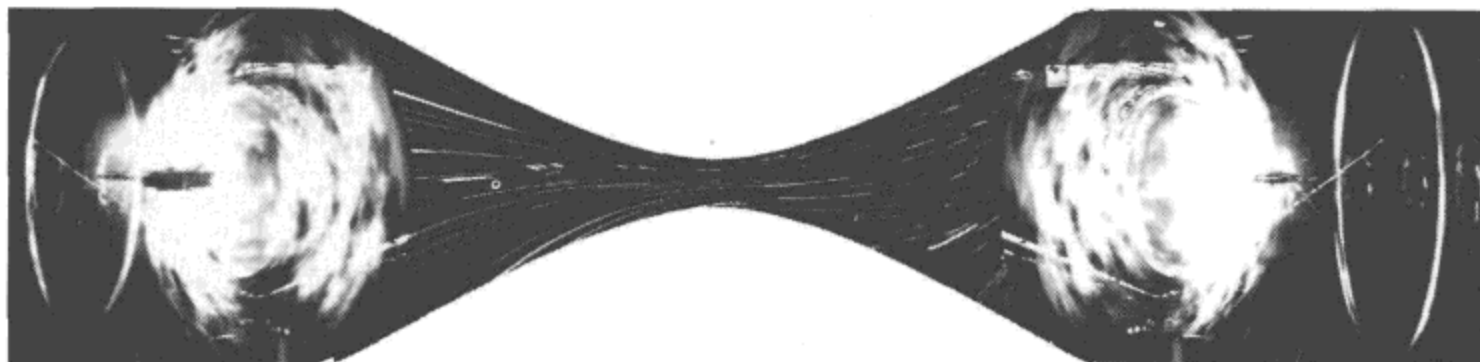


图 5-1 电话接口

利用友元访问级别者来设计 API 是一种典型的设计模式。当调用与被调用的两方需要交流的时候，任何一段外部的编码都不可能监测他们之间的互动。他们的交流是在“多维空间”进行的。对于设计模式，这是一种非常有用的设计元模式，接下来本书还会引入多种此类的设计模式。

5.6 赋予对象创建者更多权利

对于一个公开的对象，用户也不见得会用到所有的功能。要知道，“代码间也不是完全平等的”。对于面向对象语言来说，这是与生俱来的特性。因此无论 C++、Java 还是别的面向对象语言都支持访问级别，来控制各种功能的公开程度。Java 支持 public、protected、package 以及 private 四种，分别对应着全部对外公开访问、允许子类访问、包内访问，及只允许同一个类内部访问，可见 Java 对功能和数据访问的控制具有很高的细粒度。对于一些开发人员，特别是从 Smalltalk 转向 Java 的开发人员来说，他们已经习惯了 Smalltalk 中全部内容都是公开的，所以会觉得这种细粒度的控制过于复杂而且不太必要。但从另一个角度来看，即使做了这么细粒度的控制，很多

^① 超空间是科幻小说中一个假想的时空，在其中航行速度可能超过光速。——译者注

时候仍然不够用。比如说我们只想让两个指定包之间访问相应的功能，而其他包则不能访问，也即“允许友元代码进行访问”^①。这种需求非常通用，它意味着有些代码要比其他代码有更多的特权，能够访问更多的功能。

如果一个人没有 Unix 的背景知识，也没有操作过那些多用户共用的大型电脑，那么当他得知很多操作只有系统管理员才有权限执行时，一定会大吃一惊。像查询数据库，或者在 \$HOME/public_html 目录下面发布一个网页，这类常用操作无关紧要，谁都可以执行。但配置 Apache 服务器的缓存策略，或者调整数据库 Schema 这类操作，通常只允许少数用户执行该操作。而且在执行此类操作时，用户还必须先证明自己有相应的权限，也就是要验证其身份。每个操作人员在进行操作时，都要证实其身份的正确性。当然，如果没有通过相应的验证，就不能进行操作。典型的验证方式是使用账号和密码成对匹配的方式，再结合权限进行控制。

面向对象语言与上述所说的系统也有一定的相似性。运行时，对象在系统的各个组成部分之间进行传递，在此过程中，要保证对象的不变性和其内存状态的一致。如果说让系统来管理所有代码针对每个对象实例的修改和配置，那么显然不是一个明智的方案。事实上，如前文所说，还是通过访问级别来进行处理比较好。如果不想让一些功能为外部调用，那么就不要再将它声明为 public 或者 protected。这样，对于自己的代码，就像管理员一样可以调用所有功能，而其他代码就只能作为普通用户访问公开的功能。对于一个内部系统而言，这样做行之有效。但对于 API 的设计来说，还是远远不够的，因为不应该将对外的 API 变成一个像系统管理员之类的角色。要知道，你是想把你的 API 作为类库，共同为管理员和普通用户提供服务。现在来看一下要做些什么才能达到该目标呢。

首先看一下，哪些地方可能会出现这个问题呢？通常在 API 设计中，如果想将创建对象实例的操作进行分离的话，常见的方式是提供一个对外接口或者类，同时再提供一个继承该接口或者类的基类。该基类可以提供一些额外的控制，特别是一些创建操作。API 中的所有其他方法则用来实现接口或者类中定义的一些基本功能，这样第三方实现者就可以更容易地实现该类，或者是通过该类来访问一些 API 中提供的功能，这些功能对于一般的外部用户，会因为访问级别的限制而无法调用。javax.swing.text.Document 及其子类 javax.swing.text.AbstractDocument 可以看作是这种设计思想的体现。这样的设计多少有点效果，因为继承 AbstractDocument 来实现 Document 接口要比直接实现 Document 简单很多。而且 AbstractDocument 类中还提供了 writeLock 和 writeUnlock 此类 protected 方法。这样的设计比较合理，可惜只能通过继承的方式来使用。所以要针对这种情况，查找那些符合以下规律的代码：有些方法只能由创建该对象的代码来调用，如果发现某个类中存在这样的情况，那就表示该类需要这样的设计方式，以便在未来对其进行改进。下面的代码实现了 javax.swing.text.AbstractDocument 这个抽象类，证明基类可以帮助开发人员更容易实现一个接口。

```
public class MyDocument extends AbstractDocument {  
    public MyDocument() {super(new StringContent());}
```

^① 这种友元类似于给 Java 的 package 也添加了 public/private/package 的访问控制能力。——译者注

```

    final void writeLockAccess() {
        writeLock();
    }
}

```

但上面的示例代码也存在一个问题，MyDocument 类通过 writeLockAccess 方法等于提供了两个 writeLock 方法，这种做法并不可取。一个设计得更好的 API，可以避免这种重复的行为。不过这种问题只与代码风格有关，无关大局。

对于 AbstractDocument 来说，最严重的问题在于，它还含有不少对于“未授权的”API 用户而言非常有用的方法。例如，可以公开获取的 readLock、readUnlock、render（这个方法为什么要传入一个不必要的 Runnable 呢？）、getListeners（为什么不查查看自己是不是第一个 listener？如果不是，何不重新安排所有 listener，好让自己也成为第一个 listener？），还有像 replace 这个方法本就应该在类中，但很晚才提供这个方法，所以不适合原先的 Document 接口。简而言之，AbstractDocument 这个类中有很多方法，本应该只给 java.swing.text 包中的其他类来使用，但用户对这些方法也很感兴趣，想用下，于是用户就将 Document 接口的实例强制转型成 AbstractDocument，这样就可以直接调用这些方法了。这样做了以后，安全性就荡然无存了。如果调用 AbstractDocument 上的“特权”方法是件轻而易举的事情，如同那管理员可以轻易登录服务器一样，那么，互联网上的所有服务器肯定早就被“黑”掉了。正是由于这个原因，这种带子类的 API 接口方式并没有解决“特权”的问题。

NetBeans 提供的文件系统 API

你也许会奇怪，我凭什么敢承认自己对 Swing 中有关 Document API 的问题有如此深入的了解。坦诚地说，我并不是那么了解！我最了解的内容是 NetBeans 中文件系统的 API，因为在设计它时我犯了与 Document API 同样的错误。随后我检查 JDK 的源代码，希望可以找到类似的情况，发觉 AbstractDocument 正是一个这样绝佳的反例。

NetBeans 的文件系统 API 引入了虚拟文件系统的概念。开发人员可以基于 java.io.File 来构建这样一套文件系统。同样可以基于 HTTP 或者 FTP 资源来构建，还可以利用内存来存放虚拟的文件。但不管哪种实现，最基础的 API 接口都是 org.openide.filesystems.FileSystem，对于普通的开发人员来说，这是对外公开的 API。当然真正要具体地去实现这样一套文件系统并不容易，因此我们还提供了 org.openide.filesystems.AbstractFileSystem 作为基类来帮助开发人员进行具体实现。通常情况下，有一些方法没有放到 FileSystem 这个接口中。另外，系统中有一个使用 java.io.File 来处理本地文件的 LocalFileSystem 类，由于它继承自 AbstractFileSystem，所以是可以访问那些附加的方法。

在设计 API 时，我希望 API 的用户只使用 FileSystem 接口，但有一天，我发现他们并没有按照我的思路来。因为 NetBeans IDE 几乎是只使用了 java.io.File，因此大部分情况下可以安全地将 FileSystem 实例转成 LocalFileSystem，进行强制转换后，就可以随意地调用很多方法。这些方法并没有公开在 FileSystem 接口中。要知道，这些方法只是为了那些授权用户设计的。那时，由于目光短浅，我完全没有考虑到其他人也会调用这些方法。但 API 用户的创意让我吃惊，特别是他们只顾埋头苦干，只想尽快地完成手上工作时，就会完全忽视这一切。

我们必须避免用户调用这些方法。第一步，我们在二进制代码上打了少量补丁。该补丁将这些方法转成了 `protected` 级别，这样可以保持其二进制的兼容性。随后，则不再使用 `LocalFileSystem` 来处理本地文件。取而代之的则是 `AbstractFileSystem` 这个类，该类并不在任何 API 中。用户无法将 `FileSystem` 强行转换成 `AbstractFileSystem`，因为 `AbstractFileSystem` 这个类没有包含任何有价值的功能。

在 Java 中，定制对象的常用方法是将它们转换为 `JavaBean`，然后使用 `getter` 方法和 `setter` 方法。但对象的创建者是不能够通过这种方式处理安全和优先级问题的。但只需要简单的一步，就可以将这些基于 `setter` 方法进行赋值的方式变成一个可以支持优先级处理的方案。要做的事情很简单，就是添加一个 `public void setReadOnly()` 方法，同时修改所有的 `setter` 方法，使得在 `setReadOnly` 调用之前的赋值有效。反之，就可以抛出一个异常，比如 `PropertyVetoException` 或者 `IllegalStateException`。`java.security.PermissionCollection` 这个类就体现了这种设计模式，它允许在创建这个类的实例时添加相关的 `Permission`，然后通过 `setReadOnly` 将其变成不可更改的对象。其他代码即使拿到这个对象，也只能进行读操作，不具有进行修改操作的权利。事实上，除了创建这个对象的代码以外，其他代码对该对象的操作都变成了普通权限。这种方式非常不错，唯一被垢病的地方，可能就是它不够优雅吧。当然如同在 5.4 节所说，软件设计并不全然是漂亮的。但这种解决方案还是有一些缺点的。首先，一个 API 的方案关乎两类群体。无论超级用户还是普通用户都会看到相同的 API 类，但相同的东西、相同的操作却带来不一样的后果，用户肯定会感到困扰。其次，这种方式引入了基于状态的 API，也就是说调用相同的方法，有时候是有效的，有时候则会抛出一个异常，具体的原因则取决于当前的对象是处于超级用户模式还是普通用户模式。这使得该对象语义的复杂度超出了其应有的程度。最后一点也很重要，就是当一个对象从超级用户模式切换到普通用户模式以后，就不可逆了，只能进行一次此类操作。但在特殊情况下，还是希望该操作没有该限制。综上所述，如果要进行权限控制，还是考虑一下其他的可行方案。

在 Java 和其他面向对象语言中，每一个类都至少有一个成员，可以被构造对象实例的构造函数调用。这也方便我们用一种更直接的方式来支持超级用户模式：当构造一个对象实例时，所有与超级用户相关的内容都必须传入。正如同在 5.1 节中所说，使用一个工厂方法会更好解决一些问题。与前面那些用 `setter` 方法传递数据的解决方案相比，这种方案解决了语义上的疑惑，因为这种方案不会在超级用户和普通用户两种状态间进行多次切换。在超级用户模式下，对象被创建，所有的数据都会被注入到这个对象中，然后就不会再次进入超级用户模式。同时，这种方案也解决了 API 混合的部分问题。当 API 的用户引用一个对象的实例时，通常无须关注该对象的类构造函数，而且在常用的 IDE 环境中，构造函数也不会出现在代码提示中。要知道，工厂和构造函数的概念也非常易于理解。该方案看起来比较简单和自然，并没有引入新的内容。简而言之，这可能是最合适的解决方案。

然而，这种方案易于改进吗？如果需要为这些超级用户代码添加一些新的方法，又会如何？在这种情况下，你需要添加一个新的构造函数，当然也可以是一个新的工厂方法，通过参数传入

额外的数据。这种方式可以做到二进制兼容，但每添加一项都会把工厂类变得更加庞大。不过问题不算严重，不用太担心。更复杂的情况其实出现在参数方面，上下文的改变会使得一个方法的参数增长非常明显。一大长串的参数顺序排下来，特别是如果类型相同，实在是让开发人员在调用方法时因为分不清楚传入参数而拼命地搔头。同时在编写调用方法的代码中也很容易引起顺序不对之类的错误。开始时，还不见得是个大问题。但随着 API 的改进，情况变得越来越复杂。此时就需要一种方案能解决该问题，比如说将工厂方案和基于 setter 方法的方案进行混合。下面的代码简单地展示了该方案：

```
public static Executor create(Configuration config) {
    return new Fair(config);
}

public static final class Configuration {
    boolean fair;
    int maxWaiters = -1;

    public void setFair(boolean fair) {
        this.fair = fair;
    }
    public void setMaxWaiters(int max) {
        this.maxWaiters = max;
    }
}
```

通过这种方式，就不必再添加新的工厂方法，如果改进过程中有相应的需求，在 Configuration 类中添加新的 setter 方法即可。添加新的 setter 方法是安全的，因为这个类是 final，而且只被用来传递工厂创建方法中的参数。它的长处在于可以限制外部的改变，它没有提供公开的 getter 方法，外部只能通过 setter 方法设置数据。也就是说添加新方法只会影响工厂方法的内部代码，而这是 API 设计者可以控制的内容。通过这样的处理，就可以对内部的内容也实现扩展，从而应对未来的改变。

进一步说，该方案还可以解决另外一个比较突出的问题，就是如果已经有普通用户操作了对象，超级用户还希望进行特权操作，这个问题在引入了参数类 Configuration 后也得到了解决。在纯粹的工厂方法中和 setReadOnly 模式中这是不可能的。现在，超级用户代码就可以持有对 Configuration 的实例的引用，并调用其方法来调整参数，这样那些控制类实例对象即使已经被普通用户所持有，也会因为持有 Configuration 对象的用户对 Configuration 的控制权有所不同而有所区别。普通用户的代码是不能访问 Configuration 对象中的某些数据的，同时因为 Configuration 与控制类对象实例是完全不同的类，也就不可能像前面一样，通过强制转型来执行一些操作。在这种设计中，Configuration 类可以说是扮演了一个秘密令牌的角色，这样就无法通过 API 来处理 Configuration 对象。因此，该解决方案不仅可靠而且安全。

不可变性及优先访问

在设计 NetBeans 的 API 时，我们经常面对一些相似的问题。我们通常使用我们自定义的

Mutex 锁，通过它来控制读/写操作，这样可以保证在多线程环境中进行安全的读写操作。该方案一直行之有效。但我们发现有些情况下，大量使用线程对程序性能的影响比较大。所以我们考虑添加如下方法来解决性能问题：

```
lock.enterReadAccess();
try {
    // do the operation
} finally {
    lock.exitReadAccess();
}
```

只不过，我们也很担心会有一些恶意代码会故意不释放锁，这样会因为这个锁而打破整个子系统的一致性。但我们能做的事情很有限，只能尽量保证我们的代码不存在此类 bug，所以我们决定引入超级创建模式以避免恶意代码或者无意犯下错误。

我们仍然使用那种基于 Runnable 机制的 Mutex 类，但添加了一个新的 Mutex 对象。有一个静态的内部类可以访问这个对象，可以通过调用相应的 enter 和 exit 方法从而更有效地利用“enter/try/perform/finally/exit”的锁方式。为此还添加了一个使用 Mutex 作为参数的构造函数，从而创建一个 Mutex.Privileged 的对象实例。

```
private static final Mutex.Privileged PRIVILEGED = new Mutex.Privileged();
public static final Mutex MUTEX = new Mutex(PRIVILEGED);
```

现在任何 Mutex 的创建者都可以利用这样的后门让自己的代码跑得更快，但在同步处理方面可能会引起更多的错误，但这样做至少可以保证普通代码无法通过恶意代码来破坏锁的一致性。

尽管在 Java 语言中，对象的创建者没有任何特殊的访问级别，但通过一些技巧仍然可以在 API 中实现此需求。解决方案很多，从最简单、最难看的到最复杂、最优雅的，一应俱全。而在我看来，最好的处理方式则是利用两个类：一个类为对象的创建者提供高级权限，而另一个则提供公开的 API，供任何人调用。

5.7 避免暴露深层次继承

有些人认为面向对象语言之所以优秀，就是因为它支持复用。的确，有一些编码模式在面向对象语言中表现得更为出色，而老一些的语言 C，就相对差一些。但这种出色的表现并不意味着它在功能复用方面就有提升，因为代码复用的主要问题在于交流而非其他方面。如果有人共享了一些内容，另外的人则必须能够吸收并复用这些内容，这才称得上真正的共享。并不是说使用了面向对象语言，这一切就会自然而然地做到了。对此，必须深入思考如何复用，预作准备，并时刻铭记这不是一件简单的事情。我对此建议则是一切要特别小心，首要注意的是避免深层次的继承。

典型的面向对象语言非常适合用来描述自然对象，包括该自然对象的进化。有一个众所周知

的经典例子是可以定义一个 `Mammal` 类来描述一种哺乳动物，类中只需要几个方法来描述其行为，而该类的子类，如 `Dog` 或者 `Cat`，则只需要覆盖这些方法即可。当然这个例子不仅展示了面向对象语言所能做好的，而且可以证明面向对象语言在描述自然对象方面有所长。上述例子说明了面向对象语言对自然对象的描述是非常容易的，因此被认为最合适描述真实世界。如同物理学比几何学能对真实的世界做出更合理的描述，面向对象语言也被认为比传统的面向过程语言更适用于处理真实世界的各种概念。

十年来，我开发了一个 Java 框架。此间，我常见的一种情况是监听器的功能处理，即支持收到消息（如通过调用 Java 方法发出消息）后，通知第三方进行处理，但有些时候，则会通过覆盖某方法来截获这个消息，并对消息的内容进行修改。事实上，这只是众多错综复杂的现象中的一种。调用一个方法时并不见得是固定要调用哪一段代码，很可能是根据具体情况在多个具体实现中选择某一个方法的实现代码执行。比如说想执行 A 类的 x 方法，但当前拿到的对象可能是 A 子类 B 的实例，如果 B 类覆盖了 x 方法，此时原来期望调用的 A.x 方法其实就变成了 B.x 方法。所以，执行时所调用的方法则依赖于有哪些具体子类，以及这些子类对父类方法的覆盖，而且随着开发人员不断地继承，还会不停地增加方法的执行路径。当然路径的选择也只是一选择而已。我们更愿意将面向对象编程解释为一种增强的选择，而不是作为一种描述真实世界的技术来进行展示。这正是为什么计算机课程中，一直沿用那个哺乳动物的例子。随之，我要说明这种执行路径不确定性下隐藏着一个陷阱：对于编写代码的人来说，他所编写的 API 的大小和具体运行时的情况其实是不确定的。这其实也可以看作是一个优点吧，因为它可以增加代码复用的方式。但在发布了多个版本以后，如果还不确认相应的执行路径，那么就等着恶梦的到来吧。如果不加限制，允许任意的代码被你的代码所调用，那么这种开放性会使得系统无法维护。

综上所述，直接将深层次的继承体系公开出来并不能提高 API 的可用性。在设计一个 API 的时候，要注意“继承”不是用来改变具体的行为，而是用来添加一些额外的行为，像我在编写子类时，就更关注后者。相信很多人都相信 `Human` 这个类应该是 `Mammal`（哺乳动物）的子类。但在 API 设计时能正确使用这个原则的人又有多少呢？难道你真的想说 `JButton` 是 `AbstractButton` 的一个子类，而 `AbstractButton` 又是 `JComponent` 的一个特别子类，`JComponent` 又是 `Component` 的一个子类？你当然不想！这里的子类只是具体的实现，与细节相关。这种子类只是提供了一种方式，用来切换执行路径，从而允许编码更加简单。如果按照 Swing 的设计思路，`JButton` 就可以当作一个 `Container` 来使用。换句话说，你可以在 `JButton` 中添加其他的控件。尽管这也是可能的，但肯定不是 `JButton` 这个 API 的设计原意。

现在让我们来见识一下在面向对象语言中一个常见的 API 缺陷。也许你的实现代码可以很好地受益于继承机制，但决不要以为其他人也是如此，更不要以为整个 API 及其用户都能通过这种方式获益。如果去继承一个类，只是为了切换某些方法的执行路径，事实上这种做法是应该避免的。所能给出的建议则是：避免深层次继承，定义程序的接口，并让用户来实现这些接口。请牢记，如果一个类继承了某个类或者实现了某个接口，那么就可以作为相应的类和接口被使用。

Swing 框架中 `Frame` 这个类，它间接地继承了 `Component`，其实是一个错误的设计，请读者一定要铭记于心。在所有的 API 规则中，对于这种继承关系，就表示所有可以使用 `Component` 类的地方，都可以使用 `Frame` 对象，当然，真要跑起来，估计未必可行。事实上，`Frame` 之所以继承自 `Component`，完全是出于实现该类的代码比较方便，只是想复用 `Component` 类中的一部分代码。因此，这种设计与真实的 API 设计是无关的。这种面向对象的复用，更应该称为误用，但这种误用是非常普遍的，在深层次的继承体系中表现得尤为明显。所以一旦发现继承体系超过两层时，一定要打住，并多多思考一下，要想清楚：“我到底是在设计 API 还是在复用代码？”如果答案是后者，那么重新审视，并严格设计这个 API，或者做好子类化的准备。



直以来，都有一个编程规则，它可以用于指导所有代码的编写，那就是“将代码分成接口和实现两部分，在编写代码时，让系统的其他部分只依赖于接口”。规则早于 Java 语言而存在，但这个技巧中蕴含的道理可谓无价，在我们使用 Java 设计 API 时仍然要加以借鉴。本章将会从不同角度概述此规则。

首先，这条经验法则意味着，如果你已经有了一个可运行的应用程序，且这个程序需要对外提供 API，以方便外部用户访问其功能，那么这并不是简单地把当前程序的一些类以 `public` 的形式公开为 API 就万事大吉了。如果没有经过重构，那么公开的很可能不是 API，而是内部的具体实现。所以，千万不要直接把代码当成 API，还是要花一些精力来整理一下才成。先来考虑一下创建这个 API 的缘由。然后要保证任何时候你在实现代码中的改变不会影响到公开的 API。

NetBeans 平台如何提供对外的 API

NetBeans 的 API 开发始于 1997 年，当时的版本提供了一大堆可谓定义良好的 API，同时相应的实现都从中分离出去了，相应的模块间都是通过 API 进行交互的。

以那时的眼光看来，这个设计非常可靠，与那些不管三七二十一就把程序中的东西全都堆砌在一起的设计相比，要好得多。程序也都运行得不错。但这些 API 设计时对实现的分离，还有用户模块的拆分，更多是基于逻辑而非物理。例如，当时是使用一个构建脚本来编译所有的内容。这样没有从物理角度将相关的内容进行分隔，根本无法从包名上来分辨某个包是否是 API 的一部分。

当时，我们项目的邮件列表上有一个人向我抱怨：很难分辨清楚一个东西到底是 API 还是具体的实现。开始时，我们只是帮着他分析清楚哪类内容是 API。但反思再三，我们觉得系统的 API 设计上存在问题。先要对这位触发我们灵感的朋友送上我们真心的感谢。在他的帮助下，Netbeans 平台的 API 在 1999 年被完全打散，并重新调整，将 API 放入 `org.openide` 包及其子包中，将核心实现都放到 `com.netbeans.core` 包中。当然，`com.netbeans` 这个包名是在 NetBeans 平台尚未开源之前定下的^①。剩下的那些非核心模块则放入 `com.netbeans.`

^① 开源以后采用 `org.netbeans` 包名。——译者注

modules^①包中。

在重写这个系统的过程中，我们发现了原先的源代码存在很多问题，在 API 及本应独立的实现中发现了大量相互依赖的内容。也许是因为我们太相信开发人员的职业道德？看来仅仅把程序中的部分内容作为 API 是远远不够的。在设计时必须强调，从实现中分离接口，再作为 API 提供出去。

另一个争议之处则是要能够给 API 用户讲述清楚，在使用 API 时，应该遵守正确的原则：编程应该面向接口而非实现。如果需要使用一个 API，就要遵守 API 的原则，不要试图破坏。如果在开发的过程中，发现了一些你所需要的功能没有在 API 中公开，那么即使有一些类似于黑客入侵的方式可以来访问这种功能，也不要利用这一点来做事。请牢记：不要去依赖那些没有在文档中提起的属性和文件资源，正确的做事方法是向设计者要求提供合适的 API。不论是对于 API 的用户还是提供者来说，都应该理解彼此的需求，并尽量满足这些需求，这才符合双方的长期利益。要强调的是，前面所说的那种利用未公开内容来完成某种功能的做法是非常脆弱的，会因为版本、环境的不同而无法正常运行。所以最好还是清楚地说明 API 的需求，再定义相应的用例，从而帮助 API 设计者来实现这样的 API。以合作的方式来创建真正的 API，远远好于面向实现编程。

在 Java 编程中，关于这方面的技术要点其实还有很多。如面向接口编程这个建议是早于 Java 语言出现的，所以它原话中的“接口”并不是指 Java 语言中的“Interface”。我们没必要挑这句话的毛病，毕竟当时常用的语言是 C、C++，还有 CORBA。尽管 CORBA 中 IDL (Interface Definition Language，接口定义语言) 对于接口的定义与 Java 中接口的定义比较相似，但当时的语言大都没有“Class”和“Interface”这样泾渭分明的内容。即使在 C++ 这种面向对象语言中，也是不区分类和接口的。它只有类，只要包含方法和字段，就认为是一个类，但面向接口编程对于 C++ 来说同样有效，也同样可以用于 CORBA，当然对于 CORBA，它的 IDL 只支持接口，即不包含任何非静态的方法和字段。

那么这个建议是否可以同样应用于 Java 呢？绝对可以，只要你能够清楚地理解面向接口编程并不是对应着 Java 中的 Interface，这种编程方式只是将其抽象定义与实际的实现这两类内容进行分离。想要做到这一点方法很多，直接使用 Java 接口自然是一个好的方法，而使用 Java 类也同样可以做到。

在 Java 中，是使用接口还是类来编写 API 呢？这绝对是一个能够引起口水大战的绝佳话题。当然，引发一场口水大战并非本书的目的。人们在讨论接口和类这两者孰好孰坏时，总是各执一词，这主要是因为有些人善于使用接口，而有些人则善于使用类，他们的经验决定了他们的立场。这同时也说明在某些情况下，使用类会比较合适，而另外一些情况，使用接口则会更合适一些。本章的余下内容会详细讲解 API 的各种设计方案，特别是这两种风格不同的解决方案，在对 API 改进时，分别有哪些特点。通过这样的讲解，读者可以根据具体环境来选择合适的方案。

^① NetBeans 的功能模块名称为 com.netbeans.modules.applenu，com.netbeans.modules 为其根本名称，然后具体模块再加上自己的名称空间，如 com.netbeans.modules.swing 等。——译者注

6.1 移除方法或者字段

如果一个接口或者类已经对外发布了,那么是否还能从中移除一个方法或者是一个字段?事实上只要有人使用了它,就不能移除。首当其冲的问题是,这种改变在源代码层次是不兼容的。如果一段代码调用了已经移除的方法或者访问了已经移除的字段,就无法通过编译。而且这种改变在二进制层次也是不兼容的。如果你编译代码时,选择使用那些仍然带有该方法的较旧版本的 API,而执行时,使用移除了该方法的较新版本的 API,那么在二进制代码进行连接时或在调用该方法时,会抛出一个运行时异常^①。所以说从已经公开的类或者接口中移除方法或者字段是一件错误的事情。

先来做个假设,如果一个类中的某个字段或者方法有严格的访问控制,同时也没有对外开放。比如说一个方法或者字段被定义为 `private` 或者是默认的 `package` 访问级别,那么对于外部代码来说,它是不可见的,也就无法进行访问。对于这种情况,从类中移除该内容,不会引起什么问题^②,但对于那些使用反射的用户来讲,就会在运行时抛出一些异常。如前文所述,滥用反射可能超出了 API 设计者的预期,也和良好的 API 使用习惯相违背,这正是破坏规则者容易出现问题的根源所在。

那么是否可以移除一个声明为 `protected` 的方法呢?虽然这类方法不能被外部所调用,但仍然可以被子类调用,所以也同样不能移除任何声明为 `protected` 的方法。如果这个类没有任何子类,那么也许可以移走这个方法,比如说这个类声明为 `final`,因此也就不会存在子类。只不过如果是这样的话,也就没有必要把这个方法声明为 `protected` 了。当然,肯定有一些 API,在不可继承的类中将一些方法声明为 `protected`,这通常是因为疏忽造成的,而不是有意为之。针对这种问题,我实在给不出什么好的解决方案,如果勉强要给个建议,只能说在正式发布以前,最好生成 API 对应的 Javadoc,通过仔细阅读来避免这些小错误。

如果上面所说的 `protected` 方法同时还是一个抽象方法,又会怎样呢?没有代码能直接从外部调用该抽象方法,其子类也不可能通过 `super.theMethod` 来调用该方法。即使在代码中写了 `super.theMethod`,编译器也会报一个“`abstract method cannot be accessed directly`”的编译错误。因此,对于源代码和二进制来讲,移除一个抽象的 `protected` 方法是可以做到兼容的。代码可以在新版本上编译和运行。但从功能角度来说,这种改变可能并不兼容。那个方法当时被放到类中,应该有相应的目的,而子类也可能会重载该方法,并希望在合适的时候被调用。现在它们的实现就不能被调用了, `theMethod` 方法语义也发生了改变,则子类是无法重载这个方法的。尽管这种移除从技术层次上讲是可行的,但仍然存在潜藏的危险。因此在任何 API 的改进过程中,只要有其他可选的合理方案,就不建议这样做。

① 这两者情况分别对应着类和接口,作者会在后面再加以说明。——译者注

② 作者这里可能是忽略了序列化,对于方法来说,序列化是不存在影响的,但字段会被序列化,如果某一个对象被序列化了,随后新版本从中移除了一个字段,然后再将上次序列化的内容反序列化回来,就会出现这个问题,这种情况在 J2EE 服务器的热部署时可能会出现。——译者注

6.2 移除或者添加一个类或者接口

如果一个类或者接口可以被外部访问，就意味着其方法可能会被调用，或者其字段可能被引用，那么移除该类，在源代码层面还是二进制层面都会引起不兼容。在这一点上，类和接口是一致的，它们一旦被作为 API 公布以后，就像恒星一样，必须存在下去。

如果从二进制层次来看，添加新接口或者新类都是一件好事。但如果使用 `javax.swing.*` 这样的类型导入声明方式，那么源代码的兼容性就会因此而受到破坏，这一点在 4.2.1 节中已经详细描述过了。在那节中，我还指出在现有的包中添加新类和接口的时候，仍然可以保持二进制兼容性，与此相对的是，某些改进也同样可能会破坏源代码的兼容性，但这种破坏是可以接受的。

这表示添加类或者接口是可以接受的。但必须要注意的是，添加一个 Java 的类或者接口其实是一样的。一旦类和接口成为 API，两者就没有什么区别了，都必须长久地存在下去。

6.3 向现有的继承体系中添加一个接口或者类

在特定环境下，向现有的继承体系中添加新的类或者接口可以带来很多好处。比如下面代码中，存在两个名为 `sayHello` 的抽象方法：

```
public abstract String sayHello();
public abstract String sayHelloTo(String who);
```

在接下来的版本中，作者希望可以简化这个 API，所以提供了另外一个类，可以支持第一个无参方法的调用，代码如下：

```
public abstract class SimpleHelloClass {
    public abstract String sayHello();
}
```

当然，以前编写的类仍然需要维护。所以将以前的类改成继承 `SimpleHelloClass` 这个新类，同时添加了更多复杂的方法。

```
public abstract class HelloClass extends SimpleHelloClass {
}
```

这种改变对于源代码和二进制来说，都是向后兼容的，功能也是兼容的，即使对 Java 类或者 Java 接口进行了重构，也同样兼容。唯一的限制就是所有的方法（不管是继承还是自定义的方法），只要来自原始接口或者是 `SimpleHelloClass`，都要保持一致，至少不能移除某个方法。

6.4 添加方法或者字段

现在准备讨论一下方法和字段，还是先从字段开始吧。在 5.1 节中，我强调字段不应该被添加到 API 中。在对 API 进行改进时，与方法相比，字段受到的约束更多。这正是要避免在 API 中使用字段的原因；当然那种使用 `static` 和 `final` 声明的常量字段是可以添加到 API 中的。添加此类字段是完全可行的，同时也能做到二进制兼容。

从二进制的层次来看，添加一个 static 方法也是可行的。但要注意，static 方法只能写在类中。所以在某种情况下，它们可能会造成源代码级别的不兼容，比如说子类写了一个 static 方法，但父类已经提供了相同的方法，即使这个子类不是 public 级别的，也存在同样问题。因为在编译调用这个方法的代码时，无法确认这段代码调用的 static 方法是属于子类还是父类的。

向类中添加一个抽象方法，则会强迫其非抽象的子类来实现该抽象方法，这种改变是做不到向后兼容的。这种处理对于源代码来说是不兼容的，因为对于那些没有实现父类的抽象方法的类，除非它们也是抽象类，否则编译器就会拒绝编译此类代码，并警告说必须要实现父类的抽象方法。不过让人吃惊的在后面，那就是这种改变可能是二进制兼容的。造成这种奇怪现象的原因是虚拟机在处理方法时，对于接口和类分别采用了不同的方式。虚拟机在加载一个类的时候，如果该类没有实现父类的抽象方法，就会给出警告。但对于接口，则是一直等到方法被调用的时候，才会进行检查。

虽然虚拟机对接口和类的处理方式略有不同，但随后的调用也同样会抛出一个运行时异常，这多少可以算作接口与类相比的一个小优点。可以想象一下，如果在 Javadoc 中加上这样一句“不管是谁调用接口中的这个方法，都必须捕捉 UnsatisfiedLinkError 这个错误，因为有一些老的库可能没有实现这个方法”，这样的处理方式实在是太恶心了一点。事实上，这样会导致那些 API 看起来非常丑陋，最重要的是还会引发功能的不兼容性。

所以应该避免在允许继承的类或者接口中添加新的抽象方法。如果是类，请确保该类不可被继承。

那么添加一个非抽象方法，是否就不存在危险了呢？首先，能添加非抽象方法的只能是类，不可能在接口中添加一个非抽象方法。要做这件事，首先必须认识到，对于源代码来说，可被继承的类很容易造成兼容性问题。例如，在 JDK 1.4 即将发布时，NetBeans 平台的 API 中有一个异常类，它定义了如下方法：

```
public Exception getCause()
```

在 JDK 1.3 的环境中，运行没有任何问题，但 JDK 1.4 在 Throwable 中引入了如下方法：

```
public Throwable getCause()
```

我们有了大麻烦，编译器无法编译我们的 API 代码！但幸好，在发布之前的几个小时，我们找到了这个问题，并通过重命名的方式解决了该问题。如果没有这一步，我们基于 JDK 1.3 设计的 API 就完全不兼容于 JDK 1.4 了。但这只是一个源代码不兼容的小问题，而在二进制级别，则没有任何问题。这是因为虚拟机会把方法的返回值作为方法签名的一部分，所以我们提供的方法与 JDK 提供的方法在虚拟机执行的时候会被作为不同的方法，因此不会相互干扰。

只不过，在向既存类中添加方法时，意外的干扰仍旧是最大的问题。假设在 NetBeans 中，这个 getCause() 使用 Throwable 而不是 Exception 作为返回值。那么因为该方法与 JDK 1.4 中的一致，编译上就不存在任何问题。令人吃惊的是，代码仍然运行正常，但正常运行只是偶然情况，因为在这个类的超类中还引入了一个相同的 getCause 方法。现在这个方法就可能会在不合适的场所被调用。它完全破坏了原先的语义，从语义的角度来说，功能不再兼容。

添加新方法和新类型

如果因为在接口中添加方法而引发了问题，那么有个技巧是可以用来解决该问题的。在添加一个新方法的同时，开发人员也要添加一个新类型，并使得这个新类型成为方法签名的一部分。在 Java 规范中，已经说明，只要把一个类型声明为参数类型或者是返回类型，编译成二进制时，该类型就会成为方法签名的一部分。

从二进制代码及其语义的角度来看，这是可以接受的，因为这个方法签名与以前版本中任何兼容的都不匹配。这是因为签名中的这个类型在以前的版本中根本不存在。要知道这种机制可以避免使用的接口添加了新方法而出现问题，因为子类的方法不可能使用新类型。因此，以这种方式来添加一个方法，不管从二进制层次还是从功能的角度来说，都是兼容的。

出现这种情况的几率与具体类的用户数量有关，同时也与这个新添加方法的名称有关。比如说在 `java.awt.Toolkit` 中添加一个新方法看起来没有什么问题，因为这个类的实现不会多。如果是在 `java.lang.Object` 这个类中添加一个名为 `public boolean isValid()` 方法，估计很多人的代码就会很容易出现问题^①。

总地来说，无论是向一个类还是一个接口中添加一个方法，只要这两者可被继承，那么这种改变就做不到百分百的兼容。也许从二进制层次来讲没有问题，但源代码层次就不行。而且几乎可以肯定的是，语义是不兼容的。所以谨慎一点地说，只要能做到兼容，就决不要向一个可被继承的 API 元素中添加方法。

6.5 Java 中接口和类的区别

前文对接口和类作了大量分析，现在让我们开始讨论一下那个老生常谈的话题：在 Java 中，接口和类有哪些区别。

Java 接口最突出的特性就是多继承。事实上，很多人也认为这是其最重要的特性。但多年来开发 NetBeans API 的经验告诉我们，大部分情况下其实用不到。如果要选真正的理由，我只能说对性能的追求是唯一现实的理由，因为多继承可以减少对内存的占用。利用多继承仅使用一个对象，就可以实现 API 中公开的多个接口。类继承则只能有一个父类，此时如果类之间有些数据需要互访或者调用方法，那么只能用委托的方式。这样的继承对内存的占用是比较大的，可以看一下具体的数据：假设基于英特尔的 CPU，在大部分 32 位的 Java 虚拟机上^②，一个对象实例，不管它实现了多少接口，它只会占用 8 个字节的内存。但如果是一个引用了其他类型的类，那么每一个对象实例至少要 16 个字节。相比之下，一大堆这样的对象会占用很多的内存，这样继承自接口，就会好过继承类，但这种明显的内存占用，只有创建了非常多的对象时才会出现，如果只是一两个，就不是那么明显了。另外编译器中使用的抽象语法树也能从中获益。但在正常情况下，

① 作者这里是指如果一个新添加的方法名称特别普遍，而且该类的实现非常多，就会容易出现问题，如为 `java.lang.Object` 添加 `isValid` 方法。——译者注

② Java 虚拟机是一套规范，可以不同的实现（如 IBM 和 Bea）都有自己的 JDK，同时在不同的操作系统也有不同的实现，所以作者说大多数。——译者注

性能方面的考虑就不是主要因素了。

6.6 弱点背后的优点

从接口演变的角度来看，有一件事是很明显的，那就是向现有接口中添加方法总不太容易。如果要强调向后兼容的话，那么添加方法就不是有多困难的问题，而是几乎不可能完成的任务。但这样一个弱点，反而成为接口的一个最大优点。

通常来说，如果提供 API，也会有相应的版本。比如说对于 Java 语言，Java 1.3、Java 1.4 和 Java 1.5^①这些版本的特性都有所不同，对于一个接口来说，可能也需要根据具体情况有所改变。事实上，对于每个具体的 JDK 版本，其特性是固定的，不可能改变。这样对于接口的使用，就提供了一个良好的机会。分别定义 Language13、Language14 和 Language15 这三个接口，并分别实现这三个接口。用户在使用时可以根据自己的实际情况来选择相应的实现，这样就可以使用相应的功能，而且代码看起来也更加清楚。在编写方法和编译代码的过程中，对于语言的不同版本，要决定分别支持哪些功能。就这点而言，接口是一个非常合适的工具。

虽然看起来很美，这里仍然隐藏着一个陷阱。那就是，接口膨胀会极大地提高客户端代码的复杂度。NetBeans 曾经遇到过这个问题，是有关一个提供对象实例的接口。代码如下所示：

```
public interface InstanceProvider {
    public Class<?> instanceClass() throws Exception;
    public Object instanceCreate() throws Exception;
}
```

随后，我们发现一个常用操作：从提供者中获取类并判断这个类是否可以转为另外一个类。NetBeans 执行这个查询非常频繁，我们认为它是一个性能瓶颈点。我们需要找到一个方式来避免将真正的实例文件加载到内存中，这样可以提高整体性能，于是我们就创建了如下这个接口：

```
public interface BetterInstanceProvider extends InstanceProvider {
    public boolean isInstanceOf(Class<?> c);
}
```

提供这个接口以后，不管是我们的相关代码，还是那些由第三方在各个模块中提供的代码，我们都建议使用 instanceof 来判断相应的 provider 对象实例是否实现了 BetterInstanceProvider 接口。如果是，那么先强制转型，然后调用相应的 isInstanceOf 方法来完成相应的功能。

这样做引发了新的问题。首先所有使用了 InstanceProvider 这个接口的客户端代码都需要重写。对于这样的平台，代码是由各地的开发人员完成的，没有人可以控制所有的代码，显然这种修改是不可能完成的。结果就是大部分客户端代码还是沿用老的方式，性能问题仍然没有改进。而且这样使得代码越来越复杂。每个客户端都要使用 instanceof 进行类型检查，然后再进行处理，处理还要分成两步操作：如果 instanceof 返回 true，就可以使用新方法，否则还得使用原有方式。满篇都使用 if else 语句，看起来一点也不优雅。

```
if (instance instanceof BetterInstanceProvider) {
    BetterInstanceProvider bip = (BetterInstanceProvider)instance;
```

^① 即 JDK 5，但为了更方便理解这一节内容，所以沿续了 1.3、1.4 和 1.5 的说法。——译者注

```

    return bip.isInstanceOf(String.class);
} else {
    return String.class.isAssignableFrom(instance.instanceClass());
}

```

以上原因使得我在扩展功能时，喜欢明确指出一个不会改变的接口。但是，千万不要在 API 客户端遍布大量的 `instanceof` 判断语句，然后再强迫它们从一堆可送的接口中选择一个合适的。下文会讨论一些更好的处理方式。

6.7 添加方法的另一种方案

在 Java 接口完全不接受方法添加的同时，却存在如下相反的情况。在这种情况下，不仅要添加方法，而且还能完全保证二进制兼容。在 Java 中，如果有一个类声明为 `final`，就能达到这种效果。

把一个类声明为 `final`，就表示这个类是不可被继承的。因此那种在接口或者抽象类中添加方法而引起的问题，对于这个 `final` 类来说就不会出现了。唯一可能出现兼容问题的时候则是在调用那个类的方法时。由于在类的二进制文件中，对于要调用的方法，是可以通过名称、参数及返回值来唯一确定的，所以不会产生兼容性问题。但仍然有一些情况是二进制兼容，但源代码不兼容的。假设已经有了一个方法为 `void add(Integer i)`，在随后的版本中，有一个人添加了 `void add(Long l)` 方法。部分现有的源代码可能就无法编译了。比如说，像 `theObject.add(null)` 这句代码，编译器就会给出一个编译错误，说这句代码中要调用的方法具有二义性。要避免这类问题的方法很简单，就是不要加入具有相同数量的参数的同名方法。

所以，如果需要给一个类或者接口添加方法时，就选择 `final` 类吧。在 `final` 类中添加新方法，不会像在接口中添加新方法一样，把客户端代码搞得极其复杂。假设 `InstanceProvider` 是一个 `final` 类，代码如下：

```

import java.util.concurrent.Callable;

public final class InstanceProvider {
    private final Callable<Object> instance;

    public InstanceProvider(Callable<Object> instance) {
        this.instance = instance;
    }

    public Class<?> instanceClass() throws Exception {
        return instance.call().getClass();
    }

    public Object instanceCreate() throws Exception {
        return instance.call();
    }
}

```

现在就很容易在这个类中添加新方法，并提供默认实现，调整后的代码如下：


```
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;
import java.util.concurrent.Callable;

public final class InstanceProvider {
    private final Callable<Object> instance;
    private final Set<String> types;

    public InstanceProvider(Callable<Object> instance) {
        this.instance = instance;
        this.types = null;
    }
    /** 不仅需要工厂来创建对象实例，还需要一些关于创建对象实例时的额外信息。
     * @param 用来创建对象实例时的工厂。
     * @param 用来标识要创建的对象实例所属的类型
     * @since 2.0
     */
    public InstanceProvider(Callable<Object> instance, String... types) {
        this.instance = instance;
        this.types = new HashSet<String>();
        this.types.addAll(Arrays.asList(types));
    }

    public Class<?> instanceClass() throws Exception {
        return instance.call().getClass();
    }

    public Object instanceCreate() throws Exception {
        return instance.call();
    }

    /** 用来确认给定的 InstanceProvider 是否可以创建指定类型的对象实例。
     * 这种检查无需创建真正的对象或者将相应的实现类加载到内存中。
     *
     * @param 要验证的类
     * @return 判断当前给定的 Instance Provider 是否可以创建上一个参数指定的类
     * @since 2.0
     */
    public boolean isInstanceOf(Class<?> c) throws Exception {
        if (types != null) {
            return types.contains(c.getName());
        } else {
            // fallback
            return c.isAssignableFrom(instanceClass());
        }
    }
}
```

这两个示例的区别在于：如果使用 API 2.0 版本的话，相应的客户端代码就不用去考虑实例处理它们调用 `isInstanceOf` 的方式是否更好，也不觉得使用 `instanceof` 有什么问题。在 `isInstanceOf` 代码中对具体情况进行了处理。这两个接口看起来差不多，但后者将判断的代码放在方法的内部，对于客户端代码来说，完全不用关心，所以维护起来也就更加方便。

6.8 抽象类有没有用呢

说到这里，一直在讨论的内容就是如果要定义一个不可变的契约，那么就应该使用接口，如果需要为以后添加方法留有余地，那么使用 `final` 类则比较合适。所以，会有人提出疑问：是否有需要使用抽象类的地方呢？

如果要简单地回答这个问题，答案只有一个字——“不”。在 API 中，抽象类其实一直备受质疑，如果 API 中使用了抽象类，通常就会被认为这是因为设计者没有投入足够的时间来合理地设计 API。如果更全面地来看这个问题，也还是可以找出一些理由支持我们在设计 API 时使用抽象类。

首先要明确一点，对于某些情况，并不要求百分百做到二进制兼容。有时候，如果一个类的子类数量不多，那就没有必要在子类因为继承而破坏兼容性这种问题花上很多时间，也不必考虑得非常详尽。`java.awt.Toolkit` 就是一个绝佳的例子。这个类的子类只有三四个，但可能会被成千上万的客户端来调用。对于这个 API 的编写者来说，子类的兼容性问题，还有添加新的抽象方法，这些设计点都不用过于关注。对于 99.99% 的客户端来说，这么做没有什么问题，是一种可以接受的方案。如果说其缺点，就是 NetBeans 平台通常都会成为那例外的 0.01%。比如说我们就继承了这个类。每当新版本的 JDK 发布，添加了新的方法，我们就会碰到此类问题。其实大部分情况下，99.99% 的兼容性已经足够了，所以一般来说，使用抽象类也是可以接受的。

相对于 Java 接口，使用 Java 抽象类还有一些其他方面的优势，那就是抽象类可以包含 `static` 方法。即使想把所有的内容都使用抽象方法来描述，但也可能需要有些 `static` 工厂方法作为基本的入口，那么就可能需要抽象类，开发人员可以通过 Javadoc 得知相应的 `static` 方法，作为整个 API 的入口。不过也可以使用一个接口加上一个独立的工厂类，后者专用来提供 `static` 工厂方法，具体的选择完全取决于个人的喜好。

在 Java 中，类优于接口的第三个地方在于它可以限制访问权限（restricting access right）。在 Java 接口中，所有的方法都是 `public` 级别的，任何人都可以实现或调用它。也许这个时候应该轮到抽象类出马了。比如说，可以在一个抽象类中将所有的方法都声明为 `protected` 的抽象方法，然后用这种方式声明可供其他人实现的接口，但外部代码无法调用这些 `protected` 方法。在特定的情况下，这样处理有自己的优势。但一定要记住一点，这样做可能无法通过委托来实现一些功能，比如说无法使用 Decorator 模式来为这个类添加特定功能，如记录日志等。很多时候，这样处理后带来的约束会使得开发人员极为痛苦。

抽象类还能够对自己的子类进行一定的限制。可以把所有方法都声明为 `public` 的抽象方法，同时提供一个包级的构造函数，或者在运行时限制对子类的创建。代码如下：

```

public abstract class InterfaceThatJustJoeCanImplement {
    protected InterfaceThatJustJoeCanImplement() {
        if (!"impl.joe.JoesImpl".equals(getClass().getName())) {
            throw new IllegalStateException(
                "Sorry, you are not allowed to implement this class"
            );
        }
    }

    public abstract void everyoneCallThisJoeWillHandleTheRequest();
}

```

尽管这种基于运行时的验证并不完美，但它却可以对其子类进行限制。如果换作是接口，就无能为力了。

6.9 要为增加参数做好准备

如果要对 API 进行修改，那么最常见的问题就是如何通过添加方法的参数来增强现有方法的功能。这种场景非常普遍，特别容易出现在那些 API 需求变化比较大的场景中。比如说，假设现在有一个方法用来计算一个 UI 列表控件中的内容。但随后你发现，不仅仅需要得到列表控件中的内容，还需要一些额外的文本来对计算得到的数据进行说明和归类。

如果出现这种需求，就需要添加一个新方法来调用先前的那个方法，参见下面的代码：

```

public abstract class Compute {
    /**
     * @return 返回要用的字符串列表
     * @since 1.0 */
    public abstract List<String> getData();
    /** 计算要用的字符串以及配套的描述信息
     * 子类可以覆盖该方法。
     * 默认的实现，字符串和描述信息是一致的，参见 getData 方法。
     *
     * @return 返回要用的字符串以及配套的描述信息
     * @since 2.0 */
    public Map<String,String> getDataAndDescription() {
        LinkedHashMap<String,String> ret =
            new LinkedHashMap<String, String>();
        for (String s : getData()) {
            ret.put(s, s);
        }
        return ret;
    }
}

```

这样处理当然是可行的，但只限于这个方法的访问级别为 `private`，只能在一个类的内部被调用。但最大的问题是，对于客户来说，给他一个类去实现，就不如给他一个接口来实现，后者看起来更像是一种解决方案。同时在一个可以子类化的类中添加一个方法总是有些危险的。这就是为什么 `Compute` 这个类更应该是一个接口。但其实可以不用这种添加方法的处理方式。当然，也

可以像 BetterInstanceProvider 一样，做一个可扩展的接口，但考虑到 API 的改进，更好的解决方案则是使用 Request/Response 模式，简单看一下这种模式应用在代码中的示例：

```
public interface Compute {
    public void computeData(Request request, Response response);

    public final class Request {
        // 只将 getter 方法公开，其他的方法只对友元类开放
        Request() {
        }
    }

    public final class Response {
        // 只将 setter 方法公开，其他的方法只对友元类开放
        private final Map<String,String> result;
        /** 允许友元类来访问 */
        Response(Map<String,String> result) {
            this.result = result;
        }

        public void add(String s) {
            result.put(s, s);
        }

        public void addAll(List<String> all) {
            for (String s : all) {
                add(s);
            }
        }

        /** @since 2.0 */
        public void add(String s, String description) {
            result.put(s, description);
        }
    }
}
```

使用这种模式，只需要向 Compute.Request 类中添加新的获取方法，就可以为 computeData 方法添加新的参数，同时因为 Compute.Request 这个类声明为 final，所以它的方法只能被调用，不可能被重载。这就意味着向该类中添加方法是非常安全的。

从另一个角度来看，如果这个方法有太多的数据需要返回，也可以向 Response 类中添加新的设置方法。比如说加一个 add(String) 或者是 setMessage(String) 类似的方法。再强调一点，Response 这个类一定要声明为 final，这样才能为它添加新方法来扩展它。值得注意的是，这个模式也解决了 Java 中一个方法只能有一个返回值类型的问题。对于 Compute.Response 类来说，每一个设置方法都意味着一个返回值类型。

如果采取这种模式，一定要把 Request 和 Response 类都声明为 final 类。如果将它们声明为接口的话，在对这个系统进行改进时，就会平添很多麻烦。有些人可能会在某个版本中实现了这

个接口，一旦向接口中添加了新方法，就会破坏它们原有的代码。Servlet 的 API 就曾经发生过类似问题。这个 API 也是建立在 Request/Response 模式之上的，只不过在这个 API 中，Request 和 Response 不是类，而是接口。这就表示那些具体服务器的实现者，为 1.1 版本编写的代码无法简单移植到新版本中，因为它们的代码根本无法通过编译。如果你碰巧需要使用这个编程技巧，那么请一定要记住：在对方法进行改进时，方法的参数也应该可以无风险地进行改进；也就是说，这些参数类应该是 final 类而不是接口。还应该使用友元访问级别来清楚地提醒外部用户，在 Response 类中不支持数据的读取，在 Request 类中不支持数据的写。

6.10 接口 VS. 类

总而言之，编程时应该面向接口而非面向实现。请牢记一点，所谓的接口并不是指 Java 中的 Interface，而是指抽象内容。如果使用 Java 进行编程，就使用 Interface 来标识不可变类型，同时使用 final 类作为可以添加方法的类型。在设计更复杂的结构时，需要考虑扩展性，在深入思考后选择合适的方式，如前文给出的 Request 和 Response 示例。有时候，使用抽象类也是可接受的。只不过，要记得如果希望达到百分之百的二进制兼容性，决不要向接口或者是可子类化的类中添加方法。



众所周知，操作系统及其之上的应用程序都是基于模块化的开发方式完成的。最终产品由独立组件组装完成，从而使得异地协作开发可以以相当可靠的方式进行。

即使对于单个应用的开发，其实也在经历着同样的变化。随着复杂度的提高，单个应用也在转向基于独立开发的块组装完成。达成此目标的办法就是模块化开发。

随着软件规模的增大以及功能复杂性的提高，越来越有必要将单个应用拆分为独立的块、组件、模块或者插件。每个这样的部件都是模块化架构的有机组成单元。而且，每个部件都应该是独立的，并提供定义良好的对外接口供外部调用。在 NetBeans 中，我们把这样的部件称之为模块 (module)，而且本章主题就是模块化架构，我们就将符合此类标准的部件也称为模块。

将单个应用拆分成不同模块可大大改善其设计。不难理解，当大量代码相互调用频繁形成铁板一块时，相较于模块化的代码（特别是当架构上保证这样不受约束的相互调用只会局限在单个模块的内部，而不会发生在模块之间），这样的代码相互关联度大大提高而且可读性差。

模块化应用程序之于普通的面向对象应用程序，相当于 20 世纪 60 年代的结构化编程之于意大利面条代码 (spaghetti code)。意大利面条代码的概念是用来形容笨拙难以管理的 FORTRAN 或 BASIC 程序代码，往往每条语句都可能使用 GOTO 语句跳转到程序的另一个位置。这样的代码往往是以一种混乱的方式完成，也只有作者能理解（如果还有人能理解的话）。结构化编程通过引入种种代码块试图降低此种混乱：for 循环语句、while 循环语句、if 语句、过程、过程调用。这些做法确实使情况得以改善，提高了程序可读性与可维护性。如不出意外，每次方法调用都会确定有一次结果返回，除非有极特殊情况发生。^①

简单的面向对象编程在某些方面类似于结构化编程确立之前的情况：一个程序由众多的类构成，往往任何一个方法都可以调用其他的方法。尽管有 public、private、protected 访问修饰符，但面向对象的粒度大部分是在类一级层次实现的。但是类作为程序的基本构成单元来讲粒度过细，同理，类也不适合作为一个程序进化的基本单元。

模块化程序（参照 NetBeans 中的定义），是由不同模块构成的，而一个模块是一组类的集合。模块中有些类是 public 级别的，作为供其他模块访问的 API，同时另外一些类是 private 级别的，

^① 相较于上文的 GoTo 情况。——译者注

外部不能访问。此外，一个模块会依赖于其他模块，并在较高层次上声明了执行所需的功能性环境。当然，在一个模块内部开发仍然可能出现最坏的编码方式。然而，通过检查模块间的依赖，可以很好地评估一个应用程序架构的好坏。如果一个模块不依赖于另一模块，显然该模块中的类不会直接访问另一模块中的类。这使得整个程序架构更加清晰，因为它有效地避免了无关代码部分之间类似 GOTO 的跳转。

有些开发人员声称他们的应用程序规模太小，所以引入模块化架构不一定有什么好处。可能此应用程序现在确实规模太小，但只要不是学生作业那样小的项目通常都会发展变化。随着项目的演进，规模会变大。试图通过采用更优雅模块化设计来重写一个杂乱和内部相互关联的应用程序通常是一项复杂的任务，这往往会使很多人望而却步。相反，人们更倾向于保留原有代码，尽管原有代码往往庞杂且难以维护，但还能正常运行。让我们来看一个 JDK 的例子，`java.lang` 包中有些类实现了 `java.io` 包中定义的接口，有的还实现了 `java.util` 包中的接口。更过分的是，还有很多类竟然和 `sun.*` 包中的类扯上了关系。几乎难以想象谁能把 JDK 拆分成不互相关联、不出现循环依赖的模块。显然这是 JDK 曾经采用意大利面条式面向对象风格导致的必然结果。

NetBeans 设计中的脏乱差

NetBeans 中也曾经有过这种意大利面条式的代码。尽管 NetBeans 架构一直都是按照模块化架构的思路设计的，但最后结果却出现了一个庞大的模块，当时被称为 OpenAPIs。这个模块包含了所有与其他模块进行交互的 API。以今天的眼光来看，这也能叫模块化？简直是荒谬之极。回想一下，虽然 NetBeans 做得也不好，但从 1997 年到 2000 年，还没有哪个 Java 项目比得上它呢。随后又过了很长时间，我们逐渐明白了，维护这样庞杂的 API，应该考虑可扩展问题。在 2000 年前后，我们对模块化架构这个问题想得更清楚了，每个模块都应该能够对外提供公开的 API。从那以后，我们就严格按照这种规则来设计模块。只不过，原来那个庞杂的 OpenAPIs 中的部分内容还是保留下来了。

但情况还是在逐渐恶化。因为每一个版本都有一些东西要修正，开发人员开始在 OpenAPIs 的各个逻辑模块间建立内部依赖关系，这样就破坏了原有的松耦合关系。最终，就像在重复 JDK 所犯的错误——`java.lang` 包中的内容竟然去引用 `java.awt` 包，此类问题越来越多。有时候，有一些具体代码的方法中使用了某些不应该依赖的模块内容，最夸张的是有些对外的 API 竟然也出现了这种情况。到了 2003 年，我们对此终于忍无可忍了，需要从根本上更正这些错误。此时，我们迈出重要的第一步：首先清除了所有不必要的交叉引用关系，并将 OpenAPIs 按照逻辑功能进行模块分解。虽然编译后的二进制结果还是一个很大的模块，但这样做至少避免将来在逻辑模块间产生更多不必要的依赖关系。我们又花费了两年时间才将这样一个庞大的模块完全分解。

在 4.6 节中，我引入了“软件熵”的概念。只要代码开始访问其他无关模块中的内容，那么技术架构的退化就是不可避免的。除非能够有明确的方案来阻止架构的退化，否则这一过程

是不可避免的。本章讲解的模块化架构思想可以有效地解决这种问题。虽然模块化架构并不能让每一行代码变得精致，高效，甚至做了模块化以后，一个模块的内部代码可能写得比以前还乱，但模块化却是在更高的层次上以基于更大组件的方式来强化整个程序的架构，而且也有可能提供真实的结果。同样，这也符合我们说的针对性无绪的观点，即只关注那些对成功至为重要的关键组件。

最关键一点，就是要从项目一开始就得牢牢地将模块化思想应用到具体项目中。模块化架构可以使得设计更加清楚，也可以更好地管理模块间的依赖关系，在维护方面则体现为更高的灵活性。不管项目的规模如何，从一开始就使用模块化的思想去进行设计，这才是王道。

7.1 模块化设计的类型

并非所有的模块都需要提供对外 API。比如说，那些只与用户界面有关的代码就不需要提供对外的 API，这应该是最简单的模块形式：它完全封闭，不需要对外提供一些类，因为不需要外部代码来调用其功能。它往往只是一些用户界面的内容，直接与最终用户进行交互。即使有其他模块依赖于这些模块，只需要保证用户界面上的功能可用也就可以了。NetBeans 中就有这样一个 Favorites 模块，它提供了一个资源管理器的窗口用来浏览本地文件。如果某个程序需要这样的功能，直接使用 Favorites 模块即可，使用它完全能够保证用户通过相应的资源管理器窗口来浏览本地文件。然而，在此设计中，两个模块间没有任何直接的 API 调用^①。

还有一种简单的模块类型，就是提供了一些简单但又非常通用的功能的类库模块。这种通用类库的功能非常简单，易于理解，很多开源软件的开发者都会复用这种功能类库以提高开发效率，像 Apache 软件基金会提供的 Commons 集合类库就属于这种类型。这种模块的发布形式很简单，只是一个 JAR 包，公开了一些 public 的类和接口等。开发人员可以利用这些公开的内容，如直接调用相应的类，或者继承某些类并覆盖某些方法。只不过这种类型的功能类库一般都不考虑是否会有第三方来为其提供扩展功能，所以在开发时，该类库的设计人员就认为所有的代码都是放置在类似 SourceForge 的这种代码库中，不管是优化还是别的改进都只是由原作者来处理，用户会自动获得更新。

有时候，即使有其他的开发商为这种类库提供了一些新功能，也没什么意义，因为这个类库可以完全控制其对外接口，这样外部访问的功能也就是受限的，就像 Word 编辑器对外提供了脚本支持，但脚本中的 API 功能是受限的，模块中会有一些 public 的类，还有大量 private 的类。从脚本打包和使用的方式上来讲，这种脚本提供的功能与这种简单的类库并无二致。

接下来要讨论的是功能扩展问题，就是说在设计模块的时候要考虑到其他第三方基于该模块进行二次开发的可能性，也就是说要支持其他的开发商来根据自己的实际需求对模块提供功能扩展。设计者应该以 PDF 的格式来发布相关的 API 文档，对相应的规范等详细内容加以说明，同时还要提供示例代码。这样第三方开发商就可以编写相应的 API 类，而且能做到代码与发布的规

^① 其实个人并不是很赞同这个说法，作者前面也说了，配置文件等也是 API 的一种，即使两个模块间不存在代码级别的调用，但仍然可能有配置文件之类的内容。一家之言，仅供参考。——译者注

范保持一致。更重要的是,有些规范制定者通常还会提供一个该规范的默认实现。有时候,会发布一个 JAR 包或者是一个完整的系统作为实现的参考,像 Sun 就提供过 J2EE 服务器的示例。其他的开发商就会把这些提供的内容给复制过来,然后在此基础上加以扩展,最后以 JAR 包或者框架的形式加以发布,发布的内容通常会同时包含有接口和具体的实现类。客户端代码则会在这些内容的基础上来编译和执行自己的代码。这样做可能出现以下情况:某一个客户端代码会使用 A 开发商提供的库编译自己的代码,但具体运行时,则又是使用 B 开发商提供的库,只要这两者提供的接口名称和签名匹配即可。

即使做到了上面所说的各项内容,也不表示这样的系统就是一个模块化系统。其中最大的问题在于:如果有多个开发商对同一个 API 提供了多个实现产品,而这些产品很可能无法同时使用。因为不同开发商提供的 JAR 包在功能上有所重复,即不同 JAR 包中同名的类有可能只被虚拟机加载一次,也有可能通过不同的 ClassLoader 被加载多次,这样做的直接后果就是可能没有一个产品能够正常运行。假设某个开发商提供的 JAR 包中的类被 Java 虚拟机加载的时候,还有其他不同版本的类也被 Java 虚拟机加载,这两个实现类就无法正确地进行连接,也许运行时还会丢出一个某方法不存在的异常等。另一方面,如果这些类被加载了两次,每一个客户端都要提前判断到底使用哪一个开发商提供的模块。除非利用 JDK 的反射技术,否则就不可能同时使用这两者。这种情况下的唯一希望就是 Java 虚拟机在加载这两个产品的时候,加载的 API 是最新的,这样两个实现就可以共享这个最新 API。只有在这种情况下,整个系统才能正常运行,有一些模块化系统已经采用了这种方案。但要注意,这种处理方式不是由 Java 虚拟机自动完成的,必须认真地设计和改进 API,保证两个实现类编译后的二进制代码可以被正确地连接和执行。从这个角度来说,最好还是老实地承认这类 API 比较特殊,它与实现无关,而应该从其当前所在的模块中分离出来。

只有真正的模块化处理才能解决多个开发商产品间冲突的问题。模块化架构将规范与具体实现分离,分别放置在不同的模块中。可以定义一个模块专用于放置规范,即其文档中涵盖的实际的接口、抽象类等。需要实现规范的开发商可能会提供一个或多个单独的模块,用来实现具体的功能。这类设计通常有一个共同点,就是:在规范所在的模块中,至少会有一个小的“入口点”,比如说构造函数或者是静态工厂方法,这样客户端代码就可以通过这样的入口获取具体的实现内容。

对于现在的 JDK,如果想获取和注册一个具体的实现类,有很多种方案可选,但在 JDK 刚刚发布的时候,是没有这么多方案可供选择的。比如,像 `java.lang.System` 这个类中有一个 `setSecurityManager` 的方法,就是用来设置 `java.lang.SecurityManager` 类的具体实现子类,而且这个方法只能调用一次,否则会抛出相应的异常。像 `URLStreamHandlerFactory` 也有类似的设计,它通过 `setter` 方法来注册一个工厂类。在某些情况下,可以只允许注册一个实现,但有些情况下,则可能需要注册多个实现。事实上,如果允许注册多个实现的话,`SecurityManager` 和 `URLHandlerFactory` 都可以从中获益。NetBeans 就支持这样多注册机制:因为考虑到不同的模块有不同的安全处理方式,NetBeans 平台中的代码需要多个 `SecurityManager` 的实现。同样,我们也需要有多个 `URLStreamHandlerFactory` 的实现,因为很多模块都有它们自己的 URL 模式。

上文说明了为什么用户希望能注册多个工厂。例如,像 `JEditorPane` 中的 `registerEditorKitForContentType` 方法就可以为每一种具体 MIME 类型注册不同的处理机制,因为我们知道多个模块都需要使用自己的工厂来处理不同的 MIME 内容。

还有一些代码则使用属性配置的方案。比如说, `Toolkit` 会检查 `java.awt.Toolkit` 的属性中的值,保证该值所对应的 `java.awt.Toolkit` 子类有一个默认构造函数。然后使用该默认构造函数来创建一个 `Toolkit` 的对象实例。这种方式非常优雅,因为其 API 没有提供对外的(对于大多数用户而言的)无用的 `set` 和 `register` 方法,因此不会被外界误用或者破坏。但这种方案必须要在 Java 虚拟机运行前就准备好所有必须的配置信息,同时在程序正式运行前初始化所有的属性信息。

现在比较通用的注册方式是使用类路径中的 JAR 包来自动注册,该功能从 JDK1.3 开始引入^①。只需要把一个含有具体实现类的 JAR 包放置在类路径中,然后在 JAR 中的指定文件中以固化的配置方式声明要注册的具体类名,接下来该具体实现类就会在运行时被 JDK 自动发现并注册。像 Java 用来解析 XML 的工具包 (JAXP) 就是采用了这种方案。除此以外,在对外部公开时,API 的大部分内容都体现为接口,只有少数工厂类,比如 `DocumentBuilderFactory`。这些类的 Javadoc 表明,它们会按照特定的搜索路径来查找实现:浏览 `META-INF/services/pkg.name.className`^② 服务。

如果客户端的代码基于一个接口模块提供的 API 进行开发,那么在运行时至少需要一个该接口模块的具体实现模块来处理相应的功能请求,这样其工厂才能正常运作。当然,这个接口模块也可以自己来处理请求。而且,这个模块可以扮演一个注册中心的角色,开发商提供的多种实现,都可以注册到这个 API 模块中,由其代为处理功能请求。

真正模块化架构的设计方案通常会使用模块化类库,当然也可以使用简单的功能类库,只要用得合适就行。对于一个模块化系统,不应该通过复制类文件来解决多个开发方类库冲突的问题,这不算是好的解决方案,因为这通常会导致多个开发商间的类文件重复。

进一步说,这种所谓的复制类文件方案还存在其他的问题,如果规范的版本有所更新时,就会引发更多的问题,因为它无法与开发商的具体实现进行分离。你必须在每个版本中加以说明:这个 JAR 包提供了哪些规范和哪些实现,而模块化系统还必须理解这些文字说明的含义。

相比之下,在模块化系统中,将模块表现为类库的方式更加直接一些,不需要为“规范”或者“具体实现”提供特殊语义。这样只要一个独立模块对外提供很多接口,然后再加上一、两个工厂类,就可以用来定义一套“规范”,同时还可以提供一些文档对其进行说明。而“具体实现”则是一个没有直接对外暴露其功能的包,它只依赖于那个“规范”模块和其他的一些实现模块,然后,通过注册服务的方式,将自己实现的工厂注入到服务系统中。而一个“客户”也都可以看成一个模块,但它的代码只依赖于“规范”模块,并使用其工厂方法调用相应的功能,并不会直接依赖于某个具体实现模块。

① 这种服务注入的功能,其实在 Java 中用得非常多,包括 Tuscany 也用了,但不知道为什么不太为广泛的开发人员所知,直到 JDK 6,才通过 `java.util.ServiceLoader` 对外公开。有兴趣的读者可以通过阅读 <http://weblog.java.net/blog/2008/08/12/Simple-dependency-injection-serviceloader-jdk-6> 获得更多信息。

② `META-INF/services/pkg.name.className` 这里只是一个通用的例子,如 `javax.xml.parsers.DocumentBuilderFactory` 这个类对应的文件应该是 `META-INF/services/javax.xml.parsers.DocumentBuilderFactory`。——译者注

间接依赖和 NetBeans 使用的运行期容器

有时候,“间接依赖”也是必须的,这样可以允许“规范”模块在处理具体请求之前至少能够加载一个实现,否则就无法响应用户的功能请求。在 NetBeans 模块化系统中,采用了一种称为“provide-require tokens”的机制(类似于 Linux 系统中 RPM 包的依赖列表)。这样,客户端模块需要标识自己所依赖模块及其版本号,这样做一个模块就可以声明自己运行时的依赖环境。而相应的实现模块则是通过版本号来声明自己实现了哪一个特定版本的规范,这样“规范”模块和“实现”模块以及“客户端”模块就利用模块名称及版本号来建立相应的关系,比如像 `org.w3c.dom.v3` 就表示其可以替换 `org.w3c.dom.v2` 版本。有时候,“规范”模块也会依赖于一些“实现”模块^①,这也是正常的。

顺便说一下, JDBC 也在尝试使用这种模块化类库的解决方案,但它的使用方式一开始就存在问题。一个客户端必须要知道具体实现类的名称,这样才能加载相应的驱动,同时还要有相应的 `ClassLoader` 才能加载。如果程序的类路径是扁平的,即所有非 JDK 类都是通过相同的 `ClassLoader` 来加载,就没有什么问题,但如果遇到一个复杂的模块化系统,问题就大了。同时,注册驱动的方法是一个静态方法,使用了 Hash 表来存放驱动,也就表示,即使有一个驱动根本用不上,也无法卸载。所以,千万不要模仿这种处理方式! JDBC 的后续版本通过 `ConnectionPool` 和 `JNDI` 也许能解决这些问题,只不过这种方式对于一个 J2EE 程序来讲比较合适,而在不使用应用程序服务器的情况下就显得太过复杂了。

7.2 组件定位和交互

模块化的目的非常简单,就是要实现一个程序中各组成部分的松耦合。如果我们希望两个模块是独立的,那么两个模块彼此就不应该知道相互间的存在。或者像第 6 章所讲,他们之间应该通过具有良好定义的接口来进行交互。

对于一个要访问数据库的模块来说,用 `AbstractDatabaseService` 这样的抽象类肯定比直接拿个 `jdbc:mysql.mycompany.com` 要漂亮而且好用得多。通过定义 `AbstractDatabaseService` 这个抽象类,模块代码能够清晰地定义它所需的环境。如果代码写得没有问题,那么只要有人实现该服务并在适当的环境下将其传递给这样一个模块,则该模块就可以访问指定的数据库,这样做可以有效地提高可配置性和可测试性。事实上,它减少了面向对象语言中杂乱的代码。

当然,最后产品发布或者项目上线的时候,我们仍然希望模块是使用 `jdbc:mysql.mycompany.com` 来访问数据库的!基于面向接口编程的思想,我们可以将代码写得非常漂亮,但有时候还是需要解决具体的配置问题,就是说需要有人来准备模块的运行环境。所以需要有一种途径能够支持这个模块从环境配置中获取正确的 `AbstractDatabaseService` 实现,进而来访问正确的数据库。

通常这些运行环境的准备工作是由一个框架来完成的,像 Spring, Java EE5 等,它们会使用

^① 这里所说的“规范”模块依赖于“实现”模块不是指“规范”模块依赖于自己的“实现”模块,而是指“规范”模块依赖于第三方的某个“实现”模块。如 Acegi 依赖于 Spring 一样。——译者注

所谓的依赖注入来完成此类工作。这些框架通常会使用另外的一些配置文件来实现这一目标。每一个模块都要定义自身的依赖环境，比如说用 Annotation 来对运行环境进行注释性说明，也可以定义 set 方法或者是合适的构造函数。接下来，运行时整个程序会有一个负责集成的集成人员来配置数据库服务、相应的参数、还有其他的服务，这些内容都会被放置在这个框架的服务缓存池中。在运行时，这个框架再去搜索每一个模块查找其注入接口，再将相应的内容注入。换言之，它使用依赖注入的方式，将合适的实现内容放置到相应的模块中，完成初始化工作。

接下来用一个代码实例来说明这种方案具体是如何运行的，还会一并展示其他多种解决方案，与此同时，还会就各方案的优缺点、区别之处和相似性加以详细说明。现在，我们要将 NetBeans IDE 中的一个 Anagram 游戏程序转换为模块化类型。这个程序的主要功能是为用户随机显示一个经过扰乱处理的不规则单词，而用户则要将被扰乱后的单词纠正为正确的单词。遵守“面向接口而非实现编程”的原则，就很容易定义整个系统最核心三部分间的关系：一个类库用来生成原始的单词，一个业务逻辑层用来扰乱单词，还有一个用户界面模块负责将这个单词展示给用户看。下面是这三个模块的接口定义。

```
public interface Scrambler {
    public String scramble(String word);
}
public interface WordLibrary {
    public String[] getWords();
}
public interface UI {
    public void display();
}
```

我们的目标是帮助读者理解如何才能合理地使用模块库，如何才能将这些 API 解耦，并将每个模块的 API 接口放在不同的 JAR 包中。而这些接口的具体实现也会分置在不同的模块中，这些负责具体功能实现的模块只需要依赖于这个接口模块来提供这些 API 的实现类即可。注意，相同的接口可以有不同的实现，甚至可以多种实现共存。然而，对于本例，只需要有一个定义 StaticWordLibrary 的实现模块，一个定义 SimpleScrambler 的实现模块，还有一个基于 Swing 来实现 Anagrams 用户界面的实现模块。当然，也可以定义其他的实现方式比如说：一个从指定文件中读取单词的单词库；一个不限于交换两个字母的单词混淆方式；用户展现方式也不限于 Swing，可以有一个命令行界面。只不过，如果将这么多的方式都放进来，会使例子变得太复杂，而前面已经定义好的三个模块也足够描述这么多种注入方式了。简单地分析一下，就会发现这个程序需要使用注入功能。像用户界面模块就需要使用单词库来提供要扰乱的单词，还需要混淆模块来扰乱单词。所以现在会设计一个 Anagrams 的基类，它不需要说明具体如何来获取这些功能，它只简单地定义两个抽象方法，表示自己的基本功能就是扰乱一个单词的排序，然后让具体的模块来获取不同的技术实现这些抽象方法。

```
public abstract class Anagrams extends javax.swing.JFrame implements UI {

    protected abstract WordLibrary getWordLibrary();
    protected abstract Scrambler getScrambler();
}
```



```

    public void display() {
        initWord();
        setVisible(true);
    }
}

```

为了凸显模块化的意义,实现这些方法的最简单方案就是在整合模块的 API 中或集成模块中创建一些 registerXYZ 方法。为保证模块化并支持配置系统的多个实例,需要让这些方法接受类作为参数。因为使用类作为参数,所以要求注册进来的类,如 WordLibrary 和 Scrambler 的子类要有默认构造函数,对于用户界面的类来讲,则需要提供一个有参构造函数,该构造函数包含有两个参数,分别是 WordLibrary 和 Scrambler。对于很多框架来说,有参构造函数是一种常用的注入方式。除此之外,这些框架通常还允许通过 set 方法来注入依赖。只不过对于本章中这个注册的例子,使用有参构造函数会比较简单。

```

public final class Launcher {
    private static Class<? extends WordLibrary> wordLibrary;
    private static Class<? extends Scrambler> scrambler;
    private static Class<? extends UI> ui;

    private Launcher() {
    }

    public static void registerWordLibrary(
        Class<? extends WordLibrary> libraryClass
    ) {
        wordLibrary = libraryClass;
    }

    public static void registerScrambler(
        Class<? extends Scrambler> scramblerClass
    ) {
        scrambler = scramblerClass;
    }

    public static void registerUI(Class<? extends UI> uiClass) {
        ui = uiClass;
    }

    public static UI launch() throws Exception {
        WordLibrary w = wordLibrary.newInstance();
        Scrambler s = scrambler.newInstance();
        return ui.getConstructor(
            WordLibrary.class, Scrambler.class
        ).newInstance(w, s);
    }
}

```

```
public class AnagramsWithConstructor extends Anagrams {

    private final WordLibrary library;
    private final Scrambler scrambler;

    public AnagramsWithConstructor(
        WordLibrary library, Scrambler scrambler
    ) {
        this.library = library;
        this.scrambler = scrambler;
    }

    @Override
    protected WordLibrary getWordLibrary() {
        return library;
    }

    @Override
    protected Scrambler getScrambler() {
        return scrambler;
    }
}
```

如果这个游戏的 Launcher 也是这个程序的 API 的一部分，外部代码基于该接口进行编码，那么现在这个游戏程序可以说是一个非常漂亮的模块化程序了。这几个 API 接口的具体实现可以独立完成，没有任何关联关系，只有 Launcher 这个类知道如何在运行时绑定具体实现，这个绑定过程非常灵活，只需要在使用 Launcher 之前，有人把相应的工厂类注册进来即可，比如说有一个负责整合操作的开发人员使用相应的代码进行初始化工作，然后再调用相应的所有 registerXYZ 方法来完成注册工作，整个程序的运行环境就配置完成了。这种配置方式并不难，只不过代码看起来不是很漂亮，因为很多负责整合操作的开发人员并不喜欢用代码的方式来完成配置工作。他们希望什么都不写，就可以自动配置，如果一定要做点工作才能整合程序的话，他们更倾向于用配置文件的方式来替代手工硬编码。

另外，这种注册方式同样存在过程化注册的所有缺点，第 12 章中会对此有详细讲解，该方式也会降低系统的启动速度。假设有很多实现类要注册到系统中，而且会调用很多次注册方法，那么系统的启动速度就会很慢。所有的实现模块都要在应用程序启动的时候被调用，然后调用相应的方法以完成注册。相应的操作包括系统从多个模块中加载类，然后进行连接和执行，所以效率非常低效。但要做一件事总要付出代价，毕竟没有免费的午餐。

通用的注册方式

如何使用 registerXYZ 方法是显而易见的。这正是为何将这种方式作为环境配置的首选方案的原因。事实上，NetBeans 一开始的时候也是使用这种方法，但我们需要非常多的注册方法，我们很快意识到大部分注册方法的内部实现其实都差不多，除了它们处理的类型可能有所不同以外，其他内容简直像是一个模子里刻出来的。这就是我们想方设法地试图用一种比较

通用的方式来替换原有的单个注册方法的原因。

```
private static Map<Class<?>,Object[]> instances =
    new LinkedHashMap<Class<?>,Object[]>();
public static void registerClass(Class<?> impl) {
    instances.put(impl, new Object[1]);
}

public static <T> T find(Class<T> whatType) {
    for (Map.Entry<Class<?>, Object[]> entry : instances.entrySet()) {
        if (whatType.isAssignableFrom(entry.getKey())) {
            if (entry.getValue()[0] == null) {
                try {
                    entry.getValue()[0] = entry.getKey().newInstance();
                } catch (Exception ex) {
                    throw new IllegalStateException(ex);
                }
            }
            return whatType.cast(entry.getValue()[0]);
        }
    }
    return null;
}
```

最终，我们决定只用一个通用的方法来持有所有注册到系统中的内容，这样就不用写上很多注册方法，从而消除了重复的代码。这种方案能够很好地保证注册方式的一致性，也为现在 NetBeans 的方案奠定了基础。

最好不要使用那种直接调用代码来绑定服务的方式，而是使用一种声明的方式进行服务绑定，这样就不需要去写具体的代码，而是以更加说明性的方式来注册。常用的声明方式就是使用 `System.getProperty("...")` 来完成。这种处理方式在 JDK 中非常普遍，也很好用，而且它不需要通过调用具体代码来注册，而是使用后捆绑。这样只有到真正代码执行的时候，具体的实现类才会被虚拟机所加载。

```
@Override
protected WordLibrary getWordLibrary() {
    try {
        if (wordLibrary == null) {
            String implName = System.getProperty(
                "org.apidesign.anagram.api.WordLibrary"
            );
            assert implName != null;
            Class<?> impl = Class.forName(implName);
            wordLibrary = (WordLibrary)impl.newInstance();
        }
        return wordLibrary;
    } catch (Exception ex) {
        throw new IllegalStateException(ex);
    }
}
```

```

}

@Override
protected Scrambler getScrambler() {
    try {
        if (scrambler == null) {
            String implName = System.getProperty(
                "org.apidesign.anagram.api.Scrambler"
            );
            assert implName != null;
            Class<?> impl = Class.forName(implName);
            scrambler = (Scrambler)impl.newInstance();
        }
        return scrambler;
    } catch (Exception ex) {
        throw new IllegalStateException(ex);
    }
}
}

```

在这种解决方案中，仍然需要程序的系统集成人员进行一些配置工作。只不过与那种直接调用 registerXYZ 的方案相比，就不用写代码了，只需要写一个属性配置文件，程序在启动时或者启动之前会读取该文件，再根据配置内容来选择不同的组件。这样对于程序的最终发布人员来说，不用去写代码，再编译和执行代码，只需要编辑一下属性配置文件即可。只不过对于系统集成人员来说，仍然要知道实现类的全名，所以这个操作还是涉及太多的细节内容，对整个系统不甚了解的人来说，这个操作仍然是件很困难的事情。

除了使用属性配置的方式，还有很多其他的注册形式。像著名的 Spring 框架，使用的就是基于 XML 配置文件的方式。

```

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
    >

    <bean
        id="wordLibrary"
        class="org.apidesign.anagram.wordstatic.StaticWordLibrary"
    />
    <bean
        id="scrambler"
        class="org.apidesign.anagram.scramblersimple.SimpleScrambler"
    />
    <bean
        id="ui"
        class="org.apidesign.anagram.gui.AnagramsWithConstructor"
        autowire="autodetect"
    />
</beans>

```


这种使用 XML 文件的方式对于那些不是很了解整个系统的集成人员来说是比较好的。XML 文件的编辑难度与属性文件差不多，而且还有更加丰富的配置方式，比如说可以指定在依赖注入时是使用 set 方法还是构造函数，或者可以选择自动组装的方式等。XML 文件编辑起来很容易，而且还可以使用 Schema 对其格式进行明确规定。常用的 IDE 都可以对 XML 文件的编辑提供代码提示和自动完成功能，因此即使相关的集成人员不是很了解 XML 格式，也能够根据自己的需要来修改 XML 文件。Spring 会用 ApplicationContext 接口的具体实现来读取该 XML 文件，然后负责对所有的 bean 进行初始化工作，并进行装配^①。

```
public static void main(String[] args) throws Exception {
    ApplicationContext context = new ClassPathXmlApplicationContext(
        "org/apidesign/anagram/app/spring/conf.xml"
    );
    UI ui = (UI)context.getBean("ui", UI.class);
    ui.display();
}
```

Spring 框架的解决方案较那种使用属性配置的方案更加灵活，只不过它看上去仍然要求应用程序的集成人员要知道相应 JavaBean 的类名，以便进行装配。有时候这很有用，特别是对于那些自己负责具体实现的开发人员来说。但对于集成人员来说，要知道相应的类名，需要对系统有较多的了解。另外，采用这种方案，那么每个模块需要把相应的实现类公开给那些负责最终集成的人员，所以这些公开的类其实也就成了 API 的一部分。第 6 章一直在强调要将接口与具体实现进行剥离，所以应该避免直接涉及具体的实现类。

最完美的组件注入方案

Spring 框架提供的解决方案非常有效，已经成为了一种标准，如果听到谁说“依赖注入”，通常都是指 Spring 提供的这种方案。如果说某种产品成为某一种完整技术的代名词，那么 Spring 就是典范了。可以从两个方面来理解这件事，首先，它为“依赖注入”给出了定义，让我们一听到这样的词立即就想起 Spring。其次，它也有其局限性，如果我们在注入和 Spring 之间划上等号的话，我们的思路就会受限，可能无法再去思考其他更加有效的解决方案了。

上面所说的其实是一种把抽象定义的内容与具体内容混为一谈的情况，这种情况不是只有在编写程序的时候才会出现。就在我们身边的生活中，它其实随时随地都有可能发生。在我的国家，一提到 Lux^②就会想起吸尘器，不仅仅是因为生产吸尘器的公司名字是以 Lux 结尾的。还有，其实任何一种深色的苏打水都可以称为 Coke，但听到 Coke，都会想到可口可乐或者百事可乐。再举一个物理方面的例子——时间，只要一提到“时间”，大家就会想到时、分、秒，但它是时、分、秒吗？这只是牛顿用来衡量“时间”的一种表示方法，但现在我们大部分人都不会用时、分、秒来表示时间。要知道，时间这个概念对我们的日常生活是非常有用的。但要注

① 在 Spring 中提供了一个 ApplicationContext 的接口，表示将其用来管理相应的 JavaBeans，为了支持多种配置文件的加载，它还提供了多个实现类，如 ClassPathXMLApplicationContext 等支持其功能。——译者注

② 这是指的应该是 Electrolux，中文品牌名为伊莱克斯，来自意大利，为世界著名家电品牌。——译者注

意，对“时间”的定义让我们不能很清楚地理解：这个世界上对于时间的描述方式其实还有有很多种，不仅仅只有时、分、秒，爱因斯坦的相对论和很多现代科学理论都给出了其他的方式来衡量时间。牛顿对“时间”的定义是那么明确、那么实用，以至于我们都意识不到“时间”在其他语境中的含义了。

一定要以更高层次的思路来看抽象内容。比如说关注一下 Spring 框架在什么时间进行注入操作，同时还有意识到 Spring 这种方案的潜在局限性。不能因为 Spring 框架有效地解决了组件注入的问题，就盲目地认准这种方案，而不去思考其他更好的解决方案。

这正是为什么我们仍然要寻找更多更好的方式来实现依赖注入的原因。我们希望有一种解决方案，可以让相关的集成人员在不了解整个系统的情况下也能完成工作。幸运的是，Spring 框架 2.5 版中，注入的解决方案有了长足的进步，可以使用 Annotation 的方式来实现注入。

```
@Service
public class SimpleScramblerAnnotated extends SimpleScrambler {
    public SimpleScramblerAnnotated() {
    }
}
@Service
public class StaticWordLibraryAnnotated extends StaticWordLibrary {
    public StaticWordLibraryAnnotated() {
    }
}
@Service("ui")
public class AnagramsAnnotated extends AnagramsWithConstructor {
    @Autowired
    public AnagramsAnnotated(WordLibrary library, Scrambler scrambler) {
        super(library, scrambler);
    }
}
```

可以通过 Annotation 来标识要导出的 JavaBean，像该 JavaBean 的名称及它们对其他 API 的依赖关系，都可以通过 Annotation 公开出去，这样就能找到正确的实现将其注入。基于这种方案，最终负责集成工作的人就不用去写配置文件，也不用在配置文件中写上一大堆实现类的全名，使用 Spring 提供的 Annotation 是一种更好的解决方案。

```
<beans
xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd"
>
    <context:component-scan base-package="org.apidesign.anagram.app"/>
</beans>
```

虽然 Spring 2.5 对注入做了大量的简化,但在编写配置文件时,仍然需要知道相应的含所有实现的包名。但这与原先需要了解所有的类及其参数相比,已经简单多了,这几乎已经是我们最初设想到的最佳方案了。集成人员只需要选择必要的类库,包括提供 API 的类库和提供具体实现的类库,再编写一下相应的配置文件并在此基础上启动程序就可以了。与原先要使用 registerXYZ 方法的方案比起来,这真是一个巨大的进步。但集成人员还是需要去写这些配置文件,到底能不能完全避免这类配置工作呢?

通过对 Spring 注入方案进行一些扩展,是可以做到完全没有配置文件的。也许 Spring 框架的后续版本本身也会提供这样的扩展。现在让我们来换一个思路来看其他的组件注入方案。先来看一种集合了通用的注册方式和基于属性配置的注册方式优点的方案。这种注册方案使用了 JDK 1.3 版本提供的扩展机制,客户端通过 Java 6 的 `java.util.ServiceLoader` 类或 NetBeans 平台提供的查找框架可以使用这种方案。

注入术语

前面举的例子说明要解决注入问题其实有很多可行方案供选择。但我过去常常把这些方案全部归入依赖注入这一类中。然而,依赖注入的规范比我想像中要严得多。它是反转控制 (Inversion of Control, IOC) 的一种特例,大家都知道好莱坞的电影法则:不要来找我,我会来找你,这正是 Spring 解决方案的原理。但其他的方案,像调用 registerXYZ 的注册方法、使用属性配置,或者使用 Java 扩展机制,都算不上是 IOC 方案。只不过,这些注入自有其成功应用的场合,所以我决定把这些方案都称为“组件注入”。“组件”这个词这个说明技术中要操作的对象是什么,而“注入”作为一个流行词,也是非常重要的,有助于吸引开发者的眼球。

2001 年,我们给 JavaOne 大会提交了一份名为“组件定位与协作”的议题,该议题中的内容与本章多少有些关系,我们认为该议题涉及的内容对于所有的模块化程序都很重要。然而,当时 JavaOne 会议的组委会可能仅仅因为觉得这个议题的名称不够酷,或者可能因为他们觉得组件这个词用得太多滥了,要知道“组件”这个词几乎包含了所有可以想象到的内容,所以没有接受该议题。直到 2006 年,我和同事 Tim Boudreau 又提交了一份类似的议题,名为“模块化架构中那些发现注入和依赖注入的模式”。果然如我们所料,议题被组委会接受了。这说明一定要找到能够贴近目标人群的术语才能吸引他们。一旦听到“依赖注入”这个词,他们的心就被打动了。就像 API 这个词一样,恰当的名称有益交流。对于用户来说,他们越容易理解 API 这个词,就越容易接受 API 的相关内容。

常见的组件注入方案在一般原则上没有太大的区别。都是通过定义抽象的服务,要求开发方提供具体的服务实现,然后以注册的方式将服务实现放到缓存池中。然而 Spring 框架是通过外部的操作进行注入处理的。但不管哪种具体的处理方式,都需要先找到抽象定义的入口,如 static 方法,才能将具体的实现放进相应的入口,从而提供服务。在服务被调用以前,框架需要将具体的实现正确放置到位,这可以通过在类字段上添加 Annotation 或者通过配置文件结合 set 方法和

构造函数来完成。最终，框架就可以安全地将应用程序所需的资源全部准备好，然后开始正式运行程序。

NetBeans 使用的 Java 扩展机制与 Spring 不同，可以说这是两套完全不同的解决方案。Lookup 是一个很小的类库，没有使用 Spring 这类的框架。但整个程序都是建立在 Lookup 机制上，而且是基于懒加载机制，只有在必要时，才会调用相应的代码。Lookup 机制能够准确地部署各种实现，在这一过程中，既不需要通过访问类的元数据，也不需要配置文件，整个过程只需要调用 Lookup 提供的方法即可。

```
import org.openide.util.LookupEvent;
import org.openide.util.LookupListener;
class AnagramsWithLookup extends Anagrams {

    public AnagramsWithLookup() {

    }

    @Override
    protected WordLibrary getWordLibrary() {
        return Lookup.getDefault().lookup(WordLibrary.class);
    }

    @Override
    protected Scrambler getScrambler() {
        return Lookup.getDefault().lookup(Scrambler.class);
    }
}
```

第 12 章中提到两个 API 之间虽然没有使用声明式的依赖关系，但基于这个方式可以在运行时为两者建立关系，上述代码也同时体现该方案的利弊之处。要特别说明的是，这样做会使得程序的静态分析变得非常困难，根本无法做到通过分析一个程序就能知道其所有的注入口。但从另一角度来说，你对整个程序有了更强的控制能力。每一次动态调用 Lookup 都会建立一个服务入口。然后该库就会在运行环境中查找相关服务，并为可用的具体服务建立缓存池，然后再根据需求返回具体的服务。把 API 嵌入到这个库中简化了设置，因为你不需要在程序运行和服务查询之前执行代码进行初始化工作。因为每一个入口都只是一个方法的调用，所以在代码执行以前，并不需要进行注入工作。这就表示整个程序的配置变成了一项非常简单的工作。

另外要提一下，不管是谁来负责最终程序的集成工作，相应的配置都是非常简单的。因为 Lookup.getDefault() 是基于 JDK 的扩展机制来实现的，用不到任何的配置文件，只需要把类路径配置正确就可以了。像下面的代码就展示了如何通过 Lookup 来查找 org.apidesign.anagram.api.WordLibrary 服务：

```
Lookup.getDefault().lookupAll(org.apidesign.anagram.api.WordLibrary.class)
```

当这个方法被调用时，就会在当前的类路径中查找所有声明了 META-INF/services/org.apidesign.anagram.api.WordLibrary 的服务，然后对类名进行分析，如果有合适的类名，那么根据类名来初始化相应的类，最后返回类的实例。

组件注入或服务定位

一些纯粹的技术人员会认为这种解决方案只不过是一种服务定位的模式。的确如此，对 Lookup 的调用看起来就是这样一种模式。但如果仅限于此的话，也就算不上注入了。最重要的是它能够在类路径中进行查找。只有做到这一点，组件才能真正地注入到这个类中，才能成为应用程序的一部分^①。

还有，利用最终应用配置，这种方案可以很容易地使用一种服务的具体实现来替换另外一种服务的具体实现，与此同时，不需修改那些使用服务的代码。这样对单元测试就非常有用，因为在测试一个对象的时候，很容易使用 Mock 出来的服务实现来完成测试^②。

```
@Test public void testInjectionOfServices() throws Exception {
    Anagrams ui = create();

    assertNull("No scrambler injected yet", ui.getScrambler());
    assertNull("No scrambler injected yet", ui.getWordLibrary());

    MockServices.setServices(
        ReversingMockScrambler.class, SingleMockLibrary.class
    );

    Scrambler s = ui.getScrambler();
    assertNotNull("Now we have scrambler", s);
    assertEquals(
        "It is the mock one", ReversingMockScrambler.class, s.getClass()
    );
    WordLibrary l = ui.getWordLibrary();
    assertNotNull("Now we have library", l);
    assertEquals(
        "It is the mock one", SingleMockLibrary.class, l.getClass()
    );

    ui.display();

    assertEquals(
        "The word from SingleMockLibrary is taken",
        "Hello World!", ui.getOriginalWord()
    );
    assertEquals(
        "The word is rotated using ReversingMockScrambler",
        "!dlroW olleH", ui.getScrambledWord()
    );
}
```

① 这里作者的意思是指这种方式可以自动从类路径中发现新的实现，而不是局限于原有的实现。——译者注

② 这里的服务通常是指一个接口（当然也可以是类），然后通过服务定位的方式，找到实现了该接口的类，这样服务定位就可以根据配置文件或者其他方式来动态提供实现类，而不是在代码中固化实现类。也就意味着对接口编程，从而很容易用一个服务替换另外一个服务。——译者注

```

public static final class ReversingMockScrambler implements Scrambler {
    public String scramble(String word) {
        return new StringBuilder(word).reverse().toString();
    }
}

public static final class SingleMockLibrary implements WordLibrary {
    public String[] getWords() {
        return new String[] { "Hello World!" };
    }
}

```

这些才是依赖注入真正要完成的功能。事实上 Lookup 作为一个注入方案，不管注入的方式如何变化，都能满足要求。

这种类型的组件注入不仅简单，易于配置，而且还遵守了 Java 的标准。如先前所说，JDK 1.3 在 META-INF/services 命名空间中定义了注册格式，从那以后，JDK 使用这种方案解决了很多问题，如 XML 解析和转换。同时 JDK 6 版本也定义了新的 API^①，用来查询对于某种服务的所有注册实现。

```

class AnagramsWithServiceLoader extends Anagrams {

    public AnagramsWithServiceLoader() {
    }

    @Override
    protected WordLibrary getWordLibrary() {
        Iterator<WordLibrary> it;
        it = ServiceLoader.load(WordLibrary.class).iterator();
        return it.hasNext() ? it.next() : null;
    }

    @Override
    protected Scrambler getScrambler() {
        Iterator<Scrambler> it;
        it = ServiceLoader.load(Scrambler.class).iterator();
        return it.hasNext() ? it.next() : null;
    }
}

```

这样，如果使用 Java 6，那么就可以在所有的程序中都使用这种组件注入，完全不再需要第三方的库或者框架了。

Spring 和 Lookup 机制

很明显，利用 Lookup 很容易建立一个真正依赖注入的框架。从某种意义上讲，Lookup 只是一种可以实现注入的技术。就像是一种支持注入方案的元汇编语言，不管哪一种方案都可用

① 这里指的就是前面说到的 java.util.ServiceLoader。——译者注

使用 Lookup 来实现注入，就像 IOC 只是一个概念，可以有多种实现方案。

事实上，我们可以通过桥接的方式将 Spring 的 ApplicationContext 和 Lookup 进行混用。如果你拿到了一个 Lookup 的实例，可以在 ApplicationContext 的内部调用它，当然，反之亦然。这一点清楚地展示了这两种技术令人惊叹的扩展性。不仅如此，这还意味着你可以将任一个的其中部分功能与另外一个混用。比如说，你可以在 Spring 的 XML 配置文件中来定义 JavaBean，然后通过 Lookup 进行访问。或者，你可以通过 META-INF/services 来注册服务，然后在 Spring 的依赖注入框架中使用这些服务。几乎是有无数种可能的互操作方式，如果想了解更多，可以访问 <http://injection.apidesign.org>。

因为新版本的 JDK 已经提供了 ServiceLoader 这样的 API，也许有人要问还有什么理由来继续使用 Lookup 呢。原因有 3 点，首先，ServiceLoader 这个 API 只在 JDK 6 及以后的版本中才有，而 NetBeans 提供的 Lookup，可以用在 2001 年以后的所有 JDK 版本上。所以，如果你用的 JDK 是老版本，就只能选择 Lookup 了^①。

另一个选择 Lookup 的理由是要考虑基于 NetBeans 平台开发的应用程序应该具有的动态性：NetBeans 可以在运行时动态添加或者移除模块，所以对于基于 NetBeans 平台而构造的应用程序，其实际的类路径在运行时产生变化是很平常的。这也说明，通过 META-INF/services/这种命名空间注册的资源也有可能改变。如果产生了这种变化，通过 Lookup.getDefault 查询得到的结果就会受影响。如果可用的服务产生了变化，有很多相关的内容都需要进行刷新。比如说，有一个用来支持 CVS 版本控制的模块被卸载了，那么就需要对被该模块控制的文件进行刷新。这表示当有内容改变时你必须得到通知。因此，Lookup 类库引入了 Listener 以支持事件通知。

```
private Lookup.Result<WordLibrary> libraries
    = Lookup.getDefault().lookupResult(WordLibrary.class);
private LookupListener listener = new LookupListener() {
    public void resultChanged(LookupEvent ev) {
        initWord();
    }
};
{
    libraries.addLookupListener(listener);
}
```

因此，支持事件监听是用户选择 Lookup 而非 ServiceLoader 的第二个原因。JDK 仍然把自己局限在一个静态应用程序的世界里，它支持动态加载（如 NetBeans 所示），但并非主流功能。因此，对于 ServiceLoader 这个 API，就没有提供监听功能。同样，传统的依赖注入框架也不能轻易地实现或者支持监听功能。这些框架通常都是在系统初始化配置时将所有的服务准备好以便随时调用。从另一个角度来说，使用监听器和动态模型，就可以随时随地产生改变，你必须随时准备好监听相应的事件，才能对其进行处理。如果涉及此类情况，Lookup 可能是更合适的解决方案。

① 其实这个类的功能很简单，很容易就可以移植到 JDK 5 及更低的版本上。——译者注

第三个，也是最后一个选择 Lookup 而非 ServiceLoader 的理由，是因为 Lookup 提供了两类 API：一类 API 为客户端代码服务，用来查询已经注册的服务，还有一类 API 则为开发者服务，方便他们编写自己的服务池，定义如何注册和获取服务。这也意味着可以把 Lookup 的 API 看作是一个用来在两个不同的组件间进行独立通信的框架。事实上，这也是远程接口通信的另一个例子，见图 7-1。



图 7-1 Lookup 机制中的服务双方进行通信

在图中的一侧，有一个调用程序要寻找相应的服务，并使用 Lookup 的 API 来得到相应的服务。而在另外一侧，则可以看到服务提供方。而在 Lookup API 则位于两者中间，它既不了解左侧的调用，也不了解右侧的服务，但仍然可以将两方进行互联，同时还能够保证类型的安全。相应的类型从一方传递到另外一方，但此过程并不对外公开，是看不见的。这样定义出来的 API，有着良好的扩展性。

7.3 编写扩展点

用户经常会问如何使用 Lookup 来定义扩展点，所谓的扩展点就是一个抽象服务，其他模块可以通过实现该服务来为原来的基础模块提供功能扩展。

假设我们现在有多个模块，其中有一个模块用来提供核心功能，而其他模块则是用来提供扩展功能。怎样才能正确实现这样的系统呢？在 NetBeans 中应该如何使用 Lookup 来声明扩展点呢？

首先要在核心模块中声明一个扩展接口，以 API 的形式公布出去，这表示这个接口对外公开，允许外部通过实现该接口来扩展功能。（对于 NetBeans Runtime Container 而言，这个接口所在的包将会声明为 Public，以便对外公开。）假设这个模块是想要显示这一天中由其他模块提供的小贴士。所定义的扩展接口如下。

```
package org.apidesign.extensionpoint.api;
```

```
public interface TipOfDay {
    public String sayHello();
}
```

当核心模块需要执行每日一贴功能时，它会从整个 NetBeans 系统中得到所有 TipOfDay 的实现类，然后从中随机选择一个来调用。

```
Collection<? extends TipOfDay> all =
    Lookup.getDefault().lookupAll(TipOfDay.class);
List<TipOfDay> arr = new ArrayList<TipOfDay>(all);
```



```
Collections.shuffle(arr);

String msg;
String title;
int type;
if (arr.isEmpty()) {
    msg = "I do not know what to say!";
    title = "No provider registered";
    type = JOptionPane.WARNING_MESSAGE;
} else {
    msg = arr.get(0).sayHello();
    title = "Selected from " + arr.size() + " providers";
    type = JOptionPane.INFORMATION_MESSAGE;
}
```

这样就可以显示一条每日一贴了，非常简单。基于 Lookup 机制可以很简单地实现各种服务的注册，并让其他模块以扩展点的方式来注入新的扩展功能以加强系统功能。但很明显，这种功能上增强也需要其他方面的配合。每一个模块，如果想为系统添加新的每日一贴功能，那么就要实现 TipOfTheDay 这个接口，自然需要依赖于这个核心模块，这样才能访问 TipOfTheDay 这个接口，不管是编译还是执行，都需要在相应的类路径上存在 TipOfTheDay 这个接口。只有这样，才能为该模块提供每日一贴的功能。

```
package org.apidesign.extensionpoint.impl2;

import org.apidesign.extensionpoint.api.TipOfTheDay;

public class HelloWorld implements TipOfTheDay {
    public String sayHello() {
        return "Hello World!";
    }
}
```

在完成了上面的代码以后，接下来要做的事情就很简单了，只要按照 Java 标准版本指定的格式将功能注册到系统中即可：编写一个文本文件 org.apidesign.extensionpoint.api.TipOfTheDay 放在 META-INF/services 目录下面，文件只包含如下所示的一行内容，最后将文件同代码一起打在 JAR 包中。

```
org.apidesign.extensionpoint.impl2.HelloWorld
```

通过这样的扩展点机制，可以将模块进行关联。相关的配置完全由最终程序的集成人员来管控，因为其行为取决于实际运行的路径。只需要使用包含不同 TipOfTheDay 实现的 Jar 包，其行为也会随之变化。

7.4 循环依赖的必要性

有些系统在运行时是允许循环依赖的，这很有用，特别是对那些遗留代码更是如此。从另一方面来讲，如果在新写的代码中也使用循环依赖的话，简直就像一堆缠在一起的意大利面条，类

和类，模块和模块之间复杂的引用关系，想想都令人头痛。这正是 NetBeans Runtime Container 不允许出现循环依赖的原因。不是所有人都喜欢这种一刀切的处理方式。但对于一个容器来讲，其规则定义得越严格，其用户基于该容器进行程序开发的系统架构就越清晰。事实上，不允许循环依赖正是 NetBeans 对付那种面向对象的意大利面条代码所做出的最大贡献之一^①。特别是对于新开发的代码，一定要避免出现循环依赖即使是那些遗留的代码，如果要重写这些老代码，也应该尽量避免循环依赖，下面会对此加以详细说明。

模块间不应该出现像面向对象的意大利面条代码一样的相互依赖。但不管我们怎么努力，这种事情总是不能完全避免，时不时地就会出现。只要有一点儿东西没有注意到，就会出现这种问题。没过多久，系统就已经乱成一团了，各个模块间一堆复杂的引用关系。还有更坏的情况就是这种依赖关系竟然还会出现循环。整个系统变得难以维护、编译和理解。最后只得把所有的代码都放到一个包里来处理，这样就完全丧失了将系统分成多个模块的好处。

NetBeans 的 Lookup 机制或者是 Java 6 引入的 ServiceLoader 机制，作为组件注入机制都可以避免在编译时的循环依赖，从而做到模块间的分离。这种做法不一定能解决所有人的问题，但每个人都有自己的想法，如果你对意大利面条式的设计恋恋不舍，那你仍然可以把所有的代码都放置在一个巨大无比的模块里。但只要你决定将一个程序进行模块化，那么一定要保证在开发时模块间不会出现循环依赖。

有过编写大型程序经验的开发人员，应该都想把程序进行模块化。下面就是这样的例子，这是遗留代码中分离出来的两个包。

```
package org.apidesign.cycles.array;

import java.io.IOException;
import java.io.OutputStream;
import org.apidesign.cycles.crypt.Encryptor;

public class MutableArray {
    private byte[] arr;

    public MutableArray(byte[] arr) {
        this.arr = arr;
    }

    public void xor(byte b) {
        for (int i = 0; i < arr.length; i++) { arr[i] ^= b; }
    }

    public void and(byte b) {
        for (int i = 0; i < arr.length; i++) { arr[i] &= b; }
    }
}
```

^① NetBeans 不允许循环依赖是指在开发时，不允许两个模块间出现循环依赖，如 A 依赖于 B，而 B 又依赖于 A，而这种情况在程序运行的时候却有可能出现。——译者注

```
public void or(byte b) {
    for (int i = 0; i < arr.length; i++) { arr[i] |= b; }
}

public void encrypt(OutputStream os) throws IOException {
    Encryptor en = new Encryptor();
    byte[] clone = (byte[]) arr.clone();
    en.encode(clone);
    os.write(clone);
}
}

package org.apidesign.cycles.crypt;

import org.apidesign.cycles.array.MutableArray;

public final class Encryptor {
    public void encode(byte[] arr) {
        MutableArray m = new MutableArray(arr);
        m.xor((byte)0x3d);
    }
}
```

上面的例子很简单，而且也有一些凑数的感觉，但还是足以说明问题的。可以看到有两个独立的包，它们之间存在着相互调用的关系。包 `org.apidesign.cycles.crypt` 中的 `Encryptor` 用来为数组加密。而包 `org.apidesign.cycles.array` 中 `MutableArray` 则在 `Encryptor` 的 `encode` 方法中被调用。不管怎么看，这都是意大利面条式代码的杰作。

上述例子中的代码是否可以分成两个 NetBeans 的模块呢？这两个模块是否可以消除原来的循环依赖呢？没问题，只要使用 Lookup 的组件注入机制来代替现在的类路径依赖即可。要知道一个模块依赖于另外一个模块，是非常普遍的。第一件要做事情就是先研究一下哪一个模块是依赖者，哪一个模块则是被依赖者。细看前面的示例代码，看起来 `Encryptor` 类应该是提供服务的，所以很自然地就应该让 `org.apidesign.cycles.crypt` 依赖 `org.apidesign.cycles.array`，如图 7-2 所示。

对于类 `Encryptor`，不需要进行修改，因为不管是编译还是具体运行的时候，它都依赖于 `MutableArray` 所在模块，可以直接使用 `MutableArray` 类。但 `MutableArray` 类在编译的时候根本就不能直接使用 `Encryptor` 类了，当然，在程序真正运行的时候，还是能够（实际上也是必须）使用 `Encryptor` 类的，因为这是执行环境中的必要的一部分。由于无法使用 `Encryptor` 类，所以 `MutableArray` 类中的 `Encrypt` 方法就必须要进行修改。到底如何修改才能保证在开发时，即使不直接使用 `Encryptor` 类也能编译通过，转而在运行时再建立这样的依赖呢？这里需要提供一个新的接口，通过这个接口调用原来 `Encryptor`。把这个对外的接口放置在 `org.apidesign.cycles.array` 模块上，然后在 `org.apidesign.cycles.crypt` 模块来实现该接口。最后通过 Lookup 来找到具体的实现，下面是修改后的代码。

```

public class MutableArray {
    private byte[] arr;
    public MutableArray(byte[] arr) {
        this.arr = arr;
    }

    public void xor(byte b) {
        for (int i = 0; i < arr.length; i++) { arr[i] ^= b; }
    }

    public void encrypt(OutputStream os) throws IOException {
        DoEncode en = Lookup.getDefault().lookup(DoEncode.class);
        assert en != null : "org.netbeans.example.crypt missing!";
        byte[] clone = (byte[]) arr.clone();
        en.encode(clone);
        os.write(clone);
    }
}

package org.apidesign.cycles.array;
public interface DoEncode {
    public void encode(byte[] arr);
}

package org.apidesign.cycles.crypt;
import org.apidesign.cycles.array.DoEncode;
public class DoEncodeImpl implements DoEncode {
    public void encode(byte[] arr) {
        Encryptor en = new Encryptor();
        en.encode(arr);
    }
}

```

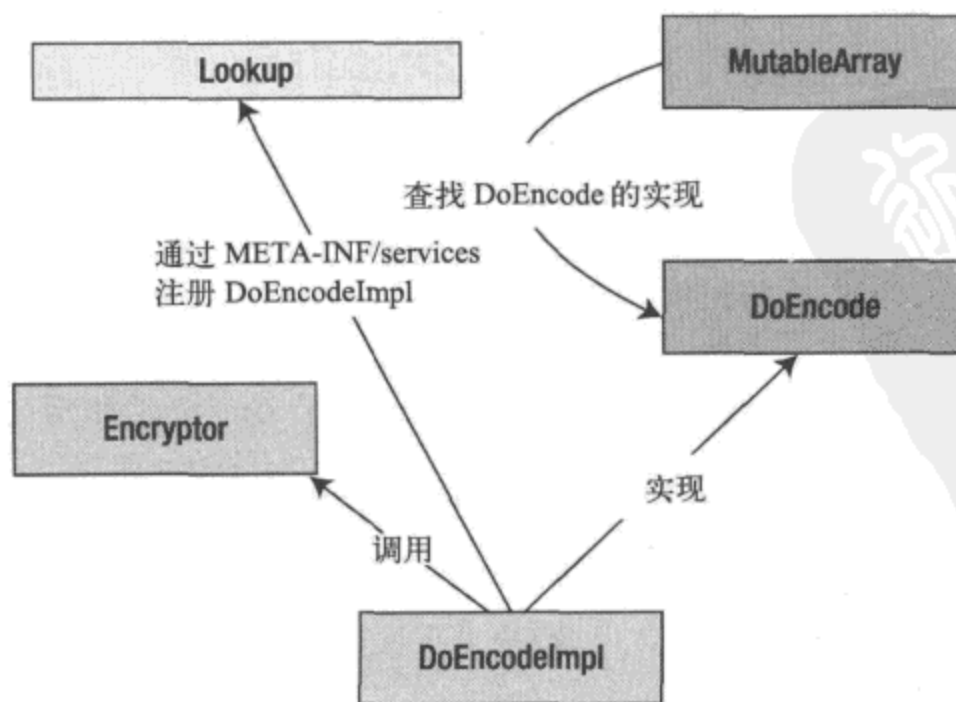


图 7-2 分成两个模块的应用

我们把这个对外的接口命名为 `DoEncode`，然后通过该接口的子类来真正调用 `Encryptor`。而 `MutableArray` 类中的代码也改成使用 `Lookup.getDefault().lookup(DoEncode.class)` 来查找 `DoEncode` 的实现类。本例中的实现类放置在 `org.apidesign.cycles.crypt` 模块中，并调用 `Encryptor` 类来实现具体的加密处理。这样就避免了代码在开发和编译时的循环依赖。

在具体程序运行的时候，这两个模块都必须可用，因为这两者是不能独立运行的。在 `MutableArray` 的 `encrypt` 方法中，对 `DoEncode` 对象进行了断言，以保证要有一个 `DoEncode` 的实例对象可供使用。整个结构对模块有着严格的定义，以保证运行环境的正确。但我们的目标是尽可能地减少模块间的耦合，所以我们希望可以有一个方式能够在更高的层次上来表达模块间的关系，而不仅仅是说在代码中通过断言这种方式来判断代码是否正确执行。`NetBeans Runtime Container` 允许在模型的 `Manifest` 文件中通过“依赖/实现”这样的标识来做到这一点。在程序被集成时，则所有的模块都会被自动校验，检查其依赖的模块是否可用，这样做可以保证整个运行环境的一致性。我们继续用这个例子来说明如何使用这一功能，`MutableArray` 类所在的模块需要声明该模块需要一个 `DoEncode` 的实现类，在 `Manifest` 文件中加入下面一行：

```
OpenIDE-Module-Provides: DoEncode
```

而 `Encryptor` 所在模块则需要告诉 `NetBeans Runtime Container` 它能够提供 `DoEncode` 的实现，它也在自己所在模块的 `Manifest` 文件加入下面一行。

```
OpenIDE-Module-Provides: DoEncode
```

完成了上述工作，`NetBeans Runtime Container` 就能知道这些模块间的相互依赖关系，可以在运行时自动启用或者禁用这两个模块。

开发人员可以按照上述思路把原来实现内部功能的模块间的循环依赖进行重构。这样，利用组件注入，就可以把程序的系统架构搭建得简洁明了，避免出现原来把代码写得乱成意大利面条一样的那种情况，所以模块化依赖和组件注入这些技术可以有效地改进 Java 程序的架构。

7.5 满城尽是 Lookup

我们一直在讨论的组件注入，其实是来自于其他框架的一个概念。也许有人会问它到底有何特殊之处呢？通过前面的说明，读者可能已经体现了 `Lookup` 作为一个功能类库的多个优点，但 `Lookup` 的优点远不止于此。想象一下，`Lookup` 可能不止一个，也许是几十个，几百个，几千个，甚至几百万个，要看你如何具体使用它。

这些优点只有真正使用 `Lookup` 机制的时候才能体会得到。与 `JDK` 提供的 `ServiceLoader` 相比，`NetBeans` 提供的 `Lookup` 可以有多个实例，每个实例都有自己的缓存池。`Tim Boudreau` 就常说，“`Lookup` 就像一个缓存池，对象可以随意进出”。可以利用这样的缓存池，结合 `Adaptable`^① 模式来提供各种服务。下面来看一下如何通过一个 `Swing` 图标类来演示该功能的使用。

① 其实也就是 GOF《设计模式》一书中所写的 `Adapter` 模式，中文翻译为适配器模式，是指把一个类的接口变换成客户端所期待的另一种接口，从而使原本接口不匹配而无法在一起工作的两个类能够在一起工作。像在 `Eclipse` 中，也同样大量使用了 `Adapter` 模式，可以算是 `Eclipse` 的核心模式。——译者注

```

public interface ExtIcon extends Icon, Lookup.Provider {
    public void paintIcon(Component c, Graphics g, int x, int y);
    public int getIconWidth();
    public int getIconHeight();

    public Lookup getLookup();
}

```

ExtIcon 这个接口继承了 Lookup.Provider，这就表示这个类可以提供一个自己的 Lookup，这样就可以提供一个额外的功能，或者临时标记图标使之看上去具有某些功能。比如说，现在假设有必要将这个图标转成 java.awt.Image。当然可以为图标到 Image 的转换提供一个通用的默认实现，但如果这个 ExtIcon 的子类知道一种优化的方式可以将自己更快地转化成一个 Image 对象，比如说，它知道 Image 文件存放在哪里，能直接读取文件内容来转成 Image 对象，自然可以通过 Lookup 返回一个 Image 对象以提高转换的效率。

```

public static Image toImage(ExtIcon icon) {
    Image img = icon.getLookup().lookup(Image.class);
    if (img != null) {
        return img;
    }
    BufferedImage buf = new BufferedImage(
        icon.getIconWidth(),
        icon.getIconHeight(),
        BufferedImage.TYPE_INT_RGB
    );
    icon.paintIcon(null, buf.getGraphics(), 0, 0);
    return buf;
}

```

每个 ExtIcon 的子类都可以自行决定是否还有更加优化的方式将自己转成一个 Image 对象，如果没有这种优化方式，则将图标输出到默认的 BufferedImage 对象中。这个例子展示如何以一种静态的方式对一个接口进行功能方面的增强，因为是否能转化 Image 对象这个功能已经是固定的。对于任意一个确定的图片，能否转化必居其一。但还可以将其用在动态的例子中，比如说要判断一个图标是否修改过，如果图标代表图片编辑器内部的物体，且需要保存或者重新加载，那么一般来说，此图标被修改过。先来定义一下 Modified 的接口。

```

public interface Modified {
    public void save() throws IOException;
    public void discard() throws IOException;
}

```

Modified 的实现可以随意进出 ExtIcon 提供的 Lookup 池，这样就可以让相应的监听者知道其状态了。比如，如果一个图标被修改了，这时，它可以调用 ic.add(new ModifiedImpl())这个方法向 Lookup 池中添加一个 Modified 对象，如下所示。

```

public final class ModifiableIcon implements ExtIcon {
    // AbstractLookup 是一个基类，避免开发人员从零开始实现这个类。
    private AbstractLookup lookup;
}

```

```

// InstanceContent 是一个特殊的类，可以给 lookup 字段提供更多的权限，比如说数据的修改。
private InstanceContent ic;

ModifiableIcon() {
    ic = new InstanceContent();
    lookup = new AbstractLookup(ic);
}

public Lookup getLookup() {
    return lookup;
}

public void markModified() {
    if (lookup.lookup(ModifiedImpl.class) == null) {
        ic.add(new ModifiedImpl());
    }
}

private final class ModifiedImpl implements Modified {
    public void save() throws IOException {
        // save somehow
    }

    public void discard() throws IOException {
        // discard changes
    }
}
}

```

通过调用这个 `ModifiedImpl` 对象的 `save` 方法来完成保存操作，待操作完成后，就可以从 `Lookup` 中移除相应的 `ModifiedImpl` 对象。这样外部代码，就可以通过 `Lookup` 机制来执行各种操作，如保存和撤销，还可以对 `Lookup` 中的内容进行监听，并更新其状态。

NetBeans 中如何处理用户选中操作

在基于 NetBeans 架构的各种应用程序中，`Lookup` 得到了广泛应用，其中最常用的一种应用场景就是用来在多个 UI 控件间跟踪用户选中的内容。此时，`Lookup` 变成所有 UI 内容共享的上下文了，这样每个对象都可以进出。全局的操作，如主工具条和菜单中的各项操作，都可以由它们自己根据相应的上下文来决定相应的操作是否可用。

全局上下文其实只是一个代理，它内部其实是指向当前用户选中的那个控件。这个控件可以自行来提供一个 `Lookup` 来对外公开自己的状态，或者继续使用内部被选中的那个控件，以代理的方式再公开出去。这样，最终公开的内容也就类似于前文所说的那个 `ExtIcon` 对象实例。如果出现这种情况，主工具条的保存操作就会去查询当前上下文中的数据，试图得到一个 `Modified` 接口的对象实例，这样才能判断保存操作是否可用，如果可用则据以调整图标状态。

对于这种场景，`Lookup` 就好像一个事件总线。这样的事件总线类似于一个对象可以进出的池，可以用来处理用户界面上的各种事件。如果使用这种机制，首先就要先创建一个 `Lookup` 的

实例,暂且将它称为事件总线。接下来,就使用 Listener 机制来监听 Lookup 的变化,先注册 Listener 到 Lookup,然后就开始监听相应对象的变化。最后,还要定义一种方式将自己的对象放到 Lookup 池中。这样就有了一个通用的事件总线,同时它还是类型安全的。当然具体要做该方案的时候,还有很多可改进之处。比如说,还可以把事件进行合并处理或者是访问网络等。

Query 模式也是一种非常有用的设计模式,它有效地融合了 Lookup 的两方面能力:既可以作为一个组件注入的门面 (Facade),还支持在 API 中使用 Adaptable 模式。

写代码的时候,开发人员可能会经常需要在类中持有一个对象,这样才能通过这个对象取得一些关于它的特别的信息。比如说,现在有一个图标,想判断这个图标是不是代表一只猫。有些绘图对象知道自己画的具体内容是什么,但大部分绘图对象只知道像素、颜色这样内容。当然,让它们了解自己内容的方方面面确实过分。也许可以使用一些第三方的算法通过相应的像素和颜色来判断所画的图标内容到底是不是一只猫。这些算法其实和具体的哪个图标没有什么关系,但适用于所有图标。而且这样的算法也很多,相应的实现也可以很多。我们能不能提供一个 API 来实现这个功能呢?

在 API 中需要有一个方法来判断该图标到底是不是一只猫。当然这个图标自身是否知道自己是一只猫并不重要。如果它不知道,就需要利用其他类的功能来检查它到底是不是一只猫。尽管设计这个查询方法的第一目标是简单易用,但可扩展性也同样重要,只有具有了良好的扩展性,外部才能够利用模块的扩展功能将相关内容注入到 API 中,以支持对图标的判断。这也正是 Lookup 机制可以用一种优雅的方式来解决问题的根本原因所在。

```
public class CatQuery {
    private CatQuery() { }

    public static boolean isCat(ExtIcon icon) {
        for (CatQueryImplementation impl :
            Lookup.getDefault().lookupAll(CatQueryImplementation.class)) {
            Boolean res = impl.isCat(icon);
            if (res != null) {
                return res;
            }
        }

        for (CatQueryImplementation impl :
            icon.getLookup().lookupAll(CatQueryImplementation.class)) {
            Boolean res = impl.isCat(icon);
            if (res != null) {
                return res;
            }
        }

        return false;
    }
}
```



```

}
public abstract class CatQueryImplementation {
    protected CatQueryImplementation() { }

    protected abstract Boolean isCat(ExtIcon icon);
}

```

每一个扩展都可以注册一个 CatQueryImplementation，并将其注入系统。每个扩展都可以判断绘制的图标是否是一只猫，也可以将这个问题留给其他查询实现来回答。

这个例子演示了组件注入的使用。这个 API 对外开放，允许外部将相应的实现注册进来以增强系统的功能。但这个例子同样受益于 Adaptable 模式。开发人员也许偶尔需要实现自己的图标，同时由他们自己来决定所绘的图标是否是一只猫。此时只需要把自己的 CatQueryImplementation 注册进来，就可以很容易地达成这一目标。这样调用时，就会返回自己期望的结果。

两害相衡取其轻

很多人在读完第 6 章中“抽象类是否有用？”这段内容以后，可能会问为什么 CatQueryImplementation 被定义成一个抽象类而不是一个接口，我曾经说过接口要比抽象类更好，但我也强调了，在特定的环境下，抽象类能够对访问权限进行控制，而这一点是非常有用的。上面就是一个典型的例子。

作为一个 API 的设计者，我最担心的问题就是 API 的用户不用 CatQuery.isCat(ExtIcon) 方法来判断一个图标是否是一只猫，而宁愿通过调用 Lookup.getDefault().lookupAll(CatQueryImplementation.class) 来得到所有的 CatQueryImplementation 实例，然后再一个个地去调用 CatQueryImplementation.isCat(ExtIcon)。这就完全绕过了我在 CatQuery.isCat(ExtIcon) 方法中苦心编写的中间代码，因此就为我将来改进这个接口埋下了隐患。在第 8 章中，关于这个问题，还会有更多的讲解。为了避免这种情况，我才将 CatQueryImplementation.isCat(ExtIcon) 这个方法改成 protected 级别。

换个角度来看，世上没有什么免费的午餐，而且这也不是那种可以双赢的例子。将这些方法变成 protected 就不可能通过调用别的 CatQueryImplementation 中的方法来实现自己的 CatQueryImplementation，使得代理模式失去用武之地。这种处理方式附加了很多限制，不过还好，不算是一个特别大的问题。放在面前的有两条路：要不为了以后方便改进 API，通过附加限制将其牢牢掌控在自己手中；要不就给 API 用户更多的自由而缩小未来 API 改进的余地。我选择前者，所以我将 CatQueryImplementation 声明为一个抽象类，将 isCat(ExtIcon) 方法声明为 protected 级别。

说到这里，看看这个例子，又让我想起那个远程接口通信的例子了。在那个例子中，居于中间的多维空间就和 Lookup 非常相似。Lookup 的一端是观察者，另一端是提供者，而它对从自身通过的对象根本不关心。它只知道将交流的内容从一方传到另外一方时，要保证其类型安全。

Query 模式可谓是将 Adaptable 模式与注入结合在一起了，从而达到极高的扩展性。如果有

必要的话,就像 Lookup 机制一样,它们也可以提供一个特定的本地的类或者方法用来查找各种接口的具体实现。在 NetBeans 中,Lookup 的优点在于它在全局范围内使用 Adaptable 模式(因此也展示了它们的相似度),从而以一致的方式来管理各种内容的注入。

7.6 Lookup 的滥用

Lookup 自身有趣的动态特性注定了它是一个很受人关注的功能,所以很多开发人员会在不同的场合来使用它,甚至在一些有着更好解决方案的场合,也去生搬硬套 Lookup。这种对 Lookup 的滥用,搞得在很多 API 设计方案中,简直把它当成一个百宝囊,通过这个百宝囊可以拿到各种有意思的实例对象。但如果 API 都这样滥用 Lookup 了,其用户就会进一步滥用它,向这个百宝囊中放置各种对象,同时得用这个百宝囊来漫无边际地查找各种给定接口的实现。在这个过程中,如果能找到想要的具体实现,程序就一切正常运行,否则直接就崩溃了。我想谁也不是想要这样的后果,所以本节将会继续讨论其他可以替代 Lookup 的方案。

曾经有人提过一个 API 的设计建议,要把 Lookup 当成一个工厂来用。它内部包含了一大堆接口,然后把 Lookup 作为参数传给一个工厂方法,然后由这个工厂方法根据 Lookup 提供的上下文环境来创建相应的对象实例。

```
public interface NameProvider {  
    public String getName();  
}  
public interface URLProvider {  
    public URL getURL();  
}  
public interface ResetHandler {  
    public void reset();  
}
```

```
public static AServerInfo create(final Lookup interfaces)
```

以这种方式设计 API,可谓是完全把赌注押在了 Lookup 这样一个百宝囊身上。如果只看这些 API,那么根本无法猜出来这个方法内部到底需要什么样的上下文参数。当然这样做的动机是好的,是为了以后添加新功能时进行扩展更加容易一些。一旦出现改善工厂方法的新需求时,只需要添加一个新的接口,然后注册到 Lookup 中即可,不用在相应的工厂类或者接口中添加新的方法。只需要调整这个方法的具体实现代码,告诉该方法的调用者,放一个合适的对象到 Lookup 中,然后再通过 Lookup 取得新的接口对象即可。但这种方法完全丢失了强类型约束。要为这个工厂方法编写具体实现的时候,就必须知道所有的上下文环境信息,因此还不如把这些所需的信息以参数的方式直接写出来。

```
public static AServerInfo create(  
    NameProvider nameProvider,  
    URLProvider urlProvider,  
    ResetHandler reset  
)
```

最新的 Java 开发工具一般都提供了代码自动完成功能，帮助开发人员更容易地编写代码。这样，程序员就不用去具体地阅读相应的 Java 文档来查找继承一个类时要写哪些方法，或者调用工厂方法创建 AServerInfo 对象时要传递哪些参数等等。同样，如果要支持关闭 AServerInfo 的资源，就再写一个接口，然后将接口作为传入参数传给工厂方法即可。

```
/** @since 2.0 */
public interface ShutdownHandler {
    public void shutdown();
}

/** @since 2.0 */
public static AServerInfo create(
    NameProvider nameProvider,
    URLProvider urlProvider,
    ResetHandler reset,
    ShutdownHandler shutdown
)
```

这种功能扩展方案的唯一缺点在于可能会出现多个同名的方法，各方法仅有参数不同，而且参数的数量可能会非常多。特别是如果是每一个方法参数都是可选的话，那么这样长长的参数列表就体现得尤为明显。极端情况下，会写出有 2^N 个工厂方法，其中 N 是指参数个数。大部分情况下不会出现这么极端的情况，但如果真的出现了这种情况，还有一个解决方案。我称其为 Cumulative（渐进）工厂模式。

```
public static AServerInfo empty() {
    return new AServerInfo(null, null, null, null);
}

public final AServerInfo nameProvider(final NameProvider np) {
    return new AServerInfo(np, this.url, this.reset, this.shutdown);
}
```

这个模式只引入了一个使用最少参数的工厂，该工厂用来为指定对象创建默认的实例，在创建对象实例的过程中使用了大量的 clone 方法。后面的方法都会从现有对象实例中复制相关内容，然后略作修改后，返回一个新创建的对象，这个对象包含与现有对象相同的数据，但对数据的修改不会影响原有实例。这样处理参数时，就比较简单了。还可以把这些方法串起来调用，只要保证方法和可用接口一一对应即可，如下面的代码所示。

```
inf = AServerInfo.empty().nameProvider(p).urlProvider(p).reset(p);
```

调节无线信号

我搞了一个无线接收中心，用来在我的电脑上收看数字影像，为了看这些数字影像，我必须调好信号，找准电视频道才行。但新的无线信号标准 DVB 比以前那些很快就能定位的老信号标准要复杂得多。找准一个电视台的频率还远远不够。需要设置带宽、传输模式、编码率、调制、频谱、安全等。

开始的时候，我只是想写一个 Tune 类再结合典型的工厂方法来创建 Tune 类的对象实例。但这么一大堆的配置参数把我给搞蒙了。最详细的工厂方法好像得要一长串的参数才能搞定。当时，我想到了 Cumulative 工厂模式，随后创建 Tune 对象实例就简单得多了。

附加一点，这种解决方案还有另外一个不错的功能。与传统的广播信号相比，DVB 信号要求每个国家或者地区共用一个频率。因为如果两个信号发射器使用同一个频率的话，会放大信号，而传统的信号发射器则会相互干扰。当然，这个细节和我们现在说的东西没有什么关系。但对于不同的地区，会有相应的工厂，它会根据具体情况，如地区编码等信息，来创建 Tune 的对象实例，有些参数值可能用不上。但必须保证大部分参数值正确。结果，往往只要取得默认的 Tune 并修改其频率就行了，其他参数不用动。如果用传统的工厂方法来实现，恐怕就没有那么简单了。

如果把 AServerInfo 设计成 final 类，而且不可变，那么对于这种情况就更合适了。本书其他章节还针对这个问题给出了一些有用的建议，如第 12 章。本章主要对 Lookup 机制加以详细说明，其目的不仅在于说明其功能，也还要再次提醒大家，Lookup 并不是一个放之四海而皆准的最佳方案。特别是，如果不是那种远程接口通信的环境，建议还是考虑其他的方案。



API还有其他类型吗？提供给客户端代码的API与为开发商提供的API有所不同吗？这两个问题的答案都是肯定的。本章将会详细讲解其中的原由及其对API设计的影响。

假设现在有一个需求，是要写一个媒体播放器的API，比如说Winamp，或者是Unix上的XMultimedia System (XMMS)。这个媒体播放器不仅能够正常播放音频文件，还可以跳到下一首歌曲进行播放，当然也要能跳回上一首歌曲进行播放，还支持播放列表的功能。通过播放列表，可以添加、移除或者对歌曲进行重新排序处理。这些功能不仅可以直接由最终用户通过界面进行操作，还可以提供相应的API，以供其他程序来调用。这样，第三方程序可以用`xmms.pause()`或者`xmms.addToPlaylist(filename)`这样的代码来进行暂停或者为播放列表添加歌曲文件。这样第三方程序可以发起通信，使用播放器的API指使它完成操作，命令执行完后，控制权返回给调用者。我们可以把这种第三方程序代码称为“客户端”，而把这些接口称为“客户端专用API”。

此外，XMMS的API还提供插件支持，第三方开发商可以通过注册“输出插件”来扩展XMMS的功能。比如说可以做一个插件，把播放的歌曲内容全部写到硬盘上，或者干脆通过网络来播放等。注意，这时是由XMMS这种播放器的内部代码来发起通信，调用这些扩展功能。收集够了数据可以播放时，XMMS会找到当前的输出插件，将数据发给它处理，这时会调用`plugin.playback(data)`方法。当播放完成后，执行过程又回到播放器，这样就可以继续采集更多的数据，从而继续整个过程。此时的插件是不是前面所说的“客户端”呢？显然，它与前面说的客户端的地位完全不同。它没有让XMMS进行任何操作。但它为XMMS增加了很多功能，所以此时的插件不是前文所说的“客户端”。XMMS的这种注册插件来扩展功能的方式是SPI(Service Provider Interface)的典型应用。

8.1 C和Java语言中如何定义API和SPI

为了说明客户端API和SPI间的区别，先来看一下两种编程语言的区别。首先使用C语言来写一个API，这个API不带有面向对象的扩展。然后可以在Java语言中写一个相似的接口。

C语言非常适合描述客户端API，只需要定义一些方法，并在相应的头文件中加以声明，其他的程序就可以编译和调用这些代码。

```
void xmms_play();
```

```
void xmms_pause();
void xmms_add_to_list(char *);
```

使用 Java 语言来定义 API 的方式则完全不同。

```
public class XMMS {
    public void play() { doPlay(); }
    public void pause() { doPause(); }
    public void addToPlaylist(String file) { doAddToPlaylist(file); }
}
```

在 Java 中声明 API 的方式有更多种。比如说可以使用 static 方法、实例方法、抽象方法、以及 final 方法，这些都是可以的。每一种的语义都有所不同。如果我们再假定这些方法没有访问级别的限制，客户端可以自动获得预定义的 XMMS 对象实例，那么不管用 C 还是 Java，处理客户端 API 的代码都差不多。但如果是编写一个 SPI 的话，那么情况就完全不同了。

如果要使用 C 语言来为 XMMS 编写一个插件的话，必须要写一个支持回放的函数，代码如下。

```
void my_playback_prints(char* text) {
    printf("%s\n", text);
}
```

当然，像 XMMS 这样的播放器软件也必须提供相应的注册功能，允许外部通过注册的方式将相应的扩展功能加入到播放器中，代码如下所示。

```
void xmms_register_playback(void (*f)(char*));
```

这样，第三方插件只需要调用这个注册方法，即可将自己的扩展功能挂入播放器软件中。

```
xmms_register_playback(my_playback_prints)
```

XMMS 会在必要的时候再调用相应的 playback 方法。在 Java 里没有指针，所以契约就始于 Playback 的接口的定义。

```
interface Playback {
    public void playback(byte[] data);
}
```

那么，第三方插件必须实现 Playback 接口，将实例注册到播放器。示例代码如下。

```
class MyCallbackPrints implements XMMS.Playback {
    public void playback(byte[] data) {
        System.out.println(new String(data));
    }
}
xmms.registerPlayback(new MyCallbackPrints());
```

只需要简单的几步，XMMS 就可以调用插件了，像前面 C 语言下一样。虽然扩展功能的行为相似，但对应的程序逻辑却完全不同。对于 Java 语言，你只用到大学里第一学期就学习过的内容，即类和接口，还有如何继承子类 and 实现接口。但对于 C 语言，则要使用函数指针，函数指针对于 C 语言的学习者来说，绝不是一个简单的东西，相信很多学习 C 语言的一年级新生都没有听过这个概念。

如果要用 C 来设计一个 SPI（比如回放）的话，那么相应的开发工作量不仅比较大而且难度

也高，对于一个新手，可能会望而却步。C 程序员要设计 SPI，在技术方面必须达到一定的高度才行。但对于 Java 程序员，只要所编写的方法不是 `private`、`final` 或者 `static` 的，那么就表示该方法是支持回调的，而且可以看成是一个 SPI。但很多程序员，甚至是教师都没有清楚地理解这一点，它也的确不是日常编码的内容。尽管很多 Java 方面的书籍在第一章（或者至少是在他们开始介绍 applet 时）都会对读者解释 `public`、非 `static` 和非 `final` 方法，但都没有指明误用这些内容的不良后果。虽然对于一般的开发工作来说不会有大碍，但如果开始着手设计 API 时，那些一开始形成的坏习惯就会给你带来许多麻烦。

8.2 API 演进不同于 SPI 演进

事实上，演进是任何契约都要涉及的。时过境迁，任何事物最终都会被淘汰。无论是 API 还是 SPI，都无法逃脱这一自然规律。所以要对 API 和 SPI 的演进预作计划，不要到出现问题的时候措手不及。

以 API 为例，它为外部客户提供方法，因而增加功能不会有问题。添加一个从播放列表中移除文件的方法，并不影响软件的二进制兼容性，只会取悦 XMMS 的新 API 的用户，让他们有更多的自主选择权，既可以使用这些 API，也可以不用。扩展客户端功能的 API 可谓是双赢。

但如果说到提供者的 API 的话，情况就正好反过来了。向接口中添加一个方法，会使得所有实现该接口的类全部被损害，因为原来的代码都没有实现这个新加的方法，只要这个新加的方法被调用，就会抛出异常。从另一个角度来说，如果停止调用方法，从根本上由接口中移除该方法，反而是可以接受的，也是有效的。很明显，如果操作流程不是契约的一部分，那么不调用方法也就不会引起太多的问题。

二进制兼容性 VS. 源代码兼容性

如果一个非抽象类继承了某个接口，但没有实现接口的某些方法，其实 JAVA 虚拟机仍然可以将这个类加载到内存中，而且不会引发连接错误。但如果有代码调用某个尚未实现的方法，就会抛出 `java.lang.AbstractMethodError` 这样的错误。如果调用者也添加了这个方法，那么还可以接受，但本书不推荐这么做。然后调用就总是带有一个异常声明。

具体的演进方案则还是取决于具体的接口类型：向 API 中添加一些内容总是可以的，但要移除一些内容则不行。但对于 SPI，移除一些内容可以允许，但不允许添加新的内容。建立契约时，必须清楚地区分哪些是 API，提供给外面调用，而哪些是 SPI，用来让外部来扩展程序功能。设计时容易犯的最大错误就是将 API 和 SPI 混在一个类里。如果出现这种错误的话，就把未来的演进空间给堵死了。根据 SPI 的契约，不能添加方法，根据 API 的契约，也不能减少方法，可谓进退两难。所以，要始终分离 API 和 SPI。

8.3 java.io.Writer 这个类从 JDK 1.4 到 JDK 5 的演进

现在来分析一下 `java.io.Writer` 这个类，看一下从 JDK 1.4 到 JDK 5 的发展过程中，它在哪

些方面有所演进。这个类开始的时候只是继承了 `java.lang.Appendable` 这个接口，并实现了这些方法，比如说 `Appendable` 中定义的 `append(CharSequence csq)`。这个新方法是必须要实现的，不能作为一个抽象方法留给子类去实现，因为 `java.io.Writer` 的子类很多，而且这些子类并不知道 `java.io.Writer` 的父接口在 JDK 5 中多了一个新方法。所以必须实现该方法。现在要考虑一下可能采用哪些方式来实现这个方法。当然，我们已经知道 JDK 5 提供了哪个具体的方案。但我们现在假装不知道，一起来寻找最佳方案。

我们已经确认不可能将这个作为一个抽象方法，也就是说必须提供一个该方法的实现。既然如此，直接抛出一个异常是否可以作为一种可选方案呢？

```
public Writer append(CharSequence csq) throws IOException {
    /* 直接抛出一个异常，用来表明这个方法是新的方法，而且子类必须要覆盖该方法。*/
    throw new UnsupportedOperationException();
}
```

当然，直接使用抛出异常的方案也是可以接受的，其实也非常合理，因为这可以用来提醒子类，`Writer` 这个类其实并没有真正地实现这个新方法，子类必须要重载该方法以提供相应的实现。从这个角度来讲，抛出异常的确是一个不错的方案。但从调用 `Writer` 功能客户端代码的角度（即一个更普遍的角度）来讲，如果这个方案不能提供一个实现的话，又何必要在 API 中添加这个方法呢？使用这个方法还需要提心吊胆，生怕抛出一个异常来。还需要使用 `try/catch/finally` 的方式来进行防御性编码。

```
try {
    bufferedWriter.append(what);
} catch (UnsupportedOperationException ex) {
    bufferedWriter.write(what.toString());
}
```

这样来调用 API 的确有点怪。调用一个方法竟然要写四行代码！如果我是这个 API 的用户，肯定想，算了，放弃这个不合理的新方法，还是沿用老的方法吧，最多只要把相应的对象转成字符串就是了。所以抛出一个异常并不见得是一种好的处理方式。最好还是为这个新方法提供一个默认实现。比如像下面的代码。

```
if (csq == null) {
    write("null");
} else {
    write(csq.toString());
}
return this;
```

以上的代码可谓给 `Writer` 类的用户提供了一个优秀的解决方案，事实上，JDK 中也是采用了这种方式。所以用户在使用 `Writer` 类的这个新方法时，可以直接使用默认的实现，或者干脆继承这个类并重载该方法，从而能够更加高效地实现相关功能。就此而言，问题解决了。

但这样做仍然存在一个问题。假设已经有了一个优化过的 `Writer` 类，它无需将数据转换成字符串，而是直接处理以提高性能，但用户却可能无法用到这种梦寐以求的优化，为什么会出现这种情况呢？再来想一下，如果要加快数据流的输出速度，要怎样做呢？可以使用 `BufferedOutput`

Stream 这个类。如果能让 Writer 类的操作更快，要怎样做呢？可以使用 BufferedWriter 这个类。但可惜的是，在使用这些新方法时，BufferedWriter 不会提高其效率。因为 BufferedWriter 在实现这些新方法时，无法兼顾效率和兼容性，如果说想为 BufferedWriter 提供一个高效而且正确的实现，实在并非易事。所以在 BufferedWriter 中还是使用 Writer 的默认实现。如果想通过 BufferedWriter 来提高效率并优化自身的代码，就需要继承这个类，并重载这些方法才可以。下面就是一个例子：

```
/** 该类是一个支持对输入的字符进行计数的 Writer 子类。
 */
public class CountingWriter extends Writer {
    private int counter;

    public int getCharacterCount() {
        return counter;
    }

    @Override
    public void write(char[] cbuf, int off, int len) throws IOException {
        counter += len;
    }

    @Override
    public Writer append(CharSequence csq) throws IOException {
        counter += csq.length();
        return this;
    }
}

/** 这个类支持字符懒加载功能。比如说，想从一个 CD 上读取数据，那么可以按需加载，而不是一开始就将所有
 * 的数据都一次加载到内存中。
 */
private static final class CDSequence implements CharSequence {
    private final int start;
    private final int end;

    public CDSequence() {
        this(0, 647 * 1024 * 1024);
    }

    private CDSequence(int start, int end) {
        this.start = start;
        this.end = end;
    }

    public int length() {
        return end - start;
    }
}

//
// 下面的测试代码来自 BufferedWriterOnCDImageTest 类：
```

```
//
CountingWriter writer = new CountingWriter();
CDSequence cdImage = new CDSequence();
BufferedWriter bufferedWriter = new BufferedWriter(writer);
bufferedWriter.append(cdImage);
assertEquals(
    "Correct number of writes delegated",
    cdImage.length(), writer.getCharacterCount()
);
```

现在假设用户已经按照上面的方式覆盖了这个新方法,但这个通过覆盖来优化性能的方法却不会被调用,效率仍然没有提高。因为 `BufferedWriter` 的实现让所有的优化操作都失去了用武之地。`CharSequence` 总是被转成一个字符串,因为转换时需要占用大量的内存,所以很难做到高效转换。说来说去,到底还有没有更好的办法来解决这个性能损失的问题呢?当然有了,现在来看一下如何在 `BufferedWriter` 中实现这个办法,同时还不产生额外的性能损失。

```
// 下面的实现代码很高效,但也危险,因为它没有调用相应的代理方法。
// 这样对于那些基于 JDK 1.4 而实现的子类,就会出现问题。
if (shouldBufferAsTheSequenceIsNotTooBig(csq)) {
    write(csq.toString());
} else {
    flush();
    out.append(csq);
}
return this;
```

上面的方法达到了预期的效果。也就是说,首先计算一下字符的数量,如果数量比较小,性能影响也较小,就使用老的方式处理,否则就先写入当前数据后,再写入大数据。然而,此过程还远远没有结束,还有一个潜在的问题需要解决。因为 `BufferedWriter` 这个类是可以被继承的,所以它的一些方法就可能被重载,子类会提供新的实现以进行一些特殊的业务操作,比如说对字符串进行加密(如下所示)。

```
public class CryptoWriter extends BufferedWriter {
    public CryptoWriter(Writer out) {
        super(out);
    }

    /* 我们必须覆盖 BufferedWriter 类中所有的已知方法,并对相应的参数,如字符、字符串或者是字符数组
    * 进行转换。
    */
    @Override
    public void write(char[] buf, int off, int len) throws IOException {
        char[] arr = new char[len];
        for (int i = 0; i < len; i++) {
            arr[i] = encryptChar(buf[off + i]);
        }
        super.write(arr, 0, len);
    }
}
```

```

@Override
public void write(int c) throws IOException {
    super.write(encryptChar(c));
}

@Override
public void write(String str, int off, int len) throws IOException {
    StringBuffer sb = new StringBuffer();
    for (int i = 0; i < len; i++) {
        sb.append(encryptChar(str.charAt(off + i)));
    }
    super.write(sb.toString(), 0, len);
}

private char encryptChar(int c) {
    if (c == 'Z') {
        return 'A';
    }
    if (c == 'z') {
        return 'a';
    }
    return (char)(c + 1);
}
}

```

上面给出的 `CryptoWriter` 类在执行数据写入（如 JDK 1.4）操作的时候会重载相应的方法，并可以使用任何一种老的 `write` 方法就能将所写入的数据进行正确加密处理。但这种加密方法存在着相应的缺陷，因为如果用户只使用了 JDK 1.4，自然没有问题，反之使用了新的 JDK，然后再调用新添加的 `append` 方法，此时使用 `append` 方法新增加的数据就不会被加密了。

```

CryptoWriter bufferedWriter = new CryptoWriter(writer);
bufferedWriter.append("VMS");
bufferedWriter.flush();
assertEquals("Converted", "WNT", writer.toString());

```

现在我们希望前面的加密代码仍然可以在新版本中正确使用。但在 `BufferedWriter` 中，因为它没有重载 `Writer` 的 `append` 方法，所以调用 `BufferedWriter` 的 `append` 方法就等于直接调用 `out` 参数对应 `Writer` 的 `append` 方法，这样就跳过了字符串加密。也就是说 JDK 1.4 提供的 `CryptoWriter` 类，它的方法都不会被调用了。所以对于 `CryptoWriter` 这个类而言，新添加的 `append` 方法还是造成了不兼容的情况。这个类本是用来对所有的输入数据都进行加密代码，但对于新版本的 `BufferedWriter`，却失效了。

事情变得越来越复杂了！为了处理含有大量字符串的对象等目的，最好是直接将数据写入相应的 `Writer` 对象。但有时候，却要调用 `BufferedWriter` 的 `write(String)` 方法，否则就无法保证向后兼容性。真是进退维谷，但有光必有影，一千个问题就有一千个解决方法。借助强大的反射功能，就很容易知道一个类是否是 `BufferedWriter` 的子类、是否重载了 `write` 方法，这样就可以根据具体情况来判断应该何时调用哪个方法了。这样既解决了性能问题，也保证了兼容性。下面给

出解决该问题的代码示例。

```

boolean isOverriden = false;
try {
    isOverriden =
        (
            getClass().getMethod(
                "write", String.class
            ).getDeclaringClass() != Writer.class
        ) ||
        (
            getClass().getMethod(
                "write", Integer.TYPE
            ).getDeclaringClass() != BufferedWriter.class
        ) ||
        (
            getClass().getMethod(
                "write", String.class, Integer.TYPE, Integer.TYPE
            ).getDeclaringClass() != BufferedWriter.class
        );
} catch (Exception ex) {
    throw new IOException(ex);
}

if (isOverriden || shouldBufferAsTheSequenceIsNotTooBig(csq)) {
    write(csq.toString());
} else {
    flush();
    out.append(csq);
}
return this;

```

第三个版本才是最终的解决方案。它使用了反射技术，所以看起来复杂一点儿，但却是一个非常有效的解决方案。它可以查明所有这些 `write` 方法的行为是否达到预期，也就是说，它们是不是没有被覆盖。如果该 `write` 方法没有被覆盖了，那么它将对 `CharSequence` 的处理直接转交给其代理的 `Writer` 对象。所以即使一个像 CD 图像那么大的字符串也能正确处理。从另外一方面来说，`BufferedWriter` 这个类的 `write` 方法有一点危险，因为可以通过覆盖该方法来改变其行为，所以要通过代理的方式来保证所有的操作都与老的代码相兼容，这样，即使有人继承了这个类，也仍然不存在问题。当然，通过这种代理操作，性能会低很多，但至少可以保证像 `Cryptowriter` 这样的子类可以正常执行。

使用反射技术让这段代码看起来不如原来那么漂亮了，但至少它可以正常运行，结果也是正确的。向后兼容就好像一把枷锁，而系统演进也是不时之需，所以反射技术有时候也是必须的，特别是如果在第一个版本中规划不足的时候，那么使用反射技术就是在所难免了。如果所设计的 API 没有为其在将来的改进而做出准备，那么这个代价是设计者必须要付出的。这个 `Writer` 类设计上的问题出在，它把一般功能性的 API 和扩展用的 SPI 混在了一起。从 API 改进的角度来看，这两类 API 分别有着不同的约束方式，特别是 SPI，如果为某个 SPI 添加了一个新方法，那么 SPI

的开发商估计就会不太高兴，一旦出现这种情况，可谓进退两难。上文对于 `BufferedWriter` 和 `Cryptowriter` 配合的说明，已经充分证实了这一点。

NetBeans 中的 Node 和 FilterNode

有些人会认为我所说的 `Writer` 和 `BufferedWriter` 这个例子过于虚假了，有点刻意为之的感觉，像对整个 CD 进行加密处理这种情况在真实的软件开发中是很少见的。说得对，这个使用 `Writer` 和 `BufferedWriter` 的例子是我编的，主要是因为大部分开发人员都知道这些内容，很多人还会用代理的方式在现有的类上来扩展新的功能。所以虽然这个例子有点刻意，但所表现的情况却是普遍的。我在设计 NetBeans Nodes API 的时候，也出现过这个问题，当时我简直不敢相信。

NetBeans Nodes API 是一个基于 JavaBeans API 的又一层抽象扩展，它在一个普通的 JavaBeans 上面添加了属性、名称、显示名称、一些额外的扩展属性及一个继承体系。这个继承体系是借鉴了 `java.beans.beancontext` 的思想，它可以把一个完全未知的对象放到一棵树上，然后与其他结点进行交互。

这个 API 在 NetBeans 中得到了广泛的应用，它可以用来显示本地磁盘上的文件内容、项目的逻辑结构、数据库结构、Java 源代码中的元素以及很多场景。但所有这些应用场景都非常相似：有人先定义一个基本框架，比如说项目结构，然后将一些独立的结点放到这个结构中。这样做的结果就是没有人对这整棵树具有绝对的控制权。每一个人只对自己那一部分的结点负责。

前面说的这些都不是什么大问题。但随着时间的流逝，默认的项目结构感觉已经不能更好地支持现有的需求了。比如说，你想在项目结构树上过滤出所有的非 Java 文件，为了处理这种情况，我们提供了一个 `FilterNode`。这个类其实是另一个 `Node` 的 `Decorator`，它的名称、属性和继承其实都是调用另一个 `Node` 来获取的。所以如果你使用的 `Node` 是 `FilterNode` 或者其子类，那么其实你拿到的是两层结点，表面上拿到的是一个 `FilterNode`，但在 `FilterNode` 内部还有原先的结点。这两层结点所在的树其实并不完全匹配，`FilterNode` 可以为自己所在的树隐藏内部 `Node` 树上的一些结点，同时还可以添加内部 `Node` 树上根本没有的 `Node`。在没有使用 `FilterNode` 之前，构造一棵树时，其上的结点是由不同的模块来提供的，对于树来说，这些内容是完全未知的。这个例子和前面所说的那个 `Writer` 和 `BufferedWriter` 例子差不多，但还要更复杂一些，因为你无法对树上的对象进行任何控制，只能通过 `Node` 以代理的方式进行调用。

有一次，我想换个方法来控制 `Node` 上弹出菜单中的内容。在 NetBeans 中原来有一个 `SystemAction[] getActions()` 方法用来支持这个功能，设计这个方法的时候，Swing 还没有得到广泛应用。我们原来方法的返回值是 NetBeans 自定义的类型。现在我们决定转向 Swing，所以我们考虑把返回值改成 Swing 的 `Action` 类，于是我们引入了 `javax.swing.Action[] getActions(boolean b)` 这个方法。然后调整我们的 `SystemAction` 类，让它实现 `javax.swing.Action`，我只需要把对新方法的调用转成对原有类的调用就好了，就和 JDK 5 中的 `Writer.append` 方法调用了 `Writer.write` 方法一样。我还把所有的代码都修改完了，使用新的方法来显示弹出菜单。

所有东西看起来非常自然,运行结果也是正确的:所有的结点不再支持 NetBeans 专用的 Action,这样,我们就可以很方便地直接使用 Swing 提供的大量 Action 了。

但过了几天,我就收到了一个 bug 报告,抱怨说某些结点上的弹出菜单不正确。经过一段时间的研究,我发现问题出在 FilterNode 上。一旦用户 FilterNode 对现有 Node 做了一层包装以后,那么在调用 FilterNode 的新 `getAction(boolean)` 方法时,FilterNode 其实是调用了自己原来的 `getAction()` 方法,而原来的 `getAction()` 方法会去调用 Node 的 `getAction()`,这样调用下来的结果根本没有调用到 Node 新写的那个 `getAction(boolean)`。现在清楚了,FilterNode 这个例子就好像努力用 `BufferedWriter` 写出 `CDSequence` 的例子。

当我发现所有问题的原因,大吃一惊。为了处理该问题,我做了两件事。首先,我使用反射来检查 FilterNode,如果子类没有覆盖它原来的 `getAction()` 方法,或者子类能够正确以代理的方式来保证该方法可以正确执行^①,那么就直接调用原来的 `getAction()` 方法,否则还是调用新的 `getAction(boolean)` 方法。其次,我编出那个 `Writer` 和 `BufferedWriter` 的例子来说明自己的经历,就是为了告诉只使用 Java 类库开发的程序员,其实他们也可能碰到类似的问题。

现在,可以说问题已经是一清二楚了,解决方案也明明白白了,我们可能要想,如何从源头防止这个问题发生。将问题提升到一定高度来看,就会发现这类问题是因为把继承和代理混合使用而引发的。让一个类可被继承,其实就意味着把它当作一个 SPI,因为其子类可以提供新的功能,或者说新的服务,但对于代理来说,它更多是让其他的代码来进行调用它的功能,属于功能 API,但如果把这两者混合在一起,像 `BufferedWriter`,事情往往就不像表面看起来那样简单、顺畅了。我不确认在 Java 语言中,是否能够找到一个好的方式,能够在两者混合使用的情况下,还能保证未来很容易对其进行演进。但我知道如何避免将这两者进行混用:要么不允许继承,要么不允许代理。

下面说得具体一点儿,来看一下如何避免继承。其实很简单,只要理解本章所说的设计原则,将客户端要使用的 API 与那些需要通过继承来实现的 API 分解清楚,就可以做到了。比如 `Writer` 类的原版本如下所示:

```
public final class Writer {
    private final Impl impl;

    private Writer(Impl impl) {
        this.impl = impl;
    }
    public final void write(int c) throws IOException {
        char[] arr = { (char)c };
        impl.write(arr, 0, 1);
    }
}
```

① 作者在 `FilterNode` 中留了一些方法,允许子类标识自己是否可以正确处理 `getActions` 这种类似的情况,默认情况下,会假设子类不能处理这种情况,子类必须正确调用 `enableDelegation` 方法来标识自己可以处理哪些情况。

```
public final void write(char cbuf[]) throws IOException {
    impl.write(cbuf, 0, cbuf.length);
}
public final void write(char cbuf[], int off, int len)
throws IOException {
    impl.write(cbuf, off, len);
}
public final void write(String str) throws IOException {
    impl.write(str, 0, str.length());
}
public final void write(String str, int off, int len)
throws IOException {
    impl.write(str, off, len);
}
public final void flush() throws IOException {
    impl.flush();
}
public final void close() throws IOException {
    impl.close();
}
```

技巧 上面这个final的Writer类中的方法与原来java.io.Writer类基本上差不多。现在写一个SPI的提供者类，这个类中全部都是工厂方法，用来提供Impl接口的具体实现，基本上就是下面的这个样子。

```
public static Writer create(Impl impl) {
    return new Writer(impl);
}

public static Writer create(final java.io.Writer w) {
    return new Writer(new Impl() {
        public void write(String str, int off, int len)
        throws IOException {
            w.write(str, off, len);
        }

        public void write(char[] arr, int off, int len)
        throws IOException {
            w.write(arr, off, len);
        }

        public void close() throws IOException {
            w.close();
        }

        public void flush() throws IOException {
            w.flush();
        }
    })
}
```

```

    });
}

public static Writer createBuffered(final Writer out) {
    return create(new SimpleBuffer(out));
}

```

最后是对外的 SPI 接口 Impl。

```

public static interface Impl {
    public void close() throws IOException;
    public void flush() throws IOException;
    public void write(String s, int off, int len) throws IOException;
    public void write(char[] a, int off, int len) throws IOException;
}

```

现在看一下，如果出现与 JDK 中 Writer 相同的问题时，这种方案是如何轻松地解决问题的。现在 Writer 需要实现 Appendable 类，也就是说要实现 append(CharSequence)方法，我们知道对于 final 类，添加一个方法完全是二进制兼容的，引入新的接口也同样是二进制兼容的。接下来的代码是演示在 2.0 版本中如何为 Writer 添加对 CharSequences 的支持。

```

public final class Writer implements Appendable {
    private final Impl impl;
    private final ImplSeq seq;

    private Writer(Impl impl, ImplSeq seq) {
        this.impl = impl;
        this.seq = seq;
    }

    public final void write(int c) throws IOException {
        if (impl != null) {
            char[] arr = {(char) c};
            impl.write(arr, 0, 1);
        } else {
            seq.write(new CharSeq(c));
        }
    }

    public final void write(char cbuf[]) throws IOException {
        if (impl != null) {
            impl.write(cbuf, 0, cbuf.length);
        } else {
            seq.write(new CharSeq(cbuf, 0, cbuf.length));
        }
    }

    public final void write(char cbuf[], int off, int len)
        throws IOException {
        if (impl != null) {
            impl.write(cbuf, off, len);
        }
    }
}

```




```
        } else {
            seq.write(new CharSeq(cbuf, off, len));
        }
    }
    public final void write(String str) throws IOException {
        if (impl != null) {
            impl.write(str, 0, str.length());
        } else {
            seq.write(str);
        }
    }
    public final void write(String str, int off, int len)
    throws IOException {
        if (impl != null) {
            impl.write(str, off, len);
        } else {
            seq.write(str.subSequence(off, off + len));
        }
    }
}

public final void flush() throws IOException {
    if (impl != null) {
        impl.flush();
    } else {
        seq.flush();
    }
}

public final void close() throws IOException {
    if (impl != null) {
        impl.close();
    } else {
        seq.flush();
    }
}
```

技巧 虽然代码与原来的API非常相似，但每一个方法加了一个判断。它要么可以调用原来的实现，要么将参数转成一个单个的CharSequence，然后调用新接口ImplSeq中的方法。现在来看如何实现这些新添加的方法。要调用这些新的方法非常简单，只需要将输入参数改一下就可以了。

```
public final void append(CharSequence csq) throws IOException {
    if (impl != null) {
        String s = csq == null ? "null" : csq.toString();
        impl.write(s, 0, s.length());
    } else {
        seq.write(csq);
    }
}
```

```
    }  
    return this;  
}  
  
public final Writer append(CharSequence csq, int start, int end)  
throws IOException {  
    return append(csq.subSequence(start, end));  
}  
  
public final Writer append(char c) throws IOException {  
    write(c);  
    return this;  
}
```

技巧 接下来是SPI上场时间了。我们新提供了一个ImplSeq接口，然后再提供一个新的工厂方法为其创建相应的Writer。其他的API就没有任何改动了；有一点要注意一下，就是对于参数为Writer Create java.io.Writerw来说，创建相应的Writer就简单多了。

```
public static Writer create(Impl impl) {  
    return new Writer(impl, null);  
}  
  
public static Writer create(ImplSeq seq) {  
    return new Writer(null, seq);  
}  
  
public static Writer create(final java.io.Writer w) {  
    return new Writer(null, new ImplSeq() {  
        public void write(CharSequence seq) throws IOException {  
            w.append(seq);  
        }  
  
        public void close() throws IOException {  
            w.close();  
        }  
  
        public void flush() throws IOException {  
            w.flush();  
        }  
    });  
}  
  
public static Writer createBuffered(final Writer out) {  
    return create(new SimpleBuffer(out));  
}
```

下面除了原来的 Impl 接口外，还添加了一个 ImplSeq 的接口，用来支持 CharSequence。

```

public static interface Impl {
    public void close() throws IOException;
    public void flush() throws IOException;
    public void write(String str, int off, int len) throws IOException;
    public void write(char[] arr, int off, int len) throws IOException;
}
public static interface ImplSeq {
    public void close() throws IOException;
    public void flush() throws IOException;
    public void write(CharSequence seq) throws IOException;
}
}

```

现在通过静态编译就可以完全避免原来的那些问题了。如果 `BufferedWriter` 这个类允许继承，那么 `Writer` 中 `append(CharSequence)` 的默认实现在代码具体执行时，其行为就会非常复杂，不好评估。你可以实现一个 `Writer.Impl`，然后提供一个恰当的工厂方法来创建 `Writer`。或者可以实现一个 `Writer.ImplSeq`，也使用一个新的工厂方法来创建 `Writer`，从而支持 `append(CharSequence)` 这个新方法。这样做可以通过静态编译来检查代码是否有问题，从而避免了在运行时对其进行检查。

实际上，设计给客户使用的 API，最好用 `final` 类；而设计 SPI 的时候，则最好用接口，然后通过一个工厂模式将它们结合在一起。虽然这个解决方案听起来有一点儿复杂，但该方案只用到了最常用的基本设计模式，还是很容易理解和接受的。新的方案借助“工厂设计模式”尽可能对客户“隐藏了”相应的复杂度，在使用 API 功能时，用户则只看到 `final` 类 `Writer`，对于要扩展 API 功能的人，则只需要看到两个要实现的接口 `Impl` 和 `ImplSeq`，所以不管是使用还是扩展这个 API，复杂度都不高。而且，这样设计出来的 API 并不难用，同时还很好地解决了许多 API 演进中的问题。虽然根据这个方案写出第一个版本的 API 时，也许工作量会多一些，但是将客户用到的功能性 API 和 SPI 分离出来，大大降低了演进的难度，避免出现将代理与继承混合时出现的那种问题。

8.4 合理分解 API

上一节讨论了设计 API 时对受众的区分是非常重要的，可以把 API 分为两类，一类是供他人调用来完成某些功能的，还有一类是他人来扩展 API 功能的，也就是前面说的 SPI。之所以要把 API 分解成这两类，主要是考虑到演进方面的问题。但还有一个非常有意思的理由来支持我们对 API 进行分类——可读性。

任何 API（特别是使用 Java 开发的 API）都会显示出一定的内聚性：表现为把相关的内容放得比较近，比如说把有关联的内容放在一个类里，或者放在同一个包中，对于 NetBeans 来说，可能就是放在同一个模块里了。面向对象语言都有内聚性的倾向。比如说，`java.*` 命名空间里的核心运行时库就有这个特点。当你想看看有没有方法能从 `String` 对象中取得一些信息时，那么在查找 Javadoc 时，最好是先从 `String` 这个类开始。尽管所有的 API 对 `String` 的用法各有不同，但都会用到 `String`，经常会带有一个 `String` 的成员。

我们再看看 Streams 这个例子。最常见的两种流分别是 `java.io.InputStream` 和 `java.io.Output`

Stream。这两者有一定的关联度，也许是因为它们之间有关系，所以才把这两者放在 `java.io` 这同一个包中。这两个接口定义了流最基本的功能，任何流都可以用这两者来描述，但同时还有一些接口和类用来对其进行补充，提供相应的功能。比如说，这两个接口都有自己的子类，用来提供一些有用的功能，或者支持 Decorator 模式，还有些类，可以把输入和输出流连接起来形成管道。这些类都放置在 `java.io` 这个包里。所以找起来就很容易，用的时候也安全简便，API 的用户也很容易就判断这些类应该在 `java.io` 这个包里。虽然其他的流放在另外的包中，比如说 `ZipInputStream` 就放在 `java.util.zip` 包里，但这些类往往都比较特殊。任何想使用 ZIP 功能的用户，都会习惯性地先去查找含有 `zip` 关键字的包。如果把 `ZipInputStream` 放置在 `java.io` 包里，那些对 ZIP 功能没有兴趣的用户也不会觉得这个类给他们带来了不必要的麻烦。

前面提到的这两个例子其实是可以（或许是应该）抽象出一些共性的。几乎每一个 API 的设计都应该遵循这些原则。为你的用户多想一下，按照他们的期望来组织你的 API，方便他们快速找到自己需要的内容。

- 把有关联的方法放置在同一个类中；
- 对于一些无关的方法，决不能因为一时无法找到更合适的位置来放置，就随便放入与它们无关的地方；
- 把有关联的类放置在同一个包中；
- 把一些只有特殊场景下才用得着的类转移到其他地方。

原则虽然简单，但只要好好地遵守，就可以有效地提高 API 的内聚性。这样用户就能很快熟悉你的 API 了。

事实上，一个 API 往往是一个集合，有不同的受众。对于 `java.io` 这个包来说也是这样。这个基本功能的包可以让任何人来使用，但 ZIP 功能却只有对 ZIP 格式感兴趣的人才会用到，还有对流进行加密这种功能也只有部分受众才会用到。每一类不同的受众都有不同的需求。所以根据他们的需求为其提供不同的入口，这也是一件非常有意义的事情。对于那些有兴趣处理 ZIP 格式文件的人来说，他们可以直接从 `java.util.zip` 这个包开始，如果有必要的话，再去了解一下 `java.io` 这个包的内容。那些对流加密功能有需要的人就可以通过选择正确的入口而避免受到其他不必要包的干扰。所以请一定要牢记，使用 API 的人可能是多个不同的用户群。按照这样的方式来组织你的 API，会让用户在使用 API 时感觉非常顺手，能够很容易地在相应的包和类里跳转。

NetBeans 项目组吃了很多亏，也花费了很长的时间才认识到 API 也是有很多类别的。经验来自错误的教训，这方面，`org.openide.filesystems` 模块是一个很好的例子。这个模块赢得赞誉是因为它将各种不同的资源进行共性抽象。编写该库的首要目的就是要提供一个访问任何资源的 API，这个 API 的目标用户群是一个非常庞大的群体。我们提供一个 `FileObject` 类，这个类可以用来描述任何文件，不管是本地磁盘文件，还是 ZIP 文件中的一个子文件，亦或是内存中的一个文件，甚至可以是远程 FTP 服务器上的一个文件。这个类抽象了 99% 人所感兴趣的功能。但我们在为这个类提供常规实现的时候犯了一个错误，把这些实现类与 `FileObject` 放在同一个包里了。当时这个包里还有 `LocalFileSystem`、`JarFileSystem` 等。这样就彻底地破坏了这个 API 的内聚性：我们原以为这些接口只有 1% 的人 would 用到，因为这些人可能想提供自己的 `FileObject` 实现。

但事实并非如此当开发人员发现这个包里还有这么多的内容时，居然有 99% 的用户都不用 `FileObject`，而是直接使用 `LocalFileSystem` 或者 `JarFileSystem` 中提供的 `FileObject` 实现了，这样他们就可以用这些类来实现一些 `FileObject` 实现不了的功能。他们努力将拿到的 `FileObject` 对象转成具体的实现类，然后调用一些本不该调用的 API。在最后总结时，我们觉得最好是把实现类分开放置在不同的包里，让用户很难找到这些类，也避免了 API 的用户有机会一下子拿到这么多的类。^①

从那天开始，我决定使用两种类型的包，分别是：`org.netbeans.api` 和 `org.netbeans.spi`。如果再给我们一次机会，从头开始重新设计 `FileSystem` 这个文件类库的话，我们很可能把 `FileObject` 放置在 `org.netbeans.api` 包里，而继承它的子类则会放置在 `org.netbeans.spi` 包中。这样，99% 的用户就不会关心 `org.netbeans.spi` 这个包里的内容了，因为他们用不着。他们也就不会想一些比较恶心的方式来调用这些本不应调用的 API 了。但这个类库已经被大范围使用了，为了不破坏向后兼容性，我不可能进行如此大的调整了。所以我们在文档里使用大幅的文字来敬告用户要根据自己的需求去使用正确的类，并不是通过一些强制转换来滥用 API。我们也从这个错误的教训中学到了很多内容，随后对于每一个新设计的 API，我们都使用了上述的分包方式。比如说，后来我们设计 `masterfs`^② 文件系统的时候，就采用了这种方式。`masterfs` 文件系统是用来表示本地文件的，`NetBeans` 现在大量地使用了这种文件系统。我们把 `masterfs` 文件系统的具体实现就放置在它自己的模块里，然后通过一个独立的 API 进行版本控制，这样就只有 1% 的用户需要去关心 `masterfs` 是怎么实现的了。其余 99% 的普通用户还是只关心 `FileObject` 类，不会受到 `masterfs` 这个包的影响。

把 API 和 SPI 进行分离并不能解决所有的问题。如果开发人员不能对 API 和 SPI 的不同点达成共识，新的问题就会出现。这种情况太平常了，你也可以做个简单的实验，问一下你的两个同事对这两个术语之间的差异有什么样的看法，他们的回答肯定是不一样的。至少，在 `NetBeans` 项目团队中，这种情况是常见的。基本的看法是这样：用来让他人调用以完成某些功能的内容，可以称作 API，而 SPI 则是由他人来具体实现的内容。通常来说，这是比较好的切入点，对于 `FileSystem` 来说，根据这个说法来判断也没有任何问题。但细化一下 `FileSystem`，就会发现 `FileObject` 这个类会被用户调用，而 `LocalFileSystem` 和 `JarFileSystem` 则是留给他人来具体实现。这里存在一个概念性的问题。任何基于 `JavaBeans` 来设计的 API，只要想使用 `Listener` 机制，那么都会出现这个问题。尽管，对于一个 `Listener` 的实现类来说，它实现了 `Listener` 接口，可以算作是一个 SPI，但 `Listener` 的内聚性决定了它与相关 API 的关系非常紧密。事实上，他们应该在同一个包中。

① 这里有一点复杂，`NetBeans` 的文件系统 API，其实有两个大的东西，分别是 `FileObject` 和 `FileSystem`，前者是一个抽象类，指对虚拟文件的抽象，而 `FileSystem` 则是对一类资源的抽象，通过 `FileSystem` 才能取得 `FileObject` 的具体实现，像 `LocalFileSystem` 和 `JarFileSystem` 都在内部提供了自己的 `FileObject` 实现，如 `LocalFileSystemImpl` 和 `JarFileSystemImpl`，请一定要分清楚。但 `LocalFileSystem`、`JarFileSystem` 和 `FileObject` 是放置在同一包里，即 `org.netbeans.filesystems`。——译者注

② `masterfs` 是一个类似于 CVS 的一个支持历史记录本地文件系统，`NetBeans` 默认使用这种文件系统来支持本地开发，这样用户就无须建立一个 CVS 才能支持历史记录了，`Eclipse` 也提供了类似的功能。——译者注

NetBeans 项目组花费了大量的时间才将思路整理清楚。幸运的是，在 NetBeans 项目组内部，一直很强调 API 和 SPI 之间的一个不同点：API 包中的类应该是自包含的，不会引入任何 SPI 包中的内容。这样就在 API 和 SPI 间划上了一条泾渭分明的边界。不管是监听器还是别的回调接口，只要与 API 有关，都要放置在相同的位置。这正是我们本章一直在强调的内聚性。它将潜在受众分成两类。大多数受众属于第一类，通过调用 API 完成某些功能，他们只需要了解 API 包中的内容即可。而第二类人数就相对少得多，他们需要一些额外的操作，比如说将自己实现的功能注册到现有系统中。这一类受众就需要了解更多的内容，不仅仅是 API，还要包括 SPI。如果你所编写的类库也遇到了此类情况，将 API 和 SPI 分离清楚是一个不错的主意。比如定义包名为 `javax.naming` 和 `javax.naming.spi` 就是一种很有效的方法。大部分人只需要关心第一个包，只有少数需要深入了解系统的人才需要关心后一个包。

其实对于类库的用户来说，各有各的需求，不一定要强行将其分为两类。有时候，把类库接口分解成更多的类型不失为一种合理的做法。NetBeans 项目把 API 分成四类，可供参考。

- **核心类型的 API** 这类 API 的用户往往只关心类库提供了哪些功能，他们需要使用这个类库来执行某些关键操作，而这个类库的使用又离不开这类 API。
- **支持类型的 API** 主要是大量的实用方法，可以让用户更容易地使用类库。这类方法不一定都会用到，提供它们只是为了让用户感到安慰。我们觉得把它们与核心类型的 API 分离开是很有意义的，因为它们只是助手类，并非必须的内容。
- **核心类型的 SPI** 这是提供给另外一种用户群体使用的接口，这类用户希望通过这些接口来扩展类库的功能。如果某个类库不允许外部来提供扩展功能，那么就不需要提供这些内容。
- **支持类型的 SPI** 有时候为了扩展功能去实现某个接口，还是一件比较复杂的事情。因此，可以提供一些助手类型的接口。但将这种类型的 SPI 与核心 SPI 进行分离也是一件非常有意义的事情，因为最好区别对待哪些是需要实现的内容，哪些只是起到辅助作用。

前面所说的这种 API 分类方案并不是最终方案，不可改变。它只是我们 NetBeans 项目组根据模块化架构而定义的一种方案。其他的项目和类库可以和我们一样，从自己的实际出发，找到适合自己的分类方案。不过，从我们所犯的上述错误中可以总结一条通用的规律：如果在设计 API 时能够按照上面的方案进行分类，那么就很容易在后续的开发中对 API 加以演进。在设计一个 API 的时候，要考虑清楚你 API 的目标用户群，然后按照其分类特性来组织 API 的结构，这样对用户来说，可以更方便地使用最符合他们需求的 API。请牢记这一点：很少有类库只提供一类 API。通常情况下，它们的目的都是多元的，受众群也是多样的。根据用户群体的不同需求来组织你的 API 结构。

软件设计的实践也在不停地改变。虽然可能都是使用 Java 语言，但与 20 世纪 90 年代相比，人们开发软件的方式已经完全不同了。我已经提过，造成这种改变的一个原因就是开源软件的兴起，这样开发人员在开发新功能的时候，就不一定要去写新的代码，可能只是使用现有的功能来进行组装了。

还有一个重要且相关的变化就是模块化程序的普及。在过去，软件复用无异于天方夜谭，但今天已经逐渐成为一个现实。成功的产品都是建立在模块化的基础上，因模块化架构而受益。模块化程序是将不同的组件恰当地整合在一起，形成自己的产品，同时还能让用户体验到很好的一致性^①。但在十年前，今天的这一切还只是一个美好的梦。

变化是如此之大，为开发人员的日常工作带来了方方面面的影响。但所有变化中，最重要的一个变化却是在上世纪末才出现，那就是软件行业开始广泛接受测试。开发人员逐渐认识到测试对于每日开发工作的重要性。而且大量的自动化测试也开始普及。当开发人员走上了单元测试之路后，他们再也不会返回到原先的开发之路上。也就是说，测试让开发人员上瘾。

本书并不想花费大量的篇幅来讲述自动化测试的重要性。如果读者想了解更多这方面的内容，我建议可以从其他地方获取这方面的知识，“Java 的测试模式”(<http://openide.netbeans.org/tutorial/test-patterns.html>)就是很重要的渠道，该文章对测试的目标给出了详细的说明，同时也提供了很多极有深度的用例。但考虑到本书毕竟是一本有关 API 设计的书，所以还是会讲述一下，测试对人们评估和写入 API 有哪些影响。

如果说，你对我前面所说的“测试会让开发人员上瘾”不太认可，想来只有一个解释，那就是你还没有尝试做这件事。测试真的很有效。比如说，如果你能在写代码时先写测试用例，或者说在写代码之前就写好了测试用例，那么就能提高你未雨绸缪的能力。有时候，程序员说某个程序已经开发完了，但在接下来的时间里他却天天向这个程序中左加点东西，右加点东西。如果有了测试，那么你就可以避免这种情况。更坏的一种情况，就是开发人员只管写代码，根本不执行自己所写的程序。通过提前进行自动化测试，可以避免程序员一厢情愿地认为代码已经完成，也能更好地判断项目当前处于何种阶段。我们可以坦白地承认，测试的确带有宗教般的虔诚！测试

^① 这里所指的一致性，是因为很多程序都使用了第三方组件进行开发，所以界面和使用方式都由第三方组件来决定，对于使用的这些组件的程序，往往都有相同的操作方式，对于用户来说，就是一种一致性的体现。——译者注

的根源在于极限编程运动（Extreme Programming，简称 XP）的兴起，说实话，如果对测试不抱有一种宗教性的狂热，极限编程根本就无法推行下去。习惯了测试的开发人员对此简直像是狂热的宗教徒。有时候，不写测试他们根本就不会开始编码。

9.1 API 设计和测试

现在要来讨论一下 API 设计和测试的关系。Javalobby 最近的一个调查表明，约有 45% 的开发人员会编写测试。他们可能正是因为进行了测试，从而才提高了自己的开发效率。他们抢在他人之前开始测试，因此，他们也能更早获得新的改进。不管从哪个角度来看，他们对于开发类库的人来说都是不可或缺的。对于开发一个类库来说，这些编写测试的开发人员都是非常关键的人，他们富于创新，拥有满腔激情。但如果想打击这些开发人员，有一个很简单的方法：把自己的类库写得让他们无法测试。当然这样做，还有一种可能就是，这些高效且富于创造性的开发人员会拒绝使用你的类库。

测试先行的开发方式，可谓是非常富有吸引力，更容易引发开发人员对其宗教性的狂热，这些开发人员如果发现一个 API 不容易进行测试，那么他们就会拒绝使用这个 API。当然，通过施加市场压力或消除可能替代你的 API 的所有其他方案，你也可以对他们这种态度视而不见。但实话实说，要做到这点可不太容易。毕竟现在已经有了那么多的开源类库可供选择，即使你消灭了所有的竞争对手，但很快就会有新的竞争对手出现。如果有其他的产品方案更容易测试，那么肯定就会有人发现这个新方案，如果运气再背一点，由直接竞争对手来提供这样的产品，那么麻烦就大了。

进行测试，特别是对于单元测试来说，基本的策略都与 Mock 对象有关。关于 Mock 对象到底是什么、做什么，已经有了很多讨论，还有很多长篇大论的文章。但为了分析测试对 API 设计的影响，我们至少要做到意识到，其实所谓的 Mock 对象，也就是某一个接口或者类的假实现，可以利用这样的一个假实现来进行测试。比如说，当真正的应用程序需要通过一个 Connection 的具体实现类来连到某一个数据库上的时候，那么在对该程序进行测试的时候，就可以用一个 Connection 的 Mock 对象来进行，这样的一个 Mock 对象可以模仿真实数据库连接的各种行为。通过这种假实现，开发人员就可以将自己的精力集中在应用程序的业务逻辑上，避免出现那些数据库进行网络连接的随机性故障。这样做，也简化了测试的安装步骤，因为测试在具体执行的过程中不需要一个真正的数据库来提供数据了，而且这种数据库连接的 Mock 对象放置在内存了，非常可靠。

从 API 作者的角度来说，API 最重要的一部分功能就是其大部分重要的接口和类，都能支持 Mock 对象的创建。创建方式可能是为这些接口写一个实现，也可能是 API 提供了一些默认的实现。当然，这种方式只对 Mock 对象有实际意义的 API 才有用。比如说，没有人会为 String 类来写一个 Mock 对象，部分原因是因为 String 是一个非常纯粹而且简单的类。它可以完成自己的功能，而且完成得很正确。另外一个原因就是 String 所有的方法都是自包含的，不需要使用外部的内容^①，也不需要外部环境对它进行支持。因此对 String 进行测试就不需要提供任何的 Mock 对象。

^① 所谓的自包含，是指 String 对外公开的内容（包括方法和字段）都没有依赖于其他的类（Java 平台的基础类型，如 char byte 不算在内）。——译者注

换个角度来说，肯定有一些场景是需要 Mock 对象来支持的。除了前面说到的数据库连接之外，还有一些其他的例子。比如说为 `URLStreamHandler` 提供一个假的 URL Mock 实现，这样在处理 `Http:` 的时候，就可以使用本地文件或者本地数据，而不用连接到远程网络上了。还有一个例子就是 `java.awt.Toolkit`，如果能为它提供一个假的 Mock 实现，那也是非常有用的，还有更多的例子可以证明 Mock 的用途。

如果 API 对外暴露的内容一点儿都没有超过对它的需求，那么在测试时就会出现一个问题：有时候，一个 API 的用户并不需要 API 支持继承，所以 API 设计时就不支持外部进行继承。这也就是说，你不可能根据 API 来创建自己的 Mock 对象，假设在开发某个程序时使用了这个 API，那么这种设计会严重地限制该程序的可测试性。正如前面解释过的，这样也许会打消用户使用你 API 的想法，所以应该为该问题提供解决方案。

有时候，正是出于这个原因，很多人在设计 API 的时候更倾向于使用接口。但要记住，使用接口也许会给 API 日后的演进带来很多问题，因为接口不支持在新版本中添加新的方法，以避免造成不兼容性问题。所以对于一个 API 来说，能够支持 Mock 对象的创建也是非常重要的，但对于设计 API 来说，必须能够在后续版本中修正自己以前的错误，并满足用户的新需求，这两者同样重要的。

要解决问题，首先就是要面对问题。要承认 API 的用例应该包括支持客户通过继承 API 中类来创建他们自己的实现类。这种处理方式可以让任何人根据自己的需求来实现自己的 Mock 对象。但从另一个角度来看，这样做也为 API 的设计和开发提出了更多的约束，当未来对 API 加以改变时，也必须遵守这一点。所以有时候，这个方案并不是很合适。API 中公开的内容过多，会将自己陷于一个不利于 API 演进的境地。编写测试代码和生成 Mock 对象可能会因此变得很容易，但却要为此付出高昂的代价。后续对 API 加以演进可能会变得比较困难，甚至演进这条路就被堵死了，所以这并不是最好的解决方案。

解决该问题的最佳方案是：在提供一个普通的 API 的同时，还要提供一个测试 API。这个测试 API 是普通 API 的一个扩展。它不会用在普通产品和应用程序中，它只会用在编写测试上。这个测试用的 API 会提供自己的实现，至少它可以支持为 API 的类创建 Mock 对象。它还可以提供其他一些有用的助手类，将 API 内部实现的一些细节暴露出来。之所以能做到这一点，是因为这些测试类通常与普通 API 的类放置在同一个包中，所以这些测试类就可以使用普通 API 中 `package` 的 `private` 方法和字段，但这些内容对于普通的 API 用户来说，是无法使用的。

传播数字视频

我在做业余项目 `dvdcentral` (<http://dvbcentral.sf.net>) 的时候，也碰到过类似的情况。它是一个完全基于模块化的项目，并没有提供一个界面给用户进行操作，它只是基于 `NetBeans Runtime Container` 来捕捉、保存和传播数字电视。

项目中有一个模块叫 `frontend`，它其实是我使用 Java 对 Linux 上 `/dev/dvb/adaptersX/` 的硬件设备进行的一个封装。它有一个类称为 `Frontend`，其中有一个工厂方法 `find(int X)`，这个方法用来定位到数字视频设备的 X 频道 (DVB)，其返回值就是这个频道的对象实例，能够通过这

个对象实例来控制这个频道。如果生搬硬套本书给出的建议来设计这个类，那就是尽量少公开内容，只要能满足用例中的各种需求即可。这也就意味着 Frontend 这个类可以设计成一个 final 类。任何人都不会继承该类以提供新的实现类。Frontend 这个类只用来操作 Linux 的这个设备。尽管这个类在程序具体执行时一切都没有问题，但如果想为它编写一些测试，就会受到很多严重的约束了。

其他很多模块都是建立在这个 Frontend API 基础之上的。需要为这些模型来编写测试，然后在独立的环境下运行这些测试。在执行测试时，我并不希望测试代码真正地去访问电视捕捉的硬件设备。但我在设计这个 API 时，是基于“只对外暴露必须暴露的内容”这个原则，这样就无法避免测试代码去访问硬件设备了。因此，我需要为测试代码添加额外的 API。

首先，我只是简单地用磁盘文件来模拟硬件设备。我添加了一个 frontend.root 的属性配置，默认情况下，它的值是 /，然后使用这个属性来指向某个文件。在测试环境下，我就创建了一个假的实现，指向 /dev/dvb/adapter0/frontend 等。这样我就可以通过一个临时的目录来完成测试，只要在测试之前，通过 frontend.root 来指定这个路径就可以了。这样测试就不会用到这个真实的视频设备，而是使用了预先准备好的一个假设备。对于某些特定的测试来说，这样做没有任何问题。我用视频文件来模拟硬件设备，运行得非常正常，但我却无法对数据传输的延迟编写测试代码了。因为对于设备来说，会因为读取失败或者堵塞而出现这种数据传输延迟的问题，但对于本地磁盘上的文件来说，它的传输速度非常快，也非常可靠，就无法测试这个问题了。

经过一些思考以后，我决定在 API 中添加一个用来支持测试的方法。这是一个新的工厂方法，称为 createVirtual(...)，它可以接受一些参数，并创建一个虚拟的 Frontend 实现。我不希望这个方法用在测试以外的用途上。尽管如此，但这个方法仍然是 API 的一部分，我必须保证这个 API 和其他 API 一样稳定，还好，到目前为止，这还不算一件太麻烦的事。

但还有一个更具有防御性、也更灵活的方案，就是把 createVirtual 方法和 FrontEnd 的实现类放置在 FrontendTest 测试包中，而不是像现在一样，直接放在 Frontend 这个类中。这个 API 就不需要进行任何修改。所有的测试都可以把相应的类放在类路径上，然后使用这个测试类。

你也许会问把一个 API 与测试 API 分离清楚有什么好处呢？为什么这些 API 的用户不是直接把测试用的 JAR 包放在类路径上，然后把它作为一个后门来用呢？当然，用户完全可以这样做。根本没有办法阻止他们来做这事。但二者间的细微区别会给他们一个警告：不要这样做。首先，你可以声称真正 API 的稳定性要比测试用 API 的稳定性高。比如说，在 NetBeans 项目中，真正的 API 可能会被称为“稳定的”，而测试用的 API 在大部分情况下则可能会被称为“正在开发中”。这样它可能会变得很“友好”，因为它要求每一个使用它的人都得告知其创建人：这个 API 正在使用中。

在普通 API 和测试 API 之间还有一个不同之处，那就是测试 API 可能更“隐蔽”。比如说，这个测试 API 会放置在另外一个 JAR 包中，还可能会用另外一套 Javadoc 对这些测试 API 进行说

明。对于那些只使用类库功能的研发人员来说，他们根本就不关心，可能也根本不知道有这类测试 API。他们不会看到这类测试 API，也就不会受到这类 API 中一些莫名其妙方法的困扰。听起来很不错吧。但在一个普通的应用程序中，还是无法阻止用户使用这些测试 API。前面已经说过了，这种情况是很难完全避免的。但这里有一个技巧可用：测试代码经常会打开断言 Assert，而产品在运行时则会把断言给关闭。这样，你就可以通过在测试代码中对断言的检查来知道是否是测试环境，如果不是，在使用测试 API 的时候，就可以使用断言来保证使用环境不正确时可以抛出异常。下面的代码就是这样做的。

```
boolean assertionsOn = false;
assert assertionsOn = true;
if (assertionsOn) {
    throw new IllegalStateException("This is a testing method only!");
}
```

这样做是不是让人很难理解，也过于苛责了？也许是吧。你不是一定要使用这种方式。但请记住，测试正变得越来越普及。如果开发一个框架的时候，不能把其可测试性提高到一定的高度，那些需要聘用的程序员精英就会被吓走。用一个不可测试的技术框架去开发程序时，简直就是告诉这些程序员：你们赶快跳槽吧，换份更高薪水的工作。

9.2 规范的光环正在褪去

长久以来，每当要开发一个软件项目时，一开始都会有一个非常重要的规划阶段，这个阶段是用来分析相关需求，制定一套切实的方案来描述这个系统所要完成的功能。直到把所有的规划都完成了，才开始设计和编码。但直到编码阶段完成的时候，才发现所有的规划竟然都是错的。现在，越来越多的项目开始使用敏捷方法，相应的规划被看作是一个不停改变的文档，随着设计和编码的深入，也要不断升级。有时候，这种开发方式是有效的，但有时候也不一定有效。我感觉很多类库、框架和编码语言的开发也开始采用敏捷方法。

有一段时期，很多喜欢理性开发方式的设计人员在设计一种编程语言时，会将相关的内容以规范的方式写在纸上，然后让那些开发人员直接去实现这些功能，他们认为这样做就可以完成项目了。但这样做的结果就是在规范和实现间产生了很大的差距。这两者分别是在不同的阶段、由不同的人员来完成的。而且对于一份规范，可能会有多份实现。对于 ALGO、Pascal 和 Ada 这些程序开发语言来说，在设计时就可以定义很多通用的库，如流资源的处理、相应的数据结构、常用的算法等。于是就有谣言说，如果开发一个软件项目，要在正式开发之前把这种编程语言的相应规范先要确定下来，这样才能保证项目的成功。在某些情况下，特别是对于 ALGOL 和 Ada^①这两种程序开发语言来说，这样的规范是不可能实现的，或者至少是很难实现的。当然，有时候也许要重写规范，然后再进行实现。就像在经历了多年的传统开发模式后，不知不觉就使用了敏捷开发模式。

^① ALGOL 和 Ada 都是老的编程语言，已经有了具体可用的新版本，而在当时并不可能先提供规范，所以作者说像这种已经存在的语言，是不可能先规划后开发的，而且也不可能有多多个实现。——译者注

虽然现在已经有了这么多的通用类库，但情况并没有什么改变。有一个 `libc` 的函数库，它提供了常用的函数，几乎每个 C 程序员都会用到这些方法，如 `printf` 函数。这种函数的声明是在头文件中，但它们的具体实现却不止一个。每个 Unix 厂商都会提供一个自己的实现。在 Windows 平台上还有一个基于 GNU 协议的 `libc` 的实现。开发程序时，可以基于这个函数库头文件声明的内容进行编码，还可以进行编译和连接，和标准的 C++ 库差不多。近些年来，无论编程语言还是新开发的库，都不再把内部的实现细节暴露给外部了，更多通过接口的方式对外声明其功能。

也许这可以归咎于，近年来基于各种开源协议的软件大量出现，所以也就不需要像以前一样，所有的实现都要自己从头开始了。要知道完全从头开发一个功能是一件很繁琐且代价高昂的事情，更糟的是，最后往往还被证明是在做无用功。有什么原因要让你去实现另一个类库呢？现在很容易就可以找到一个需要的功能类库，再根据情况改一下，做一个自己的分支版本。很多开源协议都允许这种做法。这样做，表示很多分支版本都是使用原有开源代码进行开发的，所以开发时根本无须关心原有规范的细节。要知道一个开源软件，其具体实现已经是一个规范了，所以分支版本只是在某些行为上有所不同而已。但维护这样一个分支版本成本也是非常高昂的，所以常见的做法是将比较成熟的分支版本中有所改变的内容再合并到原有的主版本中。像 Perl、Ruby 和 PHP 都采用这种模式，因为这几种语言除了源代码以外，也没有提供具体语言规范。难道这样的框架不算成功吗？

RUBY^①和 JRUBY^②

最近，Sun 公司看到 Ruby 语言日益流行起来，于是决定在 Ruby 的基础上重新实现一个新的 Ruby，它的新名字是 JRuby，但新的 JRuby 是可以在现有 Java HotSpot 虚拟机上运行的，也就是说，它和 Java 差不多。因为 Java 虚拟机可以进行各种性能优化，其容错性、健壮性也更好一些，所以大家普遍期望 JRuby 在性能和可靠性方面要优于纯粹的 Ruby 语言，特别是对于那些负载比较重的应用^③。

开发 JRuby 的那些人所做的工作，就是尽可能让原来 Ruby 程序完成的功能，在 JRuby 上执行时完全相同。我相信，JRuby 的开发人员要了解整个 Ruby 的规范。我还听说他们把原来用 Ruby 写的所有测试代码，都要在新的虚拟机中运行一遍，再把结果与原来在 Ruby 平台中的执行结果进行比较。如果出现不同结果，他们会新写一些测试代码，同时在 Ruby 和 JRuby 中运行以定位原因，直到所有的运行结果完全相同。虽然在 2007 年所有的测试代码就已经完

① 一种为简单快捷面向对象编程（面向对象程序设计）而创的脚本语言，由日本人松本行弘（まつもとゆきひろ / Yukihiro Matsumoto，外号 matz）开发，遵守 GPL 协议和 Ruby License。英文官方网站为 <http://www.ruby-lang.org/>，简体中文官方网站为 <http://rubycn.ce-lab.net/>。——译者注

② JRuby 的官方网站为 <http://jruby.codehaus.org/>。——译者注

③ Ruby 也有自己的虚拟机，但是官方原来的虚拟机是基于 AST (abstract syntax tree) 来完成的，它的速度比较慢，只能做一般纯脚本的解释工作。同时还有其他人员实现了一些高速的虚拟机，如 YARV (Yet Another Ruby VM，已经合并到了 1.9 中) 之类，但就目前来看，Ruby 的虚拟机与 Java 的虚拟机的性能差距还比较大。——译者注

全通过测试，但 Ruby 中最重要的框架，即 Ruby On Rails（也称为 ROR）^①，却无法在 JRuby 上运行，它在某些特定的环境中运行不正常。

这个问题清楚地说明了 Ruby 语言没有很好的规范，对自己各种行为没有清楚的定义。或者说，Ruby 没有独立的语言规范，它只是一个具体的实现。只不过，现在这个语言还是流行开来了。

但在开发应用程序时，这种没有规范的方式很普遍，像我们在设计核心框架时，也是使用了这种方式。大家觉得何苦弄一个规范呢？直接公开提供一个实现不就行了吗！公开以后，用得越多，反馈的 bug 也就越多，这不正是你所期望的吗？从这个角度来看的话，先定义规范，再提供实现的方式已经有些过时——至少看起来是这样的。为了让一个技术框架或者编程语言有更清楚的定义，尽可能地限制阿米巴变形虫模型。特别是对用户信息完全不清楚时，这种限制需要被最大化，而具体的用户需求则可以最小化。通过增加测试覆盖率可以有效地验证具体实现是否符合规划定义的内容。如果测试用例设计得好，那么因使用新版本而使得现有程序无法正常运行的几率就降到最低了。事实上，这些测试应该成为非正式规范定义信息中的一部分，作为缺少正式功能规范的补充内容。

有时，我看到有些开发商想为一个框架提供多种不同的开源实现，这样做无非就是将专利技术，如 Flash、C#或 Java，重新改写成开源的版本。其实这样做很难保证一切正常，先拿 Java 来打个比方。像 Java 中的 Swing 这个 UI 库，它没有什么正式的规范，也只是在源代码的层面上有一些文档。当然，即使为 Swing 提供了两套实现也没有多大的实用价值，其行为结果总是不同。所有复杂的系统，如 NetBeans 这种大项目，都只能在原先的实现上运行。谢天谢地，最近 Java 已经开源。虽然 Java 也有多个版本，但它们之间的区别比较小，还是可以接受的。我也希望 Java 以后会按照“分支/合并”的方式来运行。

这样做是否会对 API 的设计产生影响呢？答案是肯定的。如果你要写一个开源类库，你必须确认一件事，就是没有别人对你的这个类库又提供了一个第三方实现，而且这个实现并不是基于你现有代码，而是从头开始重新实现的。最多，有人会基于你的开源代码做一个分支版本。如果是这样，你可以把工作重心稍稍从文档上转到代码上。使用一些有效的 API 设计模式来保证你的 API 方便演进。加强测试可以帮你保证在未来版本中，你设计的 API 可以持续稳定。

9.3 好工具让 API 设计更简单

设计时，总会不时地想怎么做才能让自己设计的类库或者框架做得更好，能吸引更多的用户来使用。我们在设计和开发 NetBeans 的时候也经常想这个问题，这可能是因为 Java 的富客户端平台和 IDE 市场的竞争过于激烈吧。但按照这个思路做下来，我可以了解用户各种各样的想法，也清楚地知道开发的东西对用户是否有效。

^① Ruby on Rails 是一个用于编写网络应用程序的框架，它基于计算机软件语言 Ruby，给程序开发人员提供强大的框架支持。Ruby on Rails 包括两部分内容：Ruby 语言和 Rails 框架。——译者注

当发现使用自己类库的开发人员没有预期的那么多，此时如果想要找出原因，就不要将主要精力放在用户的成本方面。当然，如果说类库曾经有很多用户，却发现有用用户正在流失，那么去分析这个数据是合理的。但现在还只是刚刚开始，需要去吸引新的用户，那么成本很可能并不是最重要的评判标准。注重向后兼容性可以帮你赢得良好的声誉，因为这可以降低用户的费用，但这一点仍然不是应该首要关心的内容。

最重要的是上市时间，也就是说，在不了解类库的情况下也能尽快利用它创造收益。换言之，类库需要尽可能优化，即使不深入了解太多的内容，也能提高开发效率，这是非常重要的一点。这才是一个 API 或者一个框架有吸引力的根本所在。但问题又来了，怎么才能做到这一点呢？

我经常听到有人建议提供更多的文档和例子。当然，这样做没错，再多的文档和例子也不算多。但我们这样做的最终目的不是说让开发人员去 Google 查一下，然后下载一些很酷的东东。我们只想通过文档来帮助他们完成最想做的工作，也就是说用提供的类库做一个应用程序。为了达到这个目标，我们需要提供一些交互式的例子，这些例子不仅仅展示一些功能性的技巧，更重要的是可以帮助用户进行开发。

Ruby on Rails 和 Grails^①很清楚地知道这一点，所以提供了一个单独的命令行功能，可以使用该功能生成一个完整程序的基础框架。从某种特定角度来看，当你开始这样做的时候，其实应用程序就已经“差不多”完成了。你也许并不清楚这件事的背后有多少隐含的内容，但这个功能所完成的工作非常有用，而且也让人感觉非常酷，因为它至少可以让人立即就在浏览器上看到一些内容。如果一个框架上手的第一步是这么简单，那么它还会是一个坏的框架吗？

NetBeans IDE 也提供了帮助用户快速开发 NetBeans 平台这样一个功能，而且它不像前面所说的那些 Web 框架，它不需要调用一个命令行来完成这个功能。它把这个操作和整个工具合为一体。在 IDE 中，使用向导来创建新的内容。为了让 NetBeans 平台更具有吸引力，我们不需要完全修改 API，但要添加更多交互类型的例子。在我们引入这个向导以后，这个平台的开发效率提升了很多，可以说只用点击几下鼠标就完成了相应的工作。只要通过这些简单的操作就可以测试和试用一个新的框架，这点成本对于任何一个开发人员来说都是可以接受的。

NetBeans 平台中每一个重要的技术项都有一个相应的向导用来为其建立初始的基础框架。如果只用文档来做同样的事情，相信只有少些人才会来试用这个平台。阅读是一件很容易的事情，但将读到的内容落到实处就不容易了，要解压缩，然后创建一个项目，再把项目给运行起来，哪一项任务做起来都有不少的工作量。

好的工具对于测试来说也是很有帮助的。通常情况下，完全靠手工来构建一个测试框架并不是一件容易的事情。自动化测试通常是在一个沙箱中来进行的，所谓沙箱是指一个独立的隔离环境，正确地创建沙箱环境是一个非常复杂的任务。而且，如何创建这个沙箱环境与 API 的使用其实没有任何关系。向导的好处在于，不仅为程序创建一个基础框架，还为该程序的测试创建了正确的环境。安装正确的话，只需要几行代码来跑一下 API 的测试代码，或者修改测试来验证新添

^① Grails 是一套用于快速 Web 应用开发的开源框架，它基于 Groovy 编程语言，并构建于 Spring、Hibernate 和其他标准 Java 框架之上，从而为大家带来一套能实现超高生产力的一站式框架。它的官方网站为 <http://www.infoq.com/cn/minibooks/grails>。——译者注

加的内容是否正确。这才是 API 的真正用途。在这个世界上，并非所有的开发人员都是测试的狂热分子，所以也不是所有人都会用到生成的测试框架。但对于那些将测试看作生命的人来说，这样的向导会让他们立即可以进行测试工作，同样会让他们觉得这个 API 非常好用。

从产品快速推向市场的角度来看，最成功的 API 就是那些能够通过向导快速体现其功能的 API，不管这个 API 本身看起来如何。一个合适的创建类型工具可以将一个丑陋而且复杂的技术装饰得如钻石般夺目耀眼。虽然这只是 API 开始的第一步。当要处理的工作变得复杂，体验也不再简单时，开发人员就需要直接使用 API 的功能了，这时候就会了解 API 是否可用、易用以及兼容。但对于一个项目来说，将产品快速推向市场才是最重要的，其余的事情是第二位的。改进 API 的最好方式就是提供好的向导！基于自己的 API 来帮助用户编写最可靠的代码，其实最好的方式就是利用向导来生成所有测试用的框架。

9.4 兼容性测试套件

设计 API 时，比如设计一个模块，如果允许为同一个 API 提供多个实现，一定要考虑如何才能帮助 API 的开发者在实现 API 时能够满足 API 各方面的要求。因为 API 设计的目的是对实现的细节进行抽象，所以要让所有的实现都具有一致性，这是非常重要的一点。只有这样，API 的用户在使用 API 时，不管用哪一个具体的实现，都不需要去关注相应的实现细节。只有这样，才算做到了我们一直强调的“无绪”。

强调这种模块级 API 实现的一致性有两种方式。一种方法是 10.3 节中讨论的那种方案。还有一种则是提供兼容性测试套件（Test Compatibility Kit，简称 TCK），它是多个测试项的集合，API 的实现人员可以使用这种套件来验证自己的实现是否正确，特别是对于各种期望的方法是否具有一致性。

为一个模块编写 TCK 要比编写常用的测试代码复杂很多。在针对自己的代码编写测试代码时，要对代码及其所有的测试加以控制，先是开始时的相关配置，然后运行测试代码，最后对执行结果进行验证。但对于 TCK 来说，则是将初始化工作交给了 API 的实现者，同时把对结果的最终检查工作也是交给了他们。这样就需要考虑更多的内容，要考虑潜在的实现方式，和各种实现方式间的区别，以及在写测试代码时，要允许实现人员进行必要的初始化工作，并对执行结果进行检查。要做到这些并不是件容易的事情，需要一定的培训。另外这也要求 API 必须有良好的设计，而且容易实现。TCK 做得越好，用户对这个 API 的源代码需要了解的内容也就越少。如果需要深入了解源代码才能完成相应的功能，说明这个 API 设计得非常差。一个测试套件可以帮助用户在不需要了解 API 的时候，也能很好地使用 API，从而提高其实现的一致性。

为了说明 TCK 的价值，可以回想一下原来给 Swing 编写模型的日子。如果读者没有这方面的经验，我们可以给出一个简单的总结。如果只是使用 Swing 的控件，那么 Swing 是非常容易使用的，而且也很强大。但当你需要使用更强大的功能时，事情就会变得很复杂。比如说，每个控件都会定义一个自己的模型。像 JList 控件定义了一个 ListModel 模型，JTable 控件定义了一个 TableModel 模型，还有按钮、树等控件都定义了自己的模型。如果只是在界面上把这些控件展示出来，因为这些控件都提供了默认的实现，所以根本不用太关心这些模型，因此这是一件很简单的事情。但有时候，这些默认的实现可能不够用。用户就需要考虑提供自己的实现了。这时，

你的角色马上就从一个 API 的用户变成了一个 SPI 的实现者。只实现一个模型里的方法是远远不够的，每个模型都有自己特定的语法强约束，特别是那些与事件相关的方法。比如说当模型改变时，要通知监听者，相应的顺序以及事件中包含的信息都有一定的限制。这事做起来还是需要很多技巧的，如果不去读 Swing 的源代码并做一些调试的话，还是很难正确无误地完成这份工作。如果提供了一个 ListModelTCKTest 这样的类，用户照着抄一下，事情就简单得多了。用户只需要看一下，然后调整一下初始化的内容，变成自己的模型，然后在程序中验证一下运行是否正确就可以了。但据我所知，还没有 ListModelTCKTest 这样的代码。所以为 Swing 写一个模型就变成了一个只能由高手来完成的工作，对于新手，则只能干瞪眼了。

测试文件资源系统类库的兼容性

NetBeans 的 `FileSystem` 类库也存在与 Swing 差不多的问题。相对来说，这个类库还是比较容易使用的。但如果想为这个类库扩展功能，提供一些自己的文件资源，这可就一点儿都不容易了。我们支持事情通知、原子行为、锁、读写互斥等。一不小心，就会漏下几个功能没有实现。如果你曾经设计过一个需要他人来实现的接口，或许你会有相似的感觉。你很快就会碰到因为实现错误而带来的潜在问题。只要有一个实现类在具体实现代码中有一处不对，那么很可能就会出现错误。但对于 API 的用户来说，他们才不管文件系统背后的实现到底是磁盘文件、CVS 这种支持版本的文件系统、还是 JAR 文件、FTP 或者是其他文件。在使用这个 API 时，他们只要发现了一个 bug，然后就认为是这个 API 的 bug。就像很多普通的框架一样，不管到底是哪个实现引起的问题，都会认为是这些框架的 bug。对于框架的维护人员来说，这样会增加很多 bug 报告，加大了 bug 分析的工作量。最后，我们受够了这种没有价值的工作，于是为 NetBeans 的文件资源类库提供了一个 TCK 套件。它主要包括一个工厂类，该类有一个创建方法和一个清除方法。代码如下。

```
public static abstract class FileSystemFactoryHid extends NbTestSetup {
    public FileSystemFactoryHid(Test testToDecorate) {
        super(testToDecorate);
    }

    protected abstract FileSystem createFileSystem(
        String testName, String[] resources
    ) throws Exception;
    protected abstract void destroyFileSystem(
        String testName
    ) throws IOException;
}
```

但一个好的 TCK 需要考虑多种实现的不同需求。比如说，有些文件系统是只读的，所以这种情况下进行资源个性化的测试是没有任何意义的。对文件进行修改操作肯定会失败，这是意料之中的。这就是 TCK 中会提供配置项的原因。文件资源库中提供了大量的测试套件。每一个实现都需要选择相应的套件来进行测试。比如说，下面的代码就描述了在访问 ZIP 和 JAR 这种压缩包时相应的操作。


```

public static class JarFileSystemTest extends FileSystemFactoryHid {
    public JarFileSystemTest(Test testToDecorate) {
        super(testToDecorate);
    }

    public static Test suite() {
        NbTestSuite suite = new NbTestSuite();
        suite.addTestSuite(RepositoryTestHid.class);
        suite.addTestSuite(FileSystemTestHid.class);
        suite.addTestSuite(FileObjectTestHid.class);
        return new JarFileSystemTest(suite);
    }

    protected void destroyFileSystem(String testName) throws IOException {
    }

    protected FileSystem createFileSystem(
        String testName, String[] resources
    ) throws Exception {
        JarFileSystem fs = new JarFileSystem();
        fs.setJarFile(createJarFile(testName, resources));
        return fs;
    }
}

```

其他类型的文件资源插件也要做同样的事情,只要简单地创建一个文件系统的实例对象就可以了。

通常情况下,这样做可以提高应用系统的灵活性。而且对于一个软件开发组织,一个 TCK 套件可以改善日常的各项工作。比如说,它可以简化一个 bug 报告的生命周期,很容易就发现整个系统中哪部分存在 bug,定位 bug 原因时所需要的人手也会少很多。在 NetBeans 组织中引入 TCK 套件以后,像过去那种指派某个人来解决问题而导致 bug 转移的情况就减少了很多。从某种意义上来说,一个 TCK 套件可以让程序员的“傲慢”^①更好地表现出来:如果你实现了某个 API,也使用了 TCK 套件进行测试,或许你的 bug 报告就不那么重要了。

我相信对于一个 API 的供应方来说,他们的责任就是让用户尽可能容易地使用 API,而最好的方式就是提供一个 TCK 套件。不是说所有的人都会用到这个套件,但使用了这些套件的人将会是你 API 用户中最有价值的那一类。因为这类人最关心软件质量,不会为了因为急于解决问题而使用一些不合适的方式来使用 API 没有公开的功能。他们希望所完成的系统可以长期稳定地运行,也能升级到类库的新版本上。这类人会为 API 类库提供反馈信息,如提供使用报告和修复 bug 的补丁,还可以提供一些自行实现的新功能。换句话说,最聪明的方式就是让这类人尽可能多花时间来改善你的 API。

^① 这里所说的“傲慢”是指在平时出现 bug 时,经常会有人说这不是我的 bug,但没有办法证明这一点,有了测试套件以后,可以很容易地证明是谁的问题,所以好的程序员可以很“傲慢”地说:这不是我的问题。作者这里的“傲慢”并不带有贬意。——译者注

API 不是单独存在的。光靠它们自己可做不了什么有意义的事情；一定要被调用才能完成特定的功能。所以在设计 API 时，要注意一点，假设已经有一个大的应用程序使用了这个 API，如何能保证这个 API 可以与其他 API 共存，并进行很好的协作。

只有少数的 API 可以独自运行，无须任何上下文环境。在设计一个系统时，也要考虑后续如何对其加以演进，而且在演进以后，不会因为上下文环境的改变而出现问题，这样就必须了解如何正确地使用第三方 API，才不会造成意想不到的后果。

本章会就使用、复用、暴露、再导出第三方类这些重要的问题展开讨论。

10.1 谨慎使用第三方 API

使用第三方 API 的方式通常有两种：一种是把对方提供的 API 作为一个功能类库来使用，这并不会把第三方的 API 再暴露给自己的客户；还有一种是确实存在暴露需求时，需要将第三方 API 中的接口暴露出来给客户使用。下面的内容会对使用第三方 API 的过程中存在的风险加以分析。

把一个第三方库的 API 暴露出来，也被称为再导出。如果一个方法的返回值或者参数声明中使用了这个第三方库中的某些类型，就会出现再导出的情况。考虑到向后兼容的问题，处理这种情况还是需要很多技巧的，因为一旦一个 API 被再导出以后，它其实就变成了自己类库的一部分了。事实上，一旦第三方类库中发生改变，比如，在 API 的接口中添加一个新方法，那么在导出的 API 中立刻就体现出来了。这也意味着，一旦这个 API 出现了不兼容性的变化，即使只是使用主 API，也同样会出现问题。所以将第三方库类中的 API 再导出时，一定要认真考虑主 API 和其他被再导出 API 的稳定性。如果对主 API 的稳定性要求非常高，就表示需要对再导出的 API 进行一层包装，而不是直接暴露给用户使用。特别是在所使用的第三方 API 不是很稳定的情况下，这样处理可以减少兼容性问题。

这种问题也称为不兼容性的传递性，可以想象一下多米诺骨牌。下面的代码就演示了一个 API 引入了第三方的 API，同时又将这个第三方的 API 暴露给客户的情况。

```
public final class String {  
    private final char[] chars;
```

```

    public String(char[] chars) {
        this.chars = chars.clone();
    }

    public int length() {
        return chars.length;
    }
    public char charAt(int i) {
        return chars[i];
    }
}

```

有人决定要把 `String` 这个类中的某一个方法给删除了（拿 `String` 来打个比方而已，事实上这个类是在 `java.lang` 基础包中，是不可能移除的，而且对于 `java.lang.String` 这个类而来说，不管未来需要怎样的改进，它也要保证兼容性）。假设为了让 `String` 这个类和 `Collection` 集合类 API 看起来比较一致，决定要把 `String` 中的 `int length()` 方法改成 `int size()`。如果换了一个新的 Java 版本，原来写的 Query 代码就不能通过编译了。下面就是因为修改了这个方法而无法编译的代码示例。

```

Query query = new Query();
String reply = query.computeReply();
assertEquals("Length is correct", 5, reply.length());

```

此时 `Query` 类没有改变一行代码，但是已经出现了不兼容的问题！这只是因为你所暴露的类库出现不兼容问题而引发的，现在因为这个 `String` 类，所有使用你 `Query` 类的产品都无法兼容了。这种多米诺骨牌的传递效果很快就会引发一场风暴，不管你为自己的类库兼容性做了多少工作，都会毁于一旦。

要想避免此类问题，只有三种途径：接受这种不兼容性，拒绝这种不兼容性或者使用包装器来防范。

第一种方案就是接受这种不兼容性。如果一个 API 被再导出以后，一旦出现了不兼容问题，那么最简单的方法就是接受现实，承认自己的主 API 库也出现了不兼容。这就是说，一旦第三方类库的新版本出现了不兼容问题，那么也要发布一个新的版本来进行兼容。从本质上来说，这也就意味着要接受这种发布周期循环，以及第三方类库的开发方式和自身的稳定性。第三方类库中每一个不兼容的问题都表示主 API 中也存在一个不兼容问题。

第二种方案就是拒绝这种不兼容性。要想防止不兼容性的问题，最简单的方法就是保证自己的类库和暴露出去的第三方类库都不存在兼容问题就行了。当然说起来容易，做起来难啊！在这个过程中，必须能对第三方类库有足够的影响力，不让其出现讨厌的不兼容问题。为了达到这个目标，可以与第三方类库的开发方建立良好的关系。对于商业类库，你可以变成该公司的合作者，对于用来预览的新版本可以进行提前的测试。如果是开源项目，则可以加入该项目的邮件组，或者变成一个代码捐献者，这样可以直接影响该类库的开发方向。

上面做法只是有可能避免出现这种问题，但一旦真出现了这种不向后兼容的问题，仍然无补于事。在那种情况下，如果你不打算接受这种不兼容性并将其再导出去，唯一的修正方法就是不

升级到新版本，仍然使用兼容的老版本。尽管这个方法还算有效，但也仍然存在缺点：就是新版本往往会引入更加有用的功能。只有一些特别重要的内容修正会从新版本移植到老版本中。但这种情况也往往只会发生在客户需求特别强烈时。我曾听过一件事情，与 Linux 内核 2.4 和内核 2.2 维护人员有关，因为有些用户担心出现稳定性的问题，所以不想升级到 Linux 2.6 内核。经验表明，继续沿用老的版本库也是一种可行方案。但如果出现了有第三方也使用同样的库，问题就麻烦了，因为这个第三方可能会需要升级到一个新版本，该版本与你当前使用的版本不兼容。此时，会出现一个类库两个不同版本共存的问题。有时候不会出问题，特别是当这两个版本库被分隔开来，彼此间不知道对方的存在时。但有时候还是会出问题的。

即使能够找到一种方式允许某个类库多个不兼容的版本能够将自己的类再导出给其他类库使用，也并没有解决 NP^①这种难题。关于这个问题的探讨已经超出了本书的主旨，如果读者对这个问题比较感兴趣的话，可以通过访问 <http://reexport.apidesign.org> 获取更多的信息。

第三种方案就是使用包装模式。对于再导出第三方 API 而引发的问题，有一种可行的解决方案就是把用到的第三方 API 进行一次包装，而不是把第三方 API 直接暴露给用户。这样在内部仍然可以使用第三方类库，但对于外部来说，第三方类库的内部其实就不可见了，通过隐藏内部的实现细节来达到目的。下面是一段演示如何使用 Query 类来包装 String 类的代码。

```
public final class Query {
    public static final class Sequence {
        private String data;

        Sequence(String data) {
            this.data = data;
        }

        public int length() {
            return data.length();
        }

        public char charAt(int i) {
            return data.charAt(i);
        }
    }

    public Sequence computeReply() {
        char[] hello = { 'H', 'e', 'l', 'l', 'o' };
        return new Sequence(new String(hello));
    }
}
```

一开始就在类库中使用 Query 类，而不是直接把 String 这个类给再导出，这样他人就会使用 Query 而不是 String，即使 String 出现了不兼容的改变，Query 类可以在内部屏蔽这种变化，从而

① NP 是 Non-deterministic Polynomial 的缩写，NP 问题通俗来说是其解的正确性能被很容易检查的问题，这里“很容易检查”指的是存在一个多项式检查算法。——译者注

避免对外部系统造成影响。即使 String 类是在内部使用，像 Sequence 这个类在开始时还是使用 String 类的 length 方法。当一个新的不兼容的 String 类发布以后，只需要升级其内部的、未导出的依附物即可，并按下列方式修改编码。

```
public final class Query {
    public static final class Sequence {
        private String data;

        Sequence(String data) {
            this.data = data;
        }

        public int length() {
            return data.getSize();
        }

        public char charAt(int i) {
            return data.charAt(i);
        }
    }

    public Sequence computeReply() {
        char[] hello = { 'H', 'e', 'l', 'l', 'o' };
        return new Sequence(new String(hello));
    }
}
```

包装模式可以有效地避免不兼容性问题引发的连锁风暴。但它会增加 API 的数量，提高用户的学习成本。如果每个类库都为 String 定义一个自己的包装类，每个类都有不同的名称，那么用户在使用不同类库提供的 API 时就会非常困难。特别是如果需要在两个类库之间传递字符串信息的时候，就会更加麻烦。

使用新版本的 Java 编译 NetBeans

NetBeans 项目组一直以来都致力于能够让 NetBeans 开发工具至少可以在 Java 的两个主版本上运行。像 NetBeans 5.5x，就可以在 Java 1.4 和 Java 5 上运行，而 NetBeans 6.0 则可以支持 Java 5 和 Java 6，同时还会在更新的版本上进行测试。总的说来，这项工作还算开展得不错，只是开发人员往往都特别喜欢尝鲜。他们喜欢试用最新的 Java 版本。这使得我们的源代码在新老 Java 版本间要尽可能保持兼容。但还有一些情况是在我们意料之外的。每当我们使用主版本的 Java 来编译 NetBeans 时，我们都会发现一些其他的隐藏问题，因为 Java 中有些类库在开发时没有考虑兼容性问题。

如果说是因为我们实现了 java.awt.peer^①包中的接口，从而引发了不兼容问题，那么这

① java.awt.peer 是 SUN 为了使得同一个 Java 程序在不同的软硬件平台上运行，而提供的一个包，这个包中的内容没有在 Javadoc 中公开，在源代码中，也说明了不建议第三方使用，所以作者说，如果他们用了这种类型的包，因此引发的不兼容性是可以接受的，但他们并不是使用这些特殊的包，也仍然出现了不兼容问题。——译者注

种情况我可以接受。由于该包中的某些类添加了新的 abstract 方法，从而无法正常编译，我也能接受。尽管我们是因为要开发 Matisse 可视化界面设计器才使用这个包，但在官方的 API 中，`java.awt.peer` 是没有公开的，也就是说我们其实不该使用该包中的内容，所以出现问题也正常。

但问题出在我们实现了 `javax.sql.RowSet` 这个接口，而这个接口在新版本中添加了一些方法^①使得代码在新版本上无法编译通过。为了解决该问题，必须使用些小技巧才能绕过去，对此我们很是不爽。这个接口是一个标准的 API，所有人都会用到这个接口。注意，所谓的“用”，自然是包括了“实现”该接口。我们确实实现了这个接口，因为我们需要对 SQL 调用进行拦截，所以就实现了这个接口，然后在我们的实现类再调用真正的 `RowSet` 对象。但这样就引起了不兼容问题，因为这个 API 的开发商为该接口添加了新的方法。这就使得相应的工作比较麻烦了。在两个不同的 Java 版本上编译相同的代码比较困难。幸好我们只是在内部使用了 `javax.sql.RowSet`，并没有把它重新导出，这算是不幸中的大幸了。

与此同时，我们把 `org.w3c.dom` 包里的接口给重新导出，允许第三方使用。这样引发了同样的问题：这个 API 的维护者总喜欢给这些接口添加一些新方法。与在内部使用 `RowSet` 不同，我们把这些接口作为方法的返回值，从而暴露给外部使用。更惨的还在后面，为了提高性能，那些返回值其实只是我们用 Decorator 模式封装的内容，并非真正实现。实现这种封装并不麻烦，只需要一点小技巧结合 `java.lang.reflect.Proxy` 这个代理类就可以了，但还是引起了编译问题。我们完全没有办法编译通过这些代码，只能通过动态调用来解决。这样，不管底下的具体实现添加新方法的频率有多高，我们都能处理。我们使用 `Proxy` 可以正确处理各种对象的方法调用，这样做可以解决问题，但编写和维护这样的代码并不是一件让人最开心的事情。

因此，更好也更简单的处理方式就是在演进过程中保持兼容性。事实已经证明，能够保持 API 兼容的项目会更容易赢得用户的信任。用户在使用这些 API 时，既不需要利用对类进行再包装的这种技巧，也不会因为第三方类库的一个兼容性问题而引发多诺米骨牌一样的风暴，更不用在面对聚集开发力量的新版本时，还委曲求全地用老版本。

10.2 只暴露抽象内容

在上一节中，我们对重新导出第三方 API 可能引发的问题给出了警告。但不仅仅是第三方 API 会引发问题。如果你将自己的 API 暴露出去，也会引发相似的问题。毕竟，你暴露的内容越多，留给 API 使用或演进的余地也就越小。

NetBeans 中代码自动提示 API 设计上所犯的错

在 NetBeans 项目中，我们有一个用来支持代码自动提示的 API。这个 API 允许任何人将代码自动提示的实现以扩展的方法加到编辑器中，这个实现可以在用户输入时提供各种提示功

^① JDK 6 为 `javax.sql.RowSet` 中添加了很多方法，作者所说的应该是 NetBeans 6 系列，同时支持 JDK 5 和 JDK 6，所以有这个问题。——译者注

能。这个 API 使用了 Swing 的文本编辑控件。在运行时，它会查找所有注册的扩展提供器，然后将相关的上下文环境（即文档）传给这些类，同时还要告知提示信息将显示的位置。

出于某些原因，这个上下文环境参数中还带了一个 `getTextComponent` 方法，该方法可以返回一个 `JTextComponent` 控件实例对象。这个方法就像一个无限放大的后门，将其他的 API 都暴露了出来，包括 Swing 的控件继承体系、所有的方法、整个 AWT 类库等。结果就是这个支持代码提示功能的 API 变得非常庞大，使用该 API 所涉及的内容已经远远超出 API 设计者的初衷。拿到 `JTextComponent` 这个对象实例能做些什么呢？答案就是你可以随心所欲地做任何事情。

如果对 API 不加限制就会引发一堆问题。当我们想在 NetBeans 中支持将某种编程语言嵌入到另外一种编程语言时，如在 HTML 中使用 JavaScript 或者在 JSP 中使用 Java，就会发现很多问题。对于每一种支持语言，我们都提供了代码自动完成提示功能，而且想要尽量避免环境因素的干扰，比如 HTML 和 JSP。在调用代码提示功能时，我们希望能够正确地配置上下文环境。但上下文环境中的一部分内容就是 `JTextComponent`。我们怎么可以为整个文档返回一个上下文环境呢？完全做不到。也许你会说，提供一个假的上下文如何？也许从技术角度来说，方案是可行的，但哪些返回 `JTextComponent` 的方法需要我们来做一个 Mock 呢？`getParent()` 方法能行吗？这部分需要显示出来吗？问题还不止这些。

如果我们当时能够在 API 中不暴露这个控件，能够详细地列举用户在计算自动完成提示时可能需要的方法，那么今天的境地可能就会好得多了。

想象一下我们在自己 API 的某一个方法里暴露了 `java.io.File` 这个类，会发生什么事情呢？这样会使得 API 的所有用户在调用这个 API 方法时，可以拿到文件对象，从而可以读取这个文件的内容，同时还可以向这个文件中写入自己的内容，这样做通常也是正常的。但通过这个文件结合相对路径，就可以查找到其他相关文件。所以，一旦把一个文件暴露给外部的 API 方法，就完全对其失去控制了。其实，这个方法只需要提供一个流，允许 API 用户能够读到一些信息而已。但直接暴露文件的方式却使得用户可以利用这个文件进行更多操作，从而可能引发更多问题。

伪 封 装

现在看一下 NetBeans 的任务列表 API。这个特定的 API 允许开发人员向任务列表窗口中添加单独的新任务。这些任务与用户文件之间有关联。当一个用户双击一项任务时，就会在工作区里打开一个文件，并且光标会自动定位到一工作区的特定位置，这样用户就可以直接来调整或校正相应内容来完成相关任务了。

当我们为这个 API 设计原型时，我们定义了一个 `Resource` 的概念。所谓的 `Resource` 是指一个文件对象。这个 `Resource` 类提供了两个方法来取得资源名称和输入流，对于大部分 API 来讲，这两个方法已经够用了。但随后还是出现了不够用的情况，于是我们就添加了一个 `getter` 方法，可以返回 `FileObject` 对象，而这个返回的对象其实是对 `java.io.File` 的一个包装。

但 `FileObject` 是 NetBeans 平台对 `java.io.File` 类的一个包装，可以通过这个类来取得数

据流和一个相应的名称。Resource 原来的概念已经够复杂了。事实上，Resource 这样设计就没有提供任何的封装，因为可以通过一个 getter 方法来返回 FileObject 对象。所以最后我们决定完全放弃 Resource 这样一个抽象的内容，直接使用 FileObject 类算了。这个改变也破坏了向后兼容性，但我们还是坚持这样做，因为我们还没有对外发布任何版本呢，一切都在我们自己的控制之下。

不管怎么样，最后，NetBeans 通过定义 FileObject 类并允许外部使用这个类，避免了直接将 java.io.File 暴露给用户，这样就允许第三方可以脱离硬盘文件资源来实现自己的 FileObject。这样，完全可以为一个内存中的虚拟文件来创建一个任务项。虽然方案简化了，但任务列表的 API 很好地实现了对外封装和重用。

对 API 进行封装，其实意味着在使用一个 API 的时候，其上下文环境的配置会比较简单，从而大大提高其潜在复用度。如果轻易就能在一些简单的 String 或者 CharSequence 类上调用代码完成，那么它使用的频率要远远高于创建自己的 JTextComponent 时的情形。所以在定义 API 的环境时，一定要考虑合适的封装方式，要分析常见的用例，也要想到那些不常见的场景，从而在设计时能够在这两者之间取得一个平衡。要知道在定义一个 API 的时候，设计者对 API 的用户往往是一无所知的，对于他们所要解决的问题，也并不一定清楚。所以设计 API 时，如果想通过特殊的环境配置来将 API 变得复杂，从而限制用户使用这样一个 API，这种做法非常不明智。请一定牢记这一点：少就是多。API 中暴露的内容越少，那么重用就会更容易。

10.3 强化 API 的一致性

在使用一些 API 的时候，会发现对于同一个功能，可能提供了多种操作方式，当然结果也是相关的。而且这些操作方式是有关联的，毕竟它们的操作结果相同（至少是相关的）。其实这也正是 API 设计者所期望的，并从另外的角度反映了老版本 API 中的一些问题。虽然真正用的时候可能并不尽如人意，但有些用户还是希望 API 能够提供此类操作。

如果想利用一个简单类库来完成某个功能，却发现有多种操作方式，其实很容易就知道哪些地方出了问题，要进行修正。还记得 7.1 节提到的“简单功能类库”吗？就是指那些只提供了实现功能的类库。下面的代码就是典型的例子。

```
URL url = new URL("http://www.apidesign.org");
assertEquals(url.toString(), url.toExternalForm());
```

如果希望调用 URL 类的 toString 和 toExternalForm 这两个方法能返回相同的字符串，就需要写一个自动化的单元测试来进行验证。这个例子写起来很简单，因为 java.net.URL 是一个 final 类，有自己固定的实现，所以自动化单元测试可以保证执行结果正确。

但对于一个模块化系统，它远比“简单功能类库”复杂。我们在设计模块化类库的时候，只需要定义 API 对外公开，而其他单独的模块会根据这些 API 来提供具体的实现。如果是这样，那么表面要验证一个 API 的功能，其实要验证的却是它背后的所有具体实现。出现问题时，也不是说只需要修正当前类库里面的代码就能搞定。还有可能要强制要求该类库所有的第三方开发商都

要修正相关代码中的问题。

NetBeans 平台中 Lookup 设计上的不一致性

在 7.2 节中已经指出：NetBeans 使用了 Lookup 这样一个通用的 API 来帮助开发人员发现各种服务。在 NetBeans 中有多种方式来查询已经注册的服务。

- 最常使用的方法就像下面的代码一样，通过 lookup 方法来查找某一个类或者接口的具体实例化对象。

```
public abstract <T> T lookup(Class<T> clazz);
```

- 通过 lookupAll 方法来查找某一个类或者接口的所有具体实现的实例化对象。这个方法会返回一个实例化对象的集合。

```
public abstract <T> Collection<? extends T> lookupAll(Class<T> clazz);
```

- 通过 lookupAll 方法来查找某一个类或者接的所有具体实现类。这个方法返回的不是实例化后的对象，而是一个 java.lang.Classes 的集合。

```
public abstract <T> Set<Class<? extends T>> lookupAllClasses(
    Class<T> clazz
);
```

看完了 Lookup 的使用方法，估计在使用时，用户肯定希望在调用 lookup(clazz)这个方法时，它的返回值是一个非 null，或者说在调用 lookupAll(clazz)的时候，返回的集合中也至少应该有一个实例化对象。你也希望 lookupAllClasses(clazz)返回集合中的类和 lookupAll(clazz)中返回集合中的实例化对象也应该匹配。用户有这种想法是很正常的，作为这个 API 的作者，我当时也是这么想的。

在 NetBeans 平台上，Lookup 这个类不是一个单例，它有很多实例化对象。还好大部分的 Lookup 类都是继承自我们提供的基类（如 AbstractLookup 或者 ProxyLookup 等），而且创建 Lookup 对象实例时，也是使用我们提供的行之有效的工厂方法（如 Lookups.fixed 等），而且这些工厂方法都进行了严格的测试，可以保证所有的方法都具有一致性。要做到这一点，并不容易。不过，在多年开发之后，几乎所有的用例，包括那些不太明显的用例也都得以证明并进行了测试。这个具体的实现表现得非常稳定，如我们的预期一样平稳运行。从这个角度来讲，Lookup 的 API 和一个“简单功能类库”还是比较相像的。

但 Lookup 的 API 也仍然是 NetBeans 平台的一个组成部分，只不过模块化程度比较高而已。所以它仍然可以被看作是一个“模块化库”；这也就表示完全可以不使用该模块提供的基类，而是重新实现一个它们自己的 Lookup 类。由具体的开发人员来自行保证所有方法的具体实现保持一致。Lookup 这个 API 表面上看起来并不复杂，但一旦涉及代理、过滤器以及和其他 Lookup 对象合并调用时，就会变得相当复杂。结果，一些实现类没有正确地实现 Lookup 设计时的目标，也就不能完全履行 API 协议。

如果一个 API 的具体实现出现了不一致的情况，这决不是一件好事。但在这个案例里，情况还要更加复杂些。在一个基于 NetBeans 平台开发的程序中，假设 95% 的 Lookup 运行正

常，而另外 5% 则时不时地出点儿问题。这样在调用 Lookup 这个 API 的时候，用户完全不知道怎么回事。在查找服务时，并不一定能找到自己使用的到底是哪一个 Lookup，因为这些 Lookup 的实现类通常是 package 一级的访问方式，使用 Decorator 设计模式处理以后才公开出来。同时，由于大部分的实现都不存在问题，用户可能根本不会意识到有一些功能已经不能正常运行了。进行测试时，可能所有的用例执行正确。但一旦产品上线，换到具体的环境中，就可能会有一些 Lookup 的实现被破坏了，这样由于不一致性，可能会使得整个产品出现意外的故障。

如果要解决此类问题，方法并不是很多。首先要定义一个“策略”，也就是说在 Javadoc 中加入大量警告信息，提醒实现这个类的开发人员在实现时一定要保持一致性。虽然这样做用处不大，但也聊胜于无。但我仍然要提醒你，这样做可能真的无济于事。因为这样做了也许可以将这种不一致性的数量下降一半吧，但还是可能会有大量的问题存在，这些问题甚至还可能会增加。一旦一个客户发现了一个无效的对象实例时，运行代码也会出现问题，通常是会抛出一个像 `NullPointerException` 或者 `ArrayIndexOutOfBoundsException` 这样的异常，它们根本不含有什么描述信息。更让人难受的是，因为这个对象出了问题，所以它的相关信息非常少，甚至连出现异常中相关的类名都找不到。结果就是，虽然收到了一个运行错误的信息，但要定位这个问题却要花上几个小时甚至是几年。所以在 Javadoc 添加警告信息这种方式，只是把一些问题推给别人，对于问题的解决其实没有任何实质的用处。

解决此类问题的一种更好方式就是编写 TCK。我在 9.4 节中曾指出，每一个模块化的类库，都应该配备一个测试套件，这样开发人员就可以利用这个套件来验证相关具体实现是否正确。这个方案的确可以解决问题，但它也有一个前提条件：是所有开发人员都会使用该套件进行测试工作。我敢肯定的是，有相当一部分 API 的用户只是去实现 API，而忽略了对具体内容的测试。最后，他们会忽略所有的内容，包括一些指导性内容，他们只想尽快地完成自己的工作。学习怎么配置 TCK 对开发人员来讲是一项额外的、讨厌的工作，所以他们一上来就会自动忽略这一点。只有那些以前吃过这种亏的人，才能意识到这样做的危险性，不吃一堑，不长一智。在这种环境下，很可能要花上好几天时间才能找到一个神秘的 bug，其实如果使用 TCK，这个 bug 可以在几分钟内定位。TCK 是一件辅助开发的利器，但只有那些勤勉的程序员才会使用它。在我们 API 的用户中，有相当一部分人完全算不上勤勉，因此不敢保证所有具体实现都是一致的。那么，如果要保证一致性，只有一个方法了，那就是强制执行 TCK 的功能。

约束和 AOP^①

读到这里，我可以想象得出，很多读者能够猜出正确的解决方案：使用一种约束性语言，

^① AOP 为 Aspect Oriented Programming 的缩写，意为：面向方面编程或面向切片编程，可以通过预编译方式和运行期动态代理实现在不修改源代码的情况下给程序动态统一添加功能的一种技术。目前使用比较多的 AOP 实现分别是 AspectJ 和 Spring，前者是预编译方式，将 AOP 技术直接加入到二进制代码中，而后者则是使用 Java 自己的动态代理来实现 AOP 的，在运行期做二进制代码增强。——译者注

或者使用更好的面向方向编程方式（下文都简称为 AOP）。也就是说需要定义一个额外的方式为一个方法或者一个类添加一致性约束。使用特定的编译器编译代码，或者是做二进制增强处理，可以生成那种增强类型代码，从而对所有的具体实现一致性进行检查。

这是一种可行的方案。但在序言中提到“学习编写 API”，所以我们想尽量只用 Java 来解决所有问题，避免引入新的技术。所以使用这种约束方式来解决问题不是本书所要讨论的方案。那感觉就像引入了新的编程语言及概念。所以只好让那些 AOP 的拥护者失望了，我们希望在现有工具的范畴内能够找到一种可以加强一致性的可行方案。

如果要约束一个方法的行为，最好的方式就是避免这个方法被覆盖。如果要约束一大堆方法的一致性，最好的方法则是让相应的类变成 `final` 类，然后在自己的代码内部检查一致性。这样做可以把一个类变得像“简单功能类库”一样，不管用户做什么事情，都处于自己代码的控制之下。事实上，这样会降低模块的扩展性。所谓的扩展性，其实只是通过实现额外的接口，并结合工厂方法拿到接口的具体实现，就像下面代码的展示。

```
public abstract class Lookup {
    /** 只允许相同包中的类访问该构造函数 */
    Lookup() {
    }

    public <T> T lookup(Class<T> clazz) {
        Iterator<T> it = doLookup(clazz);
        return it.hasNext() ? it.next() : null;
    }

    public <T> Collection<? extends T> lookupAll(Class<T> clazz) {
        Iterator<T> it = doLookup(clazz);
        if (!it.hasNext()) {
            return Collections.emptyList();
        } else {
            List<T> result = new ArrayList<T>();
            while (it.hasNext()) {
                result.add(it.next());
            }
            return result;
        }
    }

    public <T> Set<Class<? extends T>> lookupAllClasses(Class<T> clazz) {
        Iterator<T> it = doLookup(clazz);
        if (!it.hasNext()) {
            return Collections.emptySet();
        } else {
            Set<Class<? extends T>> result =
                new HashSet<Class<? extends T>>();
            while (it.hasNext()) {
                result.add(it.next().getClass().asSubclass(clazz));
            }
        }
    }
}
```

```

        return result;
    }
}

```

这可以说是一种非常安全的解决方案。所有的检查都集中在一个地方，因为只有一个入口，因此能对所有用户的方法调用都进行检查。这个入口可以把需要的一致性检查、参数校验等都集中在一处。而且，这种检查可以根据环境进行调整，也就是说，只在测试环境下进行校验，而在上线系统中就关闭校验，特别是校验会引起比较大的性能问题时，这种处理显得尤为有用。对于每一个 API 我们都有一个这样的位置来保证其一致性。

在第 8 章中提过，可以将 SPI 和 API 进行分离。分离是设计的一把利器，它支持 API 的开发商把来自其他客户端的调用进行转换并进行检查，然后再调用其他的具体实现。由于模块的实现者是在不同环境进行开发的，对于由未知模块提供的不同实现，这样的处理方式统一了行为，减少了一致性。

10.4 代理和组合

面向对象语言的核心概念就是继承，所有的内容都是围绕着这个概念展开的。在 C++、Java、Smalltalk 以及现代大部分面向对象语言中，继承可谓是一等公民。不同的语言，使用不同的方式来支持继承。人们一直宣扬，应该尽可能地去使用继承。的确，继承为代码复用提供了一种方便的途径。但在设计优秀的 API 时，继承其实用处不大。在为用户提供一个功能类库的时候，应该让它们在使用类库时，无须花费大量的精力来了解类库的内容，所以在设计 API 时，是否允许外部的继承一直是一个争议的话题。在 5.7 节中，我也说过对于一个类库来说，其对外的 API 并不应该暴露深层次的继承关系。所以本节会进一步讨论该问题，并考虑是否有其他的备选方案。

下面的代码算是一个相对不错的 API，就用这段代码来开始下面的讨论。这个 API 看起来非常简单，也不会引起什么太大的问题。让我们定义一个简单的类并封装一些基本的算法。

```

public class Arithmetica {
    public int sumTwo(int one, int second) {
        return one + second;
    }

    public int sumAll(int... numbers) {
        if (numbers.length == 0) {
            return 0;
        }
        int sum = numbers[0];
        for (int i = 1; i < numbers.length; i++) {
            sum = sumTwo(sum, numbers[i]);
        }
        return sum;
    }

    public int sumRange(int from, int to) {
        int len = to - from;

```



```

        if (len < 0) {
            len = -len;
            from = to;
        }
        int[] array = new int[len + 1];
        for (int i = 0; i <= len; i++) {
            array[i] = from + i;
        }
        return sumAll(array);
    }
}

```

这段代码可以看作是一个 API，和我们每天写的代码都差不多。但按我的标准来说，这还不算一个稳定的 API，不应该公开给他人使用，因为它违反了第 5 章中的一些基本规则，如“只公开必要的内容”。但在这里用这段代码却可以向大家展示，如果在设计类的时候，不遵循本书给出的建议而直接把这个类作为 API 公开出去，会引发什么样的后果。

作为读者，有没有发现 *Arithmetica* 这个类中存在的问题呢？还是让我从阿米巴变形虫模型的角度来看一下这些代码做了些什么事情。首先，这段代码的确完成了我们设计时的功能。所有的三个方法都有各自的上下文参数，并根据这些参数返回正确的结果，下面的单元测试可以证明这一点。

```

public void testSumTwo() {
    Arithmetica instance = new Arithmetica();
    assertEquals("+", 5, instance.sumTwo(3, 2));
}

public void testSumAll() {
    Arithmetica instance = new Arithmetica();
    assertEquals("+", 6, instance.sumAll(3, 2, 1));
}

public void testSumRange() {
    Arithmetica instance = new Arithmetica();
    assertEquals("1+2+3=6", 6, instance.sumRange(1, 3));
    assertEquals("sum(1,10)=55", 55, instance.sumRange(1, 10));
    assertEquals("sum(1,1)=1", 1, instance.sumRange(1, 1));
    assertEquals("sum(10,1)=55", 55, instance.sumRange(10, 1));
}

```

因此，我们设想的功能得到了实现。现在有人需要继承 *Arithmetica* 来实现一个新的类，并在该类中添加一些成员。这样做，好像也没有什么问题，也就是说，这个类库的实际表现至少和预期的一样。那么问题到底出在哪儿呢？是因为这个应用所能完成的功能已经超出了原有的设计，它不仅仅能算加法。令人吃惊的是，这个优秀的类库还能求数字阶乘。

```

public final class Factorial extends Arithmetica {
    public static int factorial(int n) {
        return new Factorial().sumRange(1, n);
    }
    @Override

```

```
public int sumTwo(int one, int second) {  
    return one * second;  
}  
}
```

可以说通过覆盖 `sumTwo` 这个方法，使得原来 API 的功能完全改变了，这实在是太意外了！但这个修改也有自己的用途，而且运行得很好。在类 `Factorial` 中通过重载 `sumTwo` 这个方法，可以说是改变了其父类所有方法的行为。也就是说，这个类就变成了一个能够计算数字阶乘的类。

检查签名

在 Java 语言中，通过接口的方式来调用具体的实现代码可谓是一切功能调用的基础。说到类继承，它背后会使用一张虚方法表来实现很多功能，4.2.2 节对此已经有了详细的说明。

当你写了一个类以后，而且在该类中有一个可以被子类覆盖的方法，这其实意味着在这个类的虚方法表中添加了一行关于该方法的信息。一旦这个方法被调用，此时，会首先查询这张虚方法表，然后根据得到的信息来判断到底应该调用哪一个方法，然后跳转到这个方法的入口开始执行代码。

虚方法表也是从父类继承下来的。当你继承一个类的时候，也会自动继承了该类的虚方法表。一旦类中添加了新方法或者是子类覆盖了方法，就会在这个虚方法表中添加新的记录，指向新的方法。对于一个子类，这就是对虚方法表的继承。

现在来看一下前面的那个例子，假设允许被继承的类对外公开出去，这也就意味着，只要这个类中有一个 `public` 或者 `protected` 级别的非 `final` 方法，就表示外部的用户可以通过这个方法来做他们想做的了。如果说不管三七二十一，直接就建议你在代码中进行多种检查，这个未免有些危险了。如果经常有这种内容会被暴露出去，还是应该谨慎处理。

继承机制就是如此，一旦允许别人继承自己的内容，都要小心再三。

说到阿米巴变形虫模型，其实是指一个应用程序可能根据实际情况出现变化，使得其功能远远超出原先的规划。当然，也许你不把这种情况看作是个问题。但这种变化可谓精彩之极！我们写了一个类，然后作为 API 公开出去，然后用户使用并扩展这些 API，在此基础上开发出很多新颖、特殊的功能，可谓创意十足。或许可以把这种情况看作代码复用的绝佳例子。大家都说，没有面向对象编程，就不可能做到代码重用，这个例子充分地证明了面向对象的优越性，可以帮助编程人员开发出更好的软件。人们一直在鼓励软件复用！

我们的确需要代码复用，这也是我们学习开发 API 的根本目的。但需要通过规划来做到复用。不管三七二十一，就把一样自己都不是很清楚的功能暴露给他人使用，其实不是真正的复用，只会引发更多的问题。假设在 2.0 版本中你发现了一个更好的 `sumRange` 方法，下面的代码就是 2.0 版本引入的新算法。

```
public int sumRange(int from, int to) {  
    return (from + to) * (Math.abs(to - from) + 1) / 2;  
}
```

通过调整算法可以有效地提高运算速度。但这个修改却引发了不兼容问题。尽管前面所说的那个应用程序没有对自身代码进行任何调整，但在从 1.0 版本升级到 2.0 版本的过程中，应用形式发生了改变。新的算法里根本没有去调用原来的 `sumTwo` 方法，所以原来调用 `Factorial` 类中的 `sumRange` 方法来计算阶乘就行不通了。这个改变强劲得如同地震一般：只要调用了这个方法的代码都会出现问题，相当多的应用程序可能会完全无法运行。这还是代码复用吗？分明是程序的末日！

经验告诉我，这种情况几乎都是因为允许继承而引发的。原因很简单。虽然继承是面向对象技术中支持代码复用的主要手段，但事实上，它也允许为一个类留下了太多不安全的后门，这些后门可谓远超设计时的规划。就像前面这个 `Factorial` 例子，它扩展出来的功能超出设计者预期，特别是第三方使用者没有注意到设计者意图时，情况更为严重。解决该问题的更好方法是使用代理，加上组合，避免使用继承。

大量类引发的问题在于，这些类会定义大量的虚方法（也定义了很多具体实现），同一个类中的这些虚方法之间是有各种关联关系的。每一个方法都会调用别的相关方法。因为被调用的方法是虚方法，也就说子类是可以覆盖这个方法，并完全改变整个类的意义，从而将原方法的调用跳转到新覆盖的方法上。上面给出的 `Factorial` 例子就是一个最好的诠释，它继承了 `Arithmetica` 这个超类，从而改变了该类的对外行为。这个问题其实就是因为在一个类中，其方法之间有相互依赖关系，但在实现一个方法的时候，却只考虑其实现细节，而忽略与其他方法间的关联关系。如果你要继承的类中有着大量的方法，就必须仔细阅读其源代码以了解相关内容。否则在覆盖某个方法的时候，就无法清楚地知道你改变了原来类的哪些行为。如果哪个程序员要去了解 `javax.swing.JComponent` 这个类中多个方法间的关系，那么不看源代码，根本就不可能有一个清楚的概念。想来，这也正是 Sun 为什么把 `Swing` 的源代码发布出来的原因吧。没有这些源代码，相信没有人能正确地继承一个 `Swing` 的类。即使有了源代码，想完成一个继承工作还是需要做大量的工作，不过至少还是提供了可能性。当然了，这完全违背了我们所提倡的“无绪”。

如果需要去读一个 API 的源代码，那么其设计上一定存在问题。事实上，如果一个类中的方法间有着大量的关联关系，我敢肯定，它的设计一定存在问题。所以我不认为设计成这样的类算是好的 API。我给出的第一个建议，其实在 5.3 节已经提过了，就是把类定义成不可继承。实际上，对于一个不可继承的 `final` 类，由于不可继承，所以也不会出现虚方法。也就不会引发 `Factorial` 这样的问题了。但有一点很让人惊讶，那就是使用 `Java` 接口也可以有效地避免虚方法间的关联关系。如果使用 `Java` 接口，那么所有的方法都是抽象的且虚拟的。换句话说，一个接口中的方法是不可能具有具体实现的，因此不可能产生各种关联关系。很显然，这也能避免 `Factorial` 这类问题。

如果系统允许使用 `final` 类和 `Java` 接口，那么 API 中可能会出现继承问题的源头就只有那些既不是 `final` 类也不是 `Java` 接口的类型，也就是那些带有一些具体实现内容的一般的类或者是抽象类。如果这种情况也会引发问题，还有没有办法可以更安全地避免这个问题呢？是否可以使用一个 `final` 类结合一个接口来替换一个抽象类呢？当然可以，只要使用组合模式就可以做到，下

一节中我会就此具体展开讨论。

组合并不是一等公民

在常见的面向对象编程语言中，可以通过 `extends` 之类的单个关键字来实现继承，但对于组合，就没有这么简单了。实现一个组合功能，可能需要手写大量的代码，如定义一个数据结构，然后通过代理的方式加以调用等。这样可能会使得开发人员不愿意使用组合模式，有时也确实会产生这样的后果。但如果进一步分析一下这个问题，就会发现需要手工完成的繁琐工作主要在于定义这个组合模式的结构，与调用它的用户是没有关联的。与继承相比，组合模式的支持率较低，所以在面向对象编程语言中来实现它，会显得不自然，也很困难。对API的用户来说，却是件很自然的事。要知道API只有一个作者，却可能有很多人要用它。也就是说对于API的设计者，无论花多少时间来设计一个正确的API都不为过，而且要保证用户在使用该API时，应该尽可能地简单，不需要深入了解API背后的内容。

如果要避免可继承类^①中众多虚方法间的关联关系，第一步要做的事情就是搞清楚一个类中方法的含义。我们为什么要将方法声明为 `public` 或者 `protected` 呢？为什么不用 `final` 或者 `abstract` 来定义一个方法呢？提醒一下，因为 `package` 或者 `private` 一级的方法对于API的用户来说是不可见的，所以也不在这里加以讨论。除此以外，如果在一个类中定义一个方法，通常有三个基本原因。在某些类中引入一个方法，其功能不一定只有一个，这些原因都有可能涉及，所以分析时需要综合来看。但从最基础的角度来看，如果要在一个API中定义一个方法是需要三个原因的。

(1) 首先，也是最重要的，外部可以通过一个方法调用来实现某些功能。一旦一个用户能够拿到某个类的对象实例，那么就可以调用该类定义的方法。前文已经说过了，这一步骤就是先在虚方法表中查找一个合适的方法项，然后再去调用相关方法。这些类型的方法如果按照理想的方式进行设计，会被声明成 `public` 和 `final`。从而保证这些方法只能按照设计师的意图执行操作，不会超出此设计范围。

(2) 利用虚方法的另一个原因，就是让它成为可以被具体实现替换的插槽，也就是说一个可以在子类中被覆盖的方法。在这种情况下，一个子类是可以通过覆盖父类的虚方法来替换父类的实现。只要通过该子类的一个实例化对象来调用该虚方法，其实都会调用子类中的新方法，而不是父类中原有的实现。对于这种情况，这些方法应该声明为 `protected` 和 `abstract`。子类必须实现这些方法，而且因为是 `protected` 级别，所以这些方法也不会被外部调用。同时由于这些方法在基类中没有提供实现，所以子类对它的调用也没有任何意义。

(3) 方法调用的第三种方式，就是由子类通过 `super.methodName` 这种方法来调用父类的方法，而且这种情况不需要覆盖父类的方法。处理这种情境的方式很简单，只要把这些方法声明为 `protected` 和 `final` 即可。然后，这些方法就不再是虚方法，而且只允许子类来调用的方法。

^① 原文这里用的是 `abstract classes`（抽象类），但其实不只是抽象类中有虚方法，只要是可继承类中都可以包含虚方法。——译者注

对于在类中定义方法，上文给出了三种方式，相比面向对象语言中对方法进行定义的多种方式，它们只是一个子集。也就是说，面向对象语言在这方面具有更丰富的语义，但这么丰富的语义对于 API 设计来讲其实会凸显得更加模糊。如果你在一个 API 类中看到了一些声明为 `protected` 和 `abstract` 的方法，那么根据上文所说，可以很清楚地知道如何来使用这些方法——覆盖并实现。这样的设计看起来非常清晰，如果声明为 `public` 和 `final` 或者是 `protected` 和 `final`，也非常清晰。API 的用户看到这种设计，会很清楚其用法。一个 API 其实就是开发者和用户之间的一种沟通方式，在这个沟通过程中，应该尽可能清楚地进行说明，而不是把大量的信息给隐藏起来。我们得出结论：如果 API 中声明成以上三类访问方式的方法越多，那么其 API 对外的描述会更加清楚。但面向对象语言却提供了其他声明方式，而且很多人更喜欢用那些其他的声明方式。这些声明方式，其语义往往不止一种（表 10-1 给出了一个详细的说明）。使用这些声明来声明一个方法，就表示其使用方式不止一种，很容易引起误用的情况，第 8 章中对此已经加以说明了。

表10-1 在设计API时，Java方法访问级别的声明方式的不同语义

访问级别	主要含义	额外含义
<code>public</code>	允许外部代码来调用该方法	子类可以覆盖该方法 子类也可以调用该方法
<code>public abstract</code>	子类必须实现该方法	允许外部代码来调用该方法
<code>public final</code>	允许外部代码来调用该方法	无额外含义
<code>protected</code>	允许子类代码来调用该方法	子类可以覆盖该方法
<code>protected abstract</code>	子类必须实现该方法	无额外含义
<code>protected final</code>	允许子类代码来调用该方法	无额外含义

为了提高 API 的语义清晰度，同时避免不必要的副作用，最好是避免使用那些可能存在的问题的方式来声明一个方法。表 10-2 给出了这样一个例子，显示了调整前后的内容。下面的例子很巧妙地描述了在设计 API 时，由于使用了特定的方法声明方式，实际上是给用户开了后门，其后果也是超出了设计者的预期。通常情况下，有些声明方式其潜在的副作用具有很强的隐蔽性，并不会在一开始就暴露出来。所以对于 API 设计者，这正是他们需要理解这些声明方式的重要原因，只有这样才能避免此类副作用。

表10-2 调整那些功能不明确的方法

原 代 码	调整后的代码
<pre>public abstract void method(); public void method() { someCode(); } protected void method() { someCode(); }</pre>	<pre>public final void method() { methodImpl(); } protected abstract void methodImpl(); public final void method() { methodImpl(); } protected abstract void methodImpl(); protected final void someCode() { } protected abstract void method(); protected final void someCode() { }</pre>

现在，当类中的方法存在多种语义时，我们知道该如何重新编写该类，以保证其中的方法只有单一而且确定的语义。下一步，我们要关注如何对这些类再做进一步的转换，以保证每个版本都有着更清晰的语义，也就是每个类中的方法都有更清晰的语义。这样才算把组合做到了极致。如果像下面代码中的 `MixedClass` 类一样，每个类都是由单一语义的方法组成，那么就可以使用两个类和一个接口来替代原来的抽象类。

```
public abstract class MixedClass {
    private int counter;
    private int sum;

    protected MixedClass() {
        super();
    }

    public final int apiForClients() {
        int subclass = toBeImplementedBySubclass();
        sum += subclass;
        return sum / counter;
    }

    protected abstract int toBeImplementedBySubclass();

    protected final void toBeCalledBySubclass() {
        counter++;
    }
}
```

其中第一个类用来作为客户端要调用的 API，而接口则是留给用户来实现相应功能，第二个类则用来支持系统的回调。

```
public final class NonMixed {
    private int counter;
    private int sum;
    private final Provider impl;

    private NonMixed(Provider impl) {
        this.impl = impl;
    }

    public static NonMixed create(Provider impl) {
        NonMixed api = new NonMixed(impl);
        Callback callback = new Callback(api);
        impl.initialize(callback);
        return api;
    }

    public final int apiForClients() {
        int subclass = impl.toBeImplementedBySubclass();
        sum += subclass;
        return sum / counter;
    }
}
```



```

    }

    public interface Provider {
        public void initialize(Callback c);
        public int toBeImplementedBySubclass();
    }

    public static final class Callback {
        NonMixed api;

        Callback(NonMixed api) {
            this.api = api;
        }
        public final void toBeCalledBySubclass() {
            api.counter++;
        }
    }
}

```

尽管调整后的结构比原来要复杂，甚至是对于 API 的设计者来说，复杂度也高了很多，但对于 API 的用户来说，其结构却要更加清楚，也更容易理解。对于那些不需要继承功能的用户，只需要关心 `apiForClients()` 这个方法，特别是把这个方法移到一个工厂类中，那么关注范围就更小了。对于用户而言，这个 API 变得简单而且容易理解。还有一些需要去使用继承来完成某些功能的用户，现在也清楚地知道他们要去实现 `Provider` 接口。对于 `Callback` 接口的用户，他们也很清楚地知道真实调用的方法是由 `Callback` 具体实现所提供的。说到这里，我们可以知道，对于所有该 API 的用户，通过这种解耦，整个 API 的语义变得非常简单。

```

@Test public void useWithoutMixedMeanings() {
    class AddFiveMixedCounter implements NonMixed.Provider {
        private Callback callback;

        public int toBeImplementedBySubclass() {
            callback.toBeCalledBySubclass();
            return 5;
        }

        public void initialize(Callback c) {
            callback = c;
        }
    }

    NonMixed add5 = NonMixed.create(new AddFiveMixedCounter());
    assertEquals("5/1 = 5", 5, add5.apiForClients());
    assertEquals("10/2 = 5", 5, add5.apiForClients());
    assertEquals("15/3 = 5", 5, add5.apiForClients());
}

```

然而，这些代码与原来只使用一个 `Mixed` 类相比，确实看起来更加复杂。原因很简单，我们使用了组合来替换原来的继承。但因为在目前的面向对象语言中，组合不是一等公民，所以在设

计 API 的时候，其代码显得比较冗长一些。

如果想正确地设计 API，让每一个 API 类都有明确的功能以及清晰的定义，那么组合可谓是一件利器。它不仅清楚而且优雅地为 API 不同类型的用户分离出相应的接口，而且该解决方案不会使其应用变得更加复杂。在 NetBeans 中，我们一直建议 API 的设计者应该尽可能地使用组合。如果要使用 Java 来设计 API，我也建议采用组合技术。

10.5 避免 API 的误用

我一直坚持一个观点，那就是 API 不仅仅只是类、接口加上 Javadoc 这样一个简单的组合。一个 API 不仅仅要提供入门教程，还要提供从浅到深的技术指导，如类库在运行时的各种功能都应该加以说明。对于 API 来说，不管哪个层次的内容，都要兼顾。哪怕只有一点照顾不到，最终用户就可能要承受一些不良后果。如果只考虑那些深层次的内容，很多用户可能因为门槛太高就直接放弃了。如果不提供 Javadoc，用户可能连一个单独的 API 元素都无法理解。如果执行时出现问题，用户就会抱怨，也许就直接放弃了。但还有一点是最重要的，可谓重中之重：所有这些层次之间都需要保持一致性。一个 API 不能在某个层面说能够完成某种功能，然后一转眼又在另外一个层面说，根本不支持这个功能。

事实上，如果一个项目把自己夸成天上有地下无，但其用户却发现该项目有一大堆问题，自然对此项目不会满意。教程一般来说只要可行就都是正确的，它至少可以保证根据教程编写的代码能够正确地执行而且完成一些功能。但有了教程，并不能证明这个 API 的易用性，而 API 的易用性才是保证用户正确开发的根本因素。但这种说法可能会被完全推翻，尤其是当 API 的源代码及二进制层面同运行时层面之间存在明显差异时。

如果要想在不同层次上都做到一致性，那么不仅要保证使用类库时所有的内容都要输入正确，而且执行的时候，相关的函数都要运行正确。想自始至终都一直正确是不可能的。但为了做到一致性，还是应该向这方面尽量努力。而且这样做可以减少用户在使用时的困难，其实现代 IDE 都提供了代码自动完成功能，开发人员的编程工作已经做到了半自动化，但如果只做到代码编写时输入正确，但执行时不正确，那么离目标还是差得很远。

对于我来说，用来说明不一致性的最佳例子莫过于 `javax.swing.JFrame`，这个类是 `java.awt.Component` 的子类。因此，所有可以使用 `java.awt.Component` 对象的地方，理论上讲都可以用 `javax.swing.JFrame`，比如说把 `JFrame` 控件放到一个 `java.awt.Container` 控件容器中。很明显，这是行不通的，不可能把一个顶层的窗口放到一个对话框中。但从代码编译的角度来说，却可以写出这样的代码，而且通过编译。这个特例可谓众所周知，但对于有 Swing 开发经验的人来说，基本上是不会犯这种错误的。但对于一些初学者来说，Swing 类库在开发时和运行时的不一致性，却会打击他们使用 Swing 的积极性。

还有一个与此相似的例子，但要更加复杂一些，这是一个与 JDBC 驱动相关的例子。对于 `java.sql.Connection` 这个接口，它可以创建一个 `java.sql.Savepoint` 对象实例，也会用到这个对象实例。用户可以通过调用 `setSavepoint` 这个方法返回一个 `Savepoint` 接口的对象实例，注意这个方法是以 setter 打头的。这个方法返回了一个 `Savepoint` 接口的对象实例，然后利用这个对象

实例，可以调用 `Connection` 对象的 `rollback(Savepoint)` 方法。因为 `Savepoint` 被定义为一个接口，这意味着完全可以由用户来创建一个 `Savepoint` 接口的具体实现，然后再传递给 `rollback` 这个方法。但这样做是无用的，请记住这一事实。同样，把两个不同的 `Connection` 创建的 `Savepoint` 对象在这两个 `Connection` 之间进行传递也没有任何意义。这样做只会引起运行时抛出异常。API 的声明和它实际运行时的效果差得太多了。不是声明有问题，就是运行有问题，两者必定有一个是不正确的。假设运行时的功能多少符合开发人员对数据库操作的预期，那么问题就应该是出在相关内容的定义上。这两个接口在设计时就应该与运行时完成的功能相匹配，下面的代码就比较合适一些。

```
public interface Connection {
    public Savepoint setSavepoint();

    public interface Savepoint {
        public void rollback();
        // and other useful operations
    }
}
```

新写的这个版本能够保证接口的声明和运行时结果这两者之间是匹配的。不可能将一个 `Connection` 回滚到一个错误的 `savepoint` 上，因为这个 `Savepoint` 对象应该清楚与自己相关联的 `Connection`。当 `Savepoint` 对象的 `rollback` 方法被调用时，它会正确找到与自己相关 `Connection` 以便回滚。

项目配置

在 NetBeans 中，我们设计了一个接口，如果在 IDE 中，某一类项目有自己特定的构建过程和运行时的配置，那么这个项目就必须实现这个接口。有一个 UI 控件使用了这个接口，它会将配置信息显示给用户，然后允许用户在多种配置间进行切换。如果出现了配置改变的情况，UI 就需要通知相应的项目当前应该使用哪一个配置信息。一开始时，这个 API 是设计成下面这个样子的。

```
interface ProjectConfigurationProvider {
    public ProjectConfiguration[] getConfigurations();
    public ProjectConfiguration getActive();
    public void setActive(ProjectConfiguration c);
}
interface ProjectConfiguration {
    public String getDisplayName();
}
```

这样的设计也出现前面所说的那种不一致性，与 JDBC 那个例子差不多。如果你认为创建自己的配置信息，并将其作为参数来调用 `setActive` 方法也挺好，那你可就错了。当然，我们也可以依照前面的例子来修正这个设计上的问题，只需要把 `setActive` 这个方法移到 `ProjectConfiguration` 接口中。但出于某些考虑，我们没有这样做，而是引入了泛型。

```

interface ProjectConfigurationProvider
    <Configuration extends ProjectConfiguration> {
        public Configuration[] getConfigurations();
        public Configuration getActive();
        public void setActive(Configuration c);
    }
interface ProjectConfiguration {
    public String getDisplayName();
}

```

这样的代码看起来就漂亮了。但前面已经讨论过，其实漂亮对于软件工程来说没有任何价值（请参见 1.4 节）。这个例子使用了 Java 5 的新特性，倾向于使用组合技术，同时将声明和运行时两个层次的关系更加地贴近。正确声明的程序也可以保证语义的正确。但这样做还是存在一个缺点。在我们的用户中，有 90% 的人，包括那些调整这些代码的工程师，也无法正确利用该 API 编写代码。比如说，像下面的这种方法就不正确：

```

ProjectConfigurationProvider<?> provider = null; // 别处的调用
provider.setActive(provider.getConfigurations()[0]);

```

问题出在声明中泛型标识符“？”，这表示一切未知的类型。如果有两个“？”，肯定表示两个完全不同的类型。前面的例子，setActive 用了一个“？”，而 getConfigurations()[0] 也返回了一个“？”。但对于编译器来说，这两个“？”却代表了不同的类型。要让编译器接受这段代码，是一件非常麻烦的事情，代码要写成如下形式才能通过编译。

```

{
    ProjectConfigurationProvider<?> provider = null; // obtain elsewhere;
    resetToZero(provider);
}
private static <C extends ProjectConfiguration> void resetToZero(
    ProjectConfigurationProvider<C> provider
) {
    provider.setActive(provider.getConfigurations()[0]);
}

```

现在编译器认为类型的声明是正确的了。这个调整是将两个方法都使用一个类型。通过声明一个 C 的中间类型，能让编译器来检查类型声明的正确性。正如你看到的代码，让一个编译器将不同方法声明的类型认成是相同的类型，其实对编译器来说，不是一件简单的事情。先要对这个类使用泛型，声明一个类型，像 ProjectConfigurationProvider<?>，然后在具体使用时再将相应的类型加以明确，如 ProjectConfigurationProvider<C>。在 Java 语言中，没有什么语法上的技巧可以用来轻松完成这个功能，只能通过把声明和具体的方法进行分离来做到这一点。对于大部分 API 用户来说，根本没有意识到这一点，这没什么可奇怪的。而且这种情况会变得更加复杂，因为我们需要向 UI 提供数据，然后显示给用户，而且只有在用户执行具体操作之后我们才能调用 setActive 方法。这表示我们不可能使用这种 Open 的方法，只能使用 Open 类。

```
static void workWithProjectConfigurationProvider(
    ProjectConfigurationProvider<?> p
) {
    ResetToZero<?> rtz = ResetToZero.create(p);
    rtz.obtainFirst();
    // 稍候
    rtz.apply();
}

static class ResetToZero<C extends ProjectConfiguration> {
    C active;
    final ProjectConfigurationProvider<C> provider;

    ResetToZero(ProjectConfigurationProvider<C> provider) {
        this.provider = provider;
    }

    static <C extends ProjectConfiguration> ResetToZero<C> create(
        ProjectConfigurationProvider<C> p
    ) {
        return new ResetToZero<C>(p);
    }

    public void obtainFirst() {
        active = provider.getConfigurations()[0];
    }

    public void apply() {
        provider.setActive(active);
    }
}
```

上述代码段写得很优雅,不是吗?我举这个例子是想说明寻找正确解决方案的目的到底是什么。怪不得没有哪个 API 的用户能够正确地使用它。设计人员需要花费几年的时间来学习设计理论,才能写出这种正确的代码。另一方面,我始终坚信设计时和运行时这两个层面应该保持一致,这也是我们要完成的目标。但在前面的那个例子里,我们本应该讨论 `Project Configuration.activate()` 这个方法,而不是强求用户去了解这方面的理论知识。

像何时使用整数、何时使用枚举这样的例子,其学术气息就不那么重,更多的是从实际出发了。如果一个方法声明了一个整数参数,然后分别用不同的值来代表不同的状态,可以说设计和运行的一致性就被丢失了。毕竟, `int` 代表自然数,不是吗?而在运行时还需要添加额外的检查,且只接受 1、2、3 或者是其他少数几个值。在 *Effective Java*^①一书中,这个使用整数的方式可谓被批评得一无是处,书中建议如果使用 Java 5 版本,就要使用更安全的方式。

这种设计和运行时不一致的例子估计俯仰皆是。这也是没有办法的事情,因为代码运行时所表现的计算能力就像一台图灵机。事实上常用的编程语言都有自己的类型,并不完全遵守图

① Joshua Bloch, *Effective Java* (Upper Saddle River, NJ: Prentice Hall, 2001)。

灵理论，所以编程语言无法正确表达所有运行时的内容。不可能消除所有的不同，所以一旦出现这种问题，只能说是运气不好了。但对于出现的问题，API 的用户需要了解设计和运行的区别，要找到在程序运行时出现的非预期问题。如果能让用户在不了解 API 内幕的时候也能很好地开发代码，就必须将这种设计和运行的不一致性最小化。所以在下一次设计 API 时，请牢记这一点。

10.6 不要滥用 JavaBeans 那种监听器机制

3.9 节提到，为了方便 API 的用户，应该尽可能地使用那些已经广为人知的术语。比如说，像“工厂方法”、“单例”、“JavaBeans”这些概念用在一个 API 上，让人感觉很不错。但有时候也没必要一味迎合现成的 API 类型，完全可以选择那些不太通用、但更适合的方式。

JavaBeans 是一个很受欢迎的设计模式，在相应的 JVM 规范中已经存在了很久。可以做一个假设，就是所有的 Java 程序员都用到过 JavaBeans 组件。他们一般都觉得 JavaBeans 非常容易理解。所以可以进一步假设所有的 Java 程序员都知道“setter”、“getter”和“listener”分别代表着什么意思。

JavaBeans 的规范要比你想象的复杂多

作为一个 Java 的 IDE，NetBeans 对 JavaBeans 规范的看法有它特有的角度，它不仅要遵守 JavaBeans 的规范来对外提供 Beans，同时还要深入地了解它们，以便更好地使用它们。因此，NetBeans IDE 不仅能识别几乎所有现存的 JavaBeans，同时还能正确地使用它们。

为了做到这一点，不是说把 JavaBeans 规范中的章节拿过来读上两遍、学习一下就行了，只有深入了解了规范，特别是所有的细节内容以及 JavaBeans 的一些死角，才能最终达到目标。比如，有人知道该规范中有一个索引属性吗？你是否知道在 JavaBeans 组件中，可以使用 `java.util.Vector` 类来表示其属性？你是否了解 JavaBeans 中因“setter”而产生的监听事件呢？

规范远不是想象中的那么简单！但对于大多数开发人员来说，规范中最常用的那些内容还是比较简单直白的。

因为规范很容易理解，很多人就会把自己的 API 也按照相应的规范进行调整，改得像 JavaBeans 组件一样。通常情况下，这么做还是不错的。但有时候也会弊大于利，像下面这个例子：有一个用来在编辑器中将特别信息高亮显示的 API，它允许其他模块将各自实现的 `HighlightsContainers` 注册到指定位置。这些注册的内容可以用来计算一个编辑器中某部分文字内容的高亮区域，而且根据高亮区域的改变，这些内容会发出消息加以通知。这种功能的设计完全可以用 JavaBeans 的设计模式。

```
public interface HighlightsContainer {
    public void addHighlightsChangeListener(HighlightsChangeListener l);
    public HighlightsSequence getHighlights(int start, int end);
    public void removeHighlightsChangeListener(HighlightsChangeListener l);
}
```


代码写得没有什么问题，但一旦你意识到在这种情况下，可能永远都不会有一个以上的 Listener 来注册到这个容器上，问题就此出现了。事实上，这个监听器是由编辑器提供的，永远都不会被注销，因此始终是同一个。最多，这个 HighlightsContainer 对象会被垃圾回收，但即使这样，removeHighlightsChangeListener 这个方法也永远不会被调用，注意，是一次都不会被调用！发现了这种情况以后，我立刻意识到，将 JavaBeans 的这种监听器设计模式应用到这里是有问题的，其实一个简单点儿的 API 也能完成同样的功能。

一个用户在使用类似 JavaBeans 的 API 时，最先抱怨的就是相应的 Listener 机制，在相关类中需要提供 Listener 的通知功能。尽管写这样的代码不会像研究火箭技术那样复杂，但总得写上十来行吧，而且每个用户都要做这种无用的重复工作。常见的解决方案和 JavaBeans 一样，就是提供一个 AbstractHighlightsContainer 的基类，它实现了 HighlightsContainer 接口，然后实现了支持添加和删除 Listener 的方法，同时可以将这两个方法声明为 final，另外再写一个声明为 protected final void fireHighlightsChanged() 的方法。这样，开发人员只需要继承这个基类就可以避免原来那些重复的工作了。但这样做又引发了另外一个疑问：“为什么要为一个接口提供基类？”尽管这是一种比较标准的做法，但还是太复杂了，太多类和接口，太多的方法，太多的选择，用户需要学习和理解太多的内容。如何才能让用户在不深入了解 API 的情况下，也能用好 API 呢？所有的需求都指向一点，就是简化。

首先，还是放弃 AbstractHighlightsContainer 这个基类吧。有一种方法也需要遵循 JavaBeans 的规范，那就是写一个 addHighlightsChangeListener 方法，但这个方法只支持有限的 Listener，因此方法在声明中告诉用户，它会抛出一个 TooManyListenersException 的异常。

```
public void addHighlightsChangeListener(HighlightsChangeListener l)
    throws TooManyListenersException;
```

在这个基础上再实现具体的方法就变得简单多了。

```
final class MyHighlightsContainer implements HighlightsContainer {
    private HighlightsChangeListener listener;
    public synchronized void addHighlightsChangeListener(
        HighlightsChangeListener l
    ) throws TooManyListenersException {
        if (listener != null) throw new TooManyListenersException();
        listener = l;
    }
    public synchronized void removeHighlightsChangeListener(
        HighlightsChangeListener l
    ) {
        if (listener == l) listener = null;
    }
    public HighlightsSequence getHighlights(int start, int end) {
        return null; // implement
    }
}
```

这就是简化。但这段代码还是带了一个永远也不会被调用的无用方法，就是那个用来移除

Listener 的方法。除此以外，用来创建正确的事件、然后将事件发送给 Listener 的那段代码也是非常复杂。JavaBeans 的规范也指出 Listener 应该声明为一个接口。这表示不能为 Listener 类型添加新的方法，否则会引起不兼容性问题，即使未来 API 有这个需求，也不能这样做。这个约束有点多余，因为实际上这个 Listener 的实现只在这个 API 框架内部。当然了，因为 API 很清楚 Listener 需要通过 Event 传递哪些数据，而且 Event 是一个 final 类，所以对 Event 进行演变的时候，就不存在问题。但要注意，因为任何人都可以去实现一个接口，所以向一个接口中添加新的方法绝对不是一个好的思路。

代 理

你也许会问：“为什么一个 API 的客户却要去做一个只应该由内部使用的接口呢？”要知道，凡事只要有可能，那么就一定有人会去做。要牢记，API 的用户绝对是一个有创意的群体。

如果要问为什么要实现只应该在架构内部使用的那些接口，一个最常见的原因就是因为想使用代理技术。如果你写了一个 HighlightsContainer 的实现类，这个类其实持有另外一个 HighlightsContainer 对象，真实的工作其实由内部这个 HighlightsContainer 对象来完成的，在计算高亮区域的时候，可能需要过滤一些由这个内部对象产生的一些事件。出于这些目的，就需要实现 HighlightsChangeListener 接口。

所以 NetBeans 在制定 API 设计规范时，针对这种情况不建议使用 JavaBeans 的方案，而是建议使用如下的“回调”方案。

```
public interface HighlightsContainer {
    public void initialize(Callback callback);
    public HighlightsSequence getHighlights(int start, int end);
    public static final class Callback {
        Callback() { /* only for the infrastructure */ }

        public final void highlightsChanged() {
            // refresh everything
        }
    }
}
```

这个 API 清楚地向用户描述了自身的功能。用户需要去实现一个 HighlightsContainer 接口。实现这个接口有两个方法，其中一个方法 getHighlights 是用来完成具体工作的。另外一个方法 initialize 则是被他人调用的，到底是何时会被调用呢？是在类初始化的时候。该方法提供了一个声明为 Callback 类型参数，Callback 类是一个 final 类，这样它只能被调用，而不能被继承。这个类中的方法是用来完成什么功能呢？它有一个出现变化会被通知调用的方法。所以如果发现高亮局域改变时，就可以调用这个方法了。这样做让整合设计变得更加简单直白，不会像前面的设计那样混乱、不易理解，即使没有额外的说明文档，用户也会明白其用途。这样的设计可以帮助用户在不深入了解系统的前提下也能用好 API，这才是一个真正意义上设计良好的 API。

除了上述的好处以外,这种使用回调的设计方式对于 API 未来可能的演进更有帮助。不管何时,如果你需要与架构内部进行互操作,就可以向 Callback 这个类中添加一个新方法。这样的操作非常安全,因为 Callback 这个类是由 API 自行实现的,不可继承,所以添加方法只需要保证 API 内部保持一致就可以了。

不要总是一味埋头苦干

在检查 HighlightsContainer 这个类的设计时,我们发现事情其实不像表面上看起来那么简单:可能有不止一个 Listener 来监听相应容器的事件。于是我们就使用了 JavaBeans 设计模式。但在 NetBeans 平台中,还有一些其他的 API,在设计这些 API 时,回调的设计方式会比 JavaBeans 设计模式更合适一些。

虽然在调整 API 时,通常会尽量将调整的结果贴近现有的设计模式,但这样做的代价决不应该是引入混乱或者使其更加复杂。在使用 API 时,需要了解的细节内容越少,就表示这个 API 设计得越好。



迄今为止，本书所讨论内容大部分还是集中在源代码和二进制的兼容性问题。人们通常建议：设计时对外公开的类、方法、字段应该尽可能少，这样才有助于在未来对它们加以改进。这个建议是可取的，因为如果用户想将他们的程序建立在我们提供的类库之上，就需要让程序正确连接，而且在运行时也不要抛出 `UnsatisfiedLinkErrors` 这类错误。但即使程序正确地连接在一起了，也并不表示 API 的设计工作结束了。事实上，这才刚刚拉开序幕。因为还要保证连接在一起的程序各部分能正确运行。而且程序不应该只能运行一次，程序的模块升级到新版本时，程序应该也能运行。而且以上所说的这些工作只需要简单的组装就可以完成。为此，作为设计人员的你需要深入了解功能兼容的真正含义。

NetBeans 文件系统 API 性能提升中出现的惊险一幕

NetBeans 的文件系统 API 使用了 `java.io.File`，我们一直在致力于提高相关操作的性能，在此过程中出现了一个与功能兼容有关的趣事。

我们发现有一些常用的操作存在性能问题，比如说判断一个磁盘文件是否存在，如果是通过文件系统的 API 来判断，就比较慢，而直接通过 `java.io.File` 这个类来判断就会快得多。但从宏观设计的角度上看，是不应该存在这样一个问题的，于是我们决定深入研究该问题，以便提高性能。首先要进行优化的，就是磁盘文件真实状态的缓存，以及与本地磁盘中缓存的同步。文件系统的 API 在内存中缓存了很多磁盘文件的信息。不管是创建文件还是删除文件，这个 API 要做的事情不仅仅是更新磁盘上的真实文件，它还会更新内存的缓存。到这一步，还没有发现什么问题。但有一些 API 的用户的开发习惯不好，经常将我们文件系统的 API 与 `java.io.File` 混合使用，这样直接使用 `java.io.File` 进行的操作就会跳过对缓存的更新。这些用户特别喜欢直接使用 `new File("...").createNewFile()` 来创建磁盘文件，然后马上通过 `FileObject` 来访问该文件。尽管我们不建议使用这种技术，但这种恶意代码还是不时出现在系统中，因为在文件系统 API 的老版本中，在 `java.io.File` 和 `FileObject` 之间存在着映射关系，所以这种转换也是可以做到的。

这个例子也体现了一些用户在使用 API 时，完全按照经验行事，而不关心具体的操作方式是对还是错，只关心程序运行时能否达到自己的目标。因为这种开发方式可以很简单地完成

工作任务，所以 API 的用户就会觉得这样做不存在任何问题。但我们却希望避免这种磁盘文件与缓存的同步操作，因为频繁地检查和校验正是造成文件系统 API 操作性能低下的原因。

引起性能低下的另外一个原因则和物理驱动器与虚拟文件系统的映射有关，NetBeans 可以将 Windows 操作的驱动器，如 C 盘，D 盘，还有软盘、网络驱动器都映射到一个虚拟的文件系统中。这样做的后果就是，当你用外部工具修改了文件后，需要同步内存中的缓存时，必须强行刷新所有的驱动器。这种操作效率肯定很低，因为哪怕本地硬盘驱动器上很小的改变，却会触发整个文件系统的刷新机制，就有可能去刷新所有的驱动器，包括硬盘驱动器和网络驱动器。很显然，这个刷新过程肯定慢不少。正因如此，我们决定改变原有的映射关系，转而为每一个 Windows 系统的磁盘都创建一个虚拟的文件系统。这种调整不需要改变 API 签名，也没有添加新的方法。只需要对内部实现加以小小的更改即可。至少，这些内部的更改是我们可以预估出来的。

但 API 的用户却不这么想。我们收到了很多这方面的 bug 报告，说许多调用 API 的代码突然间就不能正常运行了。这些代码的维护人员非常恼火说：“我都几个月没有碰过这个模块了，现在它却突然不能正常运行！这种事情怎么能发生在我身上？”还有些类似的话。即使我们告诉他们代码中的问题，证明他们一开始的使用方式就不正确，或者没有按照文档中的说明来做，其实也补救不了什么。他们会说：“之前是运行的，现在却不运行了，这是你们的问题！”我们尽力去改正 API 中我们看到的问题，特别是那些容易误用的地方。但我们不可能找到所有的问题，就像序言中所说，对于一颗恒星，不可能清楚地知道谁在观察它，所以我们不可能知道到底哪些地方把 `java.io.File` 与 `FileObject` 混用了。于是，我们提供了一个移植指导文档，解释如何把那些存在问题的老代码模式移植到现在正确的代码上。但这样做还是不够，抱怨还是不停出现。

在优化工作进行到一半的时候，我们决定屈服于这些压力，放弃这种调整方案。我们不想在性能方面有所损失，但面对这种铺天盖地的抱怨，只能决定在提高性能的同时，还要保证兼容性。我们选择了两种不同的方案来做这件事。一旦在缓存和磁盘文件间出现了不一致的情况，就要全部刷新。但如果没有任何不一致的迹象，仍然按照新的处理方案来运行，也就是说绝不访问磁盘。为了做到这一点，我们必须能够发现那些 `java.io.File` 与文件系统 API 混用不正确的情况。我们决定借助于 `java.lang.SecurityManager` 这个类。只有创建或者删除一个磁盘文件的时候，才能引发缓存不一致的问题。为此，可以在执行的时候用 Java 代码，也可以用外部工具，如使用 `/bin/rm` 命令。但 Java 是一个沙箱环境，如果执行创建或者删除磁盘文件的命令没有通过安全认证，这两个操作都不会被执行。因此，要监控文件的改变可以覆盖 `java.lang.SecurityManager` 类的 `checkWrite(String file)` 和 `checkExec(String cmd)` 这两个方法，然后将这个新的 `SecurityManager` 注册到系统中，这样新的 `SecurityManager` 就可以替代默认的 `SecurityManager`。一旦开发者不使用 NetBeans 提供的文件系统 API，而是使用了 `java.io.File` 来操作文件，就会标识缓存中相应的文件已经失效，或者是标识整个缓存已经失效。随后在缓存失效的情况下，一旦有代码访问一个 `FileObject`，就会刷新缓存以保证其有效性。这样既提高了性能也保证了兼容性。

但这个使用 SecurityManager 类来截获操作的技巧并不能解决 Windows 系统中多个虚拟文件系统的问题。我们还是决定解决 API 的兼容性。于是我们沿用老的方案，用一个虚拟文件系统来代表所有的驱动器。为了提供更理想的刷新功能，我们添加了一个新方法。然后查找所有可能引起过度刷新的地方，调整后，使用这个新加的方法。老的 API 仍然是兼容的，同时，功能却得到了有效提高。

这样解决算是一个完美的结局吗？也许算是吧。毕竟在提高性能的同时还能保证其兼容性。唯一的弊端应该就是这件事对我们的声誉有所损伤。最近，我与一个组织的高级主管开了一次会，该组织的产品是基于 NetBeans 平台开发的。他对我抱怨了很多，主要集中在稳定性和兼容性方面。他说：“前段时间，因为你们的一些调整，我们整个系统都崩溃了。”我当然一听就知道他说的是什么事了。我没有给他解释技术细节上的内容，更没有和他说这是因为他们公司的程序员也许根本就没有留意过 API 的 Javadoc 说明书，我告诉他，这可能是因为他们使用的 NetBeans 版本只是正在开发的版本，并不稳定。只要把他的程序移植到稳定的版本上，就可以解决这个问题了！他欣然接受了我的解释。事实就是如此，特别是我们最近修正了不兼容的问题以后，我说的也都是实话。

我们应该尽力去掌握更多的技巧，学习几种 API 设计模式，这样才能提高运行时的兼容性。但仅仅知道做什么事情还远远不够。还必须避免自己设计的类库像阿米巴变形虫那种随意变形，也就是说，你必须避免在不同版本中代码相同但功能不一致的问题（图 4-2 和图 4-3 就是类似的情况）。要想达到这个目标，还必须牢牢记住这条重要建议：作为设计者，你有这个责任来避免这类问题的出现。

关于有针对性的无绪和如何编写可靠性高的代码这两者之间的关系，尽管并非本书的主旨，而且与 API 设计模式也没有直接的关系，但本章还是要对它进行一些具体的讨论。其实真正要讨论的问题，就是如何能够保证自己类库真正运行时的效果与设计目的。虽然市面上已经出版了很多关于这个话题的书，如 *Pragmatic Unit Testing in Java with JUnit*^①，但我还是想着重从无绪和可靠性这两个方面来讨论 API 的设计。

11.1 不要冒险

4.2.3 节对阿米巴变形虫模型进行了详细的说明，可以帮助开发人员捕捉在设计类库时可能出现的问题。该节一直强调类库不同版本间的区别。但还可以从时间和改进这两个角度来分析一下软件工程中与版本发布时间相关的问题。我把这种模型称为“失败者的归家之路”，这个名字就像荷马史诗中的那位主人公奥德修斯^②，他的归家之路十分漫长，几乎永无尽头。

在一个软件项目开始的时候，我们总是知道我们当前所处的阶段，也知道我们要达到的目标。虽然这话不是那么绝对，但对于大部分软件项目来说，基本上是这样的。我们或多或少知道当前

① Andy Hunt 与 Dave Thomas 合著，*Pragmatic Unit Testing in Java with JUnit* (Raleigh, NC and Dallas, TX: Pragmatic Bookshelf, 2003)。——译者注

② 奥德修斯：伊萨卡的国王，特洛伊战争中的希腊首领，在经历十年的艰苦漂泊后才最终回到家。——译者注

所处的阶段。我们可以想象在当前基础上再加以改进，最终实现目标。就像图 11-1 所示，我们认为从当前起点出发，达到目标是一件简单的事情。事实上，并非如此。



图 11-1 这个项目太简单了

显然，事情永远都不像表面上看起来那么简单。通常你不可能绝对准确地达成自己的目标。比如说，你在产品分支上添加了一个解决方案。一旦把这个方案合并到产品主干上，就会发现很多的功能都不能按照原先的设计来运作。可能会有 bug，可能有一些功能没有了，还有一些人会因为引入的新功能而恼火，这类事情非常之多。所以想在软件开发的一个周期内就达到理想的目标，是不现实的。但这种情况是可以接受的。我们可以像图 11-2 中一样，利用多次迭代来工作。我们知道前进的方向，这样不管快还是慢，只要小步前进，最终还是能够达成目标的。



图 11-2 也许用迭代的方式来做就好了

我们相信，终有一天，我们能够达成目标。但成功之路并非一帆风顺。图 11-3 告诉我们，前进的过程中可能会倒退。每一次倒退可能会让我们离目标更远。但我们却坚持信念，终有一天，会达成目标；终有一天，所有的倒退都会被战胜。但我却要说，现实未必如理想一般。他们的努力可能会付诸东流，而倒退仍然存在。我们必须接受这个现实，必须承认从开始的起点到最终的终点之间不可能是一条直线，必须接受这一过程中有多次倒退，但我们还是有机会到达终点。过程的倒退可能会浪费我们更多的时间，但终有一天，还是能够到达终点。

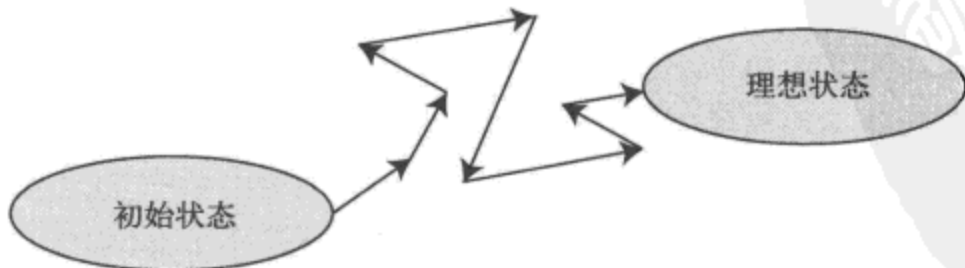


图 11-3 终有一天，一切都有圆满的结局

但如果我们真的无法到达终点，又会怎样？软件工程会变得过于复杂，最终无法完成既定的目标（参见图 11-4）。此时，我们发现我们正在原地转圈。这就像是一个恶性循环，我们一步步地移动，想更加接近我们的目标，但却迷路了。就像阿米巴变形虫模型一样，我们可以不停地改变它的形状，希望它和我们的期望一致，但怎么也无法达成目标。现实生活中的项目也是如此。

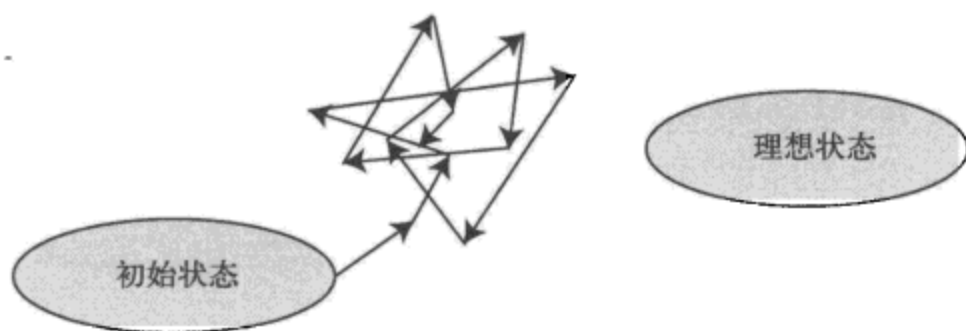


图 11-4 糟糕

现实是更加残酷的，那些项目的情况其实更糟。前面说的一些内容只涉及一个项目的某一点功能；但真实的项目中却有几十、上百、甚至成千上万的功能点。这些功能点可以单独分析并开发。但它们间的改变却不能同步进行，更惨的是，这些功能点之间会相互影响。因此，对某一个功能的调整可能会使得该功能更接近其目标，但却使得其他的功能出现倒退。就是说，完成一个功能点的代价，可能会使得其他功能点远离其最终目标，见图 11-5。

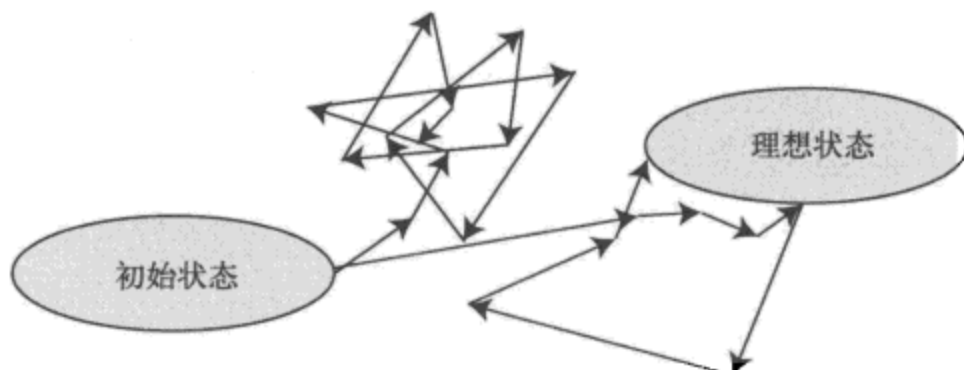


图 11-5 真实项目的情况其实更复杂

将更新的内容尽可能控制在小范围内

我夫人是一个公司的会计，工作时会使用一套会计软件系统，这套软件的生产厂商是以单词“Wizard”来结尾的。我夫人所在公司为了获得热线支持，支付了一大笔费用。我冷眼旁观了多年以后，发现热线服务只会提供一个建议，就是告诉软件用户：“你所遇到的问题其实是一个 bug 引起的，你可以这样忍耐一下，下一个新版本会修复这个 bug，等它发布的时候，升级到新版本就一切 OK 了。”这倒是和他们公司名称中的 Wizard^①绝配，换个直白的意思就是：“别急，新版本的升级可以解决一切问题。”

据我所知，这个软件至少提供过 3 个升级版本，这 3 个版本也的确修复了一些问题，但同时还会引发一些其他的问题。我夫人告诉我说，每次升级，她都要花几个小时来通过热线学习如何在数据库中执行不同的 SQL 语句，才能让升级后的版本正常运行。

软件项目通常都像上面说的那个软件一样，在原地打转。阿米巴变形虫模型在现实生活中是非常普遍的。如果有一个严重的问题，这个问题很快就会被修复。但如果只是一些普通的问题，

① 在英文中，Wizard 除了计算机常用的“向导”的意思以外，还有“男巫”的意思，作者这里暗讽这家公司像巫师一样，只会说下一个版本解决问题。——译者注

那么他们就宁愿接受这种经常出现的问题。这正是人们为什么普遍害怕升级的原因。大多数经验告诉我们，一旦人们习惯了软件的某个版本，那么就宁愿忍受该版本而不愿意冒险升级，因为升级可能会出现未知的问题，于是人们总说：“别去碰程序的那个部分”。

XServer^①需要升级吗

在2006年前后两年，所有在自己笔记本上安装Linux的用户，都有这种感觉：“好的，我有一个可以正常运行的系统。我可以让这个系统休眠^②，然后大部分时候可以让这个系统从休眠中醒来。尽管这个系统没有挂起功能，但不是什么大事情，可以接受。难道我应该为了使用挂起功能，而冒险把XServer升级到新版本上吗？要知道，如果搞不好的话，我的休眠功能就没有了。”

毫无疑问，因为害怕升级引发新的问题，从此严重地破坏了本书讲解的基本原则，即避免，用户深入了解系统。如果一个基于模块化架构的大应用程序在组装和改进的时候，需要将其中的模块升级到新的版本，这样的风险是否可以接受呢？我们是否可以做一些工作来减少这些风险，同时将升级过程变成一个没有问题的半自动化过程，我们至少希望对于那些常用的类库可以做到这一点。

11.2 可靠性与无绪

一个类库或者一个程序的功能像阿米巴变形虫一样，会渐渐产生调整和改变，并引发其外部行为的改变，进而引发的问题，这不仅仅有技术上的影响，还有社会影响。不是程序员的人们（也就是最终用户，他们购买并消费软件产品）会逐渐怀疑产品的质量。尽管他们的期望值在不停地改变，但他们始终认为：“如果某个功能在前一个版本中是正常的，那么下一个版本中，这个功能也必须是正常的。”事实上，几乎所有的软件项目都不能满足用户提出的这个基本期望，所以用户对产品越来越没有信心。

这种情况的确不妙。事实上，比这糟糕的事还多着呢。软件每次升级的时候都会保证提供了多少更强大的新功能，还有对现有功能的改善。但他们从来不说，一次升级其实是重写了原来的版本，两个版本完全不兼容，因为从产品销售的角度来说，这不是一个正确的策略。因此大部分的开发团队不承诺软件的兼容性，甚至根本不提这回事。随后，用户试用产品的时候，会发现一大堆不兼容的问题，这样使得软件工程师的信誉大打折扣。我们现在不再被用户信任了。他们把我们看成一群不会正确做事、不能兑现承诺的人。

用户这种态度的另外一个副作用就是他们把自己的悲观情绪带到了现实生活中。他们不再欢

① XServer 其实是 X Window System display server 的意思。为 X Window System 的分布式网络中连接到 X 终端的服务器。从终端用户的角度来看，X 服务器相当于一个多视窗操作的服务器。实际上，X 服务器应用程序提出客户请求，要求在各个终端运行视窗管理服务。X 服务器（为 X Window System 的一部分）一般安置在大型机、小型机或者工作站中基于 Unix 的操作系统。——译者注

② 休眠是一种省电的模式，它将内存中的数据保存于硬盘中，使 CPU、内存也都停止工作，当再次使用时需按开机键，机器就会恢复到你执行休眠时的状态，而不用再次执行启动操作系统这个复杂的过程。——译者注

迎我们及我们的产品，而是在他们的软件产品和部门之间建立壁垒，他们认为这样可以剔除我们的问题和不兼容性。不知道你对此有何想法，但我不喜欢这个样子，不想别人把我当做一个不靠谱和不可信的人。

臭名远扬

作为大部分 NetBeans 平台 API 的最初设计者，在 NetBeans 团队中，我也是第一个受到所有责难的人。因为我写的 API 正是 NetBeans 平台的核心架构，其他的每一个模块都会用到我写的功能。也正是阿米巴变形虫模型的源头所在。只要我负责的 API 有所改进，不管这个改变多么小，对功能的调整多么细微，都会对每一个使用了该 API 的模块造成冲击。最终，一个小小的调整可能会引发整个系统的一场大地震。

这件事搞得我一度声名狼藉，只要有系统出现了问题，第一个找到的就是我。我当然不能算是一个水平很差的程序员，但有这么多的开发人员和项目依赖于我开发的内容，这部分代码若出了问题，肯定备受关注，至少要比那些只有少数忠实用户使用的、不带 API 的对话框代码出问题时受到的关注要多得多。当时，我们开发 API 使用了和开发其他的潜在功能时相同的技术，但很快发现这种方式不管用。这么说来，我臭名远扬也算罪有应得了。

我当然不希望自己蒙羞。这样太不公平了，我其实可以做得更好。于是我采取各种方式来完全避免在 API 中出现阿米巴变形虫这种情况。我确信我成功了，因为那段时间，我可以进行各种调整而不用担心会引发一场系统级的地震，也不会让我们的开发过程陷入一个无休止的怪圈中。这是怎么做到的呢？难道说我变得比以前更聪明了？当然不是。我只是做一件事，就是改变我的态度。我需要更仔细、更有针对性地选择需要关注的事情。我需要有选择性地变得无绪。

一般软件开发都需要赢得用户更充分的信任，尤其是在开发 API 的共享库时。程序员必须有更可靠的声誉，必须能够赢回周围人的信任。我们必须证明给用户看，未来的版本升级根本就一点儿都不可怕。能做到这一点，只需要遵循本书的精神：提高我们开发时的透明度。

不管在什么时候，只要我们需要调整类库，就需要将注意力集中在调整的最重要的地方。那所谓最重要的地方具体是什么呢？就是指调整会改变类库的哪些功能。要知道每一个改变都必须有的放矢。毕竟，我们是想修改或者扩展某些特定的功能，或者是提供一些新功能。对系统进行调整肯定要对代码进行修改，但相比之下，这些修改最终会达到的效果，比修改的代码更为重要，这才是我们要关注的内容。也就是说，修改代码其实不是我们的目的，而修改这些代码后达到的效果才是我们最终的目的，也是最需要关注的内容。

如果要保证这些改变能够符合预先设定的目标，我们一定要投入大量的精力、时间和智慧才能做到。接下来的事，包括具体实现，如果让用户不需要了解具体细节就可以使用新的系统，其实只有一件事要做，就是设置一个**保卫者**^①。所谓的保卫者，能够提供参照基准，能够自动帮

^① 这里所说的“保卫者”其实也就是“自动化测试”。——译者注

我们来检查当前 API 是否存在问题，需要加以调整。换句话说，这个保卫者可以帮助开发人员确定当前所处的阶段是否与理想状态违背。在我们朝向目标行进时，它又能帮助我们判断与希望达到的目标到底还有多远。图 11-6 说明如果先写了保卫者代码，然后再去添加实现，你会发现自己不会再陷入那种原地打转的怪圈。每一个新的调整，我们都会要求以前使用过的保卫者来检查我们是否在走近最终目标。这样我们就不会退回到创建保卫者之前的状态。

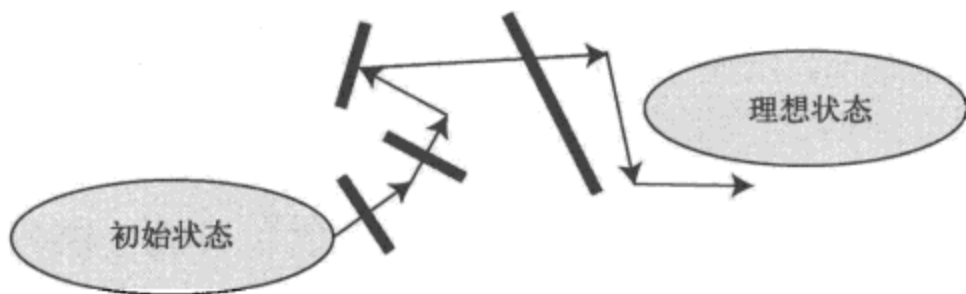


图 11-6 编写保卫者

但这样做其实也保证不了我们一定能达到最终目标。我们会逐步接近这个它，但可能永远都无法到达。虽然这话听起来比较丧气，但接近目标总比原地打转要好得多，事实上，这也是软件项目的真实写照。要知道软件项目中总会有 bug 存在，而且软件实际完成的功能与其预期的功能还是有所不同的。但有这样的保卫者可以帮助我们一步步接近目标，我们还是可以对接近最终目标充满希望。

我说了半天关于奥德赛和保卫者的隐喻，可能会让读者觉得无聊，厌烦，对此我抱以深深的歉意，我相信读者也知道这个隐喻背后的意义。的确，所谓的保卫者其实就是自动化测试。如果想避免自己编写的类库在运行时出现意外变化，只需要编写测试即可。如果需提高代码的可靠性，只需要精心挑选合适的用例，继而集中精力编写好的测试，继而再去编写自己的代码，就可以了。与其把百分之百的精力放在创建漂亮、优雅和正确的 API 上，还不如多花点时间准备自动化测试，这样做的效果会更好。的确，多多考虑 API 的设计是正确的思路，这也正是本书一直在讨论设计模式的原因所在。细节决定成败，没有什么能代替细心的工程开发。只有细心的工程开发才能保证软件的可靠性，因此必须提高开发时的透明度，并编写好的测试代码。

11.3 同步和死锁

对于存在并发的程序和系统来说，最难的地方就在于如何保证并发正确地进行。目前看来，尚未有什么好的理论或者是方法论可以指导开发人员编写出一个不存在死锁情况的应用程序。当然，还是有一些好的解决方案的。比如说，如果使用纯粹函数式编程语言，如 Haskell，就很少出现死锁的情况。但对于我们这些仍然使用传统开发语言（如 Java）的开发人员来说，像死锁或者不正确的同步这些情况都还是存在的。在编写多线程程序的时候，可能会出现相应的问题，使用纯粹函数式编程语言还是可取的。现代的计算机，很多都使用了多个处理器，或者至少是多核处理器。如果需提高性能，那么必然要引入多线程技术，不能只依靠单线程技术。

即使由你一个人来负责所有的源代码，也很难编写一个不存在死锁的程序。像前面那种内部系统，可以培养一些好的开发习惯，并重新梳理应用程序从而减少死锁的可能性。但有一个悖论，

那就是开发人员只能知道系统中不存在已知的死锁，但不能确认系统中有没有死锁。如果一个系统的源代码可以成功地部署在各种环境下，且多年来没有出现过一次死锁，那么这个系统的价格再高，也是物有所值的。这样的源代码应该放在保险箱里，每一次修改的范围都要尽可能地小，而且修改时一定要小心再小心。

但现在让我试着从“推土机”的角度来观察世界。我们把众多由他人开发出来的大模块集成在一起构建我们的程序。我们不需要知道，事实上也不可能知道每一个类库内部关于同步方面的细节信息。我们只能寄希望于这个类库的作者能够知道他们所做的工作，可以正确地处理同步代码。而且我们也相信即使将这些类库混合在一起运行，它们仍然可以很好地契合并做到运行时不出现死锁。

要做到上面所说的，可能会把很多类库的作者置于困境中。在不了解整个系统的情况下，他们设计的 API 要求能够保证其在多线程环境下也能正常运行。同时，他们也不应该把不必要的细节内容暴露给外部的 API 用户，因为那些用户应该尽可能避免对类库的深入了解。要做到这点并不容易。事实上，这是一个几乎不可能完成的任务，但有一些设计模式和建议对此方面却很有用。

11.3.1 描述线程模型

最常见的第一条建议，就是一个类库必须能够正确且恰当地描述其线程模型，换句话说，就是要描述为使用类库而需要满足的条件。事实上，这正是很多类库作者最容易忽略的地方。通常只能把这些描述信息散放在 Javadoc 的各处，而现有的 Java 平台并没有提供标准的 Annotation 或者其他的 Javadoc 内容来记录一个类或者方法中与线程相关的内容。很多类库根本就没有这方面的内容。很多人都吃过忽略和遗忘这种信息的苦头。即使用了自动化测试工具，也同样没有好方法可以保证文档中描述线程方面的内容与实际设计和运行时是相符的。

查找 bug 等内容

有一些工具可以用来提供类库在线程方面的 Annotation，并对相应的代码进行检查，看是否与其声明的一致。虽然有这类工具，但这类工具同样需要作者在开发自己类库的时候使用这些 Annotation。因为这些 Annotation 是文档的一部分内容，所以至少在 Javadoc 中需要看到这些 Annotation。有时候，这些 Annotation 也会出现在编译后的二进制代码中，这表示你的类库将这些 Annotation 作为 API 再次导出给了测试工具使用。在 10.1 节中已经说过，再次导出第三方的 API 时应该谨慎处理，几乎没有人愿意使用这些 Annotation，除非它们已经非常稳定了。

我们最近在研究是否在开发 NetBeans API 时也使用这种 Annotation。最终我们决定绝不在二进制 Class 文件的格式中对外暴露这些 Annotation。我们希望所有的内容和其现在的状态保持一致。但使用这些“标示行为特性的 Annotation”又非常有用，所以我们决定提供一套自己的 Annotation，但这些 Annotation 只在源代码层面有效：这些内容只在源代码中，编译器可以对其进行验证。但在生成 Class 文件之前，会把这些内容移除，不会体现在 Class 文件中。我们觉得这是一种可以接受的折中方案，在 Java 能提供真正正式而且稳定的方案以前，我们会采用该方案。

人们总说线程模型应该是很容易描述清楚的。他们认为如果这个模型不能用一句话解释清楚的话,只能说明这个模型很差。我不得不承认这一点,但我一度很怀疑,就算真能用一句话描述,那也是勉强压缩而成的。但现在从无绪的角度来看,如果忽略了细节性的内容(至少是在一些情况下),这个论断在有些情况下还是成立的。

“API是线程安全的”,对每一位调用API的客户而言,这是对API线程最好的说明。这意味着,在调用API的时候,不用关心API的实现,也不需要使用特殊的方式来调用API。不管怎样调用这个API都不会有问题。通过这句话让外部用户知道,这个API可以被重复调用而且可以正确地处理相应请求。这也意味着,实现一个这样的API实在并非易事。该API的提供者需要在代码中处理各种调用,即使调用的代码已经加了锁或者持有资源。但这种实现要能行得通,代码会非常复杂,难以编写。因此,即使设计了一个线程安全的API,其易用性一般也比较差,用户很难用简单的方式来使用这个API。事实上,用户必须清楚知道他们在做什么,并且能够娴熟地完成,因为即使是一个微小的错误也可能导致整个系统死锁。

一个API如果没有使用同步的话,通常就只能用在那些无须共享对象实例的场合。这些对象实例由调用的代码来创建,而且只能内部访问。这种情况不需要进行同步处理,可以用代理的方式来让API的用户访问内部数据。`java.util.Collections`这个类就是一个上好的例子,看上去它在任何场合下都可以正常运行。客户端代码只需要牢记这一点:他们自己来持有对象,保证内部访问,同时不要通过方法返回值或调用的方式对外泄露这些对象,那么就可以做到线程安全。

Swing则使用了更为专一的线程处理方式。Swing即是API,同时也是SPI,对于这种情况,使用单一线程可能是一个比较合适的折衷方案。调用者需要清楚地了解调用Swing时的限制,在调用API前,将控制权转交给这个专一的线程。从另外一个角度来说,任何编写Listener或者自己实现了Swing控件的代码,都会要求代码的调用限制在一个线程中。所以不需要使用同步,更不用担心并发的情况。

11.3.2 Java Monitors 中的陷阱

两个人做同一件事,结果可能完全不同。在写Java程序的时候也是如此,Java把原始数据类型的同步称之为Monitor。事实上,他们真正的作用与原作者的意图完全不同。造成这种不同的原因,部分是因为理论和现实的差距,还有一部分则是因为面向对象语言性的不同而造成的。简单地把一个用在面向过程语言上的解决方案,移植到面向对象语言上,其后果可能让人难以想象。

2001年4月的一次死锁

在2001年4月,NetBeans API中出现了一个问题。它没有任何预兆地突然死锁。在进行大量的调查以后,才发现是因为我们API中的锁和API用户使用的锁产生了冲突。这表明在Java中使用标准的同步操作也需要进行特定的处理,所以在设计API时也必须注意这一点。

这种死锁情况很罕见。我们API中的Children类使用了Synchronized方法来保证私有字段处于合理的状态。这些结构的同步已经被研究得很透彻,感觉没有任何会引发死锁的可能,直到发现这个API被用到了EJBClassChildren子类的定义中。此时,它仍然使用了Synchronized

方法来保护这个类内部的数据结构。

但根据 Java 语言规范，定义一个同步的方法就只如同同步对象自身^①。但因为 Children 和 EJBClassChildren 的子类其实是同一个对象实现，因此这两个 Monitor 忽然间就变成了一个 Monitor。这两个类就会相互影响，最终引发一个完全意想不到的死锁。

Per Brinch Hansen 首先提出了 Monitor 模型，他曾经在 20 世纪 70 年代创造了支持同步的 Pascal^② 语言，还写过一篇名为《在并行情况下，Java 并不安全》的论文，该论文发表于 1999 年 2 月^③。在该论文中，他反对把 Java 的同步模型称为“基于 Monitor 机制的同步模型”。因为对 Hansen 来说，一个真正的 Monitor 机制可以支持一个编译器来检查所写的同步代码是否正确，从而避免程序在运行出现问题。但 Java 只是使用了一个看起来比较像 Monitor 的机制，但并没有享受到 Monitor 的益处。Java 的编译器在编译的时候不知道代码中任何与同步相关的内容。

他针对 Java 提出的主要问题就是不安全，因为你可以将同步和非同步的方法放在一个类中。所以在访问内部数据结构的时候，很容易错误地使用同步。与原来的“基于 Monitor 机制”的同步 Pascal 语言相比，一个 Class 中只有声明为 private 的字段信息才能避免外部突然间不同步的不安全的线程访问。这样，程序员要做大量的工作来保证线程安全，要写一些深层次的同步处理，甚至是原语。他的观点完全正确，一旦了解怎么做才能维护一个含有同步方法的类，程序员就会觉得 Java 在这方面做得很差。

当你维护一个类多年并发布多个版本之后，你会很容易就忘记同步新添加的方法。避免出现这种情况的唯一方式就是使用 getter 和 setter 方法，即使是访问私有字段的时候，并且要添加 assert 以保证线程安全。

```
private int counter;

private int getCounter() {
    assert Thread.holdsLock(this);
    return counter;
}

private void setCounter(int c) {
    assert Thread.holdsLock(this);
    counter = c;
}
```

这样做可以保证这个字段在被访问的时候被安全地上锁。一眼看过去，这段代码就缺少程序

① 这里与 Java 同步机制有关，是指如果一个 static 方法被声明为同步，那么它锁定的同步对象是这个 static 方法所在的类；如果一个非 static 方法被声明为同步，那么它锁定的同步对象是当前方法所在类的实例对象，Java 的同步机制是将每个同步锁都对应一个 Monitor 锁，用来保证数据同步。书中的意思是说如果有一个类 C 实现了两个接口 A 和 B，类似于 C c=new C(): A a=c: B b=c, 此时对 a 和 b 中的方法调用，其实都是锁住 c 这个对象，于是本应该有两个 Monitor 就变成了一个 Monitor，不能保证数据同步，于是就会出现死锁问题。——译者注

② 支持同步的 Pascal 语言就是巴斯卡协同式 (Concurrent Pascal)。——译者注

③ Per Brinch Hansen, “Java’s Insecure Parallelism”(1999), <http://brinch-hansen.net/papers/1999b.pdf>。

应有的漂亮和简单，与支持并发的 Pascal 语言简直是天壤之别。我知道，漂亮不是那么重要，但可以证明即使以这种不漂亮的方式来写代码，在多线程环境下，死锁的几率仍然不会减少。

Hansen 说得太对了。Java 中，很容易就把一个对象的内部的数据结构暴露给外部，而这个对象应该是线程安全的。这种做法完全不是 Monitors 的机制。就我看来，Hansen 可能不是很清楚造成这个问题的原因，或者他知道但没有说，而我却知道，引发这类问题的最大原因就是继承，因为子类通过继承会共享锁。

最初，Monitor 的设计用在面向过程的编程语言上，该类语言肯定没有继承的概念。没有人会想到会有一种能够支持继承的 Monitor（包含数据结构和访问数据结构的方法）。事实恰恰相反，Monitor 用来处理自包含的情况，所有的数据和操作都必须在一起声明，在一个地方编写，由一个人管理，与系统的其他部分完全隔离开来。

事物不是表面上看起来的那样

当我第一次看到 Java 程序中支持同步的方式时，我也以为它是基于 Monitor 的模型，因为它看起来与我在学校里学到的那种 Monitor 模型一样。但据我所知，这种模型是对 Pascal 这种记录型数据的一种扩展。你可以创建一个新的记录，然后这个记录中的所有的方法都会声明为同步。这与 Java 的同步机制差不多，一样可以定义一个类，然后将类中的所有方法都声明为同步方法。

在 Java 中有两种同步方式：用 `synchronized` 关键字将一个方法声明为同步，或者像下面一样，在具体实现的代码中使用 `synchronized` 锁。

```
synchronized (anObject) {  
    // 关键业务代码  
}
```

要问一下你自己，喜欢用哪种方式处理同步。我必须得承认，自己已经习惯性认为前者更好一些。

要支持同步，建议在定义方法的时候就声明同步。这种方式对我来说也很有用。我可以写一个类，不出现死锁的情况。但如果有人继承我的类，也使用了同步方法呢？因为我写的同步方法可能是私有的，无法被子类覆盖，所以该开发人员不清楚我是如何使用了锁，于是也使用了和我一样的同步锁。结果怎样呢？到处死锁。

继承父类中的同步方法，并调用了新的方法，可以说是打破了父类的同步模型。这样做会使得父类设计同步模型时要满足的所有前提条件都失效，可谓是危险之极。在任何一个 Java 程序中都会出现这种情况。但如果是设计 API 的话，这种情况简直就是一个程序杀手。设计 API 时应该尽量保证其简单易用。如果用户在继承子类时，还需要考虑锁等情况，那么你的 API 对这些用户来说，就暴露了太多不必要的细节内容。如果这样做，一旦自己的类库出现死锁，那就等于在自毁名声。

避免这种情况出现的办法只有一个，就是决不要让别人知道你的 API 中哪里使用了同步。也

就是说，对于公开的 API 的类，你不能将其对外方法声明为同步。特别是如果一个类允许外部来继承它提供子类的话，这种做法是非常有害的，因为用户也许仍然会相信 Java 的同步是基于 Monitor 机制的，所以他们可以随意向 Java 类中添加自己的同步方法。能避免这种情况的唯一办法，就像是下面的代码一样只使用内部锁，当然使用 `java.util.concurrent` 包中提供的类似的锁功能也是可以的。

```
private final Object LOCK = new Object();
private int counter;

private int getCounter() {
    assert Thread.holdsLock(LOCK);
    return counter;
}

private void setCounter(int c) {
    assert Thread.holdsLock(LOCK);
    counter = c;
}
```

这些方案涉及的都是底层技术，与 Monitor 的概念已经完全无关，但它们至少可以保证同步的内容不会出现问题。代码写得不漂亮，而且有点复杂。但除非 Java 能够提供更漂亮、更正确的同步模型，否则宁愿牺牲代码的漂亮和简单性，也要保证同步代码能够正确执行。

11.3.3 触发死锁的条件

在多线程的应用程序中，要处理死锁问题可谓是棘手之极。因为每一个编写操作系统的人都会遇到大量此类问题，所以也有很多人对这个问题进行了广泛而且深入的研究。大部分程序不会像操作系统那么复杂。但一旦程序中允许执行外部代码，就需要处理该类问题了。

尽管对死锁问题已经进行了大量的研究，但仍然还没有找到一个简单的方案解决该问题。研究表明，只有同时满足下述四个条件，才可能引发死锁。

- **互斥条件** 必须有某一种资源（如锁、执行队列等）只能被一个线程访问。
- **无优先级调度** 在没有优先级调度的情况下，如果某个资源被一个线程占用了，那么就没有任何办法强行释放该资源。
- **持有和等待条件** 一个线程有可能无限等待某个资源或者无限持有某个资源。
- **可以获取多个资源** 在已经持有一个资源的情况下，还要去申请另外一个资源（锁、执行队列）。

如果代码能够破坏了上述四个条件中的至少一条，那么就可以保证写出一个不会死锁的系统。但想写出这样的代码远非易事，在 Java 中，还没有一套可遵循的编码标准能在编写同步代码这方面加以约束。在多年开发中，我们解决了代码互斥的问题，但有时候是数据持有和等待，有时候是其他问题。随后，我们将类库合并在一起，运行时却发现到处死锁。但我们仍然不知道如何才能用静态检查的方式在源代码中查找同步问题，从而找到可能出现死锁的地方。因此战胜死锁决不是一件轻松的事情。

有些编程语言，如 Java，它们支持多线程和锁定，所以有可能会出现死锁。对于相应的开发人员，基本的建议就是在调用外部代码的时候决不要加锁。如果能遵守这一原则，就可以避免前面文中死锁的第四个条件。因为只有四个条件都满足，才有可能产生死锁，所以避开这种情况就可以避免死锁，你也许觉得这是避免死锁的最终方案！事实上，有时候想满足这些条件是非常困难的事情。注意看下面的代码，你觉得是否会产生死锁呢？

```
private HashSet<JLabel> allCreated = new HashSet<JLabel>();

public synchronized JLabel createLabel () {
    LOG.log(Level.INFO, "Will create JLabel");
    JLabel l = new JLabel ();
    LOG.log(Level.INFO, "Label created {0}", l);
    allCreated.add (l);
    return l;
}
```

这段代码看起来非常安全，因为它只是调用了 `HashSet.add` 这个外部方法。注意 `HashSet` 并没有使用同步。不过，事实上，出现问题的可能性有很多种。第一个可能引发问题的点就是继承自 `JComponent` 的 `JLabel` 类。`JLabel` 会逐级调用父类的构造函数，最终会调用 `Component.getTreeLock()` 方法从而获得一个 AWT 的控件树锁^①。假设某一位开发人员覆盖了这个 `getTreeLock()` 方法，代码如下。

```
public Dimension getPreferredSize () {
    JLabel sampleLabel = createLabel();
    return sampleLabel.getPreferredSize ();
}
```

现在，代码就处于可能死锁的危险中，因为当一个控件在绘制界面的时候，经常会调用 `getPreferredSize`，但此时这棵 AWT 控件树却是被锁住的。即使你努力避免调用外部代码，最后还是避免不了。还有一个可能导致死锁的隐藏点就是 `HashSet` 类。这个类通过调用相应对象的 `hashCode` 和 `equals` 两个方法来做 Hash 数据存取，一般类默认是使用了 `Object` 自带的实现，但请注意，这两个方法是有可能被随时调用的，而且也可以被任意子类覆盖，子类覆盖该代码时涉及的数据完全是未知的^②。所以如果在具体实现的时候中去请求另一个锁，那么可能会出现你意想不到的死锁问题。

针对上述问题的建议也很简单：只要有可能的话，决不要在持有锁的情况下调用外部代码，特别要避免调用自己类库之“上”的代码。那么何谓“上”呢？简单解释一下：程序员编写代码时总要依赖于某些其他类库。比如说，某些代码中需要去读文件、操作字符串和使用集合功能。但这些要用到的功能通常是由一些“底层基础”类库提供的。JDK 的 `rt.jar` 就提供这些功能，

① AWT 是 Java 提供的跨平台 UI 控件体系，其中控件是有关系的，如某个按钮在某一个面板上，该面板又在某一个对话框中，从而形成一棵控件树，这棵树是允许加锁的。——译者注

② 如果某一个 `JComponent` 的子类覆盖了 `hashCode` 方法，而且通常在计算 hash 值的时候会用当前对象的字段数据，如 `prefSize`，而此时是在一个同步方法中调用 `hashCode`，那么这个对象及其字段已经被锁定，二次请求的时候，就会出现死锁。可以参考 11.3.4 节中的代码。——译者注

而且该类库中代码调用的内容也是受限的，不在 JDK 启动路径上的代码是不会被 `rt.jar` 这个类库所调用的。相比之下，你所编写的类库可以说是位于 `rt.jar` 之“上”，因为 `rt.jar` 根本不知道你所编写类库的存在，它只会找到 JDK 启动路径上自身提供的类，并直接调用这些类。所以除非 JDK 自己的 bug，或者 JDK 调用了其“上”的代码，否则 JDK 本身运行时是不会产生死锁的问题。

其实对类库的使用者来说，也面临着类似的情况。他们能调用的类也只限于路径上引入的类库。只有在路径中引入了指定的类库以后，才能调用该类库中的代码。同样，只有你开发的类库中存在 bug，或者你调用了外部代码，才可能触发死锁。关于这种类库中的类，在设计时可以保证锁的关系，不会有什么问题，但有两种潜在的威胁可能会破坏原有的锁关系，一个是继承，还有一个则是 Listener 机制。

如果允许继承的话，就表示，类中的每一个非 `final` 方法都可以被外部代码所替换。因此，如果一个允许继承，那么对该类中的某个可被覆盖的方法，如果它可能会持有某种锁的话，就绝不要去调用它。当然还是有可替换方案的，在 10.4 节中，你可以用多个接口来达到同样的目的，避免使用可被继承的类，这种方式会根据关注点的不同来分离不同的功能代理以完成相应的功能。

Listener 则意味着另外一个后门。它可在代码运行的时候，将自己注入到一个对象中，成为该对象方法执行时的一个代码片段。注意，Listener 做什么事情，对于设计者来说，是无从揣测的。设计者也绝不应该去假设一个 Listener 可能会做什么事情。要知道，对于 Listener 来说，大部分情况，它所在的类库其实是在你的类库之“上”，也就是说 Listener 所在类库会依赖于你的类库。它的代码可以清楚地知道你的类内容，因为它会调用 `addXYZListener` 方法来添加相应的 Listener。因此，这个类库可以随时调用你类库中的代码。这样，如果你的代码中持有锁的话，那么调用 Listener 的时候可能会引发死锁问题。所以如果要调用 Listener，那么在调用时决不能持有锁。

执行原子操作

仅仅在自己的代码中避免调用持有锁的 Listener，其实还不能完全解决问题。现在看一下 NetBeans 文件系统库中的一个例子。前文中已经说过多次，NetBeans 框架使用了 `FileObject` 作为文件访问的抽象层，而不是直接使用 `java.io.File` 这个类。`FileObject` 这个接口的优点在于它支持 Listener。所以 NetBeans 中的代码可以监听文件的改变，并获取文件对应的状态。

Listener 的调用是经过同步处理的。这样做有其优势，同时也会带来相应的副作用，因为极有可能引起死锁。开始我们实现 `FileObject` 类的时候出现了一大堆的 bug。因为在具体实现中，我们回调了相应的 Listener，而这些 Listener 却会去请求我们已经持有的锁。所以运行时总是出问题，我们必须不断修改。但让我吃惊的是，仅仅修改我们自己的文件类库并不能完全避免死锁。

问题出在利用文件系统的 API 上，当它写文件或者删除文件时，都会触发相应的 Listener。而相应 Listener 的具体实现是由第三方提供，完全未知，而此时文件系统 API 中的方法已经处于加锁状态，所以这些 Listener 的具体实现就不应该再去调用这些已经上锁的方法。当然，在特定情况下，这也是无可避免的开发人员确实需要在内部使用同步，比如说要向磁盘写入内容，

所以这种情况也很难完全避免。调用基础而且已知的类库中的操作应该是安全的而且也是常见的。但因为 NetBeans IDE 使用了 Listener 机制来对外发布文件资源改变的事件，所以就很难保证锁安全。

我们解决这类问题的方案就是使用原子操作。别误会，我们的文件系统不支持像数据库操作那样的事务处理。但它允许将 Listener 的通知延后。也就是说，你可以将代码放置在一个 Runnable 的实现类中，然后交给 NetBeans 的文件系统类库，由它来调用 Runnable 的具体实现来修改文件，然后等线程中的操作完成后，才会发事件通知相关的 Listener。这样，不管代码是一个原子操作，还是使用自己的 Listener，或者创建、删除磁盘资源，任何操作都不会调用未知代码。

在我看来，如果一个底层基础类库提供了 Listener 机制，而且想支持同步，那么就需要像我们的文件系统类库一样，必须支持类似的原子操作。否则，可能会出现死锁的情况。随着类库用户的增加，出现死锁的几率也在增加。但支持原子操作并不是那么简单。所以，如果我有机会重写文件系统 API 的话，我只会选择异步 Listener 机制。

开发 NetBeans 的经验告诉我们，其实另外那种异步 Listener 方案才是更好的方案，该方案会在一个专门进行任务调试的线程中异步调用相应的 Listener。这样做更安全。这个调试线程不会持有任何锁。但这样异步调用，会使得事件的通知延后，当 Listener 收到事件时，可能相应模型的状态已经完全不同了。所以在这种方案中，事件类就不需要或者说根本不应该包含事件发生时的大量状态信息。包含这种不准确的状态信息至少会使人误解，因为 Listener 收到事件时的模型状态已经完全不同了。所以只需要通过事件来通知相应的 Listener 有模型发生改变，然后由相应的 Listener 自行查询模型的状态即可。

异步调用是用来对抗死锁问题的一柄利器。就好像当操作系统的多个进程允许同时使用一个打印机时，打印机可以将收到的打印任务放置在缓存池中，逐个执行。这种延迟执行的方式可以避免多线程同时访问某个稀缺资源时而出出现的死锁问题。但很多时候，这些延迟执行的操作不像打印操作这样简单。打印机之所以在这方面处理得比较好，主要是因为这些需要打印操作的进程不需要等待打印任务完成。一旦程序向打印机发出了一个打印任务以后，接下来就是工作人员去取打印结果，而程序则根本不用管人什么时候去取打印稿件，而是继续打印下一份文件。但这种情况并不适用于所有的延迟操作。可能还有很多操作需要等待相关的操作完成以后才能继续自己的任务，即使这是一个异步操作，也需要等待其完成。这样，绕了一圈，又回到了起点，异步调用还是存在死锁的风险。

把 `SwingUtilities.invokeLater` 和 `SwingUtilities.invokeAndWait` 这两个方法放在一起比较一下，就能体会到异步调用的好处，当然，也同样体现了异步调用的缺点。这两个方法都放在一个 Runnable 的实例中来操作，而 Runnable 的调用则由 AWT 的调度线程来处理，该线程专门用来操作 Swing 控件。调用 `SwingUtilities.invokeLater` 这个方法通常来说是安全的，因为调用完了后，就会立刻返回，具体操作会由该类通过一个线程来完成，而调用 `SwingUtilities.invokeAndWait` 方法则有所不同，有时候我想也许这个方法应该改名为 `invokeAndDeadlock` 会更合适一些。调度线程

会直接执行这个方法，然后可能会出现死锁。特别是如果当前的这个调度线程已经持有了锁，而且更不幸运的是其持有的锁是 AWT 控件树锁，那么死锁几乎会必然出现。当然，如果在 Runnable 中使用不正确的代码来调用 `SwingUtilities.invokeLater` 也是有可能引发死锁的。但想写出这样的死锁代码也是多少要费点心思才能做到的。比如说，要利用不少技巧来写同步操作代码才可能出现这种情况。而要用 `SwingUtilities.invokeAndWait` 方法产生死锁就简单多了。所以如果一个 API 想避免死锁的话，就不应该提供这类方法。虽然说有时候提供这类方法也是必要的，比如进行测试时，可能必须要等到异步操作完成才能知道测试结果是否正确。如果遇到这种情况，这个方法也应该换个名字，而不是用现在这个名字，搞得这个方法和 `SwingUtilities.invokeLater` 看起来像兄弟俩，事实上，这两个操作的区别很大。

计划和等待

Java 5 中的并发机制为了支持延迟操作，引入了 `java.util.concurrent.ExecutorService` 这个接口。下面给出的操作代码显示该接口在等待机制方面与以往的不同之处。

```
ExecutorService service = Executors.newSingleThreadExecutor();
```

```
// 与之前不同的延迟执行
```

```
Future<?> task = service.submit(someRunnable);
```

```
// 比较下等待操作
```

```
task.get(1000, TimeUnit.MILLISECONDS);
```

之所以这样设计，是因为过去的经验表明等待操作也会引起死锁，所以如果开发人员需要等待操作，那么写代码时要尽可能小心。

但想完全避免等待操作，也是非常困难的一件事。可能需要完全基于消息机制来构建整个系统。多种基于消息机制的 Java 服务器已经证明了这种方案是可行的。但如果系统没有采用消息机制或者说系统使用了一些非消息机制的类库，仍然无法完全避免等待操作。可以将很多操作延迟处理，但等待的时间要有一个限制。现在能避免死锁的唯一方案就是允许等待超时，即控制一个线程的等待时间不得超出一个指定上限。这种方案会破坏产生死锁的第三个条件，即线程会无限地等待某个条件。除非某个开发人员非要在一个 while 循环中去运行等待操作，否则就不会产生死锁。

简单的客户端 API，延迟调用第三方代码

如果要写一个可能会被多人调用的 API，而且调用 API 的代码可能会分散在多个线程中，而你则希望对外提供给用户的 API 只是一个很简单的单线程调用，然后用户将具体的 API 调用代码放在这个线程中执行，那么前面给出的超时方案就非常有用。这样就可以保证所有内容都在一个线程内处理，从而大大减少并发执行可能出现的问题。

在收到用户请求的时候，一定要新创建一个单独的线程来进行处理工作。一个客户端代码在调用某个 API 的时候，也只是把要做的任务直接发给这个线程就可以了。如果这个方法没有返回值就立即返回；如果出现堵塞，该线程就一直处理该请求。如果出现超时，就可以直接向

调用者抛出一个异常通知该请求已经被取消了。

当我在设计 NetBeans 数据系统的 API 时，为了避免死锁问题，我采用了这种方案。因为 API 中几乎所有的方法都与文件有关联，这些方法都可能抛出 `IOException` 异常。所以创建 `TimeoutException` 很容易，它继承自 `IOException`。在出现超时的情况下，抛出 `TimeoutException` 异常。因为 `IOException` 是 `Checked` 异常，开发人员调用这些方法的时候，必须要捕捉该类异常，所以虽然加了这样一个异常类，也没有使客户代码变得更加复杂。

这样做也可以很好地消除死锁问题。但它也可能会增加整个系统中不可预测的行为。毕竟，即使没有出现死锁，也可能会出现超时异常。这正是 NetBeans 没有采用相同方案的原因所在。但我相信，配合正确的测试，该方案还是可用的，未来可以有效地避免系统中的死锁。

还有另外一个方式可以破坏无限等待（即产生死锁的第三个条件），就是从外部将某个资源从一个线程中抢走。如果一个线程已经处于同步等待状态时，那么这样的操作是很难做到的。但 `SwingUtilities.invokeLater` 方法却有可能完成这种操作。要知道 AWT 的事件调度线程也是一个资源，有很多其他线程需要得到该线程。所以可以写一个守护线程，当然也可以直接修改这个 `invokeAndWait` 方法，添加代码来监控是否出现死锁。可以通过 `Thread.getAllStackTraces` 这个方法取得当前线程的栈信息，以检查是否出现死锁。如果发现死锁情况，就可以中断当前线程从而破坏死锁。

你大概也已经明白了，与死锁问题的战争看来是一场持久战。这可能是因为没有哪种方案能从根本上解决死锁的问题。所有这些战术都很容易被攻破，因而溃败。在 NetBeans 中我们也面临着同样的问题。因为死锁是一个致命性的问题，我们一直在尽力寻找一种防御方案。我们完全放弃了漂亮的代码和优雅的设计，只想集中精力寻找一种方法来解决问题。我们最终使用强力的手段来编写测试以检查程序中是否存在已知的死锁代码。

11.3.4 测试死锁

尽管很难避免死锁的出现，但在 Java 语言中，想分析它们还是比较简单的。与编译为机器代码来执行的 C 语言相比，Java 应用在出现死锁的时候至少还可以生成线程的堆。从这个线程的堆中，你可以查找问题。找到问题之后，修复问题就变得比较简单了，可以再加一层锁，或者使用 `SwingUtilities.invokeLater` 方法延迟调用原先的代码，从而避开容易出现问题的区域。多年来我们一直使用该方案，结果就是将大量代码修改以后，其运行时的结果完全不可预测。就算是这样，我们也没有从根本上避免死锁问题。在我们修正某段代码以解决某个死锁问题时，往往又会在其他地方引发另外一个死锁问题。图 11-7 中的两段代码分别写于 2000 年 6 月 26 日和 2004 年 2 月 2 日，这也是我最喜欢的、修复死锁的两个例子。第二段代码又返回到了第一次集成之前的状态。2000 年时我们修复了一个死锁，成功地避免了我们的代码中出现阿米巴变形虫模型。但在四年后，我们还是恢复到了原来的处理方式，因为为了改善某个性能，只好退回到 2000 年时的代码。如果当时第一次修复问题的时候，能够采用更加透明的方式来编写和集成测试用例，就可以避免这种反复的情况了。

revision 1.42, 2000-06-01 15:45:21+0000	revision 1.43, 2000-06-26 08:56:40+0000
Line 229	Line 229
public final Node getNodeDelegate () {	public final Node getNodeDelegate () {
if (nodeDelegate == null) {	if (nodeDelegate == null) {
	Children.MUTEX.writeAccess (new Runnable () {
	public void run () {
if (nodeDelegate == null) {	if (nodeDelegate == null) {
nodeDelegate = createNodeDelegate ();	nodeDelegate = createNodeDelegate ();
}	}
}	}
revision 1.12, 2004-01-28 07:11:54+0000	revision 1.13, 2004-01-29 11:51:59+0000
Line 219	Line 219
public final Node getNodeDelegate () {	public final Node getNodeDelegate () {
if (nodeDelegate == null) {	if (nodeDelegate == null) {
Children.MUTEX.readAccess (new Runnable () {	
public void run () {	
synchronized (nodeCreationLock) {	synchronized (nodeCreationLock) {

图 11-7 2000 年为了避免阿米巴变形虫模型而调整的代码, 以及 2004 又改回来的代码

写一个用例来测试死锁? 对, 你没有看错。当然, 猛一看起来, 的确让人大吃一惊。但写一个可以测试死锁的用例不仅可能, 也不是很难, 通常不超过两个小时, 最多也不超过一天。编写这种测试用例, 除了能够带来自动化处理的好处, 还可以让程序员修正了一个死锁问题以后, 可以充满信心地大叫说我的确修正了这个问题。要知道死锁的 bug 是比较难重现的, 而且修复后, 质量管控部门也很难验证是否真的修复了这个 bug, 所以通过测试用例来保证死锁问题的确被修复是一个比较不错的方式。这样做还有一个好处, 如果能够有一个用例来测试这个 bug, 那么可以直接选择一个简单的方案来修复, 而不用绞尽脑汁, 寻找一个优雅而且可能过于复杂的方案。这样做可以让以前死锁修复难题变成一个简单的修复过程。任何一个开发工程师都可以修复死锁问题, 这才是我们想要的!

写一个能够测试死锁问题的用例并不是很难。像前面那个 Swing 的例子, 先写一个控件, 然后覆盖父类的 `getPreferredSize` 方法, 然后创建控件时就会出现线程被锁死的问题, 但下面的代码示例却给出了另外一种相反的死锁情况。

```
public class DeadlockTest extends NbTestCase {
    static final Logger LOG = Logger.getLogger(
        DeadlockTest.class.getName());
};

public DeadlockTest(String n) {
    super(n);
}

@Override
protected int timeOut() {
    return 10000;
}
```



```

public static class StrangePanel extends LabelProvider {
    @Override
    public Dimension getPreferredSize () {
        try {
            Thread.sleep(1000);
            JLabel sampleLabel = createLabel();
            return sampleLabel.getPreferredSize();
        } catch (InterruptedException ex) {
            Logger l = Logger.getLogger(
                DeadlockTest.class.getName()
            );
            l.log(Level.SEVERE, null, ex);
            return super.getPreferredSize();
        }
    }
}

public void testCreateLabel() throws Exception {
    final LabelProvider instance = new StrangePanel();

    class R implements Runnable {
        public void run() {
            JFrame f = new JFrame();
            f.add(instance);
            f.setVisible(true);
            f.pack();
        }
    }

    R showFrame = new R();
    SwingUtilities.invokeLater(showFrame);

    Thread.sleep(500);
    JLabel result = instance.createLabel();
    assertNotNull("Creates the result", result);
}
}

```

如果有两个线程同时运行的话，这个测试就能生效。其中一个线程创建一个控件并显示在界面上，这个控件被创建时会回调 `getPreferredSize` 方法，此时，该控件持有 AWT 控件树锁。这个时候，我们启动了另外一个线程，该线程启动后不久就会取得 `createLabel` 的锁。由于在这个代码的实现中，锁出现在 `JLabel` 的构造函数中，所以一旦第一个线程继续运行（在 1000 ms 之后），就会出现死锁。当然，修复该问题的方案很多，但最简单的方式也许就是在构造 `JLabel` 的代码中使用同步来保证不会出现死锁，示例代码如下。

```

private HashSet<JLabel> allCreated = new HashSet<JLabel>();

public JLabel createLabel () {
    synchronized (getTreeLock()) {

```

```

        LOG.log(Level.INFO, "Will create JLabel");
        JLabel l = new JLabel ();
        LOG.log(Level.INFO, "Label created {0}", l);
        allCreated.add (l);
        return l;
    }
}

```

修复这个 bug 非常简单，比写这个测试用例简单多了。但没有这个测试用例，我们就不能固定这种阿米巴变形虫模型。编写这样一个测试用例比修正该 bug 花的时间还要多很多。

通常情况下，可以在现有的 API 的基础上来编写测试用例，像上面的例子中就覆盖了 `getPreferredSize` 方法。只有在少数特殊情况下，才可能需要引入一个特定的方法在测试用例中模拟问题的产生。这很正常，因为测试用例通常会与正式 API 在同一个包中，而测试用例可能需要用到包的私有方法，这样 API 就不会因为测试的需要而对外暴露不必要的细节内容。

死锁测试其实是一种事后诸葛亮的行为。因为只有出现了一个 bug 之后才会去写这样的用例，没有人能够在事先就规划好这样一个测试用例。项目开始的时候，最好还是把精力放到如何做出一个优秀的设计上。但前文也提过了，迄今为止，还没有可以避免死锁的通用理论，所以只能在死锁出现以后，再来使用该方案解决这个问题。出于这种考虑，我建议这种测试只用在处理我们的阿米巴变形虫模型这种情况上。

11.3.5 对条件竞争进行测试

还有一些与之相类似但略有不同的用例，也可以用来测试在代码并发执行的情况下，如何查找那些因为条件竞争而引发的问题。与死锁一样，因为多线程和同步而引发的问题非常难以预测。但只要收到一个 bug 报告，最好还是写些测试用例来验证各种数据结构并发访问时的处理是否正确。有时候，预先写一些这类测试也是很明智的做法。

因为在 NetBeans 中，需要在启动时加锁，所以我们也经历过类似的问题。启动时加锁是为了避免用户重复启动 NetBeans IDE。当发现 NetBeans 已经启动了，我们会向用户发出警告，通知他们已经有一个 NetBeans 的实例在运行了。在用户收到警告后，还需要关闭这个新启动的 NetBeans 实例。这些处理与 Mozilla 和 OpenOffice.org 的处理类似。我们决定开一个套接字服务器，同时在一个特定位置创建一个文件，并写入这个套接字服务器的端口号。当 NetBeans 开发工具启动的时候，就会去读取这个端口号，并与之通信，检查该端口号是否已经被占用。

这些内容与 API 设计有关吗

答案是肯定。启动锁是一个基于文件的 API，同时也基于网络协议的用例。它允许任何进程来访问一个正在运行的 NetBeans 程序实例，所以考虑到其目的，要能保证任何第三方的代码都能可靠地使用该 API。而且，我们还在最新的 NetBeans 版本中添加了一个 `Sendopts` 的 API，这个 API 允许 NetBeans 平台中的每一个模块都能处理命令行参数。这个 API 很好用，很多开发人员都用到了该 API，但这个 API 的正确性取决于锁的启动功能是否正确，所以通过测试该 API 来验证启动锁的正确性。

我们需要通过优化,来解决的一个重要问题,就是如何让用户同时启动多个 NetBeans 实例的情况。比如说用户多次点击 NetBeans 的启动图标,或者把多个文件同时拖放到 NetBeans 的启动图标上,就会出现几乎同时启动多个 NetBeans 实例的情况。此时,会有多个进程在启动时来争抢这个特定的文件,并向文件写入内容。下面是一个进程的运行结果,但系统可以随时中断这个进程。

```
public static int start(File lockFile) throws IOException {
    if (lockFile.exists ()) {
        // 先读取端口号,再建立连接,以便进行 11 号状态
        int alive = readPortNumber(lockFile);
        if (alive != -1) {
            // exit
            return alive;
        }
    }
    // otherwise try to create the file yourself
    lockFile.createNewFile();
    DataOutputStream os = new DataOutputStream(
        new FileOutputStream(lockFile)
    );
    ServerSocket server = new ServerSocket();
    int p = server.getLocalPort();
    os.writeInt(p);

    return p;
}
```

因为执行的不是原子操作,所以在多进程下,文件的控制权可能会被另外一个执行同样操作的进程抢走。如果已经有一个进程创建了某个文件,然后另外一个进程去读这个文件,但此时第一个进程还没有向这个文件中写入端口号,那么会发生什么事情呢?如果前面进程创建的文件在进程操作完文件后被删除了,又会怎样?再假设测试时发现某个文件不存在,于是创建该文件,但这时却有某个进程已经创建了该文件,那又会怎样?

如果想保证程序中的代码没有问题,那么就需要考虑到所有这些问题。为了保证我们担心的问题不会发生,我们在启动锁的代码中加了多个检查点。下面的代码是对前面的修改,其中 enterState 在正式的版本中是一个空方法,不做任何处理,但在测试环境下,它可以用来在特定的检查点执行堵塞操作。

```
public static int start(File lockFile) throws IOException {
    enterState(10);
    if (lockFile.exists ()) {
        // read the port number and connect to it
        enterState(11);
        int alive = readPortNumber(lockFile);
        if (alive != -1) {
            // exit
            return alive;
        }
    }
}
```

```

    }
}
// otherwise try to create the file yourself
enterState(20);
lockFile.createNewFile();
DataOutputStream os = new DataOutputStream(
    new FileOutputStream(lockFile)
);
ServerSocket server = new ServerSocket();
enterState(21);
int p = server.getLocalPort();
enterState(22);
os.writeInt(p);
enterState(23);

return p;
}

```

我们可以写一个测试用例，这个用例会启动两个线程，其中有一个线程在 `enterState(22)` 处会被堵塞，第二个线程会继续运行，此时已经存在一个文件，但尚未写入端口号，这样就可以观察代码对这种情况的处理是否正确。

这个方案的确有效，除了对某些初始化操作还没有把握以外，我们认为 90% 的操作都可以被正确地处理，随后我们将代码整合到第一个版本中。当然，之后还有一些工作要做，也还有些 bug 需要修复。但我们已经有了可以自动化测试的用例，能自动测试具体实现是否正确，验证代码对边界条件的处理是否合适。我们坚信这样做可以帮助我们修复所有重大问题。

11.3.6 分析随机故障

正如我们预料，为了查找上一节中提到的 10% 的随机故障，我们花费了更大的精力，相比之下，用来测试这些问题和修复问题的时间就要少得多。事实上，没有它们也就不会有本书。因为这些故障是随机出现的，而且往往出现在别人的机器上，因此查找起来需要更加先进的跟踪技术。

并发执行引起的问题，就是没有多少信息可以帮助开发人员来分析是哪个地方出的问题。目前的方法论要么无效，要么只对特定情况有效。使用调试技术来检测时，又很难达到并发的限制。为此，开发人员只有用上一些古老的技术，比如用 `println` 命令和日志记录信息。这样做的效果通常来说是最好的。向代码中加入一些日志信息并执行多次，直到出现问题，然后再分析日志，查找原因并修复问题。还好，测试时也可以使用同样的方式。向程序和测试代码中加入日志，然后执行测试用例，在测试用例执行失败时，把所有收集到的日志信息都作为错误报告输出。

我们编写了一个 `Handler` 接口的实例来达到上述目标，`Handler` 是 JDK 中的一个抽象类，它可以处理日志和错误报告。因为这个基类在 `java.util.logging` 包中，是 JDK 提供的，这就意味着可以在任何测试或者程序代码中使用这个类，而不需要在自己的执行环境中请求任何特殊的库。为了获取相关的日志信息，只需要实现一个自己的 `Handler` 类。在运行测试用例之前，将这个实现类注册到系统中，然后就可以获取运行过程中的所有日志信息。一旦代码执行时出现问题，

就可以记录相关的错误信息了，下面的代码就是 Handler 的一个功能演示。

```
public class CLILoggerBlockingTest {

    public CLILoggerBlockingTest() {
    }

    @BeforeClass
    public static void initHandler() {
        Logger.getLogger("").addHandler(new H());
        Logger.getLogger("").setLevel(Level.ALL);
    }

    @Before
    public void setUp() {
        H.sb.setLength(0);
    }

    @Test
    public void start() throws Exception {
        File lockFile = File.createTempFile("pref", ".tmp");
        int result = CLILoggerBlocking.start(lockFile);
        assertEquals("Show a failure" + H.sb, -10, result);
    }

    private static final class H extends Handler {
        static StringBuffer sb = new StringBuffer();

        @Override
        public void publish(LogRecord record) {
            sb.append(record.getMessage()).append('\n');
        }

        @Override
        public void flush() {
        }

        @Override
        public void close() throws SecurityException {
        }
    } // end of H
}
```

在测试中，可以用日志来记录整个操作过程的各个重要组成部分。这样做的好处就是在测试过程不会引入新的内容，最多只是大量调用标准的 `java.util.logging.Logger` 类的 `log` 方法。只要这样，测试用例就可以收集所有的信息，而且一旦出现问题，还可以提供完整且全面的故障信息。随后可以用这些信息来分析问题并解决，或者让日志信息更全面、更详细，这样可以更加有效地跟踪和分析问题。

有时候,开发人员不愿意对那些在测试过程中随机出现的问题进行分析,因为他们觉得这种问题在正式运行的环境下不会出现,不会影响产品代码。也许是因为测试用例中的代码有问题,而程序的代码则是正确的,如果因为测试用例的问题就去修改程序代码,也太冒险了一点。如果不深入思考,程序代码中就有可能存在问题。即使这个问题并不总会出现,但一旦出现,后果可能不堪设想。如果想确保程序代码没有问题,最好还是向程序代码中添加一些日志信息,并编写一些能够很好处理日志的测试用例。

11.3.7 日志的高级用途

从 NetBeans 项目开始,我们引入了能够收集日志信息的测试用例,只关注对外暴露功能的地方,那些地方因为多线程调用而使得程序特别复杂,容易引发问题。对于每个用 Java 编写的程序,即使只用了一个线程,也有可能有问题,因为 Java 自身就会启动多个线程。当测试程序的运行环境变化时,有可能启动垃圾回收器。对于那些用户界面控件,至少还会有一个 AWT 的事件调度线程,而该线程在接收事件时并没有严格的顺序。因此,几乎所有测试用例中,使用日志都是非常有用的。NetBeans 项目则开发了一个 NbJunit 的单元测试框架来支持这些日志测试用例。这是一个通用类库,并不限于 NetBeans 项目使用,任何 Java 项目都可以使用该类库进行测试。

这个框架提供的最简单功能就是能够在测试用例执行的过程中收集信息。开发人员可以在 NbTestCase 的子类中通过覆盖 `logLevel` 这个方法来调整日志记录的内容,如下所示。

```
public class CLIHandlerBlockingWithNbTestCaseTest extends NbTestCase {

    public CLIHandlerBlockingWithNbTestCaseTest(String s) {
        super(s);
    }

    @Override
    protected Level logLevel() {
        return Level.ALL;
    }

    public void testStart() throws Exception {
        File lockFile = File.createTempFile("pref", ".tmp");
        int result = CLIHandlerBlocking.start(lockFile);
        assertEquals("Show a failure", -10, result);
    }
}
```

下面就是收集到的所有信息内容。如果任何一个测试方法出现了问题,在程序执行过程中产生的所有日志记录都会包含在错误报告中,格式如下所示:

```
[name.of.the.logger] THREAD: threadName MSG: The message
```

上面只截取了一部分文本信息,方便查看。同时框架还提供了完整的日志报告。日志报告文

件存放的位置可以通过 `NbTestCase.getWorkDir()` 方法获得,通常是在 `/tmp/tests/testLogCollecting-Test/testMethodOne` 目录下面,该目录的 `testMethodOne.log` 就是日志报告文件。这个文件存放了最后 1 兆的日志输出信息。这份日志报告提供了大部分最后记录的信息,应该足以应付需要深入分析复杂日志代码的情况。

比较程序运行失败和正确运行时的区别

有时候日志信息非常长,特别是对于随机故障,很难分析问题出现的原因。过去的经验告诉我们,处理这种情况的最好策略就是添加一句 `fail("OK")` 的代码。一旦测试用例执行过程中出错,就会在这段代码行处中止程序运行。这样做也同样可以生成一个日志文件。然后将这个因为错误中止程序运行的日志与那个正常运行产生的日志进行比较,找到不同的地方。另外在输出日志时,最好不要出现 `@543dac5f` 这种信息^①,因为每一次代码运行时,对象放置的内存位置都不相同,这个内存位置没有任何意义。如果能够找到两个日志文件中的不同,就可能分析出两次执行中的不同之处,从而找到问题的根源。

如果说无法从这两个日志文件得到分析结果的话,有两种可能。首先是日志文件没有足够的信息,或者是日志文件中记录的信息可能太粗略,无法就问题发生的原因提供明确的信息。如果是这个原因,就向测试用例中加入更多的日志信息。可以通过 `NbTestCase` 的 `logLevel` 方法,来输出不同级别的日志信息,比如说 `Level.FINE`, `Level.FINER` 和 `Level.FINEST`。在不同的测试用例中,可以启用不同的测试信息。如果怀疑有随机故障可能影响关键代码,可能需要在这段代码中逐行加入日志。可能还需要在日志中记录方法的各个参数,以及代码执行时各个参数的值。这样做可以得到足够的日志信息,从而可以更加深入地分析随机故障出现的原因。

但从另外一个角度来看日志,还有一个隐藏的点:向程序中添加的日志信息越多,程序的行为也就被改变得越多。每次调用 `Logger.log` 方法都会带来额外的时间消耗,同时,对日志信息进行格式化处理也会花费额外的时间。事实上,对于原来运行三次就会出现一次错误的程序,如果在程序中添加了记录日志信息的代码,运行时几乎都不会再出错。因此重现一个错误会变得非常困难,也更难以评估该问题。有时候,甚至连调试人员都想放弃重现该错误。但这个错误还是会在别人的机器时不时地露上一脸。在 `NetBeans` 开发过程中,我们有一大堆不同配置的机器用来进行每日测试。这样做有助于我们跟踪一些很罕见的错误。有时候,测试框架能报告我们在开发人员机器上都没有发现过的 bug,这样可以帮助我们更深入地查找潜在的 bug。

如果在你的程序中带有太多的日志信息,还会带来另外一个问题。遇到这种情况,也很难定位具体的消息来自于哪里。比如说,如果测试用例在执行时不停地调用某一段相同的代码,就会打出完全相同的日志信息,让你完全不知道从何下手。如果程序在某一个循环中执行了一些操作,就会重复地打出相同或相似的日志信息,这种情况尤其麻烦。关于这种情况的处理,我只能建议在测试代码中记录日志。这样,就可以把信息固定在日志的某个位置。在比较正常运行和运行失

^① 默认情况下输出一个对象,是调用它的 `toString` 方法,而 `Object` 对该方法的默认实现是通过本地 `hashCode` 方法来取自己在内存中的地址,所以没有任何逻辑意义。——译者注

败的日志时，首先查找文本信息，然后再对具体的日志信息进行比较。在处理随机故障时，最好的建议就是尽量能多加日志，然后在测试用例中获取日志信息，而在程序代码中，也应该像测试用例一样，尽可能多地提供日志信息。这样，能够大幅提高错误根源定位的准确度。

还有一个略微不同的方案，就是测试哪些功能没有记入日志。比如说，API 中有一个新方法需要子类覆盖，可能这时要让现有的类（指那些基于以前该类版本所编写的类）能够在运行时知道这个方法确实被覆盖了，如下所示。

```
protected boolean overrideMePlease() {
    Logger.getLogger(OverrideMePlease.class.getName()).warning(
        "subclasses are supposed to override overrideMePlease() method!");
};
// some default
return true;
}
```

一个只关心测试的人，肯定希望警告信息会被输出到合适的位置。可以通过自己实现的 Logger 来处理这种情况，当然也可以使用 NbJunit 类库提供的特殊的实用工具通用方法来处理这种情况。下面的代码演示了此类操作。

```
CharSequence log = Log.enable("org.apidesign", Level.WARNING);
OverrideMePlease instance = new OverrideMePlease() {
    @Override
    protected boolean overrideMePlease() {
        return true;
    }
};
if (log.length() != 0) {
    fail("There should be no warning: " + log);
}
```

这虽然只是一个小的实用工具方法，但通过该方法却可以知道在测试用例中做了什么事情，并进行分析。在处理一些未知的情况时，利用 org.netbeans.junit.NbTestCase 和 org.netbeans.junit.Log 这两个类的其他支持，这种方法可以提供更好的日志记录功能。

11.3.8 使用日志记录程序控制流程

前几节讨论了竞争问题和死锁模拟，从这几节可以得知，需要在代码中使用钩子技术在测试环境下模拟死锁问题。在多线程执行的情况下，可以使用钩子技术在修改内部数据结构的时候来暂停线程以破坏内部数据或引发死锁。前面几节还指出应该在类中放入一些特别的方法，允许测试代码来调用。这些操作都应该考虑到开发人员对执行的控制。为此，需要在程序代码中加上 enterState(10) 代码，或者提供可覆盖的方法，从而让测试用例在程序代码中加入钩子。但在测试用例中，这种举动非常疯狂，哪个程序员想在自己的代码留下这么“疯狂的”后门呢？尽管这样做是可行的，但最好还是使用其他的方案，比如说不使用钩子代码，而是只加入日志记录功能。

记录日志？对，你没有看错。这个方案非常的漂亮，因为这个日志记录是最简单的操作。这

样，开发人员就不用在代码加入 `enterState` 这样难读难看的代码。只需要像下面的代码一样记录日志即可。

```
LOG.log(Level.INFO, "Will create JLabel");
JLabel l = new JLabel ();
LOG.log(Level.INFO, "Label created {0}", l);
```

对于程序代码，这些日志记录显得水到渠成，与代码吻合得天衣无缝。

现在，在测试代码中将自己实现的 `Handler` 注册到系统中。通过实现 `Handler` 的 `publish` 方法，可以做任何“出格”的事情。现在就用不着搞乱自己的代码了，只需要在测试用例中添加一个日志 `Handler` 就可以完成原来所有的工作。然后可以得到测试用例中代码执行的所有信息，并对其进行分析。测试用例和在程序中使用日志功能时一样强大，而且不会对程序代码造成任何污染。每一次记录日志，都表示测试用例可以影响整个程序的行为。举个极端的例子，完全可以挂起指定线程以外的所有线程，只允许这个特定线程继续执行，从而实现对多线程程序的控制。可以参考下面的代码。

```
class Parallel implements Runnable {
    public void run() {
        Random r = new Random();
        for (int i = 0; i < 10; i++) {
            try {
                Thread.sleep(r.nextInt(100));
            } catch (InterruptedException ex) {}
            Logger.global.log(Level.WARNING, "cnt: {0}", new Integer(i));
        }
    }
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(new Parallel(), "1st");
        Thread t2 = new Thread(new Parallel(), "2nd");
        t1.start(); t2.start();
        t1.join(); t2.join();
    }
}
```

这个程序包含了两个线程，每个线程从 1 数到 10，每数一个数，就会随机暂停一段时间。这两个线程并发执行。读数速度是随机的。你可以通过一个简单的 `NbTestCase` 来运行这个程序，或者通过日志来轻松检验程序是否如预期一样地运行。

```
public class ParallelTest extends NbTestCase {
    public ParallelTest(String testName) {
        super(testName);
    }

    @Override
    protected Level logLevel() {
        return Level.WARNING;
    }

    public void testMain() throws Exception {
```

```

        Parallel.main(null);
        fail("Ok, just print logged messages");
    }
}

```

执行一次时，输出结果可能如下所示。

```

[global] THREAD: 2nd MSG: cnt: 0
[global] THREAD: 1st MSG: cnt: 0
[global] THREAD: 2nd MSG: cnt: 1
[global] THREAD: 2nd MSG: cnt: 2
[global] THREAD: 2nd MSG: cnt: 3
[global] THREAD: 2nd MSG: cnt: 4
[global] THREAD: 1st MSG: cnt: 1
[global] THREAD: 1st MSG: cnt: 2
[global] THREAD: 2nd MSG: cnt: 5
[global] THREAD: 2nd MSG: cnt: 6
[global] THREAD: 1st MSG: cnt: 3
[global] THREAD: 1st MSG: cnt: 4
[global] THREAD: 2nd MSG: cnt: 7
[global] THREAD: 1st MSG: cnt: 5
[global] THREAD: 2nd MSG: cnt: 8
[global] THREAD: 2nd MSG: cnt: 9
[global] THREAD: 1st MSG: cnt: 6
[global] THREAD: 1st MSG: cnt: 7
[global] THREAD: 1st MSG: cnt: 8
[global] THREAD: 1st MSG: cnt: 9

```

但再执行多次的话，输出结果可能就完全不同了。当程序每次执行的结果与上一次不可能完全相同。这是正常的。当两个线程并发运行时，其结果是无法预测的。现在做个假设，如果线程的执行顺序能够固定，比如说，按照固定的顺序执行线程来引发竞争或者死锁。如果能够找到这个固定的顺序，那么可以重现一个 bug，自然也可修复该 bug。全书一直强调要能重现错误，否则要战胜阿米巴变形虫模型就是一件没完没了的任务。应该写一个测试用例，用来模仿特定的执行顺序，破坏程序运行。比如说，我们让每一个线程递增计数，并打印出来，然后休眠，另外一个线程再运行。在这种情况下，输出结果就是可以预料的，而不是随机的数字。可能有人会怀疑能否写出这样的一个测试用例呢？答案是肯定的！可以留意下面的代码，只需要借助于日志功能，在日志方法中强行让线程等待，就可以控制每次都只是输出一个数。

```

public class ParallelSortedTest extends NbTestCase {
    public ParallelSortedTest(String testName) {
        super(testName);
    }

    @Override
    protected Level logLevel() {
        return Level.WARNING;
    }
}

```



```

public void testMain() throws Exception {
    Logger.global.addHandler(new BlockingHandler());
    Parallel.main(null);
    fail("Ok, just print the logged output");
}

private static final class BlockingHandler extends Handler {

    boolean runSecond;

    public synchronized void publish(LogRecord record) {
        if (!record.getMessage().startsWith("cnt")) {
            return;
        }
        boolean snd = Thread.currentThread().getName().equals("2nd");
        if (runSecond == snd) {
            notify();
            runSecond = !runSecond;
        }
        try {
            wait(500);
        } catch (InterruptedException ex) {
        }
    }

    public void flush() {
    }

    public void close() {
    }
}
}

```

在执行测试用例代码时，会发现每一个线程都会在自己的计数器上加 1，然后再将执行权交还给另外一个线程。那么在任何情况下，这两个线程的操作都是确定的。几乎每一次输出的数字都是按照从小到大的顺序排列。能够达到这种效果，主要是靠拦截输出消息。BlockingHandler 在程序一开始运行的时候就注册到日志系统中，然后对线程发出的日志进行分析。它会轮流挂起和恢复这两个线程，以模拟执行的顺序，这样每个线程在执行时会对计数器加 1，然后被操作系统挂起，从而让另外一个线程从等待中恢复，继续执行。

这个解决方案的优雅之处在于，程序代码完全不知道测试用例控制了它的执行流程。如果只看程序代码本身，完全想不出来测试用例可以随意地控制程序代码及其流程，因为实际的应用代码看上去很自然。只有在调用日志系统输出信息时，才将代码执行流程的控制权交给日志系统控制。这样，测试用例可以模仿任意的流程，并保证代码能按照固定的顺序执行。

当然，要正确地实现 BlockingHandler 也不是一件容易的事，特别是在有两上以上线程交互的情况下。如果传送过来的日志分析起来比较复杂，那就更困难了。所以 NetBeans JUnit 的功能扩展类库中提供了一个 Log.controlFlow 支持方法，它可以将自己注册到日志系统中，然后处理所有线

程操作。而开发人员只需要指定信息的顺序。如果覆盖了 `logLevel` 方法来输出日志，输出的格式为“`THREAD: name MSG: message`”，而这个格式正好是 `controlFlow` 方法可以支持的格式，只需要将这个结果再转给 `controlFlow` 方法，可能无须任何修改。但是，因为发给日志系统的信息可能还包含特定的内容（如 `@af52h442`，标识对象实例的内存地址），最好是使用正则表达式分析一下，得到需要的信息。下面的代码给出了一个示例，演示如何使用 `Log.controlFlow` 方法来模拟“逐步执行”。

```
public void testMain() throws Exception {
    org.netbeans.junit.Log.controlFlow(Logger.global, null,
        "THREAD: 1st MSG: cnt: 0" +
        "THREAD: 2nd MSG: .*0" +
        "THREAD: 1st MSG: ...: 1" +
        "THREAD: 2nd MSG: cnt: 1" +
        "THREAD: 1st MSG: cnt: 2" +
        "THREAD: 2nd MSG: cnt: 2" +
        "THREAD: 1st MSG: cnt: 3" +
        "THREAD: 2nd MSG: cnt: 3" +
        "THREAD: 1st MSG: cnt: 4" +
        "THREAD: 2nd MSG: cnt: 4" +
        "THREAD: 1st MSG: cnt: 5" +
        "THREAD: 2nd MSG: cnt: 5" +
        "THREAD: 1st MSG: cnt: 6" +
        "THREAD: 2nd MSG: cnt: 6" +
        "THREAD: 1st MSG: cnt: 7" +
        "THREAD: 2nd MSG: cnt: 7" +
        "THREAD: 1st MSG: cnt: 8" +
        "THREAD: 2nd MSG: cnt: 8" +
        "THREAD: 1st MSG: cnt: 9" +
        "THREAD: 2nd MSG: cnt: 9",
        500
    );
    Parallel.main(null);
    fail("Okay, just print the logged output");
}
```

代码看起来非常简单，但背后隐藏的内容并不简单。通常来说，代码执行顺序可以只被一段代码控制。对于这种情况，使用 `controlFlow` 方法来控制代码流程的脚本写起来就很简单了。像下面的例子，我们希望 1 号线程先打印 5，然后 2 号线程打印 2，接下来 1 号线程再打印 6。完成这个功能控制的脚本如下。

```
public void testFiveAndThenTwo() throws Exception {
    org.netbeans.junit.Log.controlFlow(Logger.global, null,
        "THREAD: 1st MSG: cnt: 5" +
        "THREAD: 2nd MSG: cnt: 2" +
        "THREAD: 1st MSG: cnt: 6",
        5000
    );
    Parallel.main(null);
    fail("Ok, just print the logged output");
}
```

现在清楚了，在一个程序中使用日志的话，可以将日志作为一个能够避免阿米巴变形虫模型的利器。不仅可以在程序运行时输出日志信息，还可以在测试环境下通过日志来控制流程。通过日志，不仅可以对出现问题的测试进行分析，在必要的情况下，还可以控制程序的执行流程。这样可以有效地模拟程序运行中很难出现的一些极端情况，方便重现和解决问题。

通过优化生成日志的格式和日志文件的回放，可以更好地根据日志文件控制程序。说到日志记录的问题，有一些工具，像 dtrace^①就很有用。比如说，当上下文改变的时候，如代码的执行权从一个线程切换到另外一个线程时，dtrace 就可以进行跟踪处理，功能非常强大。能够监控并回放上下文切换，就意味着能让一个在并发模式下执行顺序不确定的程序变得顺序固定，即使主机只有一个 CPU。还有一种方式是使用 AOP 技术，或者任何其他的二进制增强技术，都可以向代码中强行加入日志代码。这样做，即使原来的代码并没有对外提供相关的监控，也可以获得程序进程中的相关信息。一切皆有可能，但只有时间能最后证明哪种方案才是最合适的。就目前而言，对于 NetBeans，已经可以很成功地利用日志来控制代码的执行流程，从而避免在 API 中因为阿米巴变形虫模型带来的各种副作用。

11.4 循环调用的问题

如果在并发访问一个 API 时出现不一致，实在是一件很倒霉的事情。幸好，修复这个 bug 很简单，只需要在代码中添加 synchronized 关键字即可。当然，这样做会提高死锁的风险，但最麻烦的问题是：即使加了这个同步关键字，也无法保证结果的一致性。开发人员必须准备好接受或防止循环调用。

即使加了同步锁，Lookup 抽象类仍然出现同步问题

Lookup 这个类可以说几乎是 NetBeans 平台的核心所在。随时随地都可能调用这个类来查询各种内容。这个类是我所写的，我在编码的时候，想尽方法在各个可能出现问题的地方都使用了同步锁来保证代码执行的正确性。当时，我希望这样做，可以保证所有的功能都能正确执行，而且执行结果具有一致性。但让我吃惊的是，我的希望落空了，我们还是时不时地会收到一个“神秘”的 bug 报告，报告中所带的栈跟踪信息表明运行时这个类的内部结构被破坏了，这简直是不可能出现的问题。

我们对此不知所措，几周之后，一个同事发现这个问题可能是因为对一些修改方法的循环调用引起的。他把一段调试内容放到代码中，如下所示。

```
private boolean working;
public synchronized int sumBigger(Collection<T> args) {
    assert !working : "Shall not be working yet in order to be consistent";
    working = true;
    try {
```

① DTrace 即动态跟踪 Dynamic Tracing，是 Solaris 10 的新功能，通过此功能，用户能够动态检测操作系统内核和用户进程，以更精确地掌握系统的资源使用状况，提高系统性能，减少支持成本，并进行有效调节。官方网址为 <http://opensolaris.org/os/community/dtrace/>。——译者注

```

        return doCriticalSection(args);
    } finally {
        working = false;
    }
}

```

现在，我们得到的一些出错的断言，而不是过去那种“神秘”的栈信息。太好了。至少现在知道为什么会出错了。doCriticalSection 在特定的条件下会调用外部代码。而这个外部代码则会通过 Lookup.getDefault() 方法取得 Lookup 的全局实例对象，然后再去调用 Lookup 的某一个方法。现在真相大白，一切只是因为循环调用。

假设某个线程安全的方法内部所有的操作都是原子操作，那么只要开发人员在这个方法中调用外部代码，就有可能出现循环调用的问题。因为这段外部代码可能会再去调用这个所谓的线程安全方法，而传入的参数可能完全不同，此时原先的原子假设条件就几乎被破坏了，因为这个方法中对变量的处理不再是原子级别的了。针对该问题，有两种可能的解决方案。一种是使用锁来避免循环调用，还有一种方案则是允许循环调用，但要对其进行处理。

循环调用很常见

想重现这种循环调用引发的问题，其实不难。因为即使正确地同步执行代码，也很容易忽略循环调用问题。

像 Listener 这种机制，经常会因为事件源和 Listener 之间的通信而产生循环调用的操作。比如说你写了一个 Listener 以监听某些事件，然后调用某个 JavaBean 的 setter 方法。但有谁知道这样一个操作，会连锁引发多少个类似的事件？虽然连锁事件并不是其首要目标，但却是直接结果。

即使想方设法定义了一个合理的线程策略，就像 Swing 一样，也同样不能避免循环调用的问题。在 Swing 中，不可能在调用代码绘制界面的时候，被另外一个调用所打断。如果你不听从前文给出的建议，则绘图缓冲区会因为循环调用而完全混乱，屏幕上的绘图将不正确。

下面代码中使用了 NonReentrantLock 进行加锁，但会引发两个问题。

```

private Lock lock = new NonReentrantLock();
public int sumBigger(Collection<T> args) {
    lock.lock();
    try {
        return doCriticalSection(args);
    } finally {
        lock.unlock();
    }
}

```

首先，如果使用了这种锁，就不允许方法循环调用，这是一种不兼容原来代码的行为。在采用该方案以前，用户的代码在调用 API 时是可以循环调用的，并且利用这种方式来完成某些功能（但可能导致意外）。但采用了该方案以后，这种调用就被强行禁止了，因为在循环调用方法时，

对该锁的检查就会抛出一个异常，API 与以前的代码不兼容。当然，保证 API 的正确性要比 100% 的向后兼容更为重要，所以少许的不兼容性也是可以接受的。但还存在另外一个更严重的问题。如果不允许方法的循环调用，那么像 Swing 绘制复杂界面这种问题就无法解决。因为这种锁是根据代码执行时的栈信息来进行检查的。

```
@Test
public void testCriticalSectionWith15() {
    final CriticalSection<Integer> cs = create();
    testFor15(cs);
}
// 如果当前的操作被关键代码所调用，而且发现相应的锁不是可重入锁，
// 则会抛出一个异常
public void run() {
    testFor15(cs);
}
```

直接调用 testFor15 这个方法不存在任何问题，但如果调用时当前代码已经持有了一个锁，调用就会失败。如果这两段代码在一个类中，这个问题还是很好解决的。但如果这个锁对外开放，哪怕是间接的，那么在调用一些代码的时候，就可能导致代码执行时因为对锁进行检查而抛出异常。

综上所述，最好还是允许循环调用，允许关键的方法支持它，就好像支持多线程一样。支持循环调用的技巧在于要找出那些能够使用同步锁，但不应该调用外部代码的方法。一开始，要将类内部的数据备份，保证其不可被更改，然后再执行代码，处理相应的结果，等一切操作中结束后，再在全局状态下将当时类内部的数据与原先备份的数据进行合并处理。如果使用的数据是整数，那么合并起来就非常简单，可以参见下面的代码。

```
private AtomicInteger cnt = new AtomicInteger();
public int sumBigger(Collection<T> args) {
    T pilotCopy = this.pilot;
    int own = doCriticalSection(args, pilotCopy);
    // now merge with global state
    cnt.addAndGet(own);
    return own;
}
```

如果内部数据不是数字这种简单类型，而是更复杂的对象，处理起来就麻烦一些，会不停地重复此操作，直至循环函数被中断，或者外部有并发调用。

```
public int sumBigger(Collection<T> args) {
    T pilotCopy = this.pilot;
    for (;;) {
        int previous = cnt.get();
        int own = doCriticalSection(args, pilotCopy);
        // 如果没有出现并入或者可重入性没有产生变化，那么就直接返回，
        // 否则就再尝试一次
        if (cnt.compareAndSet(previous, own + previous)) {
            return own;
        }
    }
}
```

但这种代码可能永远都不会结束，因为代码中存在着一个死循环。所以需要在数据结构中添加额外的属性信息来保证，每一次循环调用时内部结构都进入一种“更稳定”状态。而后代码重复指定操作，直至最后“稳定”下来，即最后出现对 `doCriticalSection(args, internalParam)` 的一次调用不再是循环调用。内部结构表示了对象的状态，但应该是不变的，如果有操作需要改变结构的状态，就会把内部数据又复制一份，然后只是对这份副本操作。只有这种方式，且数据结构在每次计算后保持“稳定”，才能确信代码可用于循环访问。

11.5 内存管理

因为 Java 语言使用了垃圾收集器，所以它的内存管理与传统的编程语言完全不同。与 C 和 C++ 不同，Java 中只需要一个构造函数，无须析构函数^①。这样，开发人员就不需要像 C 语言写一个 `XCreateWindow` 和 `XDestroyWindow` 函数来创建和销毁 X Window，也不用像 C++ 语言一样写析构函数。但这不表示有垃圾收集器就可以完全忽略 API 的内存管理了。尽管很多人都希望垃圾收集器可以避免开发人员手动释放内存，但这种情况只适用那种最简单的场景。对于所有其他场景，特别是如果要设计一个通用类库时，还必须关注内存管理。

有些编程语言使用经典的内存管理模式，开发人员需要编写析构函数或者销毁函数来释放内存，因此必须尽可能多地关注运行时的情况，包括数据、对象相互间的组成和引用关系。API 设计人员需要保证正确运行，不会通过额外的释放或者析构函数来管理内存。

先从一个最简单的例子开始。看一下 `java.lang.Math` 这个类，它不能被实例化，表面看来也就不需要释放任何资源。但这个判断并不是绝对正确的。尽管它不能被实例化，但这个类自身却不可能被虚拟机的垃圾收集器所回收。这种情况不多见，而且即使回收了，也看得出来能有什么好处。

在访问 static 字段的时候出现 `NullPointerException`

1997 年的时候，我们碰到了一件与回收 Class 有关的趣事。那时候，JDK 1.1 刚刚发布，与老版本不同之处在于，这个版本能够在某个类不再有用的时候将其卸载。当时 NetBeans 有一个核心 API 类 `TopManager`，它有一个 static 字段，还有一个 static 的 getter public 方法 `TopManager.getDefault()`，另外还有一个 setter 方法。在这个类初始化的时候，NetBeans 会通过这个 setter 方法把这个类的默认实现注入到 `TopManager` 类的字段上。随后，其他的代码就可以通过 getter 方法访问 NetBeans 的所有功能。

让我们不解的是，这个 getter 方法时不时地会返回一个 `null`。我们总觉得这是一个不可能出现的问题。我们在调用 setter 方法的时候，绝对将一个非 `null` 的对象设置给了 `TopManager`，但这个问题仍然会时不时地出现。但我们去调试这个程序的时候，一切都很正常，没有出现任何问题。但正式运行的时候，还是时不时地会抛出一个 `NullPointerException`。到底是为什么呢？我们花了好几个小时才发现这是垃圾收集器惹的祸！这个类会不时地被虚拟机从内存中卸

① 析构函数(destructor)与构造函数相反，析构函数往往用来做“清理善后”的工作（例如 C++ 在建立对象时开辟了一片内存空间，应在用完该对象后，通过 `delete` 指令来释放内存，这条指令就会自动调用相应的析构函数）。

载。随后如果需要使用这个类，虚拟机会再加载这个类。但这次加载时，相应的字段就没有被初始化。这正是出现 `NullPointerException` 的原因。

我们找到了出现该问题的原因，但是解决起来却不那么容易。不管怎样，我们需要想办法，阻止虚拟机卸载这个类。最后让另外一个类持有 `TopManager` 的实例。但这种解决方案不直观，随后也证明了这个方案确实存在 bug。大概一两周后，Sun 公司又发布了一个虚拟机的更新版本，在版本中定义：如果某一个类声明了 `static` 的字段，那么只要这个类的 `ClassLoader` 还在内存中，这个类就不会被卸载。从那以后，虚拟机一直遵守这项准则。如果你希望卸载某一个类，那么只有回收该类的 `ClassLoader` 加载的所有其他类以及这个 `ClassLoader` 本身才能达到目标。

有时候确实需要卸载一个库的所有类，比如说当 `NetBeans` 的某一个模块被禁用了或者干脆被删除了。因为每一个类的内部都会引用 `ClassLoader`，一般来说，这就意味着要移除 `ClassLoader`。当一个 `ClassLoader` 从内存卸载后，由这个 `ClassLoader` 加载的类及类对象实例都会被卸载。当然，这只是一种简单的情况。

还有一种简单的场景，就是类的实例对象不会引用外部的对象。比如说，`java.awt.Dimension` 和 `java.awt.Rectangle` 这两个类只包含了一些整数字段，没有引用任何外部对象。这正是垃圾回收器的用武之地。API 用户创建这些实例，然后引用它们，等不再需要时，也就忘了引用。只要没人引用这些对象，垃圾回收器就可以将其移走。根本没有必要去调用析构函数，所以也就不需要析构函数。以上分析说明，代码中不可能访问已经被销毁的对象。这样做使得内存管理大大简化了，这正是垃圾回收机制取得成功的根本所在。

但上面说的都是最简单的场景，在真正设计那些含有复杂数据结构的 API 时，情况比这复杂多了。有时候数据结构可能相对很简单，但其背后却可能有着复杂的资源处理，如 `OutputStream` 文件描述符，或者是 AWT 控件的资源句柄。对于这种情况，就不能依赖垃圾收集器来释放这类资源，因为有可能已经太晚了，而如果你对有这些对象的引用，则完全不可能释放。如果出现这种情况，就需要添加一种类似于析构函数的机制。比如说处理流时，可以添加一个 `close()` 方法，可以释放文件资源。AWT 窗口就有一个 `dispose` 方法来释放 `Windows` 的资源句柄。这两个方法都很像析构函数，它们没有释放内存中的对象，但是却强制将内部的非内存资源给释放了。拿数据流打个比方，在 `close` 方法调用后，该对象就无法对外提供功能了，或者它可以暂时先释放数据资源，然后在需要的时候再重新加载。但即使是这样，配合使用垃圾回收器也是不错的。如果一个流或者一个窗口已经用不到了，需要从内存中移走，那么做一些清理工作也是正常的。出于这方面的考虑，当对象从内存中移走的时候，需要对外发出通知。有两个途径发出通知：使用 `finalize` 方法或者是配合 `ReferenceQueue` 使用 `WeakReference`。相对而言，`finalize` 方法差一些。首先，在 API 中 `finalize` 是可见的。其次，从逻辑角度上来说，这也是不合理的。想一下，如果一个对象都没有必要存在了，还调用这个方法做什么呢？如果调用了这个方法，使得这个对象又可以被其他对象访问了，这时还回收它吗？而且，`finalize` 是在一个专用的线程中来调用的，调用的时间也不固定。另外，也不能保证真的能收到释放该对象的信息。所以考虑到上述原因，使用 `ReferenceQueue` 会是一个比较好的选择。所有需要清除的对象都会存放在这个队列中。一旦某个对象可以被回收了，就会把引用放到相应的队列中。开发人员可以去检查这个队列中的

内容, 并进行一些清除工作。

这些例子相对比较简单, 因为为了方便读者理解, 相关 API 已经是简化过了。API 用户在内存里保存了对象, 而且 API 并没有引用任何外部对象。如果说 API 中也持有对外部对象的引用, 那么内存管理会更加复杂。很多 API, 特别是那些模仿 JavaBeans 监听机制来设计的 API, 可能会持有外部实现的 Listener 对象。这样就允许 API 客户将对自己对象的引用传入 API 内部, 被 API 持有。比如说 API 用户将一个 Listener 添加到相应的 JavaBeans 中, 这个 JavaBeans 对象就持有了对自己对象的引用。如果出现这种情况, 就意味着这个 API 也要负责管理所引用外部对象的生命周期了。如果这些 Listener 对象实例被一个单例的对象所持有, 因为这个单例永远都不会从内存中释放, 于是所有的 Listener 对象实例也就不会从内存中释放。这样就造成了内存泄漏。如果某个 Listener 的环境有所改变, 也可能会导致代码的行为不一致 (可以参考前面所说的阿米巴变形虫模型), 于是原来可以正常运行的功能就不再正常运行了。

对于 Listener 对象应该使用弱引用机制

无论何时谈到 JavaBeans 设计, 基本都会讨论 Listener 机制, 常得出的结论是, 如果在 JavaBeans 的规范中声明持有 Listener 对象时应该使用弱引用机制, 那么事情会好很多。这样做就可以解决前面说到的引用问题, 同时也算是给开发人员提供了一个好的范例。对于任何使用 Listener 机制或者其他回调接口的设计, 最好在持有相关对象时使用弱引用机制。

但 JavaBeans 的规范中却对内存管理只字未提。其实这也很正常, 因为这个规范成文是在 Java 1.1 时代, 至今也未有大的改动, 而弱引用却是在此后的版本中引入的。如果重新定义规范可能会引发不兼容问题, 或者即使改了用处也不大, 因为毕竟弱引用只是建议, 并不强制。所以到现在为止, JavaBeans 还是没有使用弱引用。

NetBeans 项目也经常出现类似的问题。为了解决这种问题, 我们写了一个 WeakListeners 类, 用法如下。

```
public static <T extends EventListener> T create(
    Class<T> type, T listener, Object source
) {
    return org.openide.util.WeakListeners.create(
        type, listener, source
    );
}
```

通过 create 方法可以将任何一个 Listener 包装成一个弱引用的对象, 而只有这个弱引用对象才去引用原始的 Listener。这样做的效果其实和在 JavaBeans 中使用弱引用差不多。在处理某些情况时, 这个方法非常有用。但它却不能解决所有的问题。开发人员还是得好好理解对象实例的生命周期, 还有垃圾收集器如何来处理这些对象实例。

在传统的项目文档中, 最少提及的内容应该算是程序的内存模型了。这也很正常, 因为目前来说, 还没有比较通用的方法可以用来设计程序的内存模型。但这件事却不应该被忽略, 因为开发人员如果不能有效地管理内存, 那么他们写出的程序没有几个能够长期稳定运行。

传统的做法都是先写程序, 然后再使用性能分析工具来检查可能出现内存泄漏的地方。如果

找到了，再去修改代码以修复这个 bug，不断重复如上的工作，直到感觉程序已经稳定为止。其实前面已经说过很多次了，如果程序会有多个版本的话，这种工作方式效率比较低。随着时间的推移，利用性能分析工具得到的所有提高将会倒退，除非能保证测试工作持续开展下去。

像 JUnit 这种单元测试框架在这方面其实也帮不上什么忙，所以我们在 NbTestCase 框架中添加了很多扩展功能，希望可以帮助分析内存使用情况。我们把这个框架称为 Insane，大家可以通过访问 <http://performance.netbeans.org/insane> 来获得该框架的更多信息。

在 Java 这种现代面向对象语言中，首要处理的内存管理问题就是内存泄漏。内存泄漏问题不是说内存中有哪部分地址被遗忘了，因为有垃圾收集器的存在，不会出现此类情况。也正是因为有了垃圾收集器，本想释放的对象仍留在内存中，因为有人在别的地方继续引用它们。如果有一个特定的操作在被执行后，总会留下一些垃圾对象，过不了多久，也许就会发现系统的空余内存越来越少，而整个程序的运行速度也越来越慢。NbTestCase 类中就可以通过 `assertGC` 方法对此类情况加以断言。

```
WeakReference<Object> ref = new WeakReference<Object>(listener);
listener = null;
NbTestCase.assertGC("This listener can disappear", ref);
```

如果你觉得在执行完某些操作后，在内存中，某个对象实例就已经用不上了，你可以创建一个 `WeakReference` 对象，然后移除对该对象的引用，然后再调用 `assertGC` 方法来尝试将这个对象从内存中释放。在 `assertGC` 方法中，它会尝试强行调用垃圾收集器来回收该对象。它会多次调用 `System.gc` 方法，以释放内存，它还会调用 `finalize` 方法，如果发现 `WeakReference` 中的对象已经被释放了，就会返回对象释放成功的信息。否则它会调用 `Insane` 类库来查找是谁持有了这个对象，使其无法从内存中释放。下面的代码就会造成此类问题。

```
Of course, this listener cannot disappear, because it is held from long living
JavaBean:
private static javax.swing.JPanel
org.apidesign.gc.WeakListenersTest.longLivingBean->
javax.swing.JPanel@779b3e-changeSupport->
java.beans.PropertyChangeSupport@1e91259-listeners->
sun.awt.EventListenerAggregate@a553e2-listenerList->
[Ljava.beans.PropertyChangeListener;@16bc75c-[0]->
org.apidesign.gc.WeakListenersTest$PropL@11bed71
```

从上文可以看到有一个 `longLivingBean` 字段，它指向 `PropertyChangeSupport` 对象，它支持内部 Java 实现，持有你想要从内存中回收的 `Listener`。

影响性能的原因还有一个，就是数据结构的大小。如果有可能将某对象的成千上万个实例同时放置在内存中，那么自然希望这个对象实例占用的空间很小，以免出现内存问题。开发人员想尽量减少对象的大小。可以通过一些性能分析工具来检查这种问题，或者提前解决，但还是会出现问题：你希望能够保证对对象大小的控制。NbTestCase 提供了一个 `assertSize` 方法来检查对象实例的大小。

```
private static final class Data {  
    int data;  
}  
Data d = new Data();  
NbTestCase.assertSize(  
    "The size of the data instance is higher than 16", 16, d  
);
```

assertSize 使用了 Insane 类库来遍历作为一个参数传入的根对象持有的所有子对象，并计算占用的内存数量。然后再将计算结果与期望值进行比较。如果计算结果小于或者等于期望值，就算验证通过。否则就认为失败，并把指定对象的子对象大小全部打印出来，方便对失败原因进行分析。

一个简单的 java.lang.Object 对象实例只会占用 8 字节。如果向一个字段中添加一个整数或者对另一个对象的引用，占用的内存就变成了 16 字节。再添加第二个字段，但内存占用还是 16 字节。如果添加第三个字段，内存占用就变成了 24 个字节。从以上现象看出，内存占用可以根据字段的多少进行偶数进位取整而计算出来。事实上这样的结算结果只用于一些特殊场景下。因为计算值只是理论值，具体占用的内存大小则会根据不同的 Java 虚拟机有所不同。但即使是这样，计算出来的理论值也足以描述程序员开发时的意图（这和实际的虚拟机架构没有关系）。

对程序的阿米巴变形虫状态进行分析时，assertGC 和 assertSize 这两个方法都非常有用。通过在测试用例中使用这两个断言，我们可以准确地定义和验证程序的内存模型。所以这些测试用例也变成了 API 功能规范的一部分。不仅如此，我们在对程序进行自动化测试的时候，都会使用这两个断言对其内存模型的有效性进行验证。



谈 到如何减少程序运行时出的问题时，有一个有趣的方案，那就是完全屏蔽运行时的存在。我一时也没有为其找到合适的名称，就暂且称之为**声明式编程**（declarative programming），当然，这个名称已经在不同场合广泛使用，并被赋予不同的含义，对于不同的人，其意义也不尽相同。声明式编程的基本思路，不是让 API 用户一步步告诉程序如何做，而只需要告诉程序他们要的结果，然后就交给 API 去完成。

声明式编程果真存在？

图灵机理论是现代计算机的基础，当我开始学习该理论的时候，我很惊奇地发现，虽然程序和数据可能有所不同，但如果从一些特定角度来看，是没有任何区别的，所以程序和数据是可以合而为一的。其实对这样一个认识也不必太过吃惊。程序源代码是一类数据，可以由其他程序打印出来，调试器可以对编译后的代码进行处理代码编译成可执行的内容。从相反的角度看，数据也可以处理程序。编译器和分析器只收到原始的文本数据，然后进行处理。比如说某一部分原始数据可以指定计算机进行什么操作，那么程序运行时不也是这样吗！从这个角度上讲，入侵计算机的一个基本技巧就是发送一个“特殊的”数据包，然后会产生一个特定的计算，进而入侵并控制计算机系统。

就这段文字而言，程序和数据这两者之间并没有本质的区别。它们都会触发指定的操作，总有一个第三方，它会启动和读入这两者，最后处理。有什么区别呢？即使有区别，也只是看问题的角度不同而已。有些内容看起来更像数据，还有一些看起来像是可以执行的程序。但很明显，依据这个答案，你无法给这两者定义严格的界线，两者之间的区别是模糊的。

我感觉所谓的声明式编程其实也存在这样模糊的定义。不管我们是用 Java 还是用 XML 写代码，也不管我们是调用方法还是使用 Annotation，都只是原始数据。把数据变成程序就是一个执行过程。所有的执行过程也都差不多：总要有个处理部件（可能是 CPU、编译器、Annotation 处理器或者是 XML 解析器），这个单元会去读这些数据，然后把数据变成一系列的行为。从这个意义上讲，即使是编译后的二进制代码也可以看成是声明式编程。

那么代码编程和声明式编程又有什么本质上的区别呢？

声明式编程的长处在于，它能够从比较高的层次来定义一些操作，然后由其他人执行相应的

操作,与普通编程相比,后者需要从细节上定义每一处操作。在7.2节中,Lookup的注册机制定义了高层次的概念。这样即使开发人员不调用registerXYZ方法也能向一些API中注册一个工厂,他们只需要通过META-INF/services/...XYZ文件就可以声明注册内容,然后由具体的服务管理器来发现该服务。

虽然Lookup这个例子很简单,但它确实可以避免调用registerXYZ方法和unregisterXYZ方法。不仅如此,这样做还可以有效地保证注册与注销的一致性,如果采用常见的API调用方式,就很难保持一致性。因为很多API的用户往往是经验主义者,他们做事的方式都是“先试一下,如果可行,就这样做了”,他们很容易找到一个方法去注册一个功能,但往往会忘记调用一个方法去注销一个功能。毫无疑问,这样做是有问题的,因为在一个模块化的系统中,往往有一个动态子系统在运行时负责模块的启动、禁用以及安装,如果忘记注销,会引起一些资源无法回收。声明式编程是解决该问题的一剂良药:开发人员只需要声明要注册什么,相应的注销和清理都会由系统自动完成。

安装及清除模块配置文件

从一开始,NetBeans平台就是通过读取模块中的配置文件来处理模块间的协作关系。比如说,开发人员可以通过文件注册的方式向工具栏中添加操作项,模块可以控制主工具条上哪些操作项可见。在早期,NetBeans要求每一个模块在初始化安装过程中都必须创建自己的配置文件。几乎所有的模块都会注册自己的钩子。一旦NetBeans调用了这个钩子,模块就会通过类似`new FileOutputStream(new File(root, "Toolbars/my.button"))`的代码调用创建自己的配置文件。然而,这种方式既容易出错,又含有令人厌烦的写文件操作,而且在模块卸载的时候还可能有问题。开发人员往往也不会用心去正确编写对应的清理代码。

我们觉得这个问题比较严重,同时这样做也加大了API使用的复杂度。它使得在NetBeans平台上开发应用程序成为一件比较痛苦的事。我花费了大量的时间和精力来解决该问题。在3.2节中,我发明了一种“merging virtual XML filesystems”方案。引入该方案以后,API倾向于那种声明式编程的风格了。每一个模块只需要通过一个XML文件来定义其内部文件的布局方式,其他的工作都留给NetBeans来自动完成。

这样做立刻就解决了卸载时的资源清除问题。与以往要保证几百个模块都正确实现注册和注销相比,现在整个NetBeans正常工作起来要简单得多。同时,它还提供了更有意思的改进。事实上,在运行时,我们根本不需要在磁盘上创建新文件,只要把这些内容通过虚拟的方式进行合并处理,然后再保留在内存中就足够了。同时,采用这种方式还能与原来的方式兼容,因为合并XML文件的处理方式和本地磁盘处理方式差不多,所以那些在安装时要解压文件的老模块,也同样可以正常运行。

最后要说的一点也同样非常重要。当NetBeans的子模块数量超过500个的时候,我们发现解析XML文件会占用过多的内存,而且在启动时对XML文件进行解析也会严重降低启动速度。所以我们将最后一级解析并合并的结果缓存起来。如果不是采用声明式编程方式,这种优化根本做不到。

一致性并不是声明式编程带来的唯一好处,有效性是另一个优点。在Java中,运行时分配

的内存大小并不与代码规模成比例。但编写那些只有一个方法的小类时，并不具备有效性。因为虚拟机至少要为每一个类准备 3K 字节以上的内存空间，也就是说每加载一个类，至少要占用 3K 内存空间。而且，即便可能，强制回收已加载的类也是非常困难的。因此，一旦类被加载，可以大致认为类会永久驻留在虚拟机中，占用宝贵的内存资源。

同样，即使是从一个 Jar 包中加载一个类文件，付出的代价也是高昂的。需要先为这个 Jar 包创建一个 ClassLoader^①，然后将一系列的文件加载到内存，所有的操作都会付出一定的代价。基于这些原因，可以假设一个程序在启动的时候，需要从每个 Jar 文件中分别加载一个类，然后让这个类执行某些注册，那么启动性能会降低很多。所以最好是使用一个注册声明文件^②，如 Java 中的 META-INF/MANIFEST.MF 文件，然后由系统加载和处理该文件，所谓的声明文件并不神秘，只是包含具体注册信息的普通文件。这样，启动时就不需要打开整个的 Jar 文件，并从 Jar 文件中加载一个类到虚拟机中了。好处还不止这一点，这样做可以保证无用的资源能够从内存中被释放，然后在需要的时候再加以创建。

NetBeans 还可以在第一次执行的时候，将相关的信息处理完后进行缓存，在后续执行时，直接使用缓存的内容即可。因为大部分声明式编程都没有什么副作用，所以进行这种优化是完全可行的。处理后的信息不会因为时间而改变，也不会受到磁盘中文件及已用空间的影响。所有这些只会影响用 Java 或其他可执行格式编写的真实代码。声明式编程的效果很好，这主要是因为其抽象性。在普通编程语言中，是通过编写和执行方法来完成具体功能，如果发现程序的性能很差，那么随后也很难进行优化。在一个方案中，如果声明的内容越多，那么其指定的操作就越少，这样就有效地增多了前置条件，从而保证后续可以进行更完善的性能优化。也许这才是声明式编程和传统编程最重要的区别所在。对于开发人员来说，如果要设计一个高层次的声明型 API 供外部使用，这一点也是最重要的理由。

12.1 让对象不可变

11.3.3 节介绍了出现死锁的四个必要条件。虽然可以把这四个条件看做等价条件，但我仍然觉得对象互斥才是避免死锁的最重要条件（即一个对象只能被一个线程所访问）。为什么这样说呢？因为这个死锁条件才是涉及了计算机和编程的核心问题。先问自己一个问题：为什么一个资源只应该被一个线程所访问呢？极端一点来说，原因只有一个，那就是：这个线程会销毁这个对象。

当然，这是一个推论，也可以认为资源并不一定是被销毁。从某种程度上讲，资源潜在的未来数据状态已经固定，不能再为第三方提供有效的状态数据。想一下，就好像有一个线程使用打印机打印了一张纸以后，这张纸上面就是相应的文字和像素。此时，这张纸就不再是一张白纸了，对于其他人来说，这张纸已经“用过”了，不可能再用一次。如果面对的是一个空白的纸，那么

^① 这个说法不是特别准确，也有可能是复用现有的 ClassLoader，特别是 JDK6 中的 instrumentation 功能更强。

——译者注

^② MANIFEST.MF 是很多程序使用的声明文件，Java 是在 jar 中的 META-INF 目录下放置一个 MANIFEST.MF 文件来声明的。——译者注

还有文章可以做。因此可以说这个打印机在打印时就已经销毁了一张纸。同样，一张 DVD 上刻录了数据以后，这张 DVD 就可以说是毁了。当然，DVD 上现在可能存储了一部精彩电影，对于 DVD 来说，这也是很不错的结局了。但在记录之前，这个 DVD 可以用来备份一本书或者把我收藏的音乐刻上去。但刻录过后就不能拿作它用了。还有类似的情况，如硬盘，如果你向硬盘上写入了新的数据，那么也可说是毁了这个硬盘。说得更加极端一些，在 Java 中，一旦改变了某一个对象某个字段的内容，也可以说这个对象不能为他人所用了。在赋值操作前，可以去读取该字段的值（如果它是整数的话），然后进行各种运算。但当字段赋值以后，原来的值就永远没有了，即使还有别的代码对这个对象有兴趣也已经太晚了，这个对象都已经算是被毁掉了，因为它被修改了，不是原来的版本了。

也许有人会这样说：“我很难过，在我修改字段时对象就被毁了。不过，这也很正常。这个世界上天天都在上演类似的事情。”的确如此，昨天我的窗户上还有一整块玻璃，但今天它碎了（就像我一开始写这本书的时候，全身是劲，但现在却是全身无力。好像我以前很年轻，很帅，但现在已经变老变丑了）。生活中的事情总是变化莫测。我们已经习惯并适应了这些变化。但想了解这个时刻变化的世界并不容易，不是说拍张照片来研究一下就可以了，何况这个世界在我们面前日新月异，每天都带给我们新的惊喜。也许有一天，我们以为自己已经了解了一切，但第二天有些事物就发生了改变，原先的理解已经差之千里了。今天的真理，可能明天就变成谬论了。

所以，我们能够理解世界万物吗？我们的确可以做到，只需要简化一下模型。从一定角度来看，几何立方体和球体都是通过对现实世界中的实物加以抽象分析得到的。同样，我们也可以只把注意力放在当前世界的一个特定状态上，在这个状态上，至少现存的事物不会产生变化，这样可以提高我们的认识。然后，我们再找出新生的事物。事实上，几个世纪以来，数学家们都在采用该方法进行研究。数学模型，特别是几何模型，根本就不曾有过改变，可以认为是永世不变的。我们从一个小的角度去观察和认识它们，尝试去理解它们之间的关系。我们掌握的越多，视野就会越发地开阔，也能认识到更多的事物。已经被发现的事物会保持原样不再变化。即使你不再专注该事物，还会有别人在观察。或许几百年后还会有人阅读你的论证或者你的构造几何。他们发现的事物与你发现的事物并无不同，他们的视野也与你相同。

数学的基础知识建立在不可修改的事物上。我们知道如何学习它们，并将它们理论化。还有一些方法可以帮助我们检验我们的理解是否正确。如果把这种思路也应用到计算机程序上，是否也能获取相应的益处呢？假设把程序看作证据，我们就能验证它们是否在所有的环境下都正确，不只是在知识有限的今天正确，如果明天加了一个新的同步锁，破坏了今天所做的前置条件，它们也应是正确的。这样的程序世界就是一个处处是真理的美好世界！

你听到这话，也许会觉得这是一个遥不可及的梦想。事实上，业内人士都知道这样的程序设计语言已经存在了。在 Haskell 这种面向函数编程语言中，就不可以修改已有的对象。开发人员能做的只是通过自己的代码来访问该对象，并利用你所知的现有对象的信息，来描述那些新创建的以及未知的对象。整个过程会描述新创建的对象，最终得到这些数据。当然，对于这种设计，运行时所创建对象的数量也可能会超出计算机的容量。但现在有了垃圾收集器，它可以检查内存对象是否有存在的必要。如果无须存在，就会移除它们以释放内存空间，然后继续创建对象。

在应用这个方案之后，可以完全避免死锁，因为找不到锁住对象的理由。对象只在一处创建，所以一个对象的数据只能被访问，却永远不可能被更改。如果只需要读取对象的状态，那么的确不需要锁住它。可以同时由多方读取，所以也就不需要限制一个资源只能由一个进程或者线程访问。这样就会破坏死锁的第一个条件，所以永远都不会出现死锁。

所以，如果想避免死锁，保证程序的正确性，同时还希望让自己的程序设计得比较优雅一些，那么可以考虑将数据结构定义成不可变的类型。但这事说来容易，做起来却很难。对于 Haskell 或者其他现代函数语言来说，这种编程方式比较容易，也比较自然。但如果想在 Java 语言中使用这种方案则不然，因为在 Java 中，只有 `java.lang.String`、`java.lang.Integer` 和 `java.lang.Class` 这几种类型算是不可变类型。事实上，Java 就是一种基于可变对象的编程语言，相比之下，像 *Concurrent Clean* (Haskell 的兄弟语言)，它为 GUI 程序的开发提供了不可变的 API。尽管如此，对于专用的 API，还是可以在内部使用一些不可变的特性。

我对优雅是否有一种狂热呢？

在我大学毕业时，我的硕士论文主题是关于对 *Concurrent Clean* 这种语言的面向对象扩展有关，我当时也在犹豫是将自己业余的主要精力花在函数语言上（这是我的最大爱好）还是说使用 Java 来开发 NetBeans IDE。随后，我认为函数语言将是未来编程语言的方向，当然我现在仍持此观点。在目前的各种解决方案中，只有函数语言能够充分地利用多核或者多 CPU 机器的能力，它不需要复杂的并行程序设计。因为编译器能够有效地生成并行代码。函数语言的时代终将到来，我不确定的只是它具体的到来时间。

因为这种时间上的不确定性，才让我决定使用 Java 作为编程语言，并开始设计 NetBeans IDE。大家需要一个 IDE 来支持新的语言的需求很清楚，而且很容易将这样一个项目转成一个商业计划。如果想用面向函数编程语言来做这件事就非常困难，因为未知的问题太多了。这类语言总是有些不同，而且这些语言仍未在学校里被用于教学。这样就使得学习的门槛很高，除非能证明该语言在并行程序的开发中有明显的优势，否则很难估计函数语言何时才会被广泛应用。

但如果这一天真的到来了，代码会变得更加漂亮，也会更加优雅，就像几何学一样稳定，而且已经证明的真理将不会改变。直到那时，我们才会意识到函数语言的时代真的到来了，至少可以让部分 API 成为不可变的。

如果要想让一个 API 不可变，第一件要做的事情就是移除所有的 setter 方法。一个 setter 方法就意味着会将一个对象赋值给一个字段，这样就会破坏该对象原先的状态。现在，应该在创建对象时就把所有的数据全部放进来。这样才是有效的。但这种做法有时候不太现实，所以提供一个 Clone 工厂方法会比较有用，如下面的代码所示。

```
AServerInfo empty = AServerInfo.empty();
AServerInfo name = empty.nameProvider(prov);
AServerInfo urlAndName = name.urlProvider(prov);
info = urlAndName.reset(prov);
```

使用这种编码方式可以克隆给定的对象，克隆出来的对象与源对象相比，只有一个字段有所不同。这也是一种使用 setter 方法的不可变版本。从某种意义上讲，还可以用更简单的方式来写

这段代码，把所有的调用都放在同一行代码中。

```
inf = AServerInfo.empty().nameProvider(p).urlProvider(p).reset(p);
```

与 setter 方法相比，这样做的唯一缺点，就是每次调用会创建出两个新的对象，但这两个对象都只是用来作为克隆模板，然后就再也用不上了。对于实时应用程序来说，这种方式在性能方面不会太好，但 Java 的垃圾收集器已经很好了，所以临时创建的对象也不应该影响其他对象才是。还要说的是，这种编程方式用得越多，虚拟机对此的优化空间也就越大，这方面的内容可以参见下面段落中的内容。

唯一类型和逃逸分析^①

在编程中，关于不可变对象最具争议的就是性能。如果改变一些数据，一定需要通过克隆的方式来创建对象，这个操作够快吗？答案其实是肯定的。我这样说不单是因为有了一个好的垃圾收集器，还因为有一些技巧可以完全避免克隆操作。

在某些情况下，如果开发人员确信没有其他代码在监听某一个对象，那么破坏性地修改一个对象的状态也是正常的。如果一个对象只有一个你监听者，那么编译器就无须创建新对象，只需要修改现有实例即可，这样就无须垃圾回收器进行回收操作，还能保持 API 不可变。

对于唯一类型，Concurrent Clean 语言有一个构造函数，而 Java HotSpot 的虚拟机借助于逃逸分析也可以支持这种方式。比如说 `AServerInfo.empty()` 会创建一个 `AServerInfo` 的对象，但该对象不会被别的代码引用。当代码调用 `nameProvider` 方法的时候，它会清楚地知道，该对象现在只有一个引用，所以在这个方法里面可以随意地修改这个对象中的数据。其他类似方法也是如此。

达到完全的不可变是不可能的，但对于一般的可变对象而言，使用不可变数据结构存储其内部状态是大有裨益的，当其被应用于多线程访问或者循环调用时，同步的处理也要简单很多。在实现 NetBeans 代码中的 `Lookup` 类时，也使用了这种优化，通过复杂的内部结构来提高计算速度。开始的时候，我在计算过程上加了锁，但这个过程会调用外部的代码，于是出现了死锁。下面的代码展示了该问题的最佳解决方案。

```
public int sumBigger(Collection<T> args) {
    for (;;) {
        ImmutableData<T> previous;
        synchronized (this) {
            previous = this.data;
        }
        int[] ret = { 0 };
        ImmutableData<T> now = doCriticalSection(args, previous, ret);

        synchronized (this) {
            // 如果没有出现并入或者可重入性没有产生变化，那么就直接返回，
            // 否则就再尝试一次。
        }
    }
}
```

^① 在编程语言的编译优化原理中，分析指针动态范围的方法称之为逃逸分析。它跟静态代码分析技术中的指针分析和外形分析类似。通俗一点讲，当一个对象的指针被多个方法或线程引用时，我们称这个指针发生了逃逸。而用来分析这种逃逸现象的方法，就称之为逃逸分析。——译者注


```
        if (previous == this.data) {
            this.data = now;
            return ret[0];
        }
    }
}
```

这样做就只是在更新缓存时向方法内部的代码上加了最重要的同步，允许多线程遍历和检查缓存的内容，这样可以避免出现死锁。

对于不可变性的需求

像 Java 这种面向对象语言可以很好地支持对象的可变性。从某种意义上讲，在 Java 中生成一个不可变对象比将对象及其字段定义为可变要麻烦得多。从另一个角度来讲，不可变性有其适用的场合。Collection API 库对此就有强烈需求。像 HashMap 会要求新加入的对象要正确实现 Object 的 hashCode 方法，而 TreeSet 则要求新加入的对象要正确实现 Comparable.compare 方法。其中一个隐含的前置条件就是不管是 hashCode 方法还是 compare 方法，计算结果都是不会改变的。也就是说，这两者往往都取决于对象的不可变数据。如果整个对象是可变的，那么 hashCode 方法的计算值就可能在这个对象的生命周期中不停地产生变化，也就是说有可能出现这种情况：把一个对象放到 HashMap 中，过段时间后再去查找这个对象，却找不到，而事实上这个对象还是存放在 HashMap 中的。

尽管在 Java 和其他主要的开发语言中，不可变性并非第一要求，但还是尽可能使用为好。尽管看起来，这个方案更加偏向于理论化一些，但的确能符合无绪这一点。开发人员编码时只需要创建新的不可变对象，不用过于关注代码执行时的内容了。这个方案的优点在于：它可以保证开发人员在不深入了解系统的情况下，也能写出高可靠性的程序，说不定程序还能写得更漂亮更优雅一些。

12.2 不可变的行为

除了在 Java 中那种只有 final 字段的对象，不可变性还有其他体现方式。所以，即使在 API 中，也确实需要不时地修改对象数据，那么减少这种变化频率也是有益的。

NetBeans 界面上各种菜单、工具条项的真正操作

基于传统 NetBeans 平台而开发的程序，对于复制、剪切和粘贴都只有一个菜单项。但当焦点处于编辑器、项目资源管理器中，或者处于 UML 绘图工具区域中时，则因其位置不同，菜单项对象的功能也各不相同。尽管这些 UI 部件之间并没有什么关系，但它们都需要控制剪贴板并处理其中内容。

开始的时候，我们采用了一种称为“执行者”的方案来处理这个问题。一旦发现某一个 UI 部件获得了焦点，就会调用其 componentActivated 方法。这样一来，当前 UI 部件可以获得

剪贴、复制、粘贴或者其他动作信息。之后，将当前 UI 部件对应的执行者（回调接口实现）绑定到当前动作。设置完成之后，所有对菜单的操作都会转交给相应的执行者，直到其他 UI 部件被选中并注册新的执行者。

一开始，这个方案看起来是一个比较合理的架构，它甚至在大多数情况下都可以正常运行。但在特定的不常见的场景下，我们碰到了一些无法解决的问题。比如说，在不需要点击操作来切换焦点的窗口管理器上，用户所执行的剪贴或者删除操作往往并不是发生在所期望的目标元素上，而是误操作至其他元素之上。你可以想象一下，如果你删除的结点不是老的备份文件，而是数据库中的内容，你必然是怒火中烧。有时候，这种误操作还会出现在弹出菜单项中！就算这种焦点切换机制是合理的，用户可能也会遇到这样的问题：一些诡异的 UI 部件模块可能会随机“醒来”，在没有获得焦点的情况下，误将自己的“执行者”设置到当前上下文环境中的复制、剪贴或者其他类似动作，导致当前真正被选中的 UI 部件和执行者不匹配。

我们需要修正这个问题。幸运的是，从 JDK 1.3 中，Swing 开始支持 ActionMap 的概念了。每一个 Swing 的组件都拥有一个 ActionMap，而且可以在特定的键盘事件和真正的“执行者”之间建立映射关系。这样看来，ActionMap 是解决我们问题的最佳方案，在稍后我们苦苦思索能够支持向后兼容性的方案时，我们就开始使用它。

现在，每一个顶层的 UI 界面都有一个自己的 ActionMap，并对其有着绝对的控制权。大部分用法就是在创建界面时把正确的回调实现类注册到界面上，然后就不必关心接下来要发生的事情了。整体架构会根据当前选中的内容或者是弹出菜单的信息，正确地将相应的操作绑定到合适的界面上。

这样做，我们就避免了焦点随机切换产生的问题。用户的代码多少可以算是“不可变”对象了，大部分的动态交互行为都由 NetBeans 基础架构来处理了，而 NetBeans 基础架构则是我们团队可以控制和测试的内容，而且可以很细心地编写代码，并验证其正确性。

用户代码的静态性越强，其正确率也就越高，即使是以透明模式来编写的代码也是如此。如果需要动态的变化，最好把这种动态性留给整体架构来实现。对于我们来说，这种动态性就只是一个功能点，我们可以集中精力正确完成该功能。

12.3 文档兼容性

到目前为止，我已经大致讲解了程序签名和程序行为。在 3.2 节中我已经指出：类库的 API 远远不止是代码。比如说，还应该包括类库读写涉及的文件。对于这样的 API 来说，这些内容与类及方法一样，都应该是简单、健壮、正确、易读的，而且在未来也容易加以改进，等等。当然，为达到这个目标采用的技术方式也许有所不同。但它们仍然很重要，特别是在大幅使用声明编程方案时。

对于 Java 的 API 开发来说，“第一个版本绝不完美”像一条诅咒。事实上，对于基于文件的 API，这话也是适用的。当向磁盘上写下第一个文件时，其格式也许会在随后有所更改，以便添加更多功能，但不管是老文件还是新文件，程序都必须能支持曾经有过的格式。最重要的一个方案是在每一个文件中都加上一个版本号或者类似的其他标识信息。

如果有可能对文件格式加以改进,那么接下来要面对的问题就是如何简化:如何设计文件格式,以便在将来可以不时修改。通常有两个思路:根本性改进或者逐步改进。根本性改进是改变文件的版本号,然后完全重新修改内容。这是一种新的格式,也就是说,将文件读入内存的代码也需要换成新的版本,以便统一处理。这种做法不好,是因为需要为读取老文件和新文件分别提供相应的代码。但这样做,可以保证老的格式不受影响,极大地降低了衰退的风险,所以也可以算是一种好的做法。

对于逐步改进的方法,则不需要两份读取文件的代码。因为它可以在读取文件的代码中增加一些额外的 if 语句,只对老的格式有少许的负面影响。这样做的好处就在于读取文件的代码只有一份。但在读取老格式文件的时候需要特别小心,才能保证其正确性。几年前,这样做可能暴露了极大的风险。但现在自动测试已经得到了普遍应用,所以同时正确处理老文件和新文件的可能性还是比较大的,主要看程序员是否够勤奋。

只有在采用逐步改进的方案时,才能认识到极其重要的一点:不仅可以向后兼容,还可以向前兼容。这就是说,如果旧版本的软件了解某个协议老版本的内容,那么它也可以了解协议新版本的部分内容。当然,理解全部内容是不可能的,否则就太不可思议了。但旧软件可以理解文件中与协议老版本相关的内容,而忽略新版本内容。比如说,假设有一个表示一个文本文档的文件,它的第一个版本只支持黑色字体。而在新版本中,它却可以支持其他颜色的字体。如果采用了向前兼容的方式,则旧的软件仍然可以读取这个文件的格式。它会忽略所有的颜色,但最终显示的结果还是有用的,至少可供他人浏览。这种只支持信息浏览的程序也是可行的。只有支持编辑功能的软件才需要关心这方面的内容,并且要在用户修改文档前给出警告信息,通知用户有版本问题。如果一个文档是使用旧的软件编辑的,那么所有的颜色信息都可能丢失。

黑白电视机和彩色电视机

协议向前兼容的奇迹其实早有证明,如黑白电视机就可以收看彩色电视内容。事实上,彩色电视的信号已经全部改变了,但好的黑色电视机仍然可以分析信号并以灰阶的方式正确显示出来,而且新的电视机可以很完美地重现其颜色。虽然这原本不是为向前兼容而设计的,但却做到了向前兼容,的确称得上是一个杰作。据我所知,生产于 1950 和 1960 年间的电视机,在五十多年后的今天,仍然还可以正常播放一些节目!这才是真正的兼容性。设计时的折中之处在于没有使用红、绿、蓝这三原色(即 RGB 颜色模型),而是采用了当时还不著名的 YUV 颜色模型^①,这种关键技术使得黑白电视机可以只从信号中取得 Y 信道内容,而忽略了新版本中添加的 YUV 颜色信息。这种方案的缺点在于它兼容电视信号的时间太长了。我这么说并不表示我不希望人们切换到更先进的 MPEG 信号上。随着视频信号编码能力的增强,我们至少可以比五十多年前多接收四倍的频道。

如果要存放额外的数据,而且还希望旧版本的软件可以读取这些数据,那么最好的存储格式

^① YUV 是欧洲电视系统所采用的一种颜色编码方法(属于 PAL),是 PAL 和 SECAM 模拟彩色电视制式采用的颜色空间。其中的 Y、U、V 几个字母不是英文单词的组合同, Y 代表亮度, UV 代表色差, U 和 V 是构成彩色的两个分量。

——译者注

应该是什么样子呢？就我所知，有两个很好的方案。一个是属性文件，还有一个则是 XML 文件。属性文件是由多个键值对组成的。如果某一个新版本添加了新的键，老版本可以忽略这个新加的键，但仍然可以正常运行。XML 文件也差不多，如果设计得好，它也可以像键值对一样，添加额外的属性信息或者是额外的元素。它可以添加新的子结点树。但老版本的软件可以完全忽略这些新的内容，只关注那些原先就认识的内容。

另外一种二进制格式

2000 年，我和一个朋友闲聊时说起 XML 与现有的其他格式相比，到底优点体现在哪里呢？在长时间的讨论中，我认为这种格式完全没有什么意义，但我那位身为 Systinet 公司创始人的朋友，其观点却与我完全相反。几个小时后，我们觉得应该把 XML 看作一种文本格式。与其他文本格式相比，它有一个突出的优点：支持树型结构。这样，当处理该格式时，可以完全跳过某一个特定的子树，而且还能够理解甚至处理余下的文档。

当时还讨论了另一个话题，就是把 XML 作为一个二进制文件格式，我对这个话题兴趣不大。但我朋友却极有兴趣，特别是当我们都认为 XML 文本格式的优点就在于它的树型结构以后，他的兴趣越发浓厚了。当我后来有机会看到他在 Web 服务方面的最新工作进展以后，我不禁又想起那场讨论。简单对象访问协议（SOAP）以及其他类似的格式并没有充分利用树结构。它们其实只是把 XML 当做另外一种二进制格式而已。

对于文件格式来说，也应该只暴露必需的内容，但这是一个很难达到的目标。类会有相应的访问级别，如 private 和 protected，但对于文件来说，就没有类似的功能来支持这种控制了。所以文件一公开，就表示所有内容都公开了。只要留下机会，肯定就会有人利用它。也许使用“混淆进行加密”的方式是降低这一风险的唯一选择。也就是说，让文件的格式令人反感，不具可读性，然后再提供一个好的类库来访问文件。这样做，希望可以让开发人员使用类库来访问文件，不需要自行读取文件，这样就不会将不必要的内容暴露给外部使用。但编程语言总是比类库要多，开发商想为每种编程语言都提供这类文件读写的功能，实在是一件不太可能完成的任务。所以还是把写到磁盘上的文件内容都对外暴露出来比较好。

优秀的 API，还有一个优点，那就是其自描述性。事实上，属性文件和 XML 文件的格式都代表了这种发展方向。如果修改过用这些格式写的配置文件，特别是，如果所用格式还允许写注释，那么你会知道修改时只需要保留现有文件即可，并不需要额外的文档说明。当然从头写一个文件还是比较困难的，但如果只关心其中一部分内容的话，就很容易正确理解其含义，并进行正确的本地修改。

ANT 脚本的兼容性

最后也要说一下，文档也是有语义的。与本章说到的例子相比，它的改进难度要大得多。以 NetBeans IDE 为例，它使用了 ANT 脚本，几乎每一个 NetBeans IDE 的项目背后都有一个 ANT 脚本。这个 ANT 脚本是由这个 IDE 生成的。但用户可以根据自己的需要在构建的过程中进行调整。IDE 的功能还在不停地改进。那么生成的脚本文件，其格式也会随之改进。但因为

ANT 也支持继承和目标任务的覆盖，所以在脚本上定义的目标任务也是可以持续调用或覆盖的。所以，我们也要密切关注生成脚本的兼容性。直接删除现有目标肯定是不现实的，重命名也不怎么样。如果我们希望改变两个目标任务间的依赖关系，或者改变一下他们定义或者依赖的属性文件，这事就更麻烦了。而且因为 ANT 的构建文件不支持闭包，所以每个目标任务都是公共的，可以被任意调用和覆盖，所以整个场景变得更为复杂。

我们现在又把视角从文件 API 的兼容性移到了面向对象语言中常见的兼容性问题。继承、覆盖等，需要我们面对 ANT 脚本相关的所有这些问题，和用 Java 设计 API 时碰到的问题一样。而且问题所影响的范围还要更大一些，因为使用 ANT 脚本这种 API 的用户是 NetBeans IDE 的所有用户，而不仅仅是只需要通过扩展为 NetBeans 添加新功能的模块开发人员。幸运的是，并不是所有的用户都需要修改项目中的 ANT 脚本。所以到目前为止，报告上来的不兼容性问题还不多。

就 API 和透明的软件集成而言，文件有一定的复杂度。问题的数量则与文档格式中明确声明和隐含的内容数量有关。对于简单的问题，像文档的版本号，简单想一下就能明白。对于复杂的问题，如 ANT 的构建脚本，其复杂度堪比面向对象的 API 设计。但至少从表面上来看，衡量文件设计是否优秀的标准和 API 设计是一样的。与 Java 类和 API 签名相比，设计文件时使用的技术虽然有所不同，但区别不大。



第三部分 日常生活

如果了解事物背后运行的规律，当然是一件好事，如果还能让事物按照规律运作，那就更好了。但了解事物规律并不代表能操作事物工作。学习理论自然重要：至少你现在了解了 API 设计背后隐藏的那些知识，可以判断每日的决策是否正确。但日常操作比理论更加动态，也更加灵活。所以在将理论应用到日常工作和生活中时，要加倍小心，一定要看清楚具体的环境。

理论是科学的产物。任何一门科学都要尽量全面地描述所研究的内容，只有这样，才能被人理解。自古希腊以来，只有一门科学能够将此做到极致——几何学。但几何学之所以精确，是因为它所描述的内容并不是一个真实的世界。它只是讨论几何对象。在纯粹几何学的世界里，所有研究的对象都是静态的，而且可见，所有的内容不仅是独一无二的而且不会移动。事实上，这与真实的世界完全不同，真实的世界里，一切都在不停变化中。当然了，将几何学应用到真实世界中，也有巨大价值。比如说，我可以用几何学知识来算一下我家篱笆的长度。但并不是所有几何学的知识都能很容易地应用到日常生活中。比如说，把一根木头从中间切成两段，再把这两段又分别从中间切成四段，连续切上 100 次。这事要是放到几何学上来讨论，太简单了，因为在几何学中，比例和规模根本就不重要。但实际动手做，就困难多了。

现实生活中充满了变化因素，理论的东西不可能一丝不变地用在生活中。不仅要知道哪些事情可行，哪些事情不可行，还要知道如何运作才能可行。对于 API 设计来说，也是如此。

本书接下来的内容将会把通用的理论知识（在本书第一部分有所描述）以及基于 Java 开发应用程序的基本规则（在本书第二部分有所描述）与真正的开发工作结合在一起。相较于科学世界而言，现实生活并不完美，因为理论往往都是对现实世界进行简化后再抽取出来的内容。每天的生活中，我们都会做出各种妥协，所以不能把那些理论的内容直接放到生活中。必须对其加以调整以适应真实世界。这也正是本部分的主旨。这一部分将列出 API 设计中要注意的内容，指出何时可以使用何种理论以及如何使其生效，这些都是非常实用的理论。在我们开发 NetBeans 项目时，这些理论已经被我们的设计实践所验证了。

在开始下面的内容之前，我们假定读者已经掌握了理论知识并且知道如何用于 Java 程序。但对于现实世界来说，这绝对还不够，接下来我会解释如何解决在创建、发布、修复 bug 以及维护 API 时出现的问题。这些内容很重要，因为我们大部分的开发工作都是在处理这些无趣的任务。理论以及最开始的设计仅仅是冰山一角，真正的工作还在后头。本部分将会对此详细论述。

极端的意见有害无益

本书的第二部分向大家展示了如何编写更优秀的API。如果大家原原本本地套用这些建议的话，那就糟糕透了。这样做既不现实，通常也不会从中受益，建议仅仅是建议而已。不同的API设计问题有不同的解决方案。本书给出的建议只是建立在我个人开发NetBeans项目的经验之上。对于其他的项目，很有可能出发点与我的不同。哪怕只是初始因素略有差别，给出的API设计建议也可能大不相同。尽管如此，任何试图归纳本书第一部分结论有效适用范围的努力，都是缔造你自己判断的很好的起点。

过来人的话不一定总是对的

在2007年的JavaOne大会上，我无意中听到两个人关于Java API设计的一些对话。其中一个人谈到，在一次API设计的项目中，领导的意见与自己相左。另外一个人听了这话以后说：“我认识那个人，他总想让事情按他的思路来进行，就认死了他的解决方案。”而第一个人听了这话就说：“不是那样的，他说服了我，我最后觉得他的方案更好一些。”

他们谈的这件事算是有个不错的结局！但我相信这只是小概率事件。这两个开发者经常会被迫接受一个不给任何解释的解决方案。我当然不喜欢这种情况，所以我在更改方案的同时，还会告诉他们为什么要改变他们原有的方案。

在我看来，如果仅仅因为某人资历比较高或者经验比较丰富，就盲从其意见，是不可取的。如果他们的建议确实有用，那么应该有据可证。我认为这多少有点像工程设计。软件工程也可以看作是一种工程，所以对于我而言，不盲从是正确的。否则软件开发不就变成了“老人治国”：建议的价值由提出者的年龄来决定。我们承认经验和直觉的重要性，但完全据此来决定一切就显得目光太短浅了。我建议，只有建议与当前问题所处环境相匹配时才值得采纳。

不久前，我试图说服负责增强JDK功能的开发人员采用我的方案，因为他们的方案对于最终用户来说并不理想，需要用户创建三个有继承关系的类，但我的方案却只需要一个类。

我竭尽全力说服了他们改变原先的设计。我引用了很多用户编码风格方面的分析报告，坚持认为，对开发人员来说，API越简单越好。他们看起来同意了我的意见，至少没有反驳。如果这件事就这样发展下去，毫无疑问会得出一个结论：只使用一个类的方案更优秀。但让我吃惊的是，他们虽然同意了我的方案，但还继续坚持使用他们自己的方案，我完全不知道是为什么。

么。换句话说，他们的行为完全不合理。我当时觉得很受打击，迄今还记得那些痛苦的争论。

我并不是说接受别人的建议不好。当别人的建议有其合理性，也会带有很多益处，你是可以接受他人建议的。某人提出的建议如果没有充足的理由，那么不能仅仅因为其资深的身份就贸然接受。比如说，你应当有坚持自己意见的权利，同时把所提建议解释清楚。

接下来，我会给出一些例子，说明固持己见会使得 API 设计这一过程变得非常痛苦。

13.1 API 必须是漂亮的

遍历本书，我们一直坚信，真理必须是漂亮的。第 1 章中已经说明了，古代文化和科学已经在不觉中将这一点深植于我们心中。

在设计 API 时追求漂亮，一个直接的结果是必须保留很多本该“弃用”的内容。对于开发者来说，为了漂亮往往会选择从 API 中移除一部分内容，而不是弃用这部分内容。毕竟设计一个都是弃用内容的 API 并不是我们的目的。但对于从旧版本迁移到新版本来说，使用 deprecated 关键字来标识某个内容已经被弃用了，相对于移除内容而言，这样做算是比较小的代价了。这样的代码，只是搞得用户在读相应的文档的时候会产生短暂的迷惑而已，代价的确不大。相比之下，如果直接把某项内容从 API 中完全移除，这样做的代价就高了很多。与其他任何破坏兼容性的行为相似，这种移除行为也意味对用户进行惩罚。用户因为使用你的类库而受到惩罚，如果从市场角度来看这件事，肯定是不对的，因为就不应该用自己的产品来折磨用户。但我经常会听到对这种行为的辩解：“总之，没人在用这个 API。”如果真是这样的话，那么更有充分的理由对其加上弃用的标识了，反正也没有人用，这样的弃用就不会给别人带来任何麻烦。

我再强调一次，优雅固然好，但并不是 API 最重要的特质。特别是谈到资金的时候，这个问题就更加明显了。项目的出资人更关心两方面的问题：尽快地将产品推向市场，同时控制各项成本。除此以外，还有一个特定因素，即这项技术酷不酷。如果这种“酷”技术没有考虑上面的基本经济指标，那么我对这项技术在未来的应用情况及使用范围表示怀疑。

从某种程度上说，弃用不是一种很酷的做法。但弃用机制不会过多影响产品推向市场的时间，也不会增加各项成本。事实上，弃用带来的效果反而是正面的：你弃用的内容越多，说明你越关注现有的用户，通过弃用的方式来保护他们原有的开发投资，有效地帮助他们降低各种成本。当然，我这样说，并不是说在自己的代码中就只能使用弃用这种方式。但弃用机制的确是一个非常重要的工具，可以鼓励 API 的老用户迁移到新版本上，也建议新用户使用更合适的新处理方案。所以，最好是把弃用机制作为一种善意的提示，而把移除作为一个不得已而为之的方案。

13.2 API 必须是正确的

还有一个好的建议，就是 API 必须是正确的，但这个建议常被违反。相对于有问题的 API，正确的 API 肯定有用得多。相对于容易被误用的 API，易于使用的 API 也容易用得更多。前者会让用户翻遍文档，才能找到正确使用该 API 的方式。如果为了正确性而在设计时牺牲了易用性，那会引发更多问题。

如果这本书的内容都是那种通用的方法，如从文件中读取字符串，那么这本书早在多年前就可以出版了。我相信在使用 Java 开发程序的人，绝不止我一个人写过那种读取文件的代码，这段写起来还是比较冗长的。不过这种冗长的代码其实也是不得已，就像用 C 来实现同样的功能，代码也同样非常麻烦。对于 C 来说，它可以读取任意的文件，不管有多大，也不管文件结构如何。但这种简化带来的则是额外引发的随机性错误。

Mercurial^① 与 Subversion

对于 Java SDK 项目和 NetBeans 项目来说，有一个比较有意思的事情，那就是它们分别使用了不同的版本控制工具。以前 NetBeans 使用了 CVS，然后迁移到了 Mercurial 上，Mercurial^② 工具与我们以前用过的大部分版本控制工具完全不同。接下来的后果也很正常：开发人员开始抱怨，而且越来越严重。所以经常会听到有人在叫：“为什么不使用 Subversion？”

那个负责将 CVS 转换到 Mercurial 的工程师给了我一个很好的解释：“CVS 和 Subversion 相当于尝试使用经典力学的方式来解释这个世界，但对于速度或者距离，都给出了错误的答案；而对于 Mercurial 来说，则在一定程度上是使用了相对论来对这个世界给出直接的说明”。

这个解释非常到位。对于 Subversion 来说，提交的结果是一种完全未知的状态，就好比开发人员向代码库里发了一个轻量级的信息，然后觉得可以在预定的时间内或按照一定的顺序完成。整个的代码库对于所有的文件都会进行版本管理，当某个开发人员提交某个文件的时候，可能该文件已经被其他开发人员修改并提交。所以必须先检查整个库的签出状态。就好像随机地向天上的星星发一束光，然后就期望哪天地球上有个人能够在这束光中看到蒙娜丽莎的微笑。很显然，这不可能发生。

使用 Mercurial，工作方式就有所不同了。首先，需要开发人员来获取代码库的状态，保证本机的代码库与其同步，然后将你所做的修改合并进去，再将状态推回去。这个状态回推只有在开发人员本机上的代码库版本完全匹配的情况下才会成功。如果此时恰巧有人改变了这个代码库，就需要重新再来一次了：拉，合并，推回去。这样可以保证开发人员清楚地知道自己在做什么：了解自己硬盘上代码库的状态。回头看一下前面那个光束的比喻，这样做就好比把光束发送到其他星球以后，由其他星球再返回相应的信号，通知相关人员说要进行哪些修改，然后再把整个光束都返回过来，等调整后再重新发过去，最终形成正确的图像。显然，如果多个开发人员在同时进行这样的操作，那么只会有一个人成功，其他人也要重复这种步骤“拉，合并，推回去”。

从理性主义的角度来看，Mercurial 的处理方式是正确的，而且非常漂亮和优雅。虽然

① Mercurial 是一种轻量级分布式版本控制系统，采用 Python 语言实现，其特点是易于学习和使用，扩展性强。它是一个基于 GNU General Public License (GPL) 授权的开源项目。项目的主页是 <http://www.selenic.com/mercurial/wiki/index.cgi/Mercurial>。——译者注

② 原文使用了 hg 作为 Mercurial 的简称，Mercurial 的原意是元素周期表当中的汞元素，但是 Mercurial 这样的单词显然不太适合日常使用。事实上 Mercurial 的命令取了元素周期表当中汞元素的化学符号——hg，所有的 Mercurial 命令都以 hg 开始。——译者注

Subversion 很容易上手，但它的方式却不是这样子。猜一猜：我们的那些开发人员碰到了什么问题，为什么是想换 Subversion 而不是 Mercurial 呢？答案是与“无绪”有关，开发人员希望在提交代码时不要关心太多其他内容。版本控制系统的正确性和易用性就讲到此。

如果你是一个新入门的开发人员，那么会如何使用 Java 语言来读取和解析一个文件呢。假设你已经找到了 `java.io.File` 这个类，然后正在寻找处理文件的方式。首先，你在 `java.io.File` 这个类里根本找不到哪个方法来访问文件的内容。于是你只能在 `java.io` 这个包里拼命地查找相关内容，经过长时间的查找，终于找到了 `FileInputStream` 这个类。但不是每个新入门的开发人员都对“流”有着很清楚的概念。在用其他一些编程语言开发时，可以完全不使用“流”这个概念就可以读取文件。不过为了简化当前的例子，我们假设新人也对“流”有所了解，并认为 `FileInputStream` 正是需要的类。既然已经创建了一个输入流的实例，就考虑从流中读取文件内容，但怎么做才能读取内容呢？先找到 `read` 方法，但这个方法用起来很怪呢：有两个 `read` 方法，一个每次读取一个字符，另一个则是每次读取一个数组。问题是我从哪里找个数组出来啊？现在我想从文件中读一个数组，竟然还要我先找出一个数组，天啊！又晕头转向地搜了半天，终于知道如何定义一个空数据数组^①，然后将文件流中的数据读取到这个空数组，然后一遍遍地重复，直到读完为止。对一个原来用 Perl 语言进行开发的程序员来说，这样做简直是酷刑。这么个小小的功能，竟然要如此费事才能完成。

我听过一些 Java 设计者解释 Java 的发展为什么没有 C# 那么快。他们觉得与 C# 的开发微软公司相比，Sun 公司更看重兼容性。如果说向 Java 语言中添加了某些内容，那么这些内容就得一直保持下来，永远都不会移除，所以对于 Java 来说，开发新功能就要慢一些：需要更加全面地对新功能的正确性进行分析处理。虽然这个说法可能很令人信服，但如果有一个开发人员只是想分析一个文件中是否有“OK”这个字，那么这个理由就很难让他接受。写单元测试时，这是一个最常见的操作。由于这种从文件中读取字符串的需求太普遍了，所以在我们的项目所有的模块中都有一个这样的通用方法来做这事。我知道如果处理的文件很大，那么这样做就会有问题，不过我不在意。因为我知道我的文件都很小，如果说真出现了大文件，那么抛出一个异常或者错误就是了。尽管这样做会给程序带来一些随机性的问题，而且从理性角度来分析，这也算是不太正确的处理，但对于这样一个简化了的 API 来说，在大部分用例中能够正确运行也就可以了。

也许单元测试是最恰当的例子之一，从文件读取字符串这种功能用得最多，但平时开发时，也会经常需要这种功能。比如说读取一个配置文件，那么最好先检查一下这个文件能够放在内存中，不会出现内存不足的问题。通常这种配置文件都是开发人员提供的，是人工完成的，也就都不会太大，所以像 `java.io.File` 这个类应该提供 `asText()`，`byte[] asByte()` 和 `Iterable<String> asLines` 之类的方法，方便直接取得文件内容。这样做不仅仅是更方便用户使用这个 API，而且用户在使用这个 API 的时候，就不需要去了解与这个 API 相关的一些内容，如流。这样做不一定是百分之百的正确，但可以缩短产品推向市场的时间。如果说想增加用户使用 API 的代价，那么

^① 原文是 Empty Array，但这里的意思并不是 null，也不是零长度的数组，而是指一定的长度，但数组中的数据没有进行任何赋值操作，所以这里翻译为“空数据组”。——译者注

原先这种先定义空数据数组，再来填充的方式倒是可以满足要求。

Java 可以考虑在下一个版本中添加这些新方法。如果真这样做了，我觉得这将是一个非常伟大的改进，不过也许要等 10 年以后了。有了这些方法，开发人员就可以把精力放在具体的功能代码，而不是陷进如何读取文件、创建数组等无关的细节中。有时候，易用性比正确性还要重要。C# 的设计者可能对此有较深的认识，因为在 C# 中，多个版本都提供了这种功能。

13.3 API 应该尽量简单

如果一个人刚刚开始使用 API，那么他会提出这样的问题。这点非常重要，因为第一眼或者说一开始上手的那种感觉对于用户来说非常重要。如果用户要使用某个 API，一上来就晕天黑地找不到入口，这样用户就会对这个 API 十分失望。显然，入门的门槛不能太高。然而，大多数 API 用户使用 API 的次数绝不止是上手时的第一次，所以这点也要考虑到。他们会在使用 API 的过程中对 API 有更加深入的了解，他们也会逐渐提出更多的需求。特别要注意的是，如果用户已经花费了大量的时间和金钱来使用你的 API，那么在简化 API 的时候，也一定要保证他们的投资物有所值。简言之，就是在设计 API 的时候，让复杂功能可能实现的同时，让简单的功能尽量地易用。

两个对话框的故事

过度简化经常会出现。我在开发过程中遇到的最复杂的事情不是与 API 设计有关，而是与界面设计有关。设计界面与设计 API 有很大的差异。一般来说，一个好的界面设计者不一定是一个好的 API 设计者，反之亦然。但如果再深入一个层次看，其实两者的设计原则还是有很多共同之处的。首先，它们都要以用例为导向，要了解自己的用户，并从多类用户中抽取通用的元素，加以设计，期望达到普遍适用的效果。

Linux 创建人 Linus Torvalds 曾经在一封邮件^①中写过：他认为 GNOME^②提供的文件选择框太难用了。在开源社区中，很多事情其实做得并不完美，所以有抱怨很正常。一般的想法是，假设你希望把事情做得更好一些，可以考虑自己动手加以改进，然后再把改进后的代码贡献给原来的社区或者是作者。但 GNOME 在提供文件选择框时，其实是为 Linux 初学者进行过优化，而对高级用户有很大限制。主要的问题在于这个文件选择框的维护者不愿意进行这种用起来会变复杂的调整。他们认为“这样做会让 Linux 初学者在使用该文件选择框时容易误用”。所以哪怕像那种用键盘输入文件名这种非常简单的事情，只要这个功能比鼠标单击来得复杂，那么就意味着至少能适用于这两类用户了。

Torvalds 的邮件在 Linux 社区引起轩然大波。很多持有相同观点的人则希望有人能设计一个 API 用来桥接不同的文件选择框，这样不同的用户就可以根据自己的需要使用不同的文件

① <http://mail.gnome.org/archives/usability/2005-December/msg00022.html>。——译者注

② GNOME，即 GNU 网络对象模型(The GNU Network Object Model Environment)，GNU 计划的一部分，开源运动的一个重要组成部分。其目标是基于自由软件，为 Unix 或者类 Unix 操作系统构造一个功能完善、操作简单以及界面友好的桌面环境，它是 GNU 计划的正式桌面。可以通过 <http://www.gnome.org> 这个网址来访问其官方网站。

——译者注

选择框了。但很多为 GNOME 贡献代码或者使用 GNOME 的人则因为 Torvalds 这种强烈的言论而感到被伤害了。

但 Torvalds 的确是提出了一个非常好且实用的观点。不管你设计什么，都不能把你的用户当成傻瓜来看。在尽力简化 API 的同时，也要避免过度简化。

人们普遍认为，复杂的功能应该被隐藏起来。如果用户需要去使用这些复杂的功能，他们自然愿意花费大量的时间研读这方面的文档。而对于那些新手，这些复杂功能不应该出现在他们面前。通过隐藏复杂度，就可以避免新用户在刚开始学习使用时放弃对 API 的学习。然而，对于高级用户来说，对待方式就有所不同，不能说因为方便新手学习而隐藏复杂度，最终造成其他高级用户无法完成一些复杂的功能，这样是不可取的。在设计时，可以将简单的功能放在一个包中，而复杂的功能则放在另外一个包中。比如说，常用的网络 API 都放置在 `net` 包中，而对于处理流之类的高级功能，相应的类就可以放置在 `net.stream` 子包中。

NetBeans 中的 `DataEditorSupport` 类就是这样一个极端的例子。在 NetBeans 平台上，这个类是用来支持编辑器功能的。事实上，它是一个非常复杂的 API，初学者是用不到这个类的。使用者需要继承它实现一个子类，然后写上相应的代码，将一个新的编辑器注册到 NetBeans 的平台中。这样的话，只要用到这个功能，再简单的例子也会变得非常复杂，因为这需要用户编写大量无用的代码。编写辅导教程的那些人和某些狂热分子，已经就这事批评我们很久了。最终我提供了一个 `DataEditorSupport.create(...)` 工厂方法来解决这个问题，这个方法里面实现了所有功能，避免了上百行没用的代码。但据我的经验，这个方法估计只适用于那种演示程序。但自从给出了这个解决方案以后，我就再没听过有人就这事抱怨了，他们不再觉得向 NetBeans 平台中注册一个编辑器是一件很复杂的事情了。对于初学者来说，示例代码更加简单而且量也更少，但对于复杂的用户来说，这种复杂还是要继续沿续下去。这个例子很好地说明了：简单的事情应该更简单，而复杂的事情不一定要做得像前者那样简单。同时这个例子也说明了：示例应该简单，不要吓坏初学者，而复杂的高级功能还要可用，喜欢深究它的，用户自然会花时间和精力去研究。

上面关于简单性的说法，与书中提到的“只公布自己需要公布的内容”这一重要原则有一点儿冲突。纵观全书，我一直坚持说尽量让功能 `private`、`final`、不让外部访问等。这种处理方式有助于 API 在未来加以改进，也有助于增量设计 API。在 NetBeans 项目中，这种方式已经被证明是行之有效的。但从另外一方面来说，因为我们暴露的东西越少越好，所以经常有一些用户在我们的邮件组里问是否可以把某个类的字段变成 `public` 的，或者允许某个类可被继承，亦或允许某个方法可被重写。如果对这些需求我们都答应下来，那么我们会把自己陷于极其不利的境地，在以后无法更好地对代码进行改进。但如果我们直接对用户说“不”，那么就会像 GNOME 的界面设计者一样，被很多人骂得体无完肤——不能因为简单化而牺牲可用性。你绝不能和用户说，“你是一个绝顶高手，所以我们没有办法满足你的需求了！”我会在第 14 章对这一问题给出说明，要鼓励用户自行找到解决方案，然后进行设计、修改，并行之于文。如果他们能做到这一点，只要你审阅一下，然后把这个方案合到现有的系统中就可以了。有段时间，Nokia 使用了 NetBeans 平台用来开发网络监控程序，我们与其合作时，就采用了这种方式。Nokia 的需求是独特的、高

级的。我们不想直接支持他们提出的需求，因为这些需求对于 NetBeans IDE 来说并不合适。但合作项目中 Nokia 那方的联系人想根据他们的需求对系统加以调整，然后由我们审阅通过。我们要做的事情比较简单，只是检查这些调整是否满足 API 设计的原则即可。我们还会去检查这些调整是否让简单的事情处理起来更容易，而不是更复杂。当 Nokia 那方给出的调整满足了我们的要求以后，我们会将这些调整合并到 NetBeans 的源代码中。我们很满意这种工作方式，同时 API 的用户也感到很满意，因为通过 NetBeans 平台可以支持更加复杂的用例开发了。

13.4 API 必须是高性能的

性能往往是破坏向后兼容性的主要原因。如果 API 设计得不好，那么用户在使用 API 时会碰到很多性能问题，如在调用方法时，代码的执行速度非常慢，亦或在执行过程中会创建很多不必要的对象。性能问题较难修复，除非改变程序既有行为。当然，从另一个方面来讲，在设计 API 时，也不应该单单为了性能过度优化以及违背 API 的设计原则。况且，这样做所获得的性能提升往往也是虚假的，或者仅仅是暂时的。对于 Java 或者其他动态编译语言更是如此。要知道编译后的程序，并不是真正执行的程序。执行之前，还需要经过解释或者动态编译。如此一来，产生的机器代码是经过持续优化的。Java 刚刚出现的时候，大量临时对象对垃圾收集器来说是一个巨大威胁。然而，当 Java 引入了分代收集的垃圾回收机制以后，这些临时对象就会被放置在“新生代”内存区域中，很容易被回收。如果为了性能优化而去调整代码避免创建这种临时对象，那么今天看来，可能就等于做了大量无用功。当然，创建的对象越少，垃圾回收器的负担就越轻。从这个意义上讲，你所做的这些工作并不全然无用。但在这个针对内存占用的优化的过程中，代码往往会变得更加难懂，这也限制了 API 自身，可谓过度优化。

以直接暴露字段的方式取代 getter 或者 setter 访问方法，是不成熟性能优化的另外一种情形，至少对于 Java 来说是这样的。相对内置优化支持较少的语言来说，例如.NET，这样做可能是一个必要的技巧。但对于 Java 来说，就完全没有这个必要了。况且，这样做并不能带来性能提升，因为 Java HotSpot 动态编译器通过采用方法内联的策略将 getter 或者 setter 方法访问转化为简单的直接字段访问，来消除额外性能开销。而且，直接暴露字段这种方式，可谓限制了未来 API 优化的空间，在 5.1 节中我们已经深入讨论过这个问题了。

简而言之，谨防不成熟的优化，因为在 API 设计中，这一点往往是“罪恶之源”。

13.5 API 必须绝对兼容

纵观本书，向后兼容被反复提及。如果能做到向后兼容，就不用担心用户升级时出现各种问题。在对类库做整合时，无论是最终用户还是应用程序装配者，都极其依赖于这种向后兼容性。所以说，努力做到向后兼容是可取的，也是必要的。然而，考虑到当前软件项目开发的实际情况，向后兼容性有时候的确是实现不了的。本书的目标之一就是向大家解释为什么每一个类库或者框架都应该做到向后兼容，同时也会介绍相关技巧，帮助我们避免犯下破坏向后兼容性的大错误。尽管如此，关于是否能够达到 100% 的兼容，相对于如何遵守本书中给出的建议，更重要的是态度。

要达到百分百的兼容性，不光成本很高，而且难度极大。事实上，就算是修正一个微小的

bug 都可能破坏兼容性。比如，原来出现某种情况是抛出一个 `NullPointerException` 异常，程序不再执行，而现在能够处理这个异常，程序可以继续运行。这种情况其实也是一种不兼容性的体现，因为有些代码可能会根据这个 `NullPointerException` 来进行某些操作。但从 Java 编码和 API 使用的角度来看，不存在任何问题。但新版本不支持这种用法，也意味着它不能与旧版本达到 100% 的兼容。

修正 Bug 当然是很重要的事情。很多时候，版本升级的主要原因就是修复 Bug。第 17 章会给出一个 API 设计的小游戏，它会描述在修复 Bug 时虽然每一步看起来都是正确的，但会破坏兼容性。也就是说，修复每一个 Bug 对 100% 向后兼容都具威胁。因此，100% 的兼容性是很难做到的。尽管第 17 章的那个游戏中说明一定程序上的不兼容性也是可以接受的，但问题在于如何学习保持百分之百的兼容性。像这个例子中，如果找不到更好的解决方法，那么保持 100% 的兼容性的成本就十分高昂。

因此，大多数情况下，能做到 99% 的兼容性就足够了！我知道在这里这么说是唱反调。因为很难定义这个所谓的 99% 的兼容性包含什么，剩下的 1% 指的又是什么。一旦允许出现不兼容情况，比如原来方法返回值不为 `null`，现在竟然返回 `null` 了，这样做会使得原来运行正常的程序无法正确执行了。再比如，配置文件的格式也有所调整，或者是把类名改掉、把方法删除等，我相信很多人都会把出现的这些问题归为那不兼容的 1%。所以，对于理解的人来说，一开始就要把目标定为达到 100% 的兼容性。只不过，有时我对同事说起 100% 的兼容性时，他们就会大笑不止。对于这些家伙，只好给他们严格地定义一下 99% 的兼容性到底包括哪些内容。

100% 的兼容性就意味着绝对兼容，也就是说不会有任何人或者任何事物会破坏兼容性。也暗指即使用错误的方式来使用一个 API，那么也会沿续下去。但一旦有这种绝对化的目标，就意味着马上要去检验所有版本的 API，避免出现任何一个不兼容问题。同样，所要检验的内容也不仅仅是这些对外的 API，对 API 的具体内部实现也要加以关注。这对开源类库的用户是很常见的。然而，研究所有 API 版本的源代码的做法并不常见，只有在需要检查版本不兼容问题时，才会逐行对比两个版本的 API 源代码。一旦发现有哪行源代码有所不同，就认为 API 不是 100% 的兼容。大部分 API 的用户不会去做这种事。他们不会检查 API 所有版本是否有所差别。

接下来的代码示例显示了一种对兼容性持有的认真态度。

```
public class Arithmetica {
    public int sumTwo(int one, int second) {
        return one + second;
    }

    public int sumAll(int... numbers) {
        if (numbers.length == 0) {
            return 0;
        }
        int sum = numbers[0];
        for (int i = 1; i < numbers.length; i++) {
            sum = sumTwo(sum, numbers[i]);
        }
    }
}
```

```

        return sum;
    }

    public int sumRange(int from, int to) {
        // 开始: 读取 arithmetica.V2 这个指定属性
        if (Boolean.getBoolean("arithmetica.v2")) {
            return sumRange2(from, to);
        } else {
            return sumRange1(from, to);
        }
        // 结束: 读取 arithmetica.V2 这个指定属性
    }

    private int sumRange1(int from, int to) {
        int len = to - from;
        if (len < 0) {
            len = -len;
            from = to;
        }
        int[] array = new int[len + 1];
        for (int i = 0; i <= len; i++) {
            array[i] = from + i;
        }
        return sumAll(array);
    }

    private int sumRange2(int from, int to) {
        return (from + to) * (Math.abs(to - from) + 1) / 2;
    }
}

```

分析上面的代码, 可以发现 1.0 版本中的 `sumRange` 方法中所采用的算法, 其性能非常差, 但这个算法可以让其子类用来计算阶乘数据。10.4 节中有一个例子, 由于覆盖了这个方法, 提供了一个更加高效的实现, 结果却破坏了兼容性。如果你希望仍然保持这个方法的兼容性, 那么可能要使用别的方案了, 比如说通过 `Boolean.getBoolean("arithmetica.v2")` 从系统配置中取得相关信息, 以便判断是沿用老的方法还是使用新的方法。这样做可以算得上是实现 99% 的兼容性了, 但绝非 100%。毕竟这两个版本还是存在着一定差别的。

前面这个例子说明了 99% 向后兼容性的意义及其限制: 也许两个版本不是 100% 兼容, 其 API 也不是绝对兼容。但这些情况对于用户的影响是非常小的, 他们往往只知道有 1.0 版本, 并不知道这个 API 在新版本中会有哪些改变, 更不知道 2.0 版本竟然引入了一个基于属性的 API, 更不用说这个属性配置的名称了。了解所有这些情况的用户多少可能会有几个, 但绝对算得上凤毛麟角, 所以正常情况下不会出什么问题。从实用的角度来看, 2.0 版本与 1.0 版本还是做到了兼容。

偷偷做那 1% 不兼容的事情

这里我必须再强调一次这种方法符合常规的思路。我知道有些人会说, 即便通

过!Boolean.getBoolean("arithmetic.v1")来进行反向判断,也能得到99%的兼容性。其实这种观点是错误的,因为这个类的含义已经完全改变了,除非用户明确要求旧版本的行为。当然,这种情况“只”会出现在计算阶乘的时候。但我相信用1.0版本的用户中,用这种继承该类的方式来计算阶乘的人肯定存在,而且数量可能还不在少数。事实上,只要我看到这种可被继承的类还带有一些实现方法,那么我就知道一定存在这类问题。所以这种反向判断的方案是一种完全不可行的错误方案,因为它肯定做不到99%的兼容性。但面对一个非理性的人,他一门心思想偷偷摸摸地做改动,你又如何能向他解释清楚呢?

所谓“99%的向后兼容性”就意味着在修复问题的同时,要严谨地考虑其可能带来的不兼容性问题,往往要在两者之间取得一个平衡。如果说发布的版本只有少数,因为修改API而引发不兼容问题是可以接受的。但这种定义性的东西也要来具体地加以解释。就像前面所说的,这种不兼容性是建立在常识之上的,所以处理兼容性问题时一定要谨慎。

13.6 API必须是对称的^①

对称,其实是对漂亮的强调。不知道为什么,如果事物成双成对出现并保持对称,我们就觉得非常优雅。而且在我们内心深处,总是把漂亮和优雅与真理联系在一起。有时候,我们也据此来判断API的正确性。假设你看到两个API,其质量差不多。它们都可以在未来加以改进,也可以维护,易用性也不错。其中有一个API看起来比较对称,此时,你就可能选择这个对称的API了。然而,绝不要为了追求对称而牺牲了API其他优秀特质。

还是先来看一个例子吧。最近NetBeans团队准备写一个读写JSON格式(JavaScript Object Notation^②)的API。当时负责这项任务的工程师让我评审一下他设计的API。他说他对设计的API不是很满意。我们一起看了一下这个API,尝试找出新的解决方案。这个API的内容益发清楚了,但我们还是对此不太满意,总觉得少了一点儿东西。一个月后,我们又说起这个API,那个同事告诉我缺少的就是“对称”。最开始设计的那个API试图在读写JSON对象时以相似的方式来处理,实际上根本就没有这个必要。在写JSON对象的时候,需要在堆中放置很多活动的Java对象,然后将这些对象写到流中。要完成这个功能,其实很简单,只需要拿到JSON的根对象和一个输出流就可以了,完全可以用一个static方法来实现这个功能。在设计读JSON对象功能时,也想用类似的方式来实现这个功能。这样做,是通过一个方法拿到一个流,然后返回一个活动的Java根对象。当然在读取时还可能有一些其他需求,比如说,从文件中读取内容生成JSON对象,如果文件数据比较大,那么就不太可能完全在内存中马上完成这样一个操作。解决这个问

① 原文中用的是 Symmetrical, 对应的意思是“对称, 整齐”, 在中文中很难找到一个恰当的词与之对应, 作者在这里用这个词的意思是指: API 设计时其使用方式应该比较一致。就像后面所举的那个读写 JSON 对象的例子, 就是希望在设计这个 API 时, 能让读写方式保持一致, 减少学习成本。——译者注

② JSON (Java Script Object Notation) 是一种轻量级的数据交换格式。易于人们阅读和编写, 同时也易于机器解析和生成。它是基于 JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999 的一个子集创建的。JSON 采用完全独立于语言的文本格式, 但是也使用了类似于 C 语言家族的习惯 (包括 C、C++、C#、Java、JavaScript、Perl、Python 等)。这些特性使 JSON 成为理想的数据交换语言。——译者注

题，要设计一个基于事件的回调 API，就像解析 XML 的 SAX (Simple API for XML) 处理方式一样。做好这个设计还是比较容易的。但我们想将 JSON 对象写到文件时，也采用同样的处理方式时，却比较麻烦，我们花了大量的时间想找出一个与读文件相似的方式来写文件，最后却无果而终。当时我们也觉得比较头痛，直到真正认识到“真理、漂亮与优雅”这句话的真谛时，我们才发现根本就没有必要把这个 API 写成对称的。

把一个 API 写得对称，肯定有助于他人更快地接受这个 API。对称也有助于提高 API 的透明度，因为用户在使用这个 API 时，只要记住一半的内容，另外一半也就很容易从前面那一半推出来了。但判断一个 API 不能仅仅根据其对称性。正如前面说的那个解析 JSON 文件的例子，请记住：有可能只写了 setter 方法，而没有 getter 方法，反之亦然。又或者说 setter 方法是 protected 的，而 getter 方法却是 public 的等，这类情况很多。所以不对称并不是什么大的错误，特别是如果是因为简化 API，又或者为了方便 API 在未来加以改进而牺牲了这种对称性，其实还是值得的。



API的设计不同于普通的内部系统设计。这就是为什么当我们把相同的规则运用到 API 设计中的时候，得到的结果会出人意料。这样的结果或许有点儿难以置信，因为它与我们通常的期望相悖。然而，出乎所料未必是错的。软件系统的规模才是造成这些区别的根本原因。就像描述一所房子与描述整个宇宙不同，当我们讨论内部软件系统设计与 API 设计时二者也有所不同。

随着视野的扩展，我们会进入未知的领域。然而，这并不意味着我们不能展望未知领域。对待未知领域我们有两种处理方式。第一种方式是放任对未知的恐惧战胜我们探索未知世界的渴望。这种情况下，我们假定未知世界是危险、混乱无序、不友好的。对未知世界的探索充满痛苦与困难，最好还是老老实实呆在家里。这是看待我们尝试把视野扩张到未知领域的一种方式。事实上，这不是一种积极的姿态。如果我们都不愿去尝试探索，就不会有任何关于新世界的发现，也不会有任何新发明。结果，我们构筑了一道墙，阻止我们认知未知的世界。然而，还是不时地有各种情况迫使我们走出去，探寻那些最遥远未知的星球。

第二种是接受一种极端但又很正常的主张，认为未知世界与我们周围的已知世界是非常类似的。这假定了我们已经知晓世界的一切。的确，从这样的预期出发探索未知世界则简单得多，因为这样感觉更安全，不会有碰到出乎意料的事物的危险。抱着这样的念头，我们就可以更惬意地背起行囊，开始探索宇宙尽头的旅程。我们坚信自己不会迷路，或许未知的宇宙与眼前的世界没什么两样。

这两种对待未知世界的方式并不是绝对化的。在我们的内心它们常常相互冲突纠结，有时候我们的恐惧战胜了探索未知的信念，有时候则是另一种情形。这两种情形各有利弊。当前者发生时，我们只是放缓了探索未知世界的步伐，谨慎而惶恐地扩展我们的认知视野。这样的探索方式会耗费很长的时间，甚至有时候会一无所获，这源于我们对未知的恐惧。然而，这样的方式下，我们从不会对新发现事物的奇怪行为感到惊讶。我们知道未知世界是混乱无序的，也预料到会发现诡异的新事物。我们对出乎预料的事物早有准备。

以这种“我全都知道”的方式对待未知世界，我们甚至会显得鲁莽轻率，会不加小心地进入未知领域，不管自己遇到什么，都认为我们能作出合理的解释。毋庸置疑，我们会认为这些新事物都和周围的已知世界并无两样，只是更大些、更快些罢了，但除此之外就像我们习以为常的已知世界一样，并无本质区别。这样的方式往往行之有效，尤其当我们未深入地进入未知世界的时候。

未知世界或许和已知世界很类似。但当我们越深入地进入到未知世界时，就越可能遇到无法解释的事物。也就是说，基于现有的知识，这样的事物不可能存在。这样的事物是“反常的”。然而，这样的反常之处并不意味着这些事物与世界自身的本质相矛盾。通常，这只是世界与我们的预期发生了冲突。

文艺复兴时期无畏的物理学

前述两种对待未知世界的态度并不仅仅存在于计算机科学领域，也存在于物理学领域。牛顿物理学之所以取得如此巨大的进步，可归因于牛顿的追随者们采用的探索未知的方式。这些追随者大多数处于“我全都知道”的状态，也只有在这样的精神状态下，他们才能对当时的探索行为深信不疑。每当遇到无法解释的事物，他们都深信只需要一定时间和努力来进行研究，就可以分析清楚并理解它们。唯有这种整个宇宙对我们都是已知的信念，才使得物理学当时的飞跃发展成为可能。

在19世纪末，人们发现了很多与现有理论自相矛盾的现象，显示世界与人们的预期远不一样，此时盲目的确信和无畏的态度就消失了，对于未知世界的恐惧又重新回来了，经过一段时间人们才从这种失望的情绪中恢复过来。又经过一段时间人们才对这些自相矛盾的发现给出了合理的解释。时至今日，当时认为的自相矛盾的现象如今已不再被认为是反常的了。对于几乎所有这类现象，都有了合理的技术解释。但换个角度来看，这类现象仍然非常重要。它们的存在时刻提醒着我们，我们曾经错误地以为宇宙与我们已知的世界差不多，而事实上两者有着巨大的差距。我们应牢记这一点，谨防将来有一天，我们又一次深信无所不知，认为世界一切事物如我们认为的那样。

本书中的大部分内容都是在逐步探索API设计领域，循序渐进，希望尽可能以平稳的方式不断扩展我们的视野。这样做，可以避免在API设计领域中迷失，也不需要去面对一些奇异或是自相矛盾的情况。然而本章有些不同，我们将稍微远离熟悉的领域，到API设计的未知领域做一次“短途旅行”，介绍一些关于API设计的内容，这些内容在API设计领域看起来很正常，但与那些内部系统设计的经验完全不同。希望读者可以享受这次“旅行”，也希望你可以正确地找到“回程之路”。

14.1 API设计中的自相矛盾

有一些API设计看起来是自相矛盾的。有一些开发人员和我说过：我让他们按照特定的方式来做的时候，对他们来说，简直就是一种折磨。Doublethink（自相矛盾）一词是由George Orwell先生在他那本著名的《1984》^①中引入。维基百科将其定义为“同时相信两种相互矛盾的观点，

^①《动物庄园》和《1984》是奥威尔的传世作品，在这两篇小说中，奥威尔以敏锐的目光观察和批判以斯大林时代的苏联为代表的、掩盖在社会主义名义下的极权主义；以辛辣的笔触讽刺了那泯灭人性的极权主义社会和追逐权力的人；而小说中对极权主义政权的预言在之后的五十年中也不断地被历史所印证，这两部作品堪称世界文坛政治讽喻小说的经典之作，其影响绝不仅仅局限于文学界。——译者注

而没有意识到这两者是矛盾的”。打个比方，像“战争即是和平”和“既精神错乱又足智多谋”就是自相矛盾的话。基于这种思维方式，对于同种情况有可能会得到两个矛盾的结论，如：一个产品应该尽快地推向市场，否则就立即放弃开发该产品。再比如，在正式发布一个 API 之前，对其判断时，经常会说：“把稳定的内容对外公开，或者把它私有化，不对外可见。”

你怎么能一方面要求 API 必须尽量稳定，另一方面又让它完全不可见甚至取消发布呢？API 要么已经非常好了，可以认为稳定得足以对外发布，或是有待稳定，尚不能对外发布。要么就是质量差得太远，无论如何也不能公开了。怎么可能同时既这样又那样呢？

在设计一个 API 的过程中要有一个非常重要的里程碑——第一个对外发布的版本。在这个点之前，整个 API 都是处于开发模式。允许修复 bug 或者问题，可以随便地调整任何内容，即使引起兼容性问题都是可以接受的，因为这个 API 根本还没有用户，或者说用户很少。在这个点之前，要做的主要事情就是创建一个 API 能够满足用户基本的功能需求，并在设计时要尽量地考虑为未来 API 的改进预留空间。即使做个全面重构，也是正常的，因为这对用户没有多大影响。特别是这种重构如果能避免以后产生不兼容的情况，那么就有理由去做。但所有这些工作只允许在第一个版本发布之前做。

一旦 API 的用户数量大幅增加，那么作者就不能随意对 API 进行修改了，此时，开发进入了维护阶段。在这个阶段，最重要的目标就是保持向后兼容性。这样做才能让现有用户感到满意。要做到向兼容，就要避免对公开的类、方法和功能进行移除、清理或者大的重构。如果说，确实有这种大规模修改的需求，应该在 API 进入维护阶段之前完成。一旦进入了维护阶段，所有对其进行的调整都必须针对兼容性进行评审。不能对现有 API 的使用造成任何负面的影响，最好是没有任何影响。

开发和维护这两个阶段，并不仅限于 API 设计，几乎各种类型的软件工程都会有这两个阶段。一般情况下，这两个阶段交替出现。当一个开发阶段结束后，产品进入了维护阶段，交由维护小组来负责，由他们来修复 bug，或者解决问题。同时，原来的开发团队继续开发产品的下一个版本，此时他们又处于开发阶段了。当新版本发布以后，还是交给维护小组加以维护，循环往复。大部分的开发人员都已经习惯这种产品运作方式，因此他们希望这种方式也能适用于 API 设计。

其实不然。严格意义上的 API 设计似乎只有一个循环：开发 API，然后正式发布第一个版本，接着就进入了维护模式。如果有添加新功能方面的需求，也是可以做的，但一定要注意其兼容性，就像软件开发中维护模式所做的那样。此外，API 设计时并不像一般软件项目中有开发和维护两个团队，而是只有一个。也就是说，并不是让某人设计了一个 API，然后将该 API 转交给另一个人去维护，然后开发人员开始去设计新的功能，继而再转交给维护人员。大部分情况下，维护工作和新功能的开发都仍然是由 API 的原作者或者开发团队来完成。由同一个人或者团队来同时进行开发和维护。

API 设计之所以存在这种矛盾，在于其设计过程的双重性，即由一个开发人员或者团队来同时负责开发和维护。必须清楚地知道开发和维护这两类角色在需求上的矛盾。也就是说，当你作为开发人员说“这个 API 还不够好，不要对外发布”，同时作为一个维护人员，你也可能会想“尽管发布好了，没有关系，只要说它是稳定的，作出向后兼容的承诺就行”。

评审 NetBeans 的架构

如果一颗恒星从来没有被人发现过，那么它就不必遵守我们这个世界的规则。它可以随意做出令人费解的巨大变化，如改变大小、颜色、轨道，或者说干脆完全消失了，都没有关系。因为没有人观察他，也就不会有人为此感到惊奇或者抱怨。但我们知道，一旦公布了一个 API，就会有人来用它，注意它，会从一个完全不同的角度来看待这个 API。此时，就要尽可能地避免破坏这个 API 了。

在 Sun 公司工作时，我们曾经做过 NetBeans 架构方面的评审。从原则上来说，所有 Sun 公司的产品在推向市场之前都应该有这样一个评审阶段。而事实上这也没有得到全面贯彻，NetBeans 也不例外。在 NetBeans 被 Sun 收购之前，其实已经对外发布了多个版本。但进行评审的时候，仍然是按照第一个版本的方式进行评审。评审人员建议我们调整文件存放的位置，把启动程序的名字改一下，还有很多内容都要调整。但这些调整对于现有的用户来说，会造成非常大的不兼容问题。但对于 Sun 公司来说，这却是第一次对 NetBeans 的架构进行评审，所以我们给他们解释说 NetBeans 已经有了很多用户，要保持兼容。但他们仍然坚持“第一个版本中，漂亮的设计才是最重要的”。

那次以后，我至少还参加了五次后续的评审。但这五次与第一次完全不同。评审人员不再提那种“改一下文件目录的名称”之类的建议，而是添加一些新的功能，但是不会对现有功能造成影响，避免与前面的版本不兼容。对于老的功能，评审人员提出的最多的建议就是“除非确有必要，否则不要动它”。因为 NetBeans 已经有了很多用户，这些用户要依赖于这些功能，所以不能轻易进行修改。

这种情况就是典型的自相矛盾，但理由却很充分。在第一个版本发布之前，尽力地对其进行改进，不管调整的内容和范围有多大，目标就是改进、改进、再改进。但一旦这个 API 进入了维护阶段，如果可能的话，就不要再改任何东西了。这充分体现了 API 设计过程中开发模式与维护模式的不同之处。

如果一个人既是开发人员，又是维护人员，虽然这两种身份的想法会自相矛盾，但都会推动 API 的向前发展。比如从开发人员的角度来看，他希望可以集中精力尽可能地让一个 API 变得更好、更完美，为此宁愿不发布这样一个类库。而从维护人员的角度来看，则认为即使当前这个 API 远非完美，也应该推出，因为只有这样，一个类库才能真正变得有用。

这种冲突可以看作是理性主义与经验主义碰撞的结果，在 1.1 节中已经就这个问题展开过讨论。试问一下，完美是否是类库和 API 的唯一目标？如果仅仅是为了让 API 变得更加漂亮，是否应该牺牲兼容性呢？我们作为开发人员来说，所秉承的正是这个观点。或者说，对于一个类库来说，保持其可用性，帮助其用户解决问题才是更重要的，不能像前面章节讲的那种阿米巴变形虫一样，变来变去。从维护的角度来说，这个观点才更实用。对于一个 API 来说，在其尚未发布之前，应该倾向于理性主义，尽可能设计得更加漂亮，更加有用。一旦发布了，漂亮就要居于次席，要将兼容性放在第一位来考虑了。

看起来，API 的这种冲突其实只是把理性主义和经验主义这对旧敌带到了前线：开始是理性

主义占上风，希望设计出漂亮优雅的 API。接着，到了某个时间点，需要将产品提交给用户使用，从此刻开始，需要留意可用性。而且有一点要强调再三：尽管我们都知道如何将优雅与实用结合在一起，但它们的结合只能维持很短时间。某个 API 从开发阶段过渡到维护阶段的时间应该很短，如一个小时或一天。而最糟糕的情况就是你假装理性主义和经验主义可以同时适用。正因如此，一种明智且合理的做法就是同时给出两种截然相反的建议：“如果一个 API 稳定，就对外公布，否则还是变成私有的，不对外公开。”要么将 API 公开，要么为了将其变得更加优雅而继续在内部使用。此时要做一个选择。千万不要假装你可以同时实现这两种方式，因为你做不到。那种自相矛盾的想法只能是想法，还是留在脑子里好了，不要放到实践中加以执行。你不可能在持有两个相反观点的时候还能把一件事情做好，必须在这两者之中选择其一，同时实现这两个目标是不可能的。即使你努力尝试去做，结局可能是两边都不讨好，不仅做不到漂亮、优雅，连正确性和兼容性也丢了。

14.2 背后隐藏的工作

要监督 API 的开发是一件非常困难的事情。首先，一个优秀的 API 架构师应该像一只预言凶兆的黑乌鸦^①：总是要看到可能会面临的种种失败，要想到未来可能出现的各种错误，还要对各种风险给出警告。如果是由一个团队来设计 API，监督就显得格外必要。对未来看得越远，这名架构师就越优秀。但这也为执行这项任务出了一个难题。只有出了问题，人们才会对其加以注意。比如说，用户只有在出现了不兼容性的问题时才会暴跳如雷，他们会拒绝升级到新版本上，然后直接使用其他竞争者的产品。另一方面，如果说所有的事情都像其预期的一样，那么就不会出现什么问题，可如果什么问题都没有出现，什么事情都不用做，那也很难说得上是成功。

这种情况有点像安全部门做事的风格。如果说有人劫持了一架飞机，然后把飞机给炸了，大家肯定都觉得安全部门没有尽到责任。人们开始抱怨说机场的安检看上去那么复杂，其实却不管用。如果不出事，人们都嫌这些东西麻烦、过分。而一旦出了事，不管怎样弥补都已经晚了。

要保证一个 API 能够向后兼容与其太相似了！开发人员抱怨的事项总是不断，如评审、编程规范，等等。但 API 监护人需要时刻保持警醒的状态，否则就可能因为一个小小的疏忽而引发问题。另一方面，不时出个小问题，至少说明了你所做的工作确实很有用。

一旦我在 API 演进的过程中发现一个问题，我会指出这是因为它违反了本书中给出的哪一条编码原则。但这样做其实无济于事，相关人员就会反问我：“你要知道的话，为什么在代码评审的时候，没有指出呢？你现在倒是知道了，但当时为什么不提醒我们啊？”针对这种情况，我们设计了一种称为“少数派报告”的方案。就像 16.1 节中讨论的一样，在正式的评审过程中，要有四个人投票表决。决定权掌握在多数人的手中，但其他少数人的意见也不可忽略，其他人有权在一个专门的环节将自己的意见记录下来。我很喜欢成为这种“少数派”，特别是提出一些不一定很严重、但很让人恼火的问题。过段时间，如果真的出现了问题，此时我就不是仅仅地指责和抱怨，因为我可以拿出“少数派报告”告诉他们，这就是证据，我一早就给出了预测。

^① 原文中使用的是 Cassandra 一词，指的是希腊神话中的一位神明，是一位凶事预言家，考虑到这个词在中文中的寓意，这里翻译成“黑乌鸦”。——译者注

我已经习惯了成为“少数派”。这往往是因为我的意见和建议都比较极端，你可以在书的第二部分看出来这一点，有时候，不是一定要遵守所有这些建议。而且在大部分情况下，都不会出现什么严重问题，但总还是有例外的时候。我承认，在那时我会因为自己的这种先见之明而暗自得意。遇到这样的问题有助于我们在以后发现类似的情况。那时我们就知道了，对于这类问题最好是寻求更极端的设计建议，从此以后，我就不再是“少数派”了。

但监督 API 的设计，这件事仍然是一件长期而艰辛的工作，可能只有在发表了多个版本以后，才会凸显效果。与常见的开发过程相比，这种对 API 设计方面进行严格管控的益处是很难直接量化的。最近我有幸开始负责提升 NetBeans IDE 的性能。其实这事听起来很大，但做起来却很容易！只需要找到一个有性能问题的功能点，然后具体对其进行分析，得到这个功能点会占用多长时间、多少资源，然后修复这个问题，就完成任务了。可以很容易通过秒或者字节这种方式来具体量化任务的完成情况，每个人至少都认为自己理解这种方式。对于那些直接给最终用户使用的功能，也很容易量化。比如说在程序中添加一个新的对话框，用户可以用这个对话框实现一些有用的功能。这个对话框虽然摸不着，但看得见，完成的功能也是具体的。但设计一个新的 API 时，就不像这么简单了。设计完一个 API，仅仅是一个开始。还需要花费大量的时间来说服其他的开发人员来使用这个 API 开发功能。但再怎么讲，这些事也是可以在一定程度上量化的。换个角度，这样一个类库的向后兼容性，是一个 API 设计中非常关键的一个点，但很难量化。因为向后兼容的目标就是“不会发生任何事”，而“什么事都没有做”是很难定义为成功的。

14.3 不要害怕发布一个稳定的 API

一般来说，在 NetBeans 团队中，工程师都知道如何编写一个 API，如何以文档的方式对其进行描述。但他们很害怕对外正式发布一个 API。他们知道这个 API 还有很多问题，还有地方有待完善，并不成熟。因为他们有这种感觉，所以不将这个 API 发布出去，仅仅是在内部使用，或者允许 Friend 类型的模块使用。在 4.5 节中，我已经告诉了读者，NetBeans 可以限制模块间对 public 级别类的访问。如果说他们要将一个 API 公开出去，也仍然不想将其标识为稳定版本。他们担心可能会出现无法修复的 bug，或者不能添加新功能之类的事情等。

首先要提醒的是，所谓的“稳定”并不是说让 API 一成不变。它更多是对 API 维护态度上的一个要求。一个“稳定的 API”也是会产生变化的，会逐渐演进。但当改变一个稳定的 API 存在着多种不同的方式时，注意一定要选择那种不会伤害现有用户的方式。而如果伤害不可避免的话，至少也应该选择伤害最小的那种。比如说，决不要因为“更漂亮”而随意地修改类名或者方法名称。值得选择的开发方式并不需要开发者有很高的技术水平，但“态度”是最关键的。这种态度就是要保证现有 API 的功能不变，也许鉴于这种态度，有时候某些做事方式会让 API 看起来不是很漂亮，比如说通过 deprecated 这种方式来标识某些内容被弃用，又或者像 LayoutManager 和 LayoutManager2 两个接口来增强某些功能。但读者应该明白，这种对 API 进行调整时的出发点，更多是基于道德而非技术。

在我认识的人中，有很多人都不愿意做出这么强有力的承诺。这并不是说这些人性格不好，只是要对未知做出承诺绝非易事。发布一个稳定的 API 就意味着对未知的明天做出了承诺。不管未来发生什么事，也不管开发人员收到了何种新的用户需求，也不管要修正的 bug 是什么，开发人员都必须要在新版本中持续兼容老版本，保证原有的用户可以升级。这显然是一个强有力的承诺。也难怪众人害怕做出这种承诺。

在说服了很多开发人员接受 API 开发应该保持兼容性以后，我发现让他们克服这种恐惧的最好方式就是把未知变成已知。我与很多开发人员讨论过他们类库在未来可能出现的几种场景，从而把未知变成了已知。此时，他们就不再有这种恐惧了。当然，没有人能够准确地预知未来的细节。但在一定的范围内，还是可以对未来加以预测的，给未来可能发生的事件画上一个边界。这些边界通常意味着最极端的情况。如果知道了如何来处理这种极端情况，你就知道如何来处理其他类似的情况了。

假设你已经决定发布自己设计的 API，并将其声明为稳定版本，那么现在可能会出现的最坏情况是什么呢？当然了，可能会有很多坏的事情，但突出的只有几件。首先是“时间不够”。优先级发生了改变。工程师的数量永远不够。那些有空的工程师需要去处理最紧急的问题。有时候你的 API 似乎对整个软件版本的发布有着重要意义，过一段时间，优先级又变了，你的 API 又不是那么重要了。可能，你已设计了一个自己认为非常优秀的 API，但它却不是你应该做的主要工作。当时要想发布它，很难，但现在它就在门外等着呢。你觉得最近没有时间来修正这些问题，所以很害怕，其实没有这个必要，因为最稳定的 API 就是没有人用的 API。就像前文里的阿米巴变形虫模型展示的那样，API 中最严重的问题就是 API 在不同版本间不停地调整自己的功能。但如果没有人来使用你的 API，那么就沒有人要添加新的功能，或者修正 bug，这样也就可以降低阿米巴变形虫问题出现的几率。所以考虑到这一点，就完全不必恐惧了，可以直接大声宣布你的 API 已经稳定了。如果你不再对其进行任何修改，那么从某种意义上讲，你的 API 非常稳定，因为它没有任何的改变。

现在，我们已经克服了第一个问题——恐惧。但等我们发布了一个稳定的 API 后，还会有什么问题呢？你也许会发现有些东西做得不太对，想在一定范围内加强一下类库的功能，或者做些功能上的调整。具体细节取决于你 API 的真实情况。但如果你能遵守本书中给出的建议，至少是遵守其中的一部分，那么相信你所设计的 API 一定能做到很好的二进制兼容性。再加上足够的测试覆盖率，那么对于功能性方面的兼容，应该可以做得非常好。

不过，假设你发现原先设计的 API 中有很多内容都存在问题，都需要完全重写，那可怎么办呢？在这种情况下，最好是放弃原来老版本的 API，而是去从零开始创建全新的 API。在做这事的时候，路要走对，要允许老的 API 和新设计的 API 能够共存，在 15.4 节中对此有详细说明。如果你可以做到让这两者共存，那么情况就会非常好。老 API 的用户感觉很好，因为你没有对 API 进行什么改动，或者说改动比较小，不会对原有代码造成什么不好的影响。同时，你也会为自己从头开始创建了新版本的 API 而感到高兴，因为你可能通过前面的错误而学习到很多东西，而且新版本的 API 也没有破坏你对老版本所做出的承诺。

金钱还是信任？

前文中说过 NetBeans 团队对 API 的稳定性曾经加以归类，有一类比较特殊——“正式的”API。所有放置在 `org.netbeans.api` 包中的 API 都归为这一类。这一类其实是对发布 API 的类库如何进行分类加以规定。对于某个版本，类库可能还处于开发状态，但下一个版本可能就是一个稳定的状态。到底是否发布该版本，是个先有鸡还是先有蛋的问题，建立这个规则是希望能够解决这样的怪问题。在许多情况下，这样做都很有效。不过，在大多数情况下，最后都会发现根本就不需要做不兼容的改变。其实这些 API 早就应该被归为“稳定的”。

但有时候，对于其他某些项目，这种对未来的承诺并没有生效，也许这个社会已经不再要求我们守信了。有人和我说，这都是手机惹的祸！在中世纪时，如果两位并肩作战十年的骑士相约在巴黎，那么他们就必须约好的时间出现在约好的地点。否则可能就再也没有见面的机会了。但现在，如果你想见哪个人，只需要约个合适的时间，然后通过电话与其确认一下时间、地点等细节性内容，就一切都搞定了。手机可以说是现代人不守信的根源所在。

考虑到现代社会的情况，也许要求现代人保证他们未来开发的程序可以保持兼容，并没有很实际的意义。就像 Tracy Chapman 在 *If Not Now...* 歌中所唱：对未来的承诺，有和没有都是一样的。如果你想用一种稳定的方式来开发一个 API，那么还是应该尽快发布它，提供第一个版本，没有任何需要等待的理由。否则你也同样在冒一个失信的风险。但让我抓狂的是，有些人还以此为荣！在一次毫无意义谈话中，我想说服他们恪守原来的承诺，但他们却说：“就算有规定说我必须把钱浪费在某件事上，我仍然会很愿意不遵守这条规定，并为自己不遵守规定的行为而喝彩。”我觉得这话用来形容我们当前的社会，实在是再合适不过了。人们会因为一点点的不方便，而不再恪守自己的承诺了。

但现代社会中还是有一些领域，在它们之中，承诺和信任比利益来得更重要。我有一些朋友在为银行做股票交易之类的工作，他们恪守承诺。如果想做一笔交易，他们会打电话给其他交易人员，在电话中商讨价格，一旦两方达成一致，交易就算做成了。当然，电话里是不会直接付钱的。几天后会有相应的支持部门来付钱。在这一过程中，价格也许会下跌，银行应该很想取消这次交易，但我还从来没有听说发生过这种事。即使银行在这单交易中有所损失，但对他们来说，还有更为重要的东西——不能丧失他人的信任。因为所有的交易都是建立在双方相互信任的基础上，如果你一次不守诺言，那么所有人都会知道这件事，没有人会再和你做生意了。没有哪个银行敢冒这种风险。

尽管我也犯过错，但我仍然把 API 设计领域看作是另一个信任重于金钱的领域。API 的设计者与其用户之间的关系是建立在相互信任的基础之上的。破坏这种信任是绝对不应该的，因为这会让 API 的设计者损失其最大的财富，也就是失去用户。所以即使多花费大量的时间和精力也要保证自己的 API 能够兼容，绝不能为了节省成本而破坏你与用户之间良好的信任关系。

但随着手机的普及，社会对于信任的要求也在逐步降低，此时，放弃对未来的承诺，代之以更实际、更快捷的行动，已经为大家所习惯了。所以我能接受 NetBeans 中一些正式的 API 在某个版本上不是很稳定。但对于那些希望编写优秀 API 的人来说，还是应该避免这种情况。如果某个人对外做出了承诺，那么就应该立即去实现自己的承诺。对未来的承诺往往没有意义。

前面所说的这些建议，可能要根据具体的 API 来加以应用。但常规的处理方式是很清楚的：如果你能够将未知变成已知，那么当你正式将自己设计的 API 声明为稳定时，你就是问心无愧的。通常情况下，还应该有一个计划，这样在未来出现一些特殊的、未知的需求的时候，可以对 API 加以调整以解决问题。虽然对未来可能出现的需求加以预测需要花费大量的精力，但这仍然是可以达到的目标。如果说你提供了 API 给他人使用，你就应该可以估计到用户想调用更多的方法来查询和配置你的 API。如果你为开发商提供了 API，就应该想到你所提供的接口仍然不够他们使用，他们希望你提供更多的接口来帮助他们扩展功能。如果你想让自己的 API 能够处理这些需求，那么必须就此给出一个演进计划！只有在这样一个计划的支持下，你才可能保持自己 API 的兼容性。

总而言之，发布一个稳定的 API 首先是一个道德上的要求，而且会一直持续地要求下去。最关心的内容就是这个 API 如何保持向后兼容。首先，你要相信有个 API 就不错，而且最好要保护用户的投资。这个话题已在其他章节探讨过。但当你真正地理解了这一观点以后，你会发现发布一个稳定的 API 根本不算难事。

14.4 降低维护费用

通常认为设计一个 API 的代价非常高，而维护该 API 所付出的代价就更高了。的确如此：设计一个 API 的工作量远非把所有内容发布给大家使用就可以了。而且这些工作是没有简单的方法来绕过去的。我们的程序是通过组件拼装的方式来开发的，因此两个组件之间是需要进行沟通和交流的。如果有这种沟通的需求，最好是让相关的模块正确地提供一个 API，而不是说挖空心思用一些不好的方式来解决。也许这么做在一开始会花费大量的时间和精力，但以后会发现这些付出都是值得的。

我相信对那些已经发布了的 API 进行维护，其成本是可以降低到接近于零的。我们先来分析一下可能出现的种种情况。首先，假设没有任何人使用你的 API。当然这肯定不是你想要的目标。但这种结果的好处在于没有人会向你上报 bug，因为根本没有人用，自然就不会有 bug 了。或者说就算 API 中问题再多，也没有人会在意。很显然，这种情况下，不需要对 bug 进行修复，时间成本为零。

现在再来看一下如果有人使用了你的类库。作为类库的作者，这样让你感觉会好一些。你知道你做的事情多少有点儿用，你知道有人需要你。此时有两种可能性。一种是你的用户很满意你的 API，没有什么抱怨，当然就不会产生维护方面的费用。第二种就是他们会对你的 API 加以抱怨。这种抱怨可能意味着他们对你的 API 有兴趣，自然就受到类库维护者的欢迎。此时，可能会产生 API 的维护费用。但对 API 的维护完全基于个人意愿，要看这个类库的作者是否愿意帮助他人了。

假设 API 的维护者对这些抱怨漠不关心，根本没有时间去搭理他们。此时，又要分成两种情况进行分析了。用户需要添加新的功能，或者要求你修复 API 的 bug。先看一下关于 bug 修复这种情况。如果一个类库已经对外发布了，那么正如 4.2.3 节中所说，你可以理直气壮地说，程序这样做不是一个 bug，只是一个功能。当然要想这样说，也要看一下具体的问题。比如说，从代

码中抛出一个 `NullPointerException` 这样的异常，肯定没有人认为这是一个功能了。另一方面，即使你 API 中使用的算法性能比较低，也可以算是一个功能，就像 10.4 节中所说，程序可以用来计算阶乘。具体的处理方式，要看身为作者的你到底把兼容性放在一个什么样的高度了。如果你认为兼容性非常重要，就要将整个程序的阿米巴变形虫效应降到最低，正如本书所一直强调的。站在这个角度，那么每一个 bug 事实上都可以看作是一个新的 API，因为简单地进行修正，其实是影响了先前版本的功能，可能有他人已经依赖了这个功能。稍后我们会继续探讨这个问题，现在我们先来看看对需要 API 新功能的抱怨。

用户往往以对新功能的需求而著称。他们对于 API 也是如此。不仅如此，很多用户还以抱怨而著称，不管你根据需求做了多少有益于他们的事情，他们还在不停地抱怨，API 的用户也不例外。从另一个角度来说，开放源代码很有好处，特别是你的一举一动都为众人所关注时，更应该这样做了。如果你已经把源代码开放，并以文档的形式告诉大家如何在源代码中加入自己的调整，那就好办多了。如果这时还有 API 的用户在抱怨，你只要回复一句：“小事，你可以自己搞定。”此时，你的 API 用户只需要行动起来，就没有那么多的抱怨了。用户可以花上一些时间来给 API 添加新的功能，或者深入分析他自己的代码。采用这种方式，对于类库的维护者来说，他们根本不需要做什么事情。最多是指导一下用户，在向源代码提交之前要遵守哪些原则就是了。

假设 API 用户已经花费了大量的时间来解决自己的问题，然后决定做一个补丁，并将其整合到源代码中。现在需要花点时间来评审一下这部分内容。此时，需要有人来通篇审阅提交的内容，以检查质量是否满足要求。据我的经验，这份评审的工作非常不错，因为你所审阅的内容可以提升你的 API。当然，如果这个补丁的质量不高，那么就尽快地拒绝该补丁。此处我可以给出一些小技巧。判断一个 API 的修改是好还是坏，有一个必要的前提条件是看看这个 API 对其功能是否有足够级别的描述，就像我在 4.3 节所说的那样。如果一个补丁做得不好，往往就会忽略这方面的内容，所以你可以要求作者添加一个合适的用例对此加以说明，然后以这个理由拒绝补丁的提交。还有一点也需要加以关注，就是那些用来修正 bug 的补丁。没有人喜欢因为接受了一个修正 bug 的补丁，就不得不在接下来几周的时间里修复因为这个 bug 而引发的其他 bug。进行代码评审自然是一个解决方式，但这样做太占用时间了，你需要去理解修正内容中的代码。可以通过全面的自动化单元测试来检查代码是否正确。你其实不用逐行去通读他人的代码来进行评审，只需要让他人在提供补丁的同时要提供一个单元测试以保证其正确性。如果说能够提供一个单元测试，那么至少可以保证对这个 API 的改变是有效的。你可以要求有更多的单元测试，这样可以增加你对 API 修改的信心，而且可以证明这个提供补丁的人很上心。他们上心当然是一件好事，这也表示，如果以后再出现 bug，该作者也会加以修正。即使他们以后不再对 bug 进行修改，你也可以用前面介绍过的技巧，那就是将这个 bug 看作是一种功能（一切都是为了保证兼容性）。

人人都知道，把新功能添加到一个 API 中，所花费的成本是可以做到很低的。只需要每个项目都支持代码评审。可以有一个邮件组和一个 bug 跟踪系统，告诉他人如何来编写相应文档。项目可以使用自动化测试，并要求每一个提交代码的人都编写测试用例。据我所知，很多项目都已经做到了这一点。当然，NetBeans 和 Apache 旗下的 Ant 都使用了这种方式来处理外部提供的代码。

在建立完前面的环境以后，我们只用处理唯一剩下的问题：如果 API 中有 bug，而且这些 bug

不能看作是功能需求，怎么处理呢？没错，这些 bug 确实需要你的维护。但你可以减少它们带来的影响。第一个有效的方案是要求报告 bug 的人提供一个自动化测试用例，该测试用例不能涉及其他无关的 API。当然这种要求不可能什么时候都做的到，但大部分情况下能做到。只要有人提交这些内容，对维护还是有所帮助的，这样你就可以将精力集中在那些使用了你的 API、并有诚意帮你改进 API 的用户身上。如果开发人员能编写单元测试，也就意味着他们有足够高的技术能找到你类库的源代码，并知道如何构建源代码，还知道如何运行测试用例。而且他们还很乐意花点时间做这些事。对于开发人员来说，与这些人一起工作，修正程序的 bug 往往是一种乐趣。在这种情况下，修正他们提出的问题，就和实现新功能差不多，都是一件很容易的事情。这样也可以有效地降低维护成本。

是不是还有一些可能会增加维护成本的情况，但我们忘记说了呢？的确是，有时候，想写一个与其他内容无关的自动化测试用例是件不太可能的事情。比如说，有人付钱给你，想让你修复程序中的一个问题。此时，可能没有针对该 API 的单元测试。对于一个对该 API 不熟悉的人来说，想做这事是比较困难的。或者这个 API 出现问题的方式不确定，很难进行跟踪。所说的这些情况都有可能出现。但不管你写什么代码，也都可能出现类似的情况。比如说你老板让你做个功能，你做了，但没有写测试，当然，缺少单元测试不管是对无绪还是对可靠性来说，都不是一件好事。如果你的代码出现的问题具有不确定性，也同样难以定位 bug。所以对于平时编写代码来说，也是存在这种情况的，和写一个 API 类库并没有什么本质上的区别。如果你写的代码比较差，怎么都会有维护上的问题。

以上就是我努力想证明的内容。如果创建 API 的方式正确，那么维护它的成本并不比维护普通代码来得更高。事实上，维护 API 类库代码可能还更容易一些，因为你可以借助于正确的 bug 报告对其进行自动化测试，而对于普通代码来说，其用户就不会提供这些内容了。唯一的条件就是你开发 API 的方式必须正确：为将来的改进预作准备、通过好的测试来减少阿米巴变形虫效应、并基于用例文档进行开发。在第一个版本发布之前，这些要求都应该做到。如果没有达到这些要求，特别是如果这个 API 没有为将来的改进预作准备，到问题出现时就已经太晚了。所以这一点才是对 API 进行评审的最大动力所在。如果对一个 API 的评审和设计都做得很好，那么对它进行维护就是小菜一碟了。



上一章是要说明，相对于维护非 API 类库代码，维护 API 类库代码的成本并不会高出很多。但这并不意味着给自己的代码添加 API 是一件简单的事情，绝对不是。但前面的事实已经证明，如果你的代码质量已经达到一定水平，尤其在遵循了本书给出的有关 API 改进建议的情况下，向现有代码添加 API 并不会给维护者带来很大麻烦。

尽管如此，我们并不是总能编写出高质量的代码。如果出现了质量不好的代码，那么即使给这些代码戴上一个 API 的帽子，也解决不了问题，质量不好仍然是个事实。但因为 API 就像恒星一样，质量不好的代码会暴露在所有人的面前。

差的代码天天有，坏的 API 也不少。没有达到高可靠性的原因可能是我们并没有将针对性无绪的思想集中于测试，即编写足够的用例对 API 运行时各个方面做全面测试。也许原来的需求已经有所变更，也许原来的项目一开始就走错了方向，没有什么实际意义。但不管原因是什么，结局毫无疑问都是一个我们并不想要的 API。我们想放弃这个 API，但还需要保护用户现有的投资。如何来做到这一点呢？

很多开发过内部系统的程序员都知道，有时候是可以对代码库进行翻天覆地的变化，虽然这样做并不总是合适。就好像关闭现在的店铺，进行重新装修，过段时间再对外营业。在店铺关闭的时候，内部是空的，你可以做任何自己想做的事情，即使有些非常危险。比如说直接停运所有电梯，让所有人爬楼梯；花点时间重新铺所有的地板。不管做什么，只要你清楚所有要做的事情，就可以估算出做这些事需要的时间，保证可以按时“重新开业”。

很多人都认为规模不是一个重要的因素：如果某类事物可以在小系统中使用，那么就算把它们放到大的系统中，也仍然可以正常使用。其实这种说法是不对的。API 就像是恒星：一旦发布，就始终会被关注到。像上面说的那种“停业整顿”对于这类大的系统来说，是做不到的。你可以把自己的店铺关上一段时间，但没有人能够让全世界都同时停止工作。所以想在放弃整个 API 的同时，让所有用户都放弃使用这个 API，显然是件不可能的事情。再强调一次，API 就像是一颗恒星，不可能突然消失。它也许会改变自己的运行轨道，也许会变为超新星并最终爆炸，但这些变化都不会在眨眼之间就出现，总会有一个过程。如果 API 在设计时能够遵守一些优秀的设计准则，那么它也会随着时间而不断演化。

宇宙的掌控者

如果你得罪了用户，那么你就会失去这些用户。假设你要采用那种推倒重来的方式来调整 API，那么你就得罪了正在使用 API 的用户。如果你的 API 就这么消失了，那么用户的全部功能将无法继续正常工作，要全部重写。也许重写这个 API 会给用户带来好处，也许可以让用户摆脱现有功能的丑陋实现，也许可以用更新更好的版本来替换原有的代码。但你的用户至少有权问一句：“为什么现在要做这事呢？不能再等等吗？”他们也有权利抱怨，因为他们认为还有很多更重要的事情要先做，为什么不把这种改进的事情向后放放呢。一定要知道，对于 API 的用户来说，他们总认为自己有更重要的事情要做。

你可以听取用户的意见，然后推迟计划，暂不进行大的调整，也可以考虑给他们一个长回复：“你们其实不应该期望我们推迟该计划。因为对于整个项目来说，放弃旧的 API 而采用新的 API，可谓成败攸关。也许你对其持有不同意见，但从全局角度来看，我们建议你最好还是停止现在做的事情，开始重写你的代码！”如果觉得这话太啰嗦，那么简短一点表达你的意见：“谁管你啊，听我的，没错。”但毫无疑问，这样一种傲慢的态度会伤害用户的感情。他们会考虑采用其他的方案来替换你的 API。

你可以向他们解释，这样的调整对他们来说也是一件好事，他们可以借此来理清自己的代码。但在 4.6 节的“软件熵”那段文字中已经清楚地指出，你不能寄希望于仅通过重写就能使得你的代码变得更好。也许一开始重写的版本很漂亮，但经过多年的维护也就面目全非了，除非开发人员从根本上改变他们的编码习惯。如果要进行这么大的调整，压力一定很大，想做到这一点也是非常困难的。

我知道很多人都觉得进行一次这样大的调整才是最好，也是最有效的处理方式。其实在 NetBeans 的 API 开发过程中，我已经见过几次这种情况，总是假设这些 API 只有 NetBeans IDE 的内部项目使用。但这种假设，只适用于那种小的内部系统，对于 NetBeans 来说，它的规模已经很大了，进行如此之大的调整会花费很长的时间，远超预期，花费自然也会超出预算。更麻烦的是，这种估算并没有包含我们的合作伙伴重写代码的成本，而且是假设我们不会为重写版本引入新的特性。而且我担心原来的那种软件熵还会再次出现，和以前一样：代码中存在很多问题，但考虑发布日期，只能妥协。

所以对这种重写，我个人是持怀疑态度的，觉得这种方式其实是一种消极的方式。我不觉得这样的一个系统，只归属于具体设计者，它是设计者的系统，更是用户的系统。

如果一个类库和相应的 API 设计得不好，此时你有两个选择：一是就像处理其他代码的方式一样，尽量去修复问题；二是干脆一点，从头开始写一个新版本，放弃旧版本。但第二个处理方式并不意味着要移除旧版本的类库。可以考虑以 deprecated 的方式来标识该类库已经被弃用，在使用该类库时可以考虑给出警告以提醒用户，但这个类库必须保留，就像前面所说恒星一样，直到它自己消亡。

15.1 让有问题的类库重新焕发活力

先来看一下让一个有问题的类库重新可用的两种方式之一——修复。修复类库是一件非常困

难的事情。我见过，也曾经亲身参与并维护过一些设计比较差的类库。这些类库提供的接口都设计得不易改进。它们提供的文档也非常少，而且在多线程环境下，还会经常出一些莫名其妙的问题。我必须承认，其中有几个类库是我设计的。我在年轻的时候写了这些类库，当时根本意识不到“无绪”这个概念。

想修正这些乱七八糟的问题只有一个办法，就是写测试用例。我用“无绪”的思路来做这件事。我不知道一个功能类库到底有什么功能，但在了解它之前，我要先写一个测试来证明这个类库的确有问题，然后我才修正类库的问题。做这种工作不需要什么技巧，实在是很无聊。但一天做下来，我发现自己对类库引入了一些小的改进。最终，整个类库的可靠性已经有了很大的提高，与旧版本不可同日而语。虽然改变不小，但仍然可以很好地保证兼容性。至少，每一个类库的新版本都能做到对旧版本宏观方面的向后兼容，也就是说原来版本完成的基本功能，新版本也能做到。但对于旧版本微观方面的向后兼容就不能保证了，因为原来的代码太乱了，在一定程度上无法准确定义其行为到底是什么，在这种情况下，根本无法确定具体行为是否兼容。我想说，我已经尽我所能解决问题了。但这样其实可以算作说谎。帮助类库优化其设计是我的一个强烈意愿，真正要做的是努力工作，小心地写一些测试来验证每一个修改。

为自己的 API 而战

每一个 API 都能被修复到某种可靠程度。问题在于需要花费多少精力来做这件事，长远看花费的时间是否值得。随之而来的问题则是，你能否找到一个人，并说服他来做这事。对于修复他人所犯的错误，开发人员都不喜欢做这事。对于遗留代码的维护者来说，也很难让他们能够把代码改得比原来还好。但如果作为一个 API 的原作者，整天被人说 API 不可用，也不可靠，那么就有足够好的理由来证明你自己能做好这件事。为自己的 API 而战！有了这个念头，这件事情肯定能搞定。

我也曾经历过同样的事情。在 NetBeans 中有一个 API 有很多问题，功能很大，经常会随机性地出些 bug。这个 API 在整个系统处于比较核心的位置，不可能被轻易地移除，我们试过多次都做不到，但这事对我来说却是一件好事。因为这样我才有机会来修复这个问题。为了尽可能地做到“透明”，我写了很多测试进行检验，即使是对那种最简单的改变，我想方设法让这个 API 变得可靠起来。修正所有随机出现的问题花费了我两三年时间。现在，这个 API 的功能都已经正确了。或者至少可以说，没有再发现什么重大的 Bug 了。

如果你努力的目标是为了自己，那么你可以修正任何 API 中的问题，同时还能保证其向后的兼容性。但这样只表示你从“工程”的角度修正了该 API。你可以让这个 API 不再存在 bug，也可以让这个 API 变得更加可靠，还可以写更多的文档来支持该 API，而且让该 API 更加易用。但你无法让这个 API 变得“优雅”。当然对于工程师来说，优雅算不上什么大事。尽管那样，有时还是对“优雅”抱有期望。对漂亮、美丽的追求已经根植于我们的灵魂之中。也许无法具体的量化，但我们一直都努力地将事物变得更加优雅。通过兼容的方式来修正旧的 API，我们只能得到很多弃用的内容，但优雅却毫无踪影。

这也是为什么我们一旦知道了某个架构已经很可靠了，就会从头开始写一个新的 API，来

替换原有的 API。我们可以用桥接的方式使用旧 API 的功能。新的 API 会为整个程序带来优雅，同时还能让老版本 API 的用户的代码继续兼容。

最有效的一种保障手段就是通过编写测试，验证其与上一个实现版本之间是否兼容。在你想保留一个 API 类的前提下，用新的更少 bug 的实现替换已有内部实现，那么这种兼容性测试非常容易进行。假设某个类出现了一些维护性问题，你需要提供一个更可靠的实现版本。当然，依据本书所倡导的理念，你应该知道虽然增量改进可能会相对缓慢，但是会减少用户的顾虑。因此，依照之前提到的阿米巴虫模型，你应该尽量少地改变 API 的对外可见行为。旧版本的对外行为应该视为我们调整所应遵守的功能规格，不能随便更改。我们的目标是在完全替换内部实现的同时，保持对外行为的相似。你可以将它看作是一个 TCK 的最低要求。我们需要一组基于旧版本的测试用例，然后在新版本上执行。如果每个测试在新旧版本之间返回的结果相同，那说明我们已经达到了目标，即在保持最大程度相似的前提下再造旧行为。

我们的第一步是将旧的代码实现从应用代码中移出，不是要删除这些代码，而是把这些旧的难以维护的代码作为功能模板，新的代码实现的功能需要与此对外保持一致，示例如下。

/** 下面的代码复制于 Arithmetica 接口第一个版本的实现 */

```
static class OldArithmetica1 {
    public int sumTwo(int one, int second) {
        return one + second;
    }

    public int sumAll(int... numbers) {
        if (numbers.length == 0) {
            return 0;
        }
        int sum = numbers[0];
        for (int i = 1; i < numbers.length; i++) {
            sum = sumTwo(sum, numbers[i]);
        }
        return sum;
    }

    public int sumRange(int from, int to) {
        int len = to - from;
        int[] array = new int[len + 1];
        for (int i = 0; i <= len; i++) {
            array[i] = from + i;
        }
        return sumAll(array);
    }
}
```

这个测试代码里有一个 compare 方法，方法中会随机生成数字分别在新旧版本的代码中执行操作。如果每次运行时，这两个测试结果都相同，那么就可以认为旧版本的原有功能，新版本都能完全匹配，详见下面的 compare 方法。

```

private void compare (Arithmetica now, OldArithmetica1 old, long seed)
throws Exception {
    java.util.Random r = new java.util.Random (seed);
    for (int loop = 0; loop < r.nextInt(5); loop++) {
        int operation = r.nextInt(3);
        switch (operation) {
            case 0: { // sumTwo
                int a1 = r.nextInt(100);
                int a2 = r.nextInt(100);
                int resNow = now.sumTwo(a1, a2);
                int resOld = old.sumTwo(a1, a2);
                assertEquals("sumTwo results are equal", resNow, resOld);
                break;
            }
            case 1: { // sumArray
                int[] arr = new int[r.nextInt(100)];
                for (int i = 0; i < arr.length; i++) {
                    arr[i] = r.nextInt(100);
                }
                int resNow = now.sumAll(arr);
                int resOld = old.sumAll(arr);
                assertEquals("sumArray results are equal", resNow, resOld);
                break;
            }
            case 2: { // sumRange
                int a1 = r.nextInt(100);
                int a2 = r.nextInt(100);
                int resNow = now.sumRange(a1, a1 + a2);
                int resOld = old.sumRange(a1, a1 + a2);
                assertEquals("sumRange results are equal", resNow, resOld);
                break;
            }
        }
    }
}

```

这里使用随机数进行测试是比较合适的，因为它们生成的随机数保证执行顺序的随机性，而且同时调用这两个具体的实现进行验证。在这个程序的执行过程中，可以验证这两个具体的实现能否提供相同的结果。除了使用随机数进行测试，还可以使用其他方式来进行测试，两个版本依然应该得到相同的结果。另一方面，使用随机数有助于检查阿米巴虫模型。在 4.2.3 节中已经清楚地指出，一个程序的真实行为总是与其对外公开的规范或我们期望的效果有所不同。一般来说，通过测试来自动验证规范中说明的功能是一种非常有效的手段。做起来也很简单：如果规范里定义说有一个功能是可以正常运行的，那么就写一个测试，执行测试，如果发现结果不对，就根据测试的结果来修正具体的实现。如果规范中指定了对于某些参数，函数在执行时会抛出一个异常，那么就写一个测试传入这个参数来检查是否真地抛出了这个异常，如果不对，就修正这个具体的实现。

从另一个角度来说，程序所做的事情可能不止规范中定义的内容。这种情况也是非常危险的，因为这样做会造成安全漏洞，或者对类库未来的改进加以阻碍。因为如果有一天，需要根据

规范对类库的某些功能加以修正,会发现修正问题的同时引入了一些与原有 API 规范中不兼容的问题,或者产生一些原有 API 规范中没有定义的行为。所以检查一个程序所能支持的功能是否比规范定义的功能还多,也是一件非常重要的事。只不过,这事做起来并不容易。常规的测试模式对此无能为力,因为常规的测试只会对预期的行为进行检查。所以这一点正是随机性测试的优点所在。从某种程度上讲,随机性测试可以发现非预期的功能,因为随机性测试能让代码按照一些不常见的路径来执行,而这些路径可能是程序员或者质量工程师无法写出来的。常规测试有一定的边界,不会触发新的代码执行流程,只能按照预先定义好的路径来执行。所以随机性测试是非常有用的测试类型,因为它能对 API 进行极限测试。

完全重写 CookieSet

NetBeans 项目组在重写 CookieSet 的时候,使用了一种新的加强的方式来重新实现 CookieSet,并提供了很多新功能。原有的实现则被复制到一个称为 OldCookieSetFromFebruary 2005 的测试用例中。这个用例用来比较新的实现与原有的实现的相应行为,通过这种方式来判断新的代码实现能否与原有代码实现保持一致。我们把它看作一种非常有效的兼容性验证方式,因为通过这种测试,我们可以不急着将新调整的 CookieSet 整合到系统中,而是等所有的兼容性验证都通过,才进行整合。

使用随机性测试,有时可能会适得其反。任何的错误都是一个随机性的错误,而且可能很难跟踪和重现。但其实这种所谓的随机性错误不是真正意义的完全随机性错误。下面的 compare 方法使用了一个初始的随机种子,当出现问题的时候,就把这个随机种子给打印出来。这样在必要的时候,可以重现每一个测试,参见下面的代码。

```
public void testRandomCheck () throws Exception {
    long seed = System.currentTimeMillis();
    try {
        CountingSubclass now = new CountingSubclass();
        CountingOldSubclass old = new CountingOldSubclass();

        compare(now, old, seed);

        assertEquals(
            "Verify amount calls to of sumRange is the same",
            now.countSumRange, old.countSumRange
        );
        assertEquals(
            "Verify amount calls to of sumAll is the same",
            now.countSumAll, old.countSumAll
        );
        assertEquals(
            "Verify amount calls to of sumTwo is the same",
            now.countSumTwo, old.countSumTwo
        );
    } catch (AssertionFailedError ex) {
        IllegalStateException n = new IllegalStateException (
```



```

        "Seed: " + seed + "\n" + ex.getMessage ()
    );
    n.initCause(ex);
    throw n;
} catch (Exception ex) {
    IllegalStateException n = new IllegalStateException (
        "Seed: " + seed + "\n" + ex.getMessage ()
    );
    n.initCause(ex);
    throw n;
}
}

```

上面的那个测试根据时间种子执行一系列随机顺序的操作。每天这个测试都会表现出不同的行为。如果出现了问题，它就会把原始的时间种子打印出来。这样，如果想重现这个出错的测试用例，那么就用这个特定的时间种子来执行测试用例，这样就能保证两次执行的顺序是一样的，可以总是重现这个错误。下面的代码用来测试 2008 年 4 月 13 日的一个问题。

```

public void testSimulateOKRunOn1208120436947() throws Exception {
    CountingSubclass now = new CountingSubclass();
    CountingOldSubclass old = new CountingOldSubclass();

    compare(now, old, 1208120436947L);

    assertEquals(
        "Verify amount of calls to sumRange is the same",
        now.countSumRange, old.countSumRange
    );
    assertEquals(
        "Verify amount of calls to sumAll is the same",
        now.countSumAll, old.countSumAll
    );
    assertEquals(
        "Verify amount of calls to sumTwo is the same",
        now.countSumTwo, old.countSumTwo
    );
}

```

基于这样的一种架构，我们就可以很容易地重现那些看起来像是随机出现的问题，只要有必要就可以重现。只需要记住这样的随机种子数，然后用下面的代码执行一下测试用例，只要使用的随机种子数与出错时的一样，那么用例会在执行过程中重现当时的问题。

```

public void testSimulateFailureOn1208120628821() throws Exception {
    CountingSubclass now = new CountingSubclass();
    CountingOldSubclass old = new CountingOldSubclass();

    compare(now, old, 1208120628821L);

    assertEquals(
        "Verify amount of calls to sumRange is the same",

```



```

        now.countSumRange, old.countSumRange
    );
    assertEquals(
        "Verify amount of calls to sumAll is the same",
        now.countSumAll, old.countSumAll
    );
    assertEquals(
        "Verify amount of calls to sumTwo is the same",
        now.countSumTwo, old.countSumTwo
    );
}

```

这是一种可以接受的方案，但它允许你长期支持原先那些实现代码的具体行为。一旦有新的 bug 报上来，或者要添加一个新的功能，你就需要写一个新的测试用例来校验新代码在行为上能够与旧代码保持一致。对于这种架构来说，你可以让它慢慢地变得透明，因为你不再需要花费大量的时间来理解、阅读以及改进原有的代码，只需要从外在行为上调整新的代码，一切也就都好了。

前面的方法可以很有效地帮助开发人员修复 API 内部的问题，而且从外面看起来，这个 API 没有任何改变。如何在修改 API 的时候，不给现有用户带来问题呢？当你需要引入新版本 API 的时候，可以把旧的 API 标识为 deprecated，表示不再建议使用。但最重要的是，新旧两个版本的 API 必须可以共存，而且可以共同正常运行。考虑到分布式开发自身的特性，而且新功能的开发也是由不同小组开发的，因此不太可能指望所有的团队或者 API 用户能同时从旧版本更新到新版本上。有的人会觉得其他事情更为重要，更新 API 版本的事情可以向后放一放，继续使用旧版本的 API 一段时间。所以不同版本 API 的共存，对于 API 的改进来说，是非常有必要的。

15.2 自觉地升级与无意识地被迫升级

前几节已经说过，保证向后兼容性有时是非常困难的，即使对类库进行小的调整，也可能对整个系统功能带来可怕的影响。就像上文讨论过的，这样小的改变可能会引发大的地震，甚至可能让 API 的用户对你完全失去信心，他们不再相信你的技术水平。

从另一方面来说，改正问题也是非常重要的。假设你的 API 中有一个 Bug，那么就应该修正这个 Bug。如果放任这个老的 Bug 不管不问，不去修正它们，那么同时也会让你 API 的用户对你失去信心，这与升级新版本而引发问题所造成的后果是一样的。在设计通用类库的时候，兼容性是一个需要着重考虑的方面。同时修正已经存在的问题也非常重要。API 设计的艺术就在于当这两者产生冲突时，能够从中取得平衡。

兼容性是一种约束

当我加入了 Sun 公司以后，有人告诉我说：“前 CEO 斯科特·麦克尼利 (Scott McNealy) 总是说‘不应该把兼容性当作目标来看，而应该把它看作一种约束。’”在 Sun 公司，这句话可谓众人皆知，而且我的很多同事都能大声地将此诵读出来。只不过，说归说，做归做，至少在我打过交道的部门里，没有谁能做到这一点。不要误会我的意思，做到向后兼容肯定是一件好

事，但绝不可以将它作为一件额外的工作或者是一个目标。也不应该把它看作是一种约束。

我花了很长时间才把这话理解透。现在我觉得斯科特的话有其历史背景。有一段时间，Sun 公司面临着其他像 IBM、HP 等这样的大公司的挑战，它们也提供其他的产品和 Sun 公司开展竞争，斯科特的这话是说给那些 Sun 操作系统和 Solaris 用户以及相应开发人员听的。为了说服用户，使用一个新的操作系统是一件合理的事情，需要证明相应的产品有更好的质量。在这种情况下，兼容性就体现为质量的一个重要标识，因为通过兼容性的保证，能表示公司非常看重用户的前期投资，所以说兼容性是一个约束。

从那以后，作为产品的一个外在表现，在其他的产品中，包括 Java、NetBeans 等，都体现出了这一思想。其实对于最终客户的程序来说，其质量要求并不需要像对操作系统的要求那么高。就像你所用的邮件客户端软件一样，你可以忍受它时不时出点儿错。这种情况下，这种类型的错误是可以接受的。但对于操作系统来说，这样的错误可能就意味着一种灾难。

斯科特从来没有说兼容性不再是一种约束，对于 Solaris 来说，兼容性仍然是一个约束。但对于新产品，或者已经有了固定用户群的产品来说，就没有人把这句话当作金科玉律来看了。这也是正常的。兼容性毕竟与用户有关，只要产品的质量能够满足用户的预期，那就不会有大的问题。但也必须清楚一点，它也有自己的底限。如果用户害怕升级到新版本，那么就说明肯定有哪些地方出问题了。

我曾经观察过，有些人更容易接受产品的不兼容性。这类用户往往能够很理智地使用产品。他们会感觉自己是决策环节中的组成部分，如果没有他们，就无法做出最终决策。这种强烈的参与感，使得他们在问题出现时，仍然可以控制自己的情绪。

对于某些情况，比如程序抛出了一个 `NullPointerException` 异常，可以认为，没有哪个用户真正指望这种抛出的异常。用户会很高兴接受一个给出结果而不是抛出异常的新版本。在这种情况下，期望的是对类库的功能进行调整，这其实是一种无意识的升级。这样做，其实是给用户提供了一个新版本，而且该版本的功能与旧版本的功能其实已经不一致了，但用户不会意识到这一点。当然，如果有人很关心这点的话，那么还是应该告诉他们，新版本在某些功能上有了一些改变。但只要你原来的评估是正确的，而且没有人会通过异常进行各种业务处理，那么就没有哪个用户去关心这个具体行为的改变。他们会很开心地接受新版本，根本不关心新版本与旧版本相比之下的改变。

在后台加载编辑器中的内容

在 NetBeans IDE 6.1 版本中，我想对编辑器加载文本的方式加以改善。我不想使用那种线程堵塞的方式，我把代码改了一下，在切换到事件队列、显示编辑器前，先在后台程序中把相关的内容都准备好。

这其实是个不兼容的改变，我也正式标注说明了这是不兼容的。我本以为这只是一个很小的调整，不会有太多的负面功能影响，和那种修正 `NullPointerException` 的 Bug 差不多。但事实上，我错了。我很快就收到了一大堆出现不兼容性的报告。问题出在 `editor` 类是可被继承

的，很多人继承了这个类，然后向其中添加了各种稀奇古怪的操作代码。我看了一些这样的代码，真是大开眼界，事实上我所作的调整不可能考虑到这些情况。

最后，我们想办法把代码稳定下来但保留了调整，让外面看起来像是修正了一个严重的 Bug。如果再有机会从头开始，我会把后台加载做成 API 的一个新功能，然后让所有的用户很清楚地知道这件事，再调整他们的代码。

有时候，即使某个错误的行为对用户来说是有用的，仍然得去改变这种错误的行为。也许是想去掉特殊的方法调用来提高性能，也许是想用另外的线程来异步处理 Listener 的事件通知，也许还想调整一个方法内部的代码顺序，原因可能很多。在原来的版本中，这些功能可能都是有问题的，就像前面所说的那个抛出异常的例子。但一定要注意，不能因为觉得这样的代码有问题，就认定用户没有用到这些功能。有些用户可能已经在这些功能的基础上开发自己的程序了。如果升级到新版本，很明显，整个系统的行为就与原来不一样了。那么，责任到底是谁的呢？对于负责集成程序的人来说，他只是简单地把类库升级到新版本，突然间一切就都完蛋了。对于这种情况，可以很容易就下个结论：问题一定是类库提供者的，没有什么可以推卸的了。也许你会说这不公平，因你做的事情很简单，就是修正原来的一些错误。但这就是 API 设计者的生活。

要解决这类问题的方式就是在修正错误行为的时候，要有条件地进行修复。这种做法可以分成几类。但有一个基本原则：除非 API 的用户修改了自己的源代码，否则原有的代码即使在新的类库上运行，也仍然是正常的。如果 API 的用户想启用新的行为，就需要调整少许的代码。比如说，如果 10.4 节中那位 *Arithmetica* API 的作者想在修改代码的同时，还能保证阶乘算法的兼容性，那么可以提供一个新的构造函数，这样就可以让用户来选择使用哪种方式了。

```
public class Arithmetica {
    private final int version;

    public Arithmetica() {
        this(1);
    }
    public Arithmetica(int version) {
        this.version = version;
    }

    public int sumTwo(int one, int second) {
        return one + second;
    }

    public int sumAll(int... numbers) {
        if (numbers.length == 0) {
            return 0;
        }
        int sum = numbers[0];
        for (int i = 1; i < numbers.length; i++) {
            sum = sumTwo(sum, numbers[i]);
        }
    }
}
```

```

        return sum;
    }

    public int sumRange(int from, int to) {
        switch (version) {
            case 1: return sumRange1(from, to);
            case 2: return sumRange2(from, to);
            default: throw new IllegalStateException();
        }
    }

    private int sumRange1(int from, int to) {
        int len = to - from;
        if (len < 0) {
            len = -len;
            from = to;
        }
        int[] array = new int[len + 1];
        for (int i = 0; i <= len; i++) {
            array[i] = from + i;
        }
        return sumAll(array);
    }

    private int sumRange2(int from, int to) {
        return (from + to) * (Math.abs(to - from) + 1) / 2;
    }
}

```

显然，这样做可以有效地保护那些用户已经写好的代码，即使他们写的代码非常古怪，就像这个计算阶乘的例子。同时，这样调整以后，对于那些希望使用 2.0 这个新版本来提高算法性能的用户来说，也就不存在问题了。默认情况下，2.0 版本的功能与 1.0 版本是一致的，不会因为升级而带来问题。如果想提高性能，只要略略调整一下代码就好了。对于用户来说，这是一件非常清楚的事情。如果用户决定这样做的话，他们就不会抱怨这种不兼容性，因为他们很清楚自己需要修改相应代码。对于软件工程师来说，他们都知道，只要对代码进行修改，就有可能带来问题，他们是可以接受这一点的。

如果一个 API 用户想要升级到新版本，那么有很多方式来帮助他们更加清楚地了解升级中的一些问题。我不打算现在就这个问题进行讨论。只要让读者知道，升级时有两种方式，前者是用户根本不清楚升级会对兼容性造成破坏，而后者则是让他们明白这一点，而这两者的区别是非常显著的。

15.3 可选的行为

从某种角度来看，设计了一个类库，并在维护时保持其兼容性，总是面临两种选择。一是为类库添加新功能，二是为类库现有功能提供可选的行为。在这两者之间并没有明显的边界。事实

上,为类库添加新的行为和添加新的功能很相似。添加新的类或者新的方法都可以看作是在现有行为上添加了新的行为。很显然,想在这两者之间划一条泾渭分明的线是一种不科学的做法。因为不管是添加新的功能还是添加可选的行为都确实对类库有所改变,对它们进行分类更多是出于心理上的因素。结果可能只是体现在这些改变对先前功能的影响到底有多大。当添加一个新功能的时候,破坏原有功能的可能性是比较小的。而引入一种新的可选行为时,我们认为因为原来已经有了类似的功能,如果再引入的话就可能带来比较大的负面影响。

上面的解释可能有点过于哲学化了。现在从技术的角度来看一下,引入新的行为时,如何能尽量减少对现有行为的影响。最简单的方式就是提供一个新的 API,它与原来的版本没有任何关系。只需要保证这个新的 API 与原有的 API 可以共存就行了。如 `java.lang.Math` 和 `java.lang.StrictMath` 这两个类。但这两者之间并不是兼容的。在 Java 的早期版本中,当时并没有 `StrictMath` 这个类,它的功能其实是放在 `Math` 中的。随着 Java 程序在多个平台上的发展,对严格意义的数字精度的要求开始下降。开发人员认为提高计算速度比兼容性更为重要,他们愿意牺牲数字的绝对精度和兼容性来换取高性能的计算速度。所以从 Java 1.3 开始,把高精度的运算放到了 `StrictMath` 中,而 `Math` 则不再保证标准的数字精度,这样 `Math` 可以充分地利用本地处理器的性能。如果忽略这些调整带来的痛苦,那么这种方案完全可以算得上是一种安全的可选方案。

通常需要在 API 中提供可选行为,主要是因为开发人员希望能够尽可能以一种安全而且兼容的方式来修正 Bug。考虑到性能提升(就像 `Math` 这个类一样),或者有改变现有行为的需求,都可以这样做。这时既要考虑改进的需要,也要考虑稳定的需要。技术上的实现会有所不同,因为要考虑执行了变化后的结果好坏,以及在多大范围内可以启用或禁用某些可选行为。最大的一个范围是由已执行的应用定义的,对于 Java 程序来说,就是运行的虚拟机决定的。引入环境变量是有条件启用可选行为的非常自然的一种方式。在 Unix 程序中,环境变量可以影响当前用户的工作目录,当前操作系统使用的语言等。对于 Java 来说,甚至可以影响更多的功能特性。这里,我们不用环境变量,而是使用 Java 自带的系统属性^①。(下面的代码就是通过判断系统属性 `arithmetic.v2` 是否为真来决定是否启用新的可选行为。)

```
if (Boolean.getBoolean("arithmetic.v2")) {  
    return sumRange2(from, to);  
} else {  
    return sumRange1(from, to);  
}
```

如果在修复 Bug 的时候,能够通过全局属性值来为其定制一个开关条件,这样就可以有效地避免任何与旧版本不兼容的问题。用户清楚地知道可以通过设置系统属性来改变 API 的行为。此时用户就可以理智地分析是要接受一个这样不兼容的风险,还是设置系统属性保证与旧版本的

^① 原文为 `System properteis`, 其实就是通过 `java.lang.System.getProperty(...)` 能够取得的变量, 可以通过 `java.lang.System.setProperty(..., "...")` 来放置各种属性, 还可以通过 `java -DVariable=true` 这样的启动方式来添加各种属性。其实 Java 已经 JDK6 中支持环境变量了, 但主要使用的仍然是系统属性。——译者注

兼容呢。如果用户不知道有这样一个全局属性，也就不会为这种属性来设置一个值，那么 API 可以沿用旧版本的行为，可以保证兼容性。但这样做仍然有一个风险，就是你所用到的属性名可能已经被用于其他目的了。不过，通过合理的命名规则可以将这种风险降低为 0。

NetBeans 平台 5.0 版本和 5.5 版本的绝对兼容

NetBeans IDE 在发布的时候，会把 NetBeans 平台^①作为其中的一个部分同时发布。事实上，NetBeans 平台的发布周期与 NetBeans IDE 并不同步。比如说，NetBeans IDE 的 5.5 版本与 5.0 版本相比，有着很大的区别，5.5 版本支持了很多新功能，特别是在企业开发方面有了长足的进步。除此之外，在 5.5 版本发布之初就会声明，发布新版本不会对系统的其他部分产生任何影响，只是修正了一些 Bug。此时，我觉得是时候尝试一下“开发时保持绝对兼容性”了。我说服了负责开发 NetBeans 平台的团队，让他们在修正 Bug 的时候要保证绝对的兼容性。

事情进展得算比较顺利。开发工作中有大量的“对抗”行为，因为每一次代码的提交都需要进行评审。事情之所以顺利是因为要修正 Bug 的数量不太多。我们能够对每一个调整而可能引发的兼容性问题进行评估。只有看起来不会破坏兼容性的代码才允许提交，我们在每一个修改的地方都加入了测试，以保证向后兼容性。

我们在写测试用例的时候，最常用的一个技巧就是系统属性。如果修正一个 Bug 有可能出现问题，那么修正该 Bug 时必须提供一个可选行为，由用户明确地通过设置某个属性来启用或禁用这个行为，而默认则沿用旧的行为。NetBeans IDE 则是在自己的启动过程中，将这些属性全部打开，以启用新的行为。而 NetBeans 平台则没有做这件事，所以它默认情况下仍然是沿用旧版本的行为，以保证向后兼容性。在这种情况下，我们可以在发布 NetBeans 平台新版本的时候，提供一些新的 API 和功能，同时还可以保持向后兼容性。

显然，修正 Bug 是非常重要的，兼容性亦是如此。考虑到这两者都很重要，我们决定在 NetBeans 平台的后一个版本中，让上一个版本中默认禁用的可选行为变成默认启用的行为。NetBeans 平台 6.0 版本是一个非常重要的版本，说它重要是因为我们不再保证每次 Bug 修复都有绝对的兼容性。我们首先在一个大版本中提供了默认禁用的可选行为，到下一个大版本则会默认打开。这样做，我们既可以保证高度的兼容性，同时也能有效地对系统加以改进。

如果谈到两种可选行为共存的时候，那么这种基于系统属性的解决方案就不再适用了。比如说你既想使用行为 V1，又想使用 V2 的行为，那么你就得启动两个 Java 虚拟机，而且设置不同的系统属性。对于一个基于 NetBeans 平台开发的模块化系统来说，是不可能这样做的。也许在同一个虚拟机中需要提供两个模块，每一个模块负责提供不同的行为。

在一个模块化系统中，更正确的处理方式是让每一个模块自己来选择。如图 15-1 所示，让

^① NetBeans 其实从 5 版本后，就和 Eclipse 一样分成了几个项目，其中有一个核心项目称为 NetBeans Platform，即 NetBeans 平台，其目标是支持基于 Java Swing 技术开发的桌面应用。而 NetBeans IDE 就是基于 NetBeans Platform 结合 NetBeans 插件而提供的一个编程开发环境。这一点和 Eclipse 的发展方式基本一样。——译者注

API 来猜测用户的目的，然后在正确的时间调用正确的实现。为了做到这一点，就需要利用一个模块版本依赖的声明了。比如说，假设有一个 API 想同时支持两个可选行为。第一个行为是在 2.5 版本中加入的，而第二个行为则是在 2.6 版本中加入的。不管是 2.6 版本还是其后续的版本，都需要提供与 2.5 版本相类似的行为，对于 2.5 版本用户^①来说，是感觉不到 2.6 版本有什么改变的。这样做只需要分析模块依赖的相关声明即可。

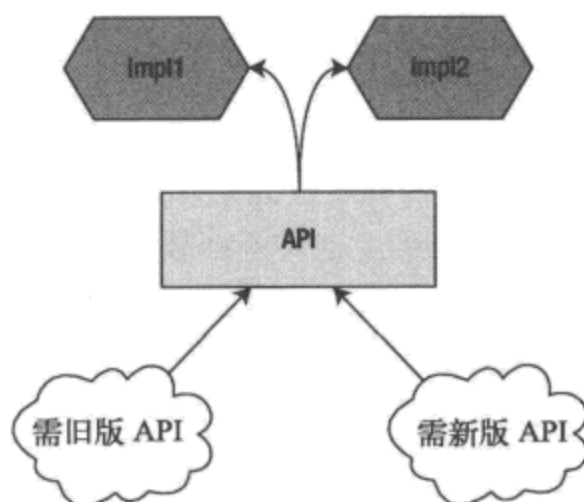


图 15-1 在运行时通过分析来选择具体行为

使用 2.5 版本或者更早版本的用户，在创建一个模块时，可以声明自己所依赖的某个 API 的版本号需要满足什么条件，如 `api >= 2.5` 或者 `api >= 2.4`, `api >= 1.7`……具体的声明内容，则要看用户何时创建和编译这个模块了。在一个模块化系统中，如 NetBeans Runtime Container (即 NetBeans 运行期容器，它是整个 NetBeans 体系的基础)，你可以在运行时获取一个模块的依赖关系声明，因为它通常会通过不同的 ClassLoader 来加载各自模块中的类。

```
StackTraceElement[] arr = Thread.currentThread().getStackTrace();
ClassLoader myLoader = Arithmetica.class.getClassLoader();
for (int i = 0; i < arr.length; i++) {
    ClassLoader caller = arr[i].getClass().getClassLoader();
    if (myLoader == caller) {
        continue;
    }
    if (RuntimeCheck.requiresAtLeast("2.6", "api.Arithmetica", caller)) {
        return true;
    }
}
return false;
}
return true;
```

调用类需要有一个依赖关系的声明。如果这个依赖声明说它所用的版本低于或等于 2.5 版本，那么这个模块就会提供原有的功能，兼容 2.5 版本。但如果这个依赖声明说它所用的版本高于或等于 2.6 版本，很显然这个模块是基于 2.6 以上的新版本来编译的。不管这个模块是一个升

^① 这里说的用户是指具体的模块，该模块声明了自己要使用 2.5 版本的 API，这样就可以通过分析依赖声明，知道该模块期望的行为，这时的版本号也同样扮演了系统属性的角色。——译者注

级到新版本的旧模块，还是基于新版本开发的新模块，此时就没有任何理由来使用 2.5 版本原有的功能，直接使用新功能即可。

想支持这种功能，其实只需要一点加载 Class 的技巧，特别是被调用者要去了解调用者所需要的版本号。尽管乍一听起来感觉有点复杂，但对于 C 语言来说，这其实只是一个很简单的动态程序 Linker 而已。你可以为一个函数提供两个版本，然后根据通用库的需求来告诉程序的 Linker 连接正确的函数版本库。使用这种方案，你可以调整 printf 的行为，同时还能在修复 bug 的问题上保证兼容性。有兴趣的话，可以去读一下 *How to Write Shared Libraries*^① 这本书，书中有很多这方面的内容。当然，对于 Java 来说，具体实施该方案的时候还是有所不同的。因为在 Java 程序中，程序 Linker 只会在连接程序时做一次绑定，因为 Java 出于性能的考虑，要在代码调用的时候进行检查。当然你可以用其他的方式来避免这种不必要的代价。第一种方式就是做二进制增强，模仿 C 程序库加载器，或者在 ClassLoader 一级玩些技巧，具体方式可以参见 19.2 节。还有一种方式，是在每一个运行时的容器上动脑筋，你可以优化容器的架构，将版本信息给缓存起来，加快查询版本号时的速度。虽然缓存的有效性也会发生变化，但对于那种大家最期待的状态来说，是很容易进行优化的，而且优化效果也很有效，在那种状态下，系统的模块只需要可选行为的最新版本。还可以考虑在加载模块的时候，就全部记录下来，这样在调用的时候，就不需要再通过方法去查询了。事实上这种基于版本号来提供相应功能的方案，同时兼顾了向后兼容性以及升级新版本。即使模块化系统中的模块还没有升级到新版本，它运行的环境仍然是它所期望的。当然，由于没有升级，就不能享受新版本带来的性能提升，可能会让整个系统的性能有所降低。但只要开发人员清楚地认识到新版本的调整，决定升级到新版本，即使不修改任何代码，只调整一下模块依赖的声明，那么这个性能问题也就自然而然地解决了。这样就为用户升级到新版本提供了动力，但同时又不至于强迫用户升级。至少从技术上来说，如果你的项目组中有一个团队专门负责性能，那么就可能会急于升级到新版本以提高性能。

关于运行时如何提供可选行为的讨论，就先告以段落吧。接下来，我们看一下如果想在代码编译的时候来提供可选行为的话，有哪些方式呢？其实最简单的方法前面已经说过了：只需要把一个类的内容复制出来，放到另外一个新的类中就可以了。就像 Math 和 StrictMath 这两个类一样，在其他情况下也能很好地运行。但一定要记住，Math 这个类非常简单——这两个类之间甚至没有任何共同的内容。但很多时候，事情并没有这么简单。有时候你需要为这些可选行为提供一个通用的接口，特别是如果你期望用户可以用相同的代码来使用不同的可选行为，就得这么做。比如说，使用 JDK 5 开发的工程师发现因为 StringBuffer 使用了同步，所以在处理字符串时的性能就不是很好，即使我们不需要在多个线程时共享 StringBuffer，也仍然需要承受这个性能损失。而且很多时候不用 StringBuffer 也同样会有性能损失，因为 Java 编译器可能会将字符串连接的代码，比如说 "a" + "b" + "c" + "d" 这样的代码，编译成使用 StringBuffer 的代码。

```
return new StringBuffer().append("a").append("b").append("c").append("d").toString()
```

直接把同步处理从 StringBuffer 代码移走，这样做肯定不是一个向后兼容的解决方案。设计

① Ulrich Drepper, "How to Write Shared Libraries (2006), <http://people.redhat.com/drepper/dsohowto.pdf>。——译者注

者决定提供一个处理字符串连接的新类——`StringBuilder`。除了它的方法没有使用同步以外，这个类几乎就是 `StringBuffer` 的复制版本。从这一点来看，和前面的那个 `Math` 和 `StrictMath` 例子差不多。但考虑到让代码能同时使用这两个类，所以 JDK 5 还引入了一个父接口 `Appendable`。这个接口是通过对 `StringBuffer` 和 `StringBuilder` 两个类进行抽象后归纳出来的，它还考虑了一些其他的功能，如将字符添加到一个现有的文本上。针对这种设计，你在写代码时就可以使用 `Appendable` 接口进行编码，然后根据实际情况使用同步的 `StringBuffer` 或者不同步的 `StringBuilder`，当然混着用也可以。

我不知道这两个方案的设计者是否考虑过另外的处理方式，即使不需要引入一个新的类也能提供可选行为。可以通过构造函数来指定这个 `StringBuffer` 是否需要保证线程安全。可以写一个 `StringBuffer(boolean threadSafe)` 这样的构造函数，通过一个字段来保存传入的参数，然后在每一个方法中根据这个字段来判断是否使用同步。这样可以在保证向后兼容性的时候，还允许编译自动根据场景来使用无须同步处理的 `StringBuffer`。

有时，基于构造函数来提供可选行为的方式比引入一个全新的类要更好一些，特别是这两个行为如果非常相似的话，就再合适不过了。你唯一需要关注的只是有可能会出非常多的构造函数。如果有 8 个版本的话，分别提供不同的行为，搞不好要写 8 个构造函数，每一个构造函数都有 1 个到 8 个布尔量参数。在这种情况下，也许用一个参数类或者枚举类会更合适一些。可以参考下面的代码。

```
public class Arithmetica {
    private final Version version;
    public enum Version { VERSION_1_0, VERSION_2_0 }

    public Arithmetica() {
        this(Version.VERSION_1_0);
    }
    public Arithmetica(Version version) {
        this.version = version;
    }

    public int sumRange(int from, int to) {
        switch (version) {
            case VERSION_1_0:
                return sumRange1(from, to);
            case VERSION_2_0:
                return sumRange2(from, to);
            default:
                throw new IllegalStateException();
        }
    }
}
```

之所以没有采用我所说的构造函数方案，可能有一个原因，就是向类中添加一个额外的字段来判断是否使用同步操作，这样会占用额外的内存空间，有时候不希望出现这种情况。解决这个问题的方案就是使用工厂方法。不是添加一个新的构造函数，而是提供一个新的工厂方法。这个

工厂方法不需要创建一个返回值所声明的类对象，可以是它的子类。下面的代码就是通过工厂方法返回了一个子类的对象实例。

```
public static StringBuffer createUnsyncronized() {  
    return new StringBufferUnsynch();  
}
```

这样做就不需要通过一个内部字段来判断采用哪些具体的行为，而是通过工厂方法来创建一个合适的子类。这样做，可以让 `StringBuffer` 中的代码非常简单，不需要根据一个字段值来进行功能切换，因为通过工厂方法创建的 `StringBufferUnsynch` 可以自动地完成所有功能。这个优秀的解决方案展示了工厂方法的威力所在，不过像处理 `StringBuffer` 这种不可继承的 `final` 类时，就无能为力了。但处理其他情况还是很不错的。

当你对外提供 SPI 接口的时候，也会出现可选行为。比如说，你对外提供了接口，他人实现了这些接口，然后将这个具体实现注入到系统中，由系统来调用这些接口，这就是提供了可选行为。在 6.4 节中已经提到过，向一个现有接口中添加一个方法是不兼容的。向一个抽象类中添加一个方法，虽然从二进制角度来说并非百分之百的兼容，但多少还是可以接受的。所以，如果想在未来的版本中更好地支持新需求，那么最好是添加新的工厂方法，由这些工厂方法来创建指定接口的实现，然后注入到系统中，还有一种方案，就是提供像 `LayoutManager2` 这样的新接口，作为 `LayoutManager` 接口的增强。后面的这种方案，也是可行的。但如果想保证百分之百兼容，最好还是添加新的接口，而不是调整现有的接口^①。比如说，有时候开发人员会调整原来的契约为：顺便说一下，如果你的接口实现了 `Runnable`，那么可能会在某个点去调用你的 `run` 方法。这样说实在不太安全，特别是先使用了某个不支持 `Runnable` 的 API 版本，就有可能出现问题。因为 `Runnable` 这个接口可以用于其他一些目的。也许你的服务提供商可能已经因为某种原因实现了这个接口，然后你却在新版本中调整了原有的契约，在某种情况下会去调用这个 `run` 方法，这可能导致不可预测的行为。

15.4 相似 API 的桥接和共存

有时候想让两个 API 共存是非常困难的。但在作出结论之前，先来看一些基础的内容。有一类功能库，对于它们来说，共存却是非常容易的事情。如果你有一个 7.1 节中所定义的那种“简单类库”，那么同时再使用另外一个功能差不多但其 API 不同甚至更好的类库，是一件非常容易的事情。因为这样一个简单的类库通常都是一些方法的集合，这些方法和 `java.lang.Math` 类中的方法差不多。这些方法单独地被封装来完成各自的功能，它们不会调用 API 用户提供的任何其他代码，这样的类库通常都是自包含的。所以很容易就可以复制 `java.lang.Math` 的代码以引入 `java.lang.StrictMath` 这个新类，这以一种更好的方式提供相似的功能。这两个类的共存不会引发任何问题，因为它们与外部交互时都是通过数字来完成的，接受数字参数，返回相同的数字结果，没有引入复杂的类型。换言之，这两个类处理数据的范围也都是一样的，尽管它们在进行相同操

^① 这里作者认为像 `LayoutManager2` 这种方式也是对原有接口的调整，不算是新添加接口。——译者注

作的时候会有少许的不同。对于这种情况，完全可以放弃原来的 API，而提供一个新的 API，而用户根本都不会考虑原有 API 的存在。

透过图 15-2，你也可以看到这个原则。通常来说，每一个用户在使用 API 的时候，总是在考虑这个 API 可以完成什么功能。然后他才会去分析如何利用这个 API 来实现这个用例。用户会调用这个 API，然后由 API 内部的实现来完成相应的功能。在这种情况下，如果两个简单的类库完成的功能差不多，而且都提供对外的 API 和相应的实现，那么用户在开始编写代码之前，其实就会决定要使用哪个类库了。一旦定下来了，那么这两个 API 之间就不会存在相互影响的情况。每一个 API 都会以自己的方式来处理用户的需求。

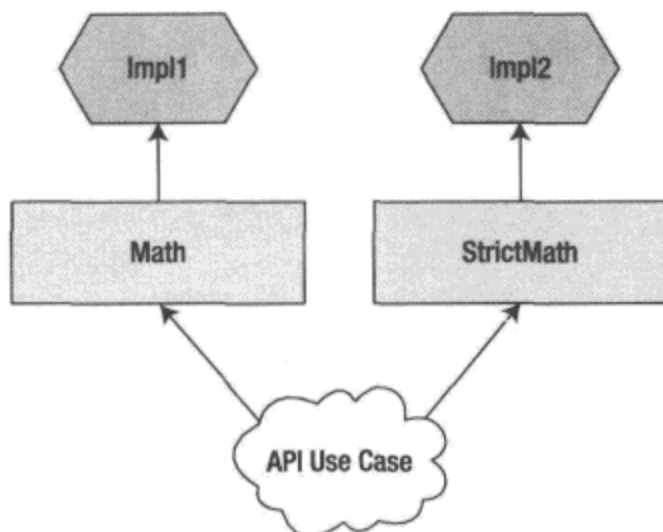


图 15-2 通过两个完全不同的 API 来提供可选的功能

使用枚举时用现有的包还是 enum 关键词

NetBeans 中有一些最古老的 API，这些 API 包含了一大堆类，用来在较高的层面来支持枚举，同时也支持懒加载。这些 API 大概写于 1997 年的夏天，然后被放在 `org.openide.util.enum` 包中，同时还提供了像 `SingletonEnumeration`、`ArrayEnumeration`、`SequenceEnumeration` 和 `FilterEnumeration` 这些类给开发人员使用。

我们一直使用这个包，直到 JDK 5 为 Java 语言引入了新的关键词——`enum`。这样，如果使用 JDK 5 的编译器来编译我们的源代码，就无法编译通过，因 `enum` 变成关键字了。所以我们必须为其提供替代方案，才能弃用这些类。我们有比较多的时间对其进行修正，所以我用了很多好的 API 设计技巧来处理这个问题，以避免那些早期就开始使用 NetBeans 开发软件的用户犯错。最终，我们设计了一个用来处理枚举的工厂类，这个类提供了很多方法，如 `singleton`、`array`、`sequence` 还有 `filter`。原来的实现其实还在，但只是作为内部类隐藏起来了，通过工厂方法来供外部用户使用。

我使用了一些兼容性测试的技巧来保证调整后的新 API 和原有的 API 在对外功能上保持一致。我写了大量的测试以不同的方式来测试这些枚举代码，其中我用了两种不同的配置。第一种配置方式是用在原有的 API 上，而第二种配置则用在新的 API 上。一旦两个配置下的测试都能通过，我就可以确认新工厂方法所提供的功能不仅正确，而且与原有 API 的功能是兼容的。

我把原有的枚举类标识为 deprecated, 建议用户不再使用这些类了, 而是迁移到新的 API 上。但我仍然要保证原有的那些类继续存在, 这样才能让那些无法升级或者不愿意升级到新版本上的用户能够继续运行他们原有的程序。但最好还是能让用户升级到最新版本上, 所以我为那些愿意升级的用户提供了一些额外的奖励: 新版本 API 使用了泛型, 这样在使用枚举的时候就能够进行静态检查了, 有助于提升代码质量, 减少不必要的错误。而老的版本就没有这些功能了。这样用户就会觉得新的可选方案更好一些, 也就愿意进行升级。

利用各种类型的奖励来鼓励用户升级到新版本上总是一件好事。比如说, 为新版本添加一个新的功能, 而老版本就没有这个功能。当然也可以通过拒绝修正老版本中的 bug, 或者把老版本搞得巨慢无比, 也达到同样的目的。但上面说的那个枚举 API, 这个奖励的意义就不太大了, 因 JDK 5 已经提供了很多的新功能: 任何人只要想使用 Java 5 来编译自己的代码, 只需要在编译命令中加入 -source1.5 就可以做到了。这样, 只要升级到 JDK 5, 那么就必须处理自己的枚举代码, 因为 enum 已经作为关键词, 原先的代码无法通过编译了, 他们必须调整代码, 使用 JDK 5 的新枚举。

如果出现老版本和新版本两个类库同时抢夺共享的资源, 那么情况就会复杂了。“共享的资源”可以是任何内容。比如说, 像 AWT 库, 就不大可能用另外一个图形 Toolkit 来替代它, 同时还能让部分程序使用老版本, 另一部分使用新版本。这其实可以做到, 但并不是一件容易的事情, 因为最终这两个 Toolkit 会同时争抢唯一的共享资源——屏幕。

对于模块化类型的库来说, 也存在类似的情况, 特别是那种提供了大部分 API, 但没有提供实现的模块来说, 更是如此, 因为有很多第三方开发商会根据 API 接口来提供对应的实现。如果说你想放弃这个类库, 并提供一个新版本, 那么也许第三方开发商就成为稀缺资源了。在这种情况下, 你要做的工作不是仅仅设计一个新的 API 就可以了。你还必须让这个 API 能够与那些原有的第三方开发商提供的程序共存。还有一点, 就是因为提供给客户使用的 API 和提供给开发商使用的 API 之间是存在着关联关系的, 你为客户提供了新版本的 API, 那么给开发商提供的 API 也应该是新版本的, 因为两者需要进行相互协作。在这种情况下, 不管是新开发商, 还是老开发商, 他们提供的模块都必须能够共存。不管用户是基于新版本的 API 来编写程序, 还是基于老版本的 API 来编写程序, 模块都必须保证能正确地处理其相应的请求。

在重新编写 NetBeans 向导 API 中多次失败的尝试

在 NetBeans 中, 向导 API 在最初设计时, 用来满足当时一些特定的需求, 随后重新编写了这个 API 以满足新的需求。一开始, 这个向导只是一个对话框的功能扩展。它会提供一些额外的按钮, 比如“下一步”、“上一步”还有“完成”按钮。随后我们发现还需要支持图形和有效性验证, 以及对向导中各步操作的概述有所控制。此时, 我们需要重新实现这个向导以支持这些新需求。但重写的效果不是太好。最初的那个 API 几乎违反了书中给出的所有设计原则: 应该设计为 final 的内容偏偏不是 final、到处都是 setter 方法……。意料之中, 修改后的 API 更是乱麻一团。在外部用户看来, 这个 API 没有使用任何设计方面的技巧, 搞得非常难用, 即使对于那些内部用户, 也有同样的感觉。现在向导 API 在变大的同时, 设计也更差了。同时还

提供了一些测试代码，用来验证这个向导 API 中的一些莫名其妙的行为，事实上根本没有人敢去修改这个 API，害怕略微一动，就搞得 API 不再稳定。

关于为这个 API 提供替代方案的事情，至少尝试过 3 次。替代方案会使用更多现代的设计原则、更多现代的编码方法、提供更好的文档，等等。按照这种思路是可以提供一个更好的 API。而且对于那种基本的用户场景来说，用户可以使用 API 来创建整个向导及向导上的页面，然后显示向导，再进行某些操作，不管如何，想让新旧两个 API 共存是比较容易的。之所以容易共存，是因为用户在使用 API 的时候，必然是二选一，不用新版本的 API，就得用旧的，不会同时使用这两者。

但这些提供替换方案的尝试都没有能解决 NetBeans IDE 需要的协作功能：对于这个 IDE 来说，它向导核心的接口是围绕模板来设计的，不同类型的文件使用不同的模板。我们分别为 Java、C++、Ruby、HTML、XML 和其他类型的文件都提供了模板。另外，用户也可以创建自己的模板。一种项目类型也会有一系列的模板文件。一旦想通过向导创建一些新的内容，只需要打开“新建向导”，然后选择合适的模板，就能做到了。模板内还支持扩展功能，允许将已经存在的模板再嵌入到其他的向导中，然后由用户一步步地按照指示来创建最终的对象。在 NetBeans IDE 中，这个类型的向导使用频率是最高的，大概占了 90%。但涉及协作的时候，这个向导却碰到了问题。这个向导是用老版本向导 API 来编写的。如果想迁移到新版本上，那么工作量太大，也太复杂了。但这个向导允许任何一个模板将自己的模板注册到这个向导中。到目前为止，这些注册进来的内容，可能全部是使用老的 API 编写的，当然了，也有可能是使用新的 API 来完成的。换句话说，也不大可能让所有人都把原来写的内容全部迁移到新版本上。就像分布式的项目开发一样，不太可能在一夜之间做到这一切。事实上，对于我们来说，NetBeans IDE 开发的控制还是比较容易的，有点像内部系统，我们对模块有绝对的控制权，但就是这样，升级到新版本的 API 上，也有巨大的工作量，不可以通过一个版本的发布周期来完成所有的迁移工作。这就意味着需要通过桥接的方式来完成这个目标。有一些模板可以继续使用老版本的向导 API，而有一些模板则可以使用新版本的向导 API。对话框需要了解这两个版本，然后根据其版本来提供必要的运行环境。

出于某种原因，这个额外的需求却会强迫每个人都要重写模板。我不是很清楚为什么，但我知道通过桥接的方式应该也可以处理这种情况。但想做桥接的话，需要对这两个 API 都要有较深的了解才行。而且需要对原有版本进行一些调整，以便进行相互协作。此外，这还意味着开发人员不能从零开始做一件事，而是在现有的老代码上进行调整，而前者往往是程序员的最爱。这样搞得我们仍然是使用老版本的 API。不过至少我们提供了一个工具来为向导生成基本内容，这样，用户在使用该功能时更方便一些。的确，我们用一个向导来生成另外一个向导，我们把它称为向导的向导。这也就是说，尽管这些重新编写向导 API 的尝试可以算是提供了可行的解决方案，但没有人有足够的意愿去推动这些方案，让这些方案成为产品的一部分。

如果新旧两个 API 需要共享一个资源，它们之间需要进行交流，一方要知道另外一方的存在。或者是，要有第三方知道这两者的存在，并在这两者之间进行协调使其共享资源。很明显，如果

你有两个 API，一个是新版本，还有一个老版本，你最终的目标应该是让这个老版本的 API 慢慢地淡出用户的视线，最终不再有人使用这个老版本的 API。这也就意味着新的 API 不能依赖老的 API，否则它们两个就会一同消失，而这就违背了创建新 API 的初衷。不过，如果是老版本的 API 要依赖新版本的 API，这是可以接受的，如图 15-3 中所示。这样，即使老版本的 API 消失了，也不会对新版本产生影响。

还有一个选择，就是让这两个 API 间没有依赖关系，而是通过一个桥接模型来做这件事。如图 15-4 所示，一个桥接模块清楚地知道这两个 API，然后正如“桥接”这个名字一样，把对一个 API 的调用转给另外一个 API 来处理，所以它们就可以协作和访问共同的资源了。

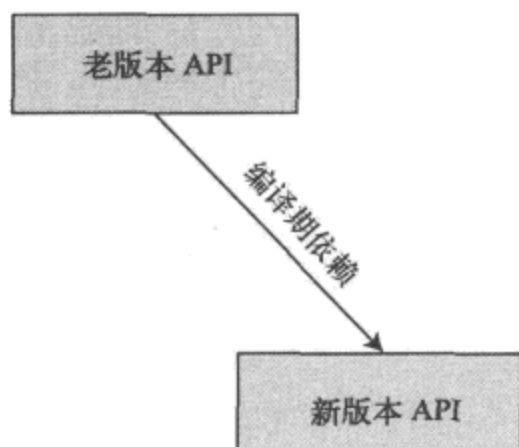


图 15-3 老版本的 API 依赖于新版本的 API

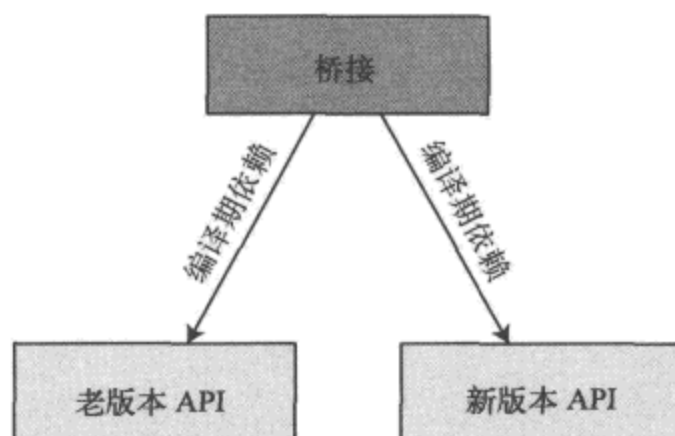


图 15-4 老版本和新版本的 API 之间不存在依赖关系

NetBeans 中的“Recommand”声明

假如你使用了一个桥接模块，那么不管你代码中使用的是新 API 还是旧 API，你都需要确定系统中一定要包含这个桥接模块。在 NetBeans 项目中，我们通过“recommend”声明来保证一定要包含桥接模块。这种声明不是一个强依赖关系的声明，所以即使依赖的模块不存在，也不会有什么大事。但这样声明也就是建议系统，如果有这样一个模块，那么最好是启动这个模块以方便其他模块使用它。

在代码编译时，新版本 API 是绝不能依赖老版本的 API，不管是直接依赖还是间接依赖，都是不允许的。在运行时，则可以有所不同。老版本的 API 和新版本的 API 需要知道彼此的存在以便正确地共享资源。彼此都要了解相互的信息。在运行时做到这一点，是通过组件注入的方式完成的，在 7.2 节中对此有详细的说明。在 NetBeans 中，是通过 Lookup 类来完成组件注入的，而在 JDK 中则通过 ServiceLoader 类来完成组件注入^①。不管哪种处理方式，新的 API 都必须提供一个扩展机制，能够让原有的 API 或者是桥接模块能够注册回调接口，并能处理回调接口。如果没有人向这个扩展中加入任何接口，那么就不会有任何事会发生。新的 API 在运行时也不会有任何问题，因为它不需要与其他组件来共享资源。如果有一天老版本的 API 不再有用，将其从系统中移除以后，那么它原来为新版本 API 提供的扩展实现也就用不上了，就会出现上面所说的那种情况，程序仍运行正常。

① 只有 JDK 6 才支持这种注入方式。——译者注

来看一个例子，对于 `java.security.MessageDigest`^① 这个类，很多人都认为它很复杂，或者说它违反了很多好的 API 设计原则。不过这里只是把它当作一个例子来讨论一下：这个 API 还不算太差，虽然它的设计并没有遵守本书建议的所有原则，比如说，它没有把 SPI 和功能 API 分离清楚，客户必须要继承 `MessageDigestSpi` 来扩展功能。这还算是小事，更大的缺点在于它的注册方式非常古怪，是通过系统属性 API 而不是基于 `java.util.ServiceLoader` 这种标准的 Java 扩展机制。这样设计的后果，就是你在使用这个类的时候，不是说把类所在的 Jar 放到类路径上就没问题了，还得去配置系统属性才行。这是一个非常大的缺点，不过还不值得为这一点就重写这个 API。现在假设这个 API 的问题是无可修正的，而我们想为它提供一个新的可替代方案。那么现在最大的问题就是这个 API 类继承了一个 SPI 的类，所以我们想修正这个问题，看下面的代码。

```
public final class Digest {
    private final DigestImplementation<?> impl;

    /** 工厂方法优于构造函数 */
    private Digest(DigestImplementation<?> impl) {
        this.impl = impl;
    }

    /** 通过工厂方法来创建一个支持指定算法的 Digest 对象实例。
     */
    public static Digest getInstance(String algorithm) {
        for (Digestor<?> digester : ServiceLoader.load(Digestor.class)) {
            DigestImplementation<?> impl = DigestImplementation.create(
                digester, algorithm
            );
            if (impl != null) {
                return new Digest(impl);
            }
        }
        throw new IllegalArgumentException(algorithm);
    }

    //
    // 这些方法与 Message Digest 中的原始方法保持一致，
    // 但为了简化起见，只抽取了原始方法的几个用来演示
    //

    public byte[] digest(ByteBuffer bb) {
        return impl.digest(bb);
    }
}
```

① `java.security.MessageDigest` 这个类是用来做消息摘要的，它可以为一个任意长度的一个数据块产生唯一的标识（对于 SHA1 是产生一个 20 字节的二进制数组），其特点在于两个不同的数据难以生成相同的摘要，难以对指定的摘要生成一个报文，而由该报文反推算出该指定的摘要。代表性的算法为美国国家标准技术研究所的 SHA1 和麻省理工学院 Ronald Rivest 提出的 MD5。——译者注

我们新设计的 API，则避免了让用户去了解这些细节性的内容，那些内容只有服务提供商才需要。更重要的是，用户很容易就知道哪些是用来生成消息摘要的功能性 API，因为它们都被设计成 final 类了。当然，这样做和原来比也不算什么大的改进，但不要忘记，这里只是举个简单的例子而已。新的 API 是一个模块化的类库，虽然可以为它添加扩展机制，允许用户来扩展其功能。但扩展机制对于新版本的 API 来说并不是必需的，因为新版本的 API 完全可以调用老版本的 API 库来实现相应功能。除非发现原先的类库有问题的时候，才需要提供扩展机制，让开发人员提供新的扩展实现，此时就可以放弃原有的 API。只有按照这种方案来完全替代原有的 API，才算是一种真正意义上的重写，可以慢慢地替换原有的 API 类库，等到没有人再使用这些类库的时候，就可以把它们丢给一边去了^①。在 NetBeans 团队中，我们的工程师把这种方式称为 singletonizer^②。

```
public abstract class Digestor<Data> {
    protected abstract byte[] digest(Data data);
    protected abstract Data create(String algorithm);
    protected abstract void update(Data data, ByteBuffer input);
}
```

对于负责具体实现的开发人员来说，这种处理方案的好处在于简化了整个开发过程。他们要做的事情非常简单，只需要实现一个接口，然后再将这个接口注册到系统中，最后再用一个 private 的对象来持有相应的数据就可以了。这样，SPI 中就只需要提供一个接口，相比之下，如果工厂模式来做，也需要两个类才行。此外，负责具体实现的开发人员在选择数据结构的时候有着更高的自由度。比如说想统计有多少字节被处理了，可以使用 int[] 这样一个简单的数组作为自己的数据结构，代码如下。

```
public final class CountingDigestor extends Digestor<int[]> {
    @Override
    protected byte[] digest(int[] data) {
        int i = data[0];
        byte[] arr = {
            (byte) (i & 255),
            (byte) ((i >> 8) & 255),
            (byte) ((i >> 16) & 255),
            (byte) ((i >> 24) & 255)
        };
        return arr;
    }

    @Override
    protected int[] create(String algorithm) {
```

① 这段文字比较难理解，但仍然是 Bridge 的一个延续，是指原有的 java.security.MessageDigest 不好用，可以提供一个新的 API，它默认可以使用老的 API，但允许通过扩展的方法来替换原有 API 的功能，这样等于提供了一个更好用的全新 API，同时还可以做到与老 API 的兼容。——译者注

② 这个词是由 NetBeans 项目组发明的，可以在 <http://wiki.apidesign.org/wiki/singletonizer> 上面了解更多该模式的内容。——译者注


```

    return "cnt".equals(algorithm) ? new int[1] : null;
}

@Override
protected void update(int[] data, ByteBuffer input) {
    data[0] += input.remaining();
    input.position(input.position() + input.remaining());
}
}

```

NetBeans 项目中经常采用这种方案，特别是在映射深层次继承的时候，比如树，只需要一个接口就能处理所有的操作。它把内存管理交由系统来控制，只保证自己代码实现正确。可能说的内容有一点偏题了。

现在要解决的主要问题变成了：不管基于老版本 API 开发的功能，还是基于新版本 API 开发的功能，都能同时用在新老两个版本的 API 上。如图 15-5 所示。情况开始变得有一点复杂了。

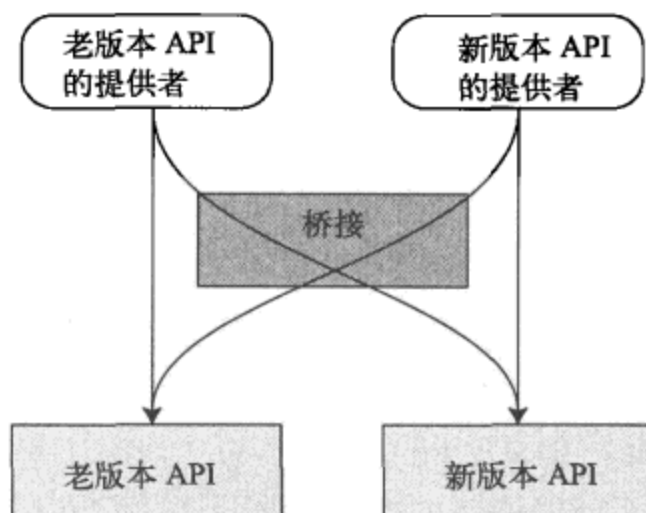


图 15-5 信息从已经注册的 API 供应者流向 API 的用户

这个桥接的设计需要知道所有注册的 `Digester` 实现，而且要做个二重桥接。就是说这个设计中，不仅要把老版本的 API 转成新版本的 API，这样新的 API 才能使用老的 API，同样，还得把新版本的 API 转成老版本的 API，这样老的 API 的客户才能使用新的 API。这样在设计中，就引入了两个类。有一个类实现了新 API 的接口，并在实现代码中使用老 API 来完成相应的功能。

```

# registration in bridge's META-INF/services/
org.apidesign.impl.security.extension.BridgeToNew

```

```

public class BridgeToNew extends Digester<MessageDigest> {
    /** 创建桥接类，而且不会引发堆栈溢出 */
    private static final BridgeToOld oldBridge = new BridgeToOld();
    @Override
    protected MessageDigest create(String algorithm) {
        // 开始：进行循环检查
        if (oldBridge.isSearching()) {
            // if the call is initiated from the other bridge, do not do
            // any delegation
            return null;
        }
    }
}

```

```

    }
    // 结束: 进行循环检查
    try {
        return MessageDigest.getInstance(algorithm);
    } catch (NoSuchAlgorithmException ex) {
        Logger.getLogger(BridgeToNew.class.getName()).log(
            Level.FINE, "Cannot find " + algorithm, ex
        );
        return null;
    }
}

@Override
protected byte[] digest(MessageDigest data) {
    return data.digest();
}

@Override
protected void update(MessageDigest data, ByteBuffer input) {
    data.update(input);
}

static {
    new BridgeToOld();
}
}

```

这样做就可以保证,只要用户想通过新 API 来使用一个由旧开发商提供的一个 *Digester* 算法,通过这样一种桥接的方法,就可以保证新 API 能够使用合适的旧算法对象来满足用户需求。但对于反过来的情况,你仍需要代理。

```

public final class BridgeToOld extends Provider {

    public BridgeToOld() {
        super("spi.Digestor", 1.0, "");
        Security.addProvider(this);
    }
    // 开始: 进行循环检查
    private ThreadLocal<Boolean> searching = new ThreadLocal<Boolean>();
    final boolean isSearching() {
        return Boolean.TRUE.equals(searching.get());
    }

    @Override
    public synchronized Service getService(String type, String algorithm) {
        Boolean prev = searching.get();
        try {
            searching.set(Boolean.TRUE);
            if ("MessageDigest".equals(type)) {
                Digest dig = Digest.getInstance(algorithm);
            }
        }
    }
}

```

```

        if (dig != null) {
            return new ServiceImpl(
                dig, this, type, algorithm, "",
                Collections.<String>emptyList(),
                Collections.<String,String>emptyMap());
        }
    }
    return null;
} finally {
    searching.set(prev);
}
}

// 结束: 进行循环检查
private static class ServiceImpl<Data> extends Service {
    Digest dig;

    public ServiceImpl(Digest dig, Provider provider,
        String type, String algorithm, String className,
        List<String> aliases, Map<String, String> attributes
    ) {
        super(
            provider, type, algorithm, className, aliases, attributes
        );
        this.dig = dig;
    }

    @Override
    public Object newInstance(Object constructorParameter)
        throws NoSuchAlgorithmException {
        return new MessageDigest(getAlgorithm()) {
            private byte[] res;

            @Override
            protected void engineUpdate(byte input) {
                ByteBuffer bb = ByteBuffer.wrap(new byte[] { input });
                res = dig.digest(bb);
            }

            @Override
            protected void engineUpdate(
                byte[] input, int offset, int len
            ) {
                ByteBuffer bb = ByteBuffer.wrap(input);
                bb.position(offset);
                bb.limit(offset + len);
                res = dig.digest(bb);
            }

            @Override
            protected byte[] engineDigest() {

```

```

        return res;
    }

    @Override
    protected void engineReset() {
        dig = Digest.getInstance(getAlgorithm());
    }
};
}

}
}

```

代码看起来有点乱，因为老的 API 使用了经典工厂模式的设计，而不是 singletonizer。工厂模式通常需要实现多个类，而 single-tonizer 只需要一个。最大的问题出在注册上。不管怎么做，也一定要把 BridgeToOld 给注册到新的 API 系统中。但直接通过 META-INF/services/java.security.Provider 的方式是无法注册成功的。可能需要一种内置或者更复杂的方式来注册，也可以在把类加载到内存的时候通过桥接的方式来将自己注册到系统中。在测试用例中，用下面的方法将 BridgeToOld 注册到系统中。

```

// java.security.Provider 是无法通过 META-INF/services 这种服务声明方式注册到系统中的。
// 所以必须通过配置或者明确调用相应的代码来初始化桥接类。
// 这样，相应的桥接类才能将自己注册为 MessageDigest 的提供者。
//
// 下面是明确注册自己的代码：
Digest initialize = Digest.getInstance("MD5");

```

这种做法不算优雅，但至少代码不必直接依赖于桥接类了。只希望有一天，提供安全策略的类都能通过标准的 Java 扩展机制来注册到系统。到那个时候，只需要在路径上添加一个用来支持桥接功能的 Jar 包就可以扩展安全机制，即使原有的代码只使用了 java.security.MessageDigest 这个类，也同样可以使用新的安全机制。

还有一点要特别加以留意，就是可能会出现严重的循环依赖，最终引发 StackOverFlowErrors 这种堆栈溢出错误。造成这一问题的原因却很简单：假设有一个提供桥接功能的类将自己注册到原有 API 中，但内部实现是调用了新的代码，同样另外有一个桥接类却将自己注册到新的 API 中，但内部实现却是调用了老的代码。如图 15-6 所示，这样很容易就会出现一个死循环，直至将堆栈空间用尽，程序出错为止。

可以有多种方式来修正这个问题。但最透明的修正方式就是使用一个线程局部变量。将这个变量

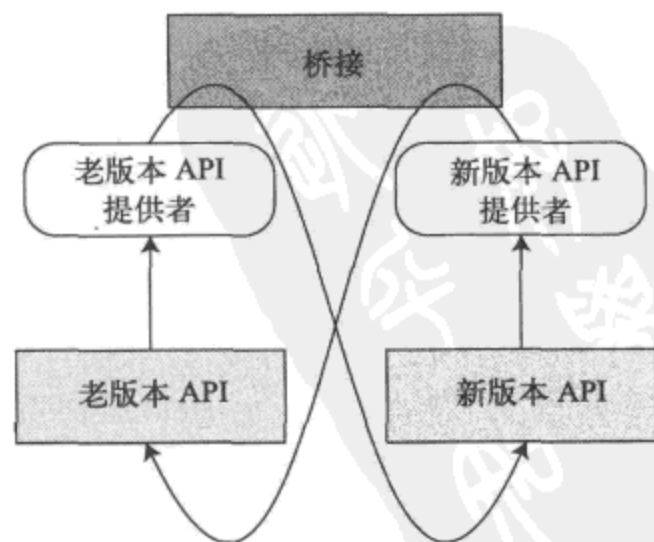


图 15-6 在运行时，如果出现双重桥接，信息流就会出现严重的循环依赖

放到一个桥接类中，以标识当前进程中有一个直接的桥接引用。

```
private ThreadLocal<Boolean> searching = new ThreadLocal<Boolean>();
final boolean isSearching() {
    return Boolean.TRUE.equals(searching.get());
}

@Override
public synchronized Service getService(String type, String algorithm) {
    Boolean prev = searching.get();
    try {
        searching.set(Boolean.TRUE);
        if ("MessageDigest".equals(type)) {
            Digest dig = Digest.getInstance(algorithm);
            if (dig != null) {
                return new ServiceImpl(
                    dig, this, type, algorithm, "",
                    Collections.<String>emptyList(),
                    Collections.<String,String>emptyMap());
            }
        }
        return null;
    } finally {
        searching.set(prev);
    }
}
```

在另一个桥接类中，对这个线程局部变量进行检查，然后跳过代理操作。

```
if (oldBridge.isSearching()) {
    // 如果当前已经被其他的桥接对象用来初始化了，就不必重复操作
    return null;
}
```

本书的大部分章节中都将优雅看作一个不重要的内容，认为向后兼容性比优雅要更为重要，因为对兼容性的需求是很容易分析出来的。但本章给出的例子也指出一个 API 不能越来越难看。完全可以对其进行修复，让 API 变得优雅一些。通过创建一个可选 API，你可以创建一个非常优雅的 API，就像一颗极其漂亮的恒星一样，可以吸引到所有看见它的人。这样关注这类功能的用户就会只关注新的 API，完全忽略了老的 API，老的 API 就会慢慢地退出自己的历史舞台。通过在两者之间以桥接的方式将旧恒星的光通过新恒星来加以反射，可以让那些老 API 的观察者们继续自己的观察。即使旧的恒星走向灭亡，它仍然可以由客户来决定其死亡的速度。只有还有人愿意关注，那么它就会继续存在，这颗恒星就会像刚刚创制时一样，仍然继续存在，发光发热。直到最后一个观察者将自己的注意力转移到新的恒星上，此时旧的恒星就会立刻消失了。当然，它可能还会潜藏在某个角落，但已经超出了 API 用户的视野，没有人会再去关心它了。它慢慢地隐入黑暗之中，直到有一天完全消失，就好像最终成为一个黑洞一样。

一个 API 死亡的过程也是差不多的，完全是由它的用户来决定的，与用户对兼容性和漂亮的需求相关。对于我们来说，好的一方面是，在 API 的世界，它能够折射出真理、漂亮和优雅。

成功的项目也并非一蹴而就，也都是从一个小项目开始的，由一个小的团队一直开发下来。但这些小项目会越变越大，直到没有任何一个人能够再独控所有的内容。因为开发人员也越来越多，人们发现他们都不认识团队中的所有人，更不用说去了解其他人工作的内容了。架构师这时候也不直接去写代码了，只是通过敲打键盘来讨论产品下一个版本中要进行哪些必要的调整，可能还要和自己现有或者未来的客户打一下高尔夫球来谈谈生意。在这种情况下，就容易跟不上事情发展的进度。事实上，这并不是必然出现的情况。没有人能理解整个系统，系统的透明度却开始增加起来。除此以外，对于一个松耦合的组织，也同样要求这样的组织能开发出软件的新版本。想高效地做到这一点只有一条路可选——分析项目，找出最重要的部分，然后把精力集中在这上面。本章在最后部分列出一些 NetBeans 项目中总结出来的经验。

16.1 在提交代码时进行代码评审

如果说一个开发团队的规模比较大，开发人员不是说都相邻着坐在一个房间里，还有外部的人员来提交代码，那么就需要在调整 API 之前进行代码评审。不管做什么事，多大多小，都比什么事都不做来得更好。但不管怎么做，都应该包括一个细致的代码评审。另外，经验告诉我们，在代码整合到版本中之前，最重要的事情就是用正确的评审方式来检查代码。只有这样做了，负责整合版本的人才愿意将调整后的内容加入到产品中去。

当然了，调整一个 API 需要进行一序列的工作，上面所说的并不是唯一的理由。特别是所做的内容为一个开源项目时，那么你所做的一切都会暴露在大家的注视之下，如果有外部的人员想对你的开源项目做一些修改，你必须对他们所做的修改给出回应。对于每一个要求更改 API 的人而言，完美而简约的回答就是，对整合 API 更改所需要采取的步骤要进行适当地整理。

还有一条理由也可以解释为什么不希望 API 变更“突然出现”，那就是你不想让一些你认为不合适的 API 调整进入产品。比如说，你不想让团队成员错误地提供一些不兼容性的调整，因为这些会马上伤害到 API 的用户。另外，即使他们的调整是兼容的，你可能也希望他们做这事的时候，是的确考虑到了兼容性，否则在未来同样有可能伤害到 API 的用户。在发布了多个版本以后，你可能会发现有一些在多年以前设计的类或者方法，现在却妨碍了 API 的改进。但你现在却不能按照自己的想法来随意修改这些代码了。

由一个团队来设计

这话已经说过很多次了，在一开始的时候，是由我一个人来设计所有 NetBeans 对外公开的 API。在那种情况下，完全不需要有什么正式的步骤，也不需要有什么规则。所有的内容都在我的脑子里。我清楚地知道每一项内容的目标，也知道每一个 API 特定的调整要按照什么样的策略来进行。至于维护其一致性，就更简单了。但这种情况仅仅适用于那些只有一个设计师的项目。

直到有一天，有一个 NetBeans 的工程师说要调整一个控制 NetBeans IDE 主窗口上工具条的 API。我当时觉得这事不算大，让他自己处理就是了。结果这一举措立即打破了我设计 API 时的工作方式。我本可以坐下来，和他一起就所做的每一个调整进行检查，但我当时还有别的事情要处理，所以他就一个人完成这项任务，然后提交了代码。

第一个问题在于，我都没有注意到 API 有所改变。我们没有一个专门的人或者组织来负责监控这样的一种改变。当我发现 API 的改变后，我很不喜欢这种调整。但我却无法为我的这种不喜欢倾向给出原因，这件事发生在很久以前，远在本书成文之前。而且，是在某个版本发布了以后，我才注意到这一改变的。考虑到我们要做到很强的向后兼容性，所以已经不可能从 API 中将它移走了。我们只能接受它的存在。

其实对于这个问题，我的印象已经有一点模糊了，记得不是很清楚。可能是在 API 中对外公开的一个类，这个类实现了 MouseListener 接口，它对外公开的时候还暴露了内部的一些实现细节。这个类还带了很多本不应对外公开的内部字段。关键在于，一旦你允许一个团队而不是一个人来调整你的 API，你必须要对他们所做的事情加以组织和监管。

当然，不是说所有 API 的调整都要下这么多的心思。有时候，只需要大概地监督一下 API，确信没有什么调整会对用户造成伤害也就行了。最值得关注的地方应该是看其是否有能力在未来支持对 API 的改进，要看调整时选择的版本是否合适，还有调整的内容是否与整个 API 体现出一致性，以及是否提供了相应的文档和测试覆盖率。如果要做一些小的调整，比如说向现有的框架中加几个新方法或者类，那么一般来说观察以上这几点也就够了。在正式提交代码前，把补丁发给相关人员，通过评审来确定是否提交。一般来说，这类调整都比较小，补丁包自身已经可以很清楚地说明问题和调整方案了。就像开源软件的开发人员都喜欢说：“直接把代码给我。”不管怎样，这才是唯一重要的事情。

标准方式 VS. 快速跟踪

在 NetBeans 开发过程中，代码的评审是公开的，对于提交代码的人和评审人员是一样的。每个人都可以参与进来。可以在 <http://openide.netbeans.org/tutorial/reviews/> 上找到整个评审过程的细节步骤，这种评审更多是记录在 bug 跟踪系统中。每一次评审的请求都被作为一个议题来跟踪，对评审的过程和结果，也同样进行跟踪，就像对 API 的调整进行跟踪一样。

对于小的调整，我们使用了一种乐观锁的优化机制来对其进行快速跟踪。代码提交者需要将 API 调整的内容准备好，然后将内容公开，并等一个星期。如果在此期间，没有人给出反对意见，那么就算通过，可以提交代码。我们觉得对于代码评审者和产品整合人员来说，这是比

较合适的方式。相对而言，任何人只要对这个问题有兴趣，都可抽出一点时间对其进行评审，在这种情况下，一个星期也就足够了。只不过，一个星期也不是特别长的时间，代码提交人员还是得采取特殊方式。另外，对于负责产品整合的人来说，代码提交者调整的内容往往与自己的理解还是有一点偏差：在这一星期里，代码提交者还可以对代码进行自评审，把提议考虑得更细一些，调整得更好一些。这也就是说，即使没有这样的评审过程，这样一个星期下来，最终的结果也比一星期前提交的内容要好，事实上，这种评审经常会没有任何评审人员提出建议。

还有一种评审的方式，我们称为标准方式，它包括两轮评审。在没有任何编码工作开展之前，就要进行第一轮评审，而第二轮评审则是在将代码整合到产品中之前举行。不管是哪一轮评审，相关讨论的内容都会记录到 bug 跟踪系统或者是专用的 Wiki 页面上。可以通过电话会议的方式来进行讨论，至少有四位评审人员需要参与投票，给出他们的反对意见。这样，可能某一个项目就会被毙掉，或者在解决了所有问题以后继续下去。

但项目中有可能出现大的调整。为了处理这种情况，上面所说的标准评审方式就不太合适了。假设你要设计一个全新的 API，同时能让这个 API 与整合框架保持一致，那么一个一个方法的评审并没有太大的意义。在方法都已经写完的时候，再去评审，已经太晚了。应该在确定了要开发该 API 之后，此时还可以给开发的内容加以限定和约束，这个点会是一个比较合适的评审时间。如果说代码已经写好了，再去说“应该使用 JavaCC^①来设计你的 API”、“要是用 ANTLR^②就好了”这类的话，估计没有哪个程序员会听着顺耳。所以对于程序员来说，要说服他们尽早地开始对设计和代码进行评审，这是一件非常重要的事情。当然做起来比较困难，特别是如果程序员认为你这样做其实是在阻碍或者拖慢他们开发进度的时候，就更困难了。以前没有参与过评审的程序员通常都会这样想，因为他们往往有错误的估计，不清楚什么样的架构级评审对于他们来说是合适的。

我第一次参与的架构评审

开发团队在发布一个产品前，需要让一家名为 Sun 的全球架构评审委员会来做次评审，提些意见。对于这个流程的发展历程，我并不太清楚。我猜测这起源于很早以前，一个与 Solaris 操作系统和 SPARC^③处理器有关的项目。我当时的印象就是说我读了一份“20 个问

① JavaCC，即 Java Compiler Compiler，是一个用 Java 开发的最受欢迎的语法分析生成器。这个分析生成器工具可以读取上下文无关且有着特殊意义的语法并把它转换成可以识别且匹配该语法的 Java 程序。官方网站为 <http://javacc.dev.java.net/>。——译者注

② ANTLR(ANother Tool for Language Recognition)，它是这样的一种工具，它可以接受词文法语言描述，并能产生识别这些语言的语句的程序。作为翻译程序的一部分，它可以使用简单的操作符和动作来参数化你的文法，使之告诉 ANTLR 怎样去创建抽象语法树(AST)和怎样产生输出。ANTLR 知道怎样去生成识别程序，语言包括 Java、C++、C#。官方网站为 <http://www.antlr.org/>。——译者注

③ SPARC，全称为“可扩充处理器架构”(Scalable Processor ARchitecture)，是 RISC 微处理器架构之一。它最早于 1985 年由 SUN 公司所设计，也是 SPARC 国际公司的注册商标之一。这款处理器以 VHDL 语言写成，并采用 LGPL 授权。——译者注

题^①”的文档，那份文档是随时会被更新的，我在第一次读的时候，文档里只有上百个与 Unix 相关的问题，根本就没有 Java 什么事。说到这里，估计读者也猜出来了，慢慢地，文档里就开始加入一套与 Java 相关的问题了。

在 Sun 公司收购了 NetBeans 产品以后，在没有经过任何评审的情况下，我们还发布了好几个版本。他们要求我准备一下资料，要进行架构评审。你可以想象得出来我当时有多晕！可能会有一些甚至不知道 Java 这个词怎么拼写的人，竟然要来指导我这个设计 NetBeans 的人，他们要来告诉我所谓的架构都是什么内容！他们竟然敢这样做！我都想象不出来，这群对 NetBeans 一无所知的人能给我提出什么建议呢？

其实这些指导性的内容并不是针对 Java 的，负责评审的人也并不是 Java 方面的专家。但当我看完并回答了这“20 个问题^②”以后，我有点明白了。在此之前，我一直认为所谓的架构和 API 就是类和方法的集合。在此之后，我发现了自己观念上的错误。我在第 3 章中已经谈过我这方面的经历了，要做架构和 API 的评审人员，并不一定要是 API 领域的专家。在不了解 API 具体工作的情况下，也仍然有很多通用的内容需要被评审。对于这些通用的内容来说，并不需要先去了解 API 才能进行评审。听起来很耳熟吧？的确如此，我在本书的第一部分就已经强调过了。事实上，我敢说，如果我没有参与过这样一次架构评审，可能根本就不会有本书的面世。我从来没有想过在为 Sun 公司工作的过程中，能学到这么多技术方面的技巧。对我来说，架构评审委员会比我在 Sun 公司遇到的其他所有事物都更有价值。因此，我想真诚地谢谢 Sun 公司，是它教会我如何进行高效的架构评审。

不过在评审时有一件事着实让我吓了一跳：评审时需要我提供一份文档，可是评审后我却不能拿回这份文档。我极其讨厌做这种无用功，我觉得这种事太没有意思了。我决定在 NetBeans 项目组中改变 API 的评审方式，让我们的技术文档直接用于架构评审。我们不但可以用它来应对评审，还可以为使用 NetBeans 平台的开发人员提供更详细的文档资料。我对架构评审流程进行开源，把名字给改了，还改了很多技术术语，我还重写了那“20 个问题”的文档，让它更符合 NetBeans 和 Java 的特点（可以在 <http://openide.netbeans.org/tutorial/questions.html> 上了解更多这方面的内容）。但在调整过程中，我没有对整体结构进行任何改动。我保留这个公开的秘密已经多年了。但现在 Sun 公司也正在快速地推进这种开源的开发方式，所以我也不需要再偷偷摸摸地隐藏这段历史了。我把调整后的内容称为 NetBeans 开放源码的评审流程。

读者可以通过阅读 NetBeans API 评审指导意见书来了解更多这方面的内容。我推荐大家去阅读它，因为我觉得在 Sun 公司，最值得称道的就是架构评审这一点了。

不管项目采取何种工作方式，都应该在代码提交之前进行一些评审工作，至少说对于架构一级的调整要进行评审。经验告诉我，这样做是唯一能够有效解决 API 协作的方式。另外，把这一

① 这是西方的一个传统游戏，由一个游戏者选择某样事物，其他参与者来问他问题，他只能回答“是”、“否”或“不确定”，一般在 20 个问题后，就可以猜出他所想的事物。本书作者在本处使用这个游戏的名称，表示在评审中通过一系列的问题，来找到真正要做的事情和方案。在 1998 年，为了展示人工智能，计算机中也引入了这个游戏，称为 20Q，有兴趣的读者可以去 <http://www.20q.net/> 这个网站试一下这个游戏。——译者注

② 20 个问题只是一个泛化，并不是说一定只有 20 个问题。——译者注

过程也同样公开给外部的代码提交人员，可以减少维护费用，在 14.4 节中也说过这个问题。

16.2 说服开发人员为他们的 API 提供文档

任何从事过开发工作的人都知道写文档是一项非常乏味的工作。每一个尝试说服开发人员写文档的人也都知道说服他们同样是一件很困难的事情。结果，大家都会认为开发人员不喜欢也不会为他们自己的代码提供文档。但一份好的文档可以有效地帮助用户，避免他们在使用 API 时去了解内部细节性的内容，可以在不深入了解 API 的情况下就能将 API 用到自己的开发工作中去。而且，API 是作者与用户之间主要的交流工具，文档写得越好，那么用户了解和使用一个类库所花费的精力也就越少。所以对于任何成功的框架来说，优秀的文档是个必备条件。

其实，一个开发人员的态度往往是这样的：“代码都在那儿，也能正常运行，只管用就是了！”这些开发人员觉得（至少是我觉得）写文档是重复已经做过的工作。任何人都知道，重新做相同或者相似的工作是非常无聊的一件事。这也许是因为开发人员一提到要为他们自己的代码写文档，就会非常反感。同样，如果强迫他们来写文档的话，文档的质量也没有保证，可能会比较差。即使他们认真、努力地去写文档，也不见得能介绍得好，因为他们太了解自己的 API，而且又与用户的视角不一样，不管怎么努力，也不太可能客观地从新用户角度来写这样的文档。即使他们是想对代码写一份概述，也经常不由自主给出一些细节性的内容。“提供这个类库就是帮助大家来画图用的，”这话听起来的确具有概述性。但工程师则会忍不住立刻对功能如何实现大加描述，他们会展示一堆代码，其内容之详细程度完全超出新用户阅读概述的目的，因为这些新用户只是想了解这个类库的功能。

上面所说的都是事实，但我仍然相信在适当的帮助下，开发人员是可以写出优秀的文档来的。做起来也很简单，考虑周详一些就行了。“倒推法”是最好的方法之一。这是我从一个朋友的博客上看到的，一读完介绍，我就认为这个方法肯定有效。但是，在心动和行动之间还是有很长一段距离的，我花了一年的时间才把这个方法理清楚并付诸实践。但效果却是出乎意料地好。

基本的理念很简单。不要从代码或者任何技术方面入手，而是从最后需要的事物（如概述信息）开始。先写一个项目的目标是什么，表现出你的自豪感，然后再介绍一下这些目标。接下来就是一份常见问题解答（FAQ）文档。在这份 FAQ 文档里，你可以讨论和解答一些深层次的细节性内容。随后，就像 4.3 节中所说：通过 FAQ 来描述用例，然后再提供真正的代码和相应的 Javadoc。

仅仅是把流程反转过来处理就能提高有效性，根源在何处呢？要知道在写最初概述信息的时候，作者是不可能对产品的技术细节了若指掌的。先写概述信息，这对于开发人员来说确实需要一点自我约束，因为开发人员总会迫不及待地想写代码。所以要在开始写代码之前，先克制一下自己的喜悦之情，把自己的想法全部整理出来。尽管这样做需要一点自我控制，但也算得上是一件很自然的事情，因为在开始编码之前，大家在脑子里也总该对全局有一个宏观性的认知。

两 头 怕

不是说只有开发人员才害怕写文档，就算是文档工程师也害怕改 Javadoc！我选择了

Geertjan Wielenga 作为本书的编辑。如果说没有他的帮助，这本书都不知道猴年马月才能出版，因为我记录的实验室笔记不是用普通的白话文所写，其中含有大量术语还有一些别的晦涩内容，因此需要编辑对其进行很多额外处理。（我很想看看这个备注栏里的内容经他修改之后会剩多少，因为这一篇内容其实带有一点对他的批评。）

Geertjan 喜欢写一些关于 NetBeans 平台相关的文档。同时他也是一位非常著名的博主，他负责维护一个 NetBeans 平台相关的站点，还不时地在上边写一些教程、采访稿、故事等。毫无疑问，他绝对是 NetBeans 文档作者中不可或缺的主力。夸张一点说，如果没有他，可能今天使用 NetBeans 平台的人就会少上很多。

从另一个角度来看，我无意抹杀 Geertjan 的功绩，但他所做的工作只是整个项目的一部分，而且只是入门的部分，比较适用于初学者。为了完善文档，我们还需要让 API 的设计人员来写一些资料，也就是说开发人员还是要自己上阵。如 Javadoc、用例等，都需要他们来完成。尽管不同的文档其复杂程度并不相同，但有一个共性，那就是告诉别人如何使用 NetBeans 平台。除此以外，这些不同类型的文档之间几乎没有任何关联。这真让人感到惭愧，因为这些文档都针对相同的用户群，而且这些用户都想从文档中得到尽可能多的信息。他们希望所有信息在需要用的时候都能信手拈来。当用户使用 IDE 进行编码并正在使用代码自动提示功能的时候，他们希望能从编码的状态直接跳到当前指定方法的相应教程。要实现这个功能，也不是太难，其实也只需要在 Javadoc 中添加一个指向相应教程的链接就可以了。

但现在就会出现一个沟通方面的问题。开发人员不了解这些教程，所以他们无法在写 Javadoc 的时候给出这样一个链接。而且这种教程类的内容通常是在 Javadoc 完成后才开始写的。另一方面，Geertjan 也不太敢自己动手去改这些 Javadoc 吧。我和他就这个问题讨论过多次，我尽力去说服他，告诉他这件事是必须要做的，但到目前为止还是说服不了他。所以开发人员害怕写文档，而写文档的人则害怕修改任何开发人员提供的资料。这样的直接后果就是，我们有两类文档，但两者之间基本没有关联。

我确信这不是什么好事，但却不知道该怎么处理这种情况。也许我们应该使用那种自下而上的方式，一定要先在 Javadoc 中加上一个教程的链接，然后才允许编写相应的教程。这样做也许可以解决这种困境，而且 Geertjan 也不用再害怕编辑 Javadoc。但我还是需要实践来检验一下，这种工作方式真的有效吗？

而且，开发人员之所以不喜欢写文档还有一个原因就是他们写完代码以后，估计就忘记了这事。当然他们不会永远都不记得这事，但他们忘记的次数足以让我们对此事提高警惕。但按前面说的那种倒推的工作方式来做肯定是不行的。只有当项目结束的时候才会正式需要概述信息，至少对于市场部来说是这就是信息源。而最终用户（至少是负责编写教程的文档工程师）则比较喜欢 FAQ 的内容，同时用例和 Javadoc 的需求市场也很明确。所以越来越需要工程师去写文档，但对于他们来说，当事情做完了再去写文档，就是变相地做重复事情，自然很无聊。

但做事的顺序也很重要。首先肯定是写概述信息，然后才会一步步地涉及代码这种细节性内容。如果不按这个顺序，前面所说的倒推的流程工作方式就无效了。前不久，我在为一个完成的

项目写一份概述信息的时候，发现这个过程真是痛苦，而结果也一点都不好。我先写概述，却在第三段突然加入了一个命令行操作的例子来说明如何使用这个项目的产品。这根本就不是概述信息！但对于工程师来说，事情就会这样：他们想分享尽可能多的有用信息，所以他们如果在项目结束以后才去写这个概述信息，就会很不顺利。唯一的解决方式就是在开始编码之前，就先写这份概述信息。

我必须承认，我越用这种工作方式，就越喜欢它。最后，写概述信息和 FAQ 竟然变得非常有趣。但是，是不是每个人都适合这种工作方式还有待验证。我希望对于开发人员来说，它不仅成为一个放松的机会，还可以适用于更多情况下。

16.3 尽职尽责的监控者

世事并非尽如人意。不管你有多么优秀，你的团队多么出色，但肯定时不时地会有人犯点错误，偶尔地改错了一个 API。只要你的团队不是那种完美的团队，或者说有新成员加入的时候，那么你都需要对团队成员的工作进行检查。你需要检查团队没有对外作出不切实际的承诺，因为一旦做出这样的承诺，就意味着一定要做到。当然我这样说，并不意味着领导者就要天天在每个程序员身边晃，告诉他们每一步工作应该如何开展。其实你需要的是一个自动化系统来对 API 中重要的调整加以监控并随时通知你。

这样一个自动化监控系统要做的事情其实是非常明确的。但问题在于，哪些内容是比较重要的，对其进行检查的频率是多少呢？很明显，在一般项目中只要涉及 API 和类库的内容都应该进行检查。如果能在每一次提交后，代码都会自动构建一次，或者至少做到日构建，这样就会非常有用，因为这样做可以确认提交的代码可以正常通过编译，这一点非常重要。而且最好在构建后也进行自动化测试。可以通过自动化测试得知类库的测试覆盖率，这样就了解了还有哪些内容根本没有经过测试。而且，还有一点也很有意义，那就是在必要时提醒说，整合新的内容可能会降低测试覆盖率，有可能给类库未来的稳定埋下一个炸弹。

如果没有足够的测试怎么办

在写测试代码时，开发人员经常会问到底要写多少才算合适。简单的答复就是，只要写的单元测试有效，就继续写。下面有一个更加精准的答案，但不必将其作为一个固定的教条。

有很多工具可以统计测试的覆盖率。我们选择了 EMMA (<http://emma.sourceforge.net>) 来统计我们程序中测试代码的覆盖率。比如说，在一个项目上面点击右键菜单中的某一项，就会执行很多程序代码和自动化测试。在这个执行过程中，EMMA 就会收集所有调用方法的信息、访问过的类和代码行，然后在浏览器中给出一个总结性报告。

根据访问过的方法来计算测试覆盖率，这种标准要求并不高。但即使这样，也很难达到百分之百的覆盖率。即使达到了，也不能确保最终的产品能正确运行。每一个方法都会有不少参数。使用某些参数，测试用例运行正确，但这并不表示换一些参数，会仍然正确。

最后则是通过代码分支或代码行来进行覆盖率的统计。如果在某个方法中有 `if (...) { x(); } else { y(); }` 这种条件代码，你也许想确认 `x` 和 `y` 这两个方法都会被调用到。EMMA

这个工具就可以支持这种需求。它可以帮助用户确定是否每一行代码都被调用到了，通过该工具，我们可以确信程序中的代码没有一行是无用的。

只不过，即使所有的代码行都被调用过了，也并不表示程序代码就没有问题了。

```
private int sum = 10;

public int add(int x) {
    sum += x;
    return sum;
}

public int percentageFrom(int howMuch) {
    return 100 * howMuch / sum;
}
```

如果在测试时通过不同的参数来确保 `add` 和 `percentageFrom` 这两个方法都被调用了，那么肯定是件好事。但如果我们先调用 `add(-10)`，然后再调整 `percentageFrom(5)`^①方法的话，那么前一个方法将 `sum` 设置为 0，而第二个方法则会去除 0，程序就会抛出一个异常，执行失败。为了确认我们的程序不会出现类似的问题，我们必须在各种情况下对每一个方法都进行测试，特别是在内存数据彼此相关的情况下。只有这样，才能确认我们的程序代码在单线程环境下能正常运行。

但还有另外一个问题：Java 不是只有单线程支持。很多程序都会自己使用到多线程技术。即使程序不使用多线程，虚拟机也会自动创建 AWT 的事件调度线程、有负责清理数据的线程及其他的线程等。你完全不能确定有多少线程在运行。有时候，垃圾收集器会从内存中清除那些已经无法访问的数据对象，此时也有可能改变程序的行为。我们曾经碰到过一次死循环的问题，要重现这个问题，必须要同时使用 Mozilla 浏览器和 Evolution 客户端^②才行，因为这两个条件会占用大量的内存，因此 Java 垃圾收集器会不停地运行以回收内存。这种类型的问题就很难测试到。

所以我们建议测试覆盖率工具主要是用来检查还有哪些内容没有测试过，而不是说测试过的内容就正确了。开发人员仍然需要提醒自己，不管测试的覆盖率有多高，在测试过的代码中仍然可能存在 bug。为了解决程序的这类不确定性问题，我们只能建议，在出现问题的时候，要写一个测试对其进行验证。只要有 bug 报上来了，就写一个测试用例对其进行验证，避免出现修正 bug 不彻底，一再反复的问题。这样做，测试覆盖率就集中在那些已经确认有问题的代码上。

除了进行常规测试以外，还需要对类库进行很多特别的测试。就好像写完了 Javadoc 以后，

① 原文这里是 `percentage`，而不是 `percentageFrom`，从上面的代码看应该是 `percentageFrom`，此处可能是作者的笔误。
——译者注

② Evolution 是世界上使用最广泛的 Linux 协作软件。它将电子邮件、日历、联系人管理和任务集成到一个易用的应用程序中。官方网站为 <http://www.gnome.org/projects/evolution/>。——译者注

除了改那些错误字、标点以外，还要对 Javadoc 中的链接进行验证，确保没有失效的链接。搞不好受到本书的影响，你会觉得 Javadoc 还不够。在这种情况下，你还需要通过一种方式来描述其他的 API，而且 API 的用户可以很容易地就找到这些帮助内容。另外你可能想根据 API 的稳定性对其进行分类，就是按照 4.5 节所说的那样做。可能还有别的原因，如你希望告诉用户，新的 API 与旧版本相比，都做了哪些调整。出于以上的考虑，你可能就会想使用 NetBeans 为 Javadoc 提供的扩展功能。

NetBeans 对 Javadoc 功能的扩展

为了更好地在开发中遵守本书给出的一些规则，我们需要对标准的 Javadoc 生成工具做一些功能增强。我们希望它可以支持版本标识，可以用一种更简单和更标准的方式从上至下地对用例和单个的 API 都进行描述。我相信任何一个开发类库或框架的开发人员，只要想遵守本书中所给出的这些建议，那么就会需要这样的一个工具。所以尽情地使用 ANT 脚本^①和 XSL^②转换来完成这些工作吧，因为它们都不是 NetBeans 特有的内容，可以独立使用。

基于“20 个问题”的模板中包含了很多固有的重要问题，根据它来生成 Javadoc 时，就能在 Javadoc 中包含各种指导性的问题，从而帮助每一个模块的开发人员认识到自己负责的类库到底对外暴露了哪些内容。比如说，有一个问题是关于读取系统属性的，它使用了反射技术来访问了非 public 的类，来读取文件和开启 Socket 连接。这样做并不是要以一种强制的方法提醒开发人员，说你做得不对，它的目标在于以文档方式来确认是不是做了这事。这样做的目标是将做的事情归类整理，以方便开发人员之后来解决这些问题。

在 NetBeans 项目中，使用一个称为 arch.xml 的文件来存放这些问题的答案。arch.xml 这个文件的格式说是 XML，其实更像 HTML。只不过这个文件里加入了两个非常重要的扩展。一个是<api>，它可以用来标识一个 API，还有一个是<usecase>，它可以用来描述一个用例。如果你使用了这两个标签，标签里的内容就会被抽取出来，放置到 Javadoc 的概述文档中。还有一个 API 的入口页面，就是任务列表 API 页面^③，它里面放置所有用例的概况，并使用一个表格对用例的属性、反射和文件等 API 加以说明，用户可以通过这份文档在 Javadoc 中找到自己需要的类。于是就很容易通过这样一个表格了解一个 API 到底能完成什么功能。如果了解更多的内容，API 的用户就可以去阅读那份完整的“20 个问题”的文档。

这些内容并不是 API 的用户唯一感兴趣的地方。他们对新 API 提供了哪些新功能也很感兴趣。特别是那种在设计时就考虑到快速改进的类库，它们经常一个月就更新多个版本，在这种情况下，用户肯定很关心每个版本都提供了哪些新功能，做了哪些调整。为了支持这个功能，需要另外一个 XML 文件，我们把它定义为 apidesign.xml。这个文件中放置了 API 变化的相关信息。不管是向现有 API 中添加一个方法还是一个类，我们都会把模块的版本号提高一些，表

① ANT 是一个将软件编译、测试、部署等步骤联系在一起加以自动化的工具，大多用于 Java 环境中的软件开发。由 Apache 软件基金会提供，官方网址为 <http://ant.apache.org>。——译者注

② XSL (Extensible Stylesheet Language)，即可扩展样式表语言，是 w3c 推荐的一种标准，用以定义 XML 文档的转换与格式化。XSL 语言家族主要包括三大部分：XSLT、XPath 及 XSL-FO。——译者注

③ <http://bits.netbeans.org/6.0/javadoc/org-netbeans-spi-tasklist/overview-summary.html>。——译者注

示有所升级，同时还会向文件中添加相关的修改信息。这些信息包括了他人可以读懂的描述信息，告诉用户都改了哪些东西，还包括可以快速跳转到对应的 Javadoc 的链接、发生改变的版本，以及对于那些对 API 所有信息都有兴趣的人来说，还有我们的 bug 追踪系统的提示。这个链接会包括那些与 API 的细节有关的内容，包括在 API 评审中提及的所有建议、回复以及考虑到的相关内容，对于用户来说，提供这些内容有时候还是很有必要的，因为这些内容决定了最终文档的内容，也可能解释了为什么会对一个 API 进行某种调整，其原因为何，目的所在。

我们的构建脚本会将 API 最新的五个调整内容放置到一个概述页面中。而其他的调整内容则会放置在一个独立的页面中，该页面中放置了 API 所有改进的内容。

API 的调整信息可以包含很多内容，如源代码兼容性、二进制兼容性、功能兼容性等，当然，也可以什么都不放。对于那些需要将代码移植到新版本上的用户来说，这样做是非常有用的。不过有时候，可能很难判断某个调整是否是兼容的。我们也说不清这个调整是 100% 的兼容，还是 99% 的兼容。而在这里，只允许我们对兼容问题说“是”或者“否”，从事实角度来说，也只能是二选一，但是否允许我们说这个调整是 99% 的兼容呢？据我所知，我们经常是写成兼容，但我们还会加一句说明，指出在某些情况下，并不是 100% 的兼容。

如果你想了解更多 Javadoc 增强工具方面的内容，可以通过访问 <http://Javadoc.apidesign.org> 来获取。

将做过的调整都记录成文是件好事。但只做这一件事还是不够的，还需要对 API 本身进行跟踪。首先（也是最重要的一点）就是检查对外内容的变化，包括 public 类及其内部的 public 和 protected 方法。你需要了解这两类内容的变化。如果从无缝集成类库的角度来看，最重要的事情就是检查它在向后兼容性方面的调整。一旦对外发布了类库的一个新版本，就需要对 API 做一个快照作为历史参照，然后在后续调整工作中，将调整的内容也做一个快照，并与原来版本的快照进行比对，以保证没有信息被遗失，也没有任何不兼容的变化。

还要对另外一些内容进行针对性检查，就是防止偶然性的变化。在 16.1 节中我们已经指出一定要避免在 API 中出现随机性的调整内容。一旦它们出现，就意味着在未来都要对它们进行维护了，而且维护的时候还要保证其兼容性。所以我们每天都会自动列出相应的调整，与前一天进行比较，并对两者的差距进行比较，以避免出现类似的情况。

NetBeans 的 API 签名测试工具

JCP^①提供了一个 API 签名的测试工具，NetBeans 在这个工具的基础上设计了一个自用的 API 签名测试工具^②。在原有工具的基础上添加了一些新功能，以便能和我们的 ANT 构建脚本

① Java Community Process（简称 JCP）是一个开放的国际组织，主要由 Java 开发者以及被授权者组成，职能是发展和更新 Java 技术规范、参考实现（RI）、技术兼容包（TCK）。Java 技术和 JCP 两者的原创者都是 SUN 公司。然而，JCP 已经由 SUN 公司于 1995 年创造 Java 的非正式过程，演进到如今有数百名来自世界各地 Java 代表成员一同监督 Java 发展的正式程序。——译者注

② 作者这里指的是 Signature Test tool，它也是一个开源软件，放置在 java.net 这个开源代码库网站上，可以通过 <https://sigtest.dev.java.net/> 来访问它的官方网站。它其实是 JCP 提供的 TCK 中的组成部分。——译者注

配合得更好,同时也能支持模块的版本机制。有了这个工具以后,在进行快照比较时,一旦发现有任何不兼容的情况出现,就会发出相应的报告。如果你想深入了解这个工具,可以访问 <http://sigtest.apidesign.org> 这个网站。

签名并不是对外暴露的唯一 API。开发人员还需要对其他一些重要的内容进行检查。通常来说,各种文件的放置位置也非常重要。对于类似于 NetBeans 的这类模块化系统来说,还需要去关注多个组件间的依赖关系,因为新添加了些依赖关系以后,就可能会对组件的运行条件形成新的约束,对一些程序进行集成时就有可能产生问题。

检查框架

为了测试这些额外的 API,NetBeans 提供了一个特殊的框架,在构建过程中来做这些工作。基本思路是使用一个特殊的 ANT 任务,它可以通过配置来输出纯粹的文本信息。可以使用常见的文本比较工具来比较这些输出的文本信息,找到不同之处。为了执行这个任务,需要指定一系列模块对应的 Jar 包文件,这个任务会读取这些文件的依赖信息,并对其进行分析。要知道,在 NetBeans 平台上的所有模块,都是通过 Jar 包中的 manifest 文件^①来定义依赖关系的。

可以对这个任务进行配置以产生不同的输出格式,并将这些输出写入一个给定的输出文件里。这些输出信息可能包括所有公开包的列表,该列表随后可以被传给签名处理工具;所有模块及其版本的列表,以提醒我们添加了新模块;带有依赖关系的所有模块的列表。我们经常让开发人员自行对他们所做的调整进行测试,正确的测试结果就会放入版本控制系统,与源文件放在一起。如果在构建时发现测试结果与正确的结果不匹配的话,就会出错。这样做就可以避免我们在无意识的情况下做了一些不正确的调整。但我们仍然发现在很多情况下,对结果文件进行改变的意义并不大。因为这样只表示在执行构建任务的机器上,其验证结果是正确的。将上一次构建产生的正确文件与新构建产生的文件进行比较。如果有什么不同,就会给所有的开发人员都发一封电子邮件进行通知。这样每个对这事感兴趣的人就会尽快地进行检查,看是否添加了一些不合适的依赖关系。

我也不确认哪种方式更好。那种引起构建失败的处理方式需要开发人员投入更多的关注。现在使用的方式则不需要他们投入太多的关注。也许应该在这两者之间取一个平衡,想出一个更优的方案。如果我们能够明确哪些是不想要的依赖关系,比如说“文本编辑器不应该依赖于编译器”,或者说“支持 Html 的时候,不需要支持 Java”,这些内容都可能会引起构建失败。我们会继续尝试以找出更有用的方式。

听了 NetBeans 项目是如何使用这些工具,不知道是否可以激发读者这方面的灵感呢,不过这并不重要。重要的是,读者可以根据这些思路来创建自己的“监控者”系统,对开发人员的工作加以检查。只有做到这一点,才能让一个团队在设计 API 的时候,不会产生一个让人无法接受

^① 在 NetBeans 的模块 Jar 文件中,有一个 META-INF 目录,下面有一个 MANIFEST.MF 文件,其中放置的 NetBeans 指定格式的内容,包括模块的基本信息和依赖关系。——译者注

的最终方案。

还有一种可选方案，就是用一个人来做这件事，他会收到要做的补丁，然后决定接受补丁或者拒绝补丁。这样做也很有效。但它会受到规模方面的限制。本书也不鼓励这样做，因为想这样做，还需要有一个“全知全能”的人。要想从这种工作方式中解脱出来，就要做到充分透明，而不是基于个人的知识。如果有好的工具、好的测试用例来对一个 API 常见的因素加以测试，那么效果肯定比用一个人去做这事要好得多。

16.4 接受 API 的补丁

在 14.4 节中已经指出，如果一个 API 设计得好，那么维护它的成本和维护普通的代码并没有什么区别，有时候可能还会更简单一些。但这个结论的前提就是要保证那些由 API 用户所提供的代码的质量。如果这些用户只是简简单单地抱怨一下，或者要求类库 API 做一些调整，而这些调整并没有多大意义，或者并不合适，那么，你的工作可就不简单了。如果你接受了不好的建议或者代码，那么日后的维护工作就变得路途坎坷了。要避免这种后果，就只能接受好的建议和好的补丁。

去它的多线程

想分辨一个建议是好还是坏，有的时候并不容易。MPlayer^①项目就是这样的例子。MPlayer 是一个开放源码的电影播放工具，也是一个视频编码工具，它是使用 C 语言开发的，有人觉得这个软件应该充分利用多核处理器的优势，建议作者应该重写代码以支持多线程。听起来这个建议也是蛮合理的。利用双核处理器，视频的压缩速度可以快一倍。现在，几乎每个人都在用多核处理器，就算现在还没有完全实现这一点，估计过不了多久，大部分人都会用上多核处理器了。只不过这个项目拒绝接受这个建议。奇怪吗？不，一点也不，因为就算只使用单核处理器，这个软件有时候还会崩溃。如果把代码改写以支持多核处理器，估计出的问题会更多。所以在我看来，MPlayer 开发团队拒绝接受这个建议是非常正确的，虽然他们这样做可能会使自己的项目产生分支，但相比之下，听从建议的风险更大。事实上，有不少多核处理器的爱好者已经建立了自己的项目^②，也就是说 MPlayer 已经有了一个分支。但在我看来，也许多年后，原始的 MPlayer 项目可能还在，而分支项目可能已经消失了。

通常来说，所谓好的建议就是能将产品的开发成本降低，并缩短产品推向市场的时间。但不管怎么说，如果你有了一个 API，而且有人希望你对 API 做些调整，你不太会有市场方面的压力。你的 API 已经出现在市场上了。除非这个需求是所有人都希望有的功能，否则你根本不用有时间方面的压力。你只需要关心总的开发成本。如果你接受了一个补丁，你就成为这个补丁的主人。

① MPlayer 是一款开源的多媒体播放器，以 GNU 通用公共许可证发布。此款软件可在各主流操作系统使用，例如 Linux 和其他类 Unix 操作系统、微软的视窗系统及苹果计算机的 Mac OSX 系统。MPlayer 是建基于命令行界面，在各操作系统可选择安装不同的图形界面。它的官方网站为 <http://www.mplayerhq.hu/>。——译者注

② 这个源自于 MPlayer 的项目称为 MPlayerXP，官方网站为 <http://mplayerxp.sourceforge.net/>。——译者注

也就是说，你可能需要花费很长的时间或精力来对这个补丁进行维护。

有什么理由需要冒这个风险呢？你接受的补丁可能带了一堆 bug，补丁的作者并没有做到所承诺的内容。这种风险是不可能完全规避的。但如果你要求补丁的作者提供全面的测试，要覆盖他所修改的代码，那么你可以至少对这个补丁所提供的基本功能保有信心。接下来的问题是，这个补丁在修正问题的时候，可能会引起反复的情况。假设能对你类库中其他部分的内容也正确地加以测试，那么这个风险也是可以合理规避的。但如果提交补丁的人也是第一位为你的类库提供测试的人，那么补丁整合到系统后出现问题的风险几率还是很大的。从这一点上来考虑，也需要为代码提供全面的测试覆盖。换个角度来看，问题的反复并不是唯一要规避的风险。这个补丁也许会阻碍 API 在未来进行改进。因此你需要在接受补丁之前进行相应的评审，可以参考本书给出的各项建议对其加以衡量，还可以参考那些如何方便 API 在未来加以改进的建议。补丁还可能存在一个问题，就是没有提供相应的文档。不过，这件事情处理起来比较简单，就是要求补丁的提交者再补齐相关的文档和一些较高层面上的用例。不过这个问题可以通过更好的测试覆盖率来加以弥补，因为好的测试代码往往就是最好的文档。最后要关注的就是正确的版本号。在支持模块版本的系统中，每一个 API 的改变都应该提高 API 的版本号，并提供相应的文档公开描述该版本提供了哪些新的功能。

在接受一个补丁之前，最好要进行一次评审。当你收到外部提交的一个补丁时，这事做起来并不难。而且，“在提交代码前进行评审”并不是一个仅仅处理外部提交内容的最佳方案，即使提交补丁的人是你团队中的成员，也应该做这事。在没有将代码整合到系统中之前，提交补丁的人为了让其提供的补丁能被成功接受，会非常重视你要解决的问题。通常来说，在整合以后，提交代码的人就不那么愿意去修复问题了。这也就是说，如果想最小化维护成本，关键就在接受补丁之前。在代码提交以后，如果出现了问题，那么就是维护人员的工作了，提交补丁的人是不会来做这些工作的。

接受补丁很有讲究

作为一个开源框架的维护人员，我敢说我一直都非常乐意接受他人提供的优秀补丁。有时候，我感觉我对他们提的要求很高，要全部满足这些要求还真不容易。我有时候甚至很怀疑，对于一个没有正确测试，或者没有提供文档的补丁，到底要不要接受呢？我很清楚，如果接受了一个满是 bug 的补丁，那么以后维护的工作量就非常大了。通常来说，我会让补丁的作者对补丁再加以改进。他们要么就改进后给我，要么就不给我了。要是没给我，那恰恰说明了我的判断是对的，因为这样的提交人也不太可能在以后会为这次的变更提供良好的支持。

如果从类库的用户角度来看，他可能会问：“向类库提供一个补丁有意义吗？”特别是提交补丁的门槛还很高，这样做的意义何在？还不如直接从类库主干上延伸出一个分支来，或自己绕过这个问题不就好了。当然，这要看整体情况而定。但新建分支或逃避问题所引发的后果是非常清楚的：维护成本会越来越高。更不用说还得自己发现问题，并找出问题点，然后知道如何来修正这些问题，这每一步的代价都不菲。尽管如此，你还是宁愿自己承受这些麻烦，也不愿意将修复交给提供类库的项目。你需要自己来修改代码，而且这个修改还要一直继续下去，一旦有新版

本发布，就得把这个修改放到新版本。有时候在新版本上进行修改并不是一个很顺利的操作，因为类库作者根本不知道你做了一个类库代码的分支版本，因此可能会有各种冲突，不能正常运行之类的事情可能会时有发生。长期做下来，成本高昂啊。相比之下，还是花点时间，根据项目的需要来提供一个正确的补丁。这件事只要做一次，你所捐献的这个 API 补丁就会进入新的版本，由类库维护人员来继续维护工作，这样多好啊。

每一个开源项目对于是否接受一个补丁都有自己的要求。但通常来说，所有的需求都是在衡量了多方面内容后所做出的一个妥协。质量不好的补丁是不能接受的，因为这意味着它会在后续过程中带来维护问题。另一方面，如果想让一个 API 的用户捐献补丁的话，那么就不能让捐献补丁这事太复杂，其相应的成本决不能高过用户自动创建 API 分支的成本。是否接受一个补丁，最终是要考虑 API 用户的利益。但一个开源项目应用得越广，那么 API 会变得更加优秀，更多的用户会明白捐献补丁才是长久利益的根本所在。对于每一个人来说，捐献补丁才是最节约成本的处理方式。



17

利用竞赛游戏来提升API设计技巧

我所在部门的领导总是说：“正确的判断来自于经验，而经验来自于错误的判断。”如果你犯了足够多的错误，那么你就能够从中学到很多东西，也就可以成为一个好的 API 设计者。我相信我们以前所做的很多错误决策，反而促成了我们今天很多深入的思考。只不过这种学习曲线在开始的时候是很痛苦的。它需要我们花费很长的时间，而且从中学到的知识还不能轻易地传授给他人。

在设计中最常见的一个主要问题就是专业术语的缺乏。人们会就某个东西是不是 API 争论很久。这么多年来，我已经疲于向他们重复本书第一部分中所说的内容，都已经疲了。同样，人们常常把 API 的改动混同为不兼容的改动，我也懒得再一个个地和他们讨论这些事了。所以我决定举办一个 API 的设计研讨会。所谓 API 的设计研讨会其实就是一个游戏，用稍嫌极端的方式来证明一个观点。所有参与 NetBeans 核心 API 开发的工程师都加入到这个游戏中。从那以后，像那种某件东西是否为 API，某些改动是否兼容之类的话题，讨论起来就比较省心了。我只消说：“你把这个调整放在 API 设计研讨会这个游戏中，试着看会不会失败。”事情立刻就一清二楚。他们会为这个游戏设计好规则，而且假设有人在检查这个调整是不是一个对 API 的调整，同时该调整是不是百分之百的兼容。正因为这个游戏能够清楚说明 API 设计中遇到的一些问题，在此将其推荐大家。

17.1 概述

这个 API 的设计研讨会游戏采用的是比赛的形式。它的目标是教参与者有关设计 API 时要考虑的演进问题。参加者需要完成一个简单任务：编写一个 API。经过评估之后，会收到一个修改任务，对之前完成的 API 加以改进。以上过程往往会重复多次。之后，会再次接受评估。在这一过程中，不会有一个评判委员会来评选最佳方案，而要由参与人员自己来做相应的评估。每个人都会看到所有的解决方案，然后从这些解决方案中找出一些与改进有关的问题。为了达到这一目标，他们会在前一个 API 版本的基础上写一个测试用例。如果这个测试用例可以用于前一个版本，但却不能用于后续版本，那么就可以得分。如果能证明别人提供的解决方案中存在问题，又要自己写一个无懈可击的 API，这两方面都能挣分。

第一次玩这种 API 设计研讨会游戏全球首演是在 2006 的 7 月中旬。当时只有 NetBeans 核心成员参与了这个游戏。目标很简单，就是演练 API 设计技巧，并检查一下是否能够处理未来可能遇到的类似设计争论。同时我还想和 OOPSLA 2006^①大会的一些与会者进行这样的讨论，但我需要事先检验这一思路是否可行。所以我现在还期望在以后的 OOPSLA 大会中有一个这样的专题再尝试一次。只有把那些专门负责设计 API 的人找到一起来演练，只有这样，才能通过辩论的方式来积累最佳实践。事实上，当 NetBeans 核心团队的成员同意来试着参与这件事的时候，效果好极了。

那次比赛始于 7 月 12 日，星期三。参与者集中在一个会议室中，每个人都可以访问一个框架性的 NetBeans 项目，同时每个人还拿到一份文档，列明了这个要设计的 API 的用例。他们大概用了一个小时，然后把各自关于 API 改进方面的不同意见提交给我。

第二天，这些参与者又碰面了。这次给了他们一些新增加的用例，他们又用了一个小时来实现这些用例。然后再把结果提交给我。

接下来的第三天，我发了一份电子邮件告诉他们如何来查看其他参与者提交的源代码。这一阶段的目标是攻破他人的 API，大家编写测试代码来证明其他人设计的 API 有演进问题。到周末的时候，就要把他们测试的结果发给我。

到 7 月 17 日星期一的时候，我公布了游戏的结果。如果发现他人 API 的漏洞，那么就得分。如果写出了无懈无击的 API，也就是没有被他人发现问题的设计者，则可以获得五分。这种竞赛的目标是为了磨炼 API 的设计技巧，并学会如何在设计完的 API 中发现错误。掌握了这些技巧就会成为游戏的赢家。但一定要知道，游戏的目标绝不是要证明某位开发者没有足够的 API 设计技巧。其实在这个游戏中，没有输家，所有提交的内容都是匿名的，我随机地为这些人都起了一个笔名。只有做得好的人才会公布出来，接受大家的祝贺。

获取 API 的设计研讨会的源代码

如果需要的话，你可以去看一下这个游戏参与者所写的源代码，学习一下，执行一下，再修改一下。源代码可以从 <http://source.apidesign.org> 这个网址来下载。

以上就是对第一次 API 的设计研讨会的简述。接下来我会对此进行详细的描述，讲解在这个比赛过程中都发生了哪些事情。

17.2 第一天

有史以来的第一次 API 的设计研讨会始于 2006 年 7 月 13 日，星期三。一开始的任务很简单：要模仿 Boolean 类的功能来写一个 Circuit（布尔电路）类。与会者都会收到一个项目的模板，模板中只有一个空白的 Circuit.java 文件，让大家用来写 API 定义。与会者还会收到一个测试文件，列举了这个 API 应该完成的三个功能。

^① OOPSLA 2006 是在 2006 年 10 月 22 日至 26 日举办的，所以作者在此之前在 NetBeans 举办了这个竞赛。

```

/** 这个 API 设计研讨会的初始需求就是创建一个类似 Boolean 类功能的 Circuit 类。
 * 这样的一个 API 能够根据基本计算规则来创建一个 Circuit，或者根据输入值来进行计算。
 * <p>
 * 所谓的基本运算规则包括以下内容：
 * <ul>
 * <li>非运算，也就是对于一个输入值就对应一个输出值，当两个输入值都为 1 的时候，输出才为 1，否则输出为 0
 * <li>或运算，它有两个输入值，并对应一个输出值，只有当两个输入值都为 0 的时候，输出才为 0，否则输出为 1
 * <li>or - has two inputs and one output. The output is always 1,
 * except in the case when both inputs are 0
 * </ul>
 *
 * <p>
 * Circuit 这个类可以像布尔量一样支持各种公式的计算，根据输入值得出相应的结果。
 * 参见下面的测试任务
 *
 * <p>
 * 相关链接：
 * <ul>
 * <li>
 * <a href="http://en.wikipedia.org/wiki/Truth_table">
 * 值为 True 的表格</a>
 * <li>
 * <a href="http://en.wikipedia.org/wiki/Tautology_(logic)">
 * 恒真式</a>
 * </ul>
 */
public class CircuitTest extends TestCase {
    static {
        // 可以在此处放置任意代码
    }

    public CircuitTest(String testName) {
        super(testName);
    }

    protected void setUp() throws Exception {
    }

    protected void tearDown() throws Exception {
    }

    /**
     * 创建一个 Circuit 对象实例来计算表达式 (x1 and x2) 时的输出值。
     * 对于该表达式，当输入为 (false, true) 时，对应的输出值应该是 false。
     * 当输入为 (true, true) 时，对应的输出值应该是 true。
     */
    public void testX1andX2() {
        fail("task1");
    }
}

```

```

/**
 * 创建一个Circuit对象实例来计算表达式 (x1 and x2) or x3 时的输出值。
 * 当输入值为 (false, true, false) 的时候, 输出值是否为 false。
 * 而输入值为 (false, false, true) 的时候, 输出值是否为 true。
 */
public void testX1andX2orX3() {
    fail("task2");
}
/**
 * 创建一个Circuit对象实例来计算表达式[x1 or not(x1)]or x3 时的输出值。
 * 对于该表达式, 任何输入对应的输出值都应该是 true。
 */
public void testAlwaysTrue() {
    fail("task3");
}
}

```

目标很简单, 就是要设计 Circuit 这个 API, 然后为 3 个空白测试提供一个实现, 这个实现要能很好地对 API 进行测试。

之所以在这个 API 的设计研讨会中用创建 Circuit API 这样一个初始任务, 是考虑到整个电路不过是可执行三种操作 (NOT、AND 和 OR) 的电路单元联成的网, 再加上一些输入元素。

□ NOT 一个输入, 一个输出。输入为 false 的时候, 输出为 true; 输入为 true 的时候, 输出为 false。

□ AND 两个输入, 一个输出。只有两个输入都为 true, 输出才为 true, 否则输出就为 false。

□ OR 两个输入, 一个输出。只有两个输入都为 false, 输出才为 false, 否则输出就为 true。

由于需求比较简单, 这个 API 也就相对较小。一个小时左右就可以写完了。但它其实并不算简单, 因为这个 API 的几个版本都要允许 API 的产品为 API 所用, 于是出现了一种很有意思的自引用。在这种情况下对 API 加以改进, 就有可能引发很难处理的向后兼容的问题。只不过这些都是以后才会出现, 可以先不考虑。更有意思的是在 CircuitTest.java 文件中对于初始任务的描述。

(1) 创建一个 circuit 对象, 对 x1 and x2 求值, 再验证输入 (false, true) 的时候, 结果为 false, 而在输入为 (true, true) 的时候, 结果为 true。

(2) 创建一个 circuit 对象, 对 (x1 and x2) or x3 求值, 再验证输入为 (false, true, false) 的时候, 结果为 false, 而在输入为 (false, false, true) 的时候, 结果为 true。

(3) 创建一个 circuit 对象, 对 [x1 or not(x1)] 求值, 再验证不管 x1 的值是什么, 结果一定是 true。

这个 API 的设计研讨会的参与者表现都不错。大部分人在一个小时后都做完了这个设计。大部分的设计都非常好, 质量非常高。

自己试一下

读者可以从我们的网页上下载这个项目的模板。然后按照 CircuitTest.java 这个测试文件给出的提示来自行设计这个 API。读到这里, 我要提醒一下, 如果你想试一下, 那么现在就是

最后的机会，要抓紧，因为接下来的内容我们将着重分析这些参与者都是怎么想的，又做了哪些事，还犯了哪些错。

在这次 API 的设计研讨会上，有一些解决方案出现了一些问题。我想在这里重复一下，以免其他人在玩这个游戏的时候也犯同样的错误，这样在设计的时候就更容易一些了。

17.2.1 非 public 类带来的问题

至少在两个解决方案中，忘记将一个非常重要的 API 变成 public 类（例如，day1/inputandoperation 项目中的那个）。这个错误在这个阶段还注意不到，但在游戏的后期，就会引发很多问题。这是因为 boolcircuit.CircuitTest 这个测试类和 boolcircuit.Circuit 这个 API 类，其实都放置在同一个包中。因此，这两个类可以用包中私有的保护方法相互调用，往往出了问题才发现。

要吸取的教训就是：再组织设计时，绝不要将 API 类和测试类放在同一个包中。如果采用这种方式，那么编译器就能够检查出你使用了非 API 的方法和字段。

17.2.2 不可变性带来的问题

使用 NetBeans 平台的开发人员都知道，一项内容如无必要，就不要对外公开，其实第 5 章中也给出过这个建议。所以，很多解决方案中，对外公开的内容都非常少。尽管这样的代码完成了既定目标，但仅仅是完成，并没有做一些多余的工作。结果有一些方案根本就不是布尔电路！但它却满足所提出的任务，你可以用这个类来运行 AND、OR 和 NOT 操作。但在参数值不同的情况来执行一个方法的时候，每次都得创建一个新的布尔电路！可以看一下目录 day1/inputandoperation 下面的代码，就是这样的。请看这个 API 是如何完成用例指定的功能的。

```
/**
 * 创建一个Circuit对象实例来计算表达式 (x1 and x2) 时的输出值。
 * 对于该表达式，当输入为 (false, true) 时，对应的输出值应该是 false，
 * 当输入为 (true, true) 时，对应的输出值应该是 true。
 */
public void testX1andX2() {
    inTrue = Factory.createSimpleBooleanInput(true);
    inFalse = Factory.createSimpleBooleanInput(false);
    Operation op1 = Factory.createAndOperation(inFalse, inTrue);
    assertFalse(Circuit.evaluateBooleanOperation(op1));
    Operation op2 = Factory.createAndOperation(inTrue, inTrue);
    assertTrue(Circuit.evaluateBooleanOperation(op2));
}

/**
 * 创建一个Circuit对象实例来计算表达式 (x1 and x2) or x3 时的输出值。
 * 当输入值为 (false, true, false) 的时候，输出值是否为 false，
 * 而输入值为 (false, false, true) 的时候，输出值是否为 true。
 */
public void testX1andX2orX3() {
```



```

    inTrue = Factory.createSimpleBooleanInput(true);
    inFalse = Factory.createSimpleBooleanInput(false);
    Operation op1 = Factory.createAndOperation(inFalse, inTrue);
    Operation op2 = Factory.createOrOperation(
        Factory.createOperationBasedBooleanInput(op1), inFalse
    );
    assertFalse(Circuit.evaluateBooleanOperation(op2));

    op1 = Factory.createAndOperation(inFalse, inFalse);
    op2 = Factory.createOrOperation(
        Factory.createOperationBasedBooleanInput(op1), inTrue
    );
    assertTrue(Circuit.evaluateBooleanOperation(op2));
}
/**
 * 创建一个Circuit对象实例来计算表达式 (x1 or not x2) or x3 时的输出值。
 * 对于该表达式，任何输入对应的输出值都应该是 true。
 */
public void testAlwaysTrue() {
    inTrue = Factory.createSimpleBooleanInput(true);
    inFalse = Factory.createSimpleBooleanInput(false);
    Operation not = Factory.createNotOperation(inTrue);
    Operation or = Factory.createOrOperation(
        Factory.createOperationBasedBooleanInput(not), inTrue
    );
    assertTrue(Circuit.evaluateBooleanOperation(or));
    not = Factory.createNotOperation(inFalse);
    or = Factory.createOrOperation(
        Factory.createOperationBasedBooleanInput(not), inFalse
    );
    assertTrue(Circuit.evaluateBooleanOperation(or));
}

```

day1/alwayscreatenewcircuit 下的解决方案也出现了类似的问题。和上一个解决方案差不多，它也提供了一个工厂方法通过不变的 boolean 量来创建 circuit 元素，只不过它还多了一个后门，可以将值由 true 变为 false。

```

/**
 * 创建一个Circuit对象实例来计算表达式 (x1 and x2) 时的输出值。
 * 对于该表达式，当输入为 (false, true) 时，对应的输出值应该是 false，
 * 当输入为 (true, true) 时，对应的输出值应该是 true。
 */
public void testX1andX2() {
    boolean x1 = true;
    boolean x2 = true;

    Circuit outputCircuit = Circuit.and(x1, x2);
    assertTrue(outputCircuit.output());

    x1 = false;

```

```

    x2 = true;
    outputCircuit = Circuit.and(x1, x2);
    assertFalse(outputCircuit.output());
}

/**
 * 创建一个Circuit对象实例来计算表达式 (x1 and x2) or x3 时的输出值。
 * 当输入值为 (false, true, false) 的时候, 输出值是否为 false,
 * 而输入值为 (false, false, true) 的时候, 输出值是否为 true。
 */
public void testX1andX2orX3() {
    boolean x1 = false;
    boolean x2 = true;
    boolean x3 = false;
    Circuit outputCircuit = Circuit.or(Circuit.and(x1,x2),x3);
    assertFalse(outputCircuit.output());

    x1 = false;
    x2 = false;
    x3 = true;
    outputCircuit = Circuit.or(Circuit.and(x1,x2),x3);
    assertTrue(outputCircuit.output());
}

/**
 * 创建一个Circuit对象实例来计算表达式[x1 or not(x1)]or x3 时的输出值。
 * 对于该表达式, 任何输入对应的输出值都应该是 true。
 */
public void testAlwaysTrue() {
    boolean x1 = true;
    Circuit outputCircuit = Circuit.or(x1,Circuit.negate(x1));
    assertTrue(outputCircuit.output());

    x1 = false;
    outputCircuit = Circuit.or(x1,Circuit.negate(x1));
    assertTrue(outputCircuit.output());
}

```

问题的根源在于对这个任务的理解有分歧, 游戏的参与者认为能够支持这种情况。但我从来没有希望他们允许一个"circuit"的值可以改变。有一些参与者觉得我希望他们提供这种功能, 下面是该处理方案的代码。

```

/**
 * 创建一个Circuit对象实例来计算表达式 (x1 and x2) 时的输出值。
 * 对于该表达式, 当输入为 (false, true) 时, 对应的输出值应该是 false,
 * 当输入值为 (true, true) 时, 对应的输出值应该是 true。
 */
public void testX1andX2() throws Exception {
    Circuit c = Circuit.construct(
        Element.createAnd(
            Element.createInput(0),

```

```

        Element.createInput(1)
    )
);

assertFalse ("false AND true is false", c.evaluate(false, true));
assertTrue ("true AND true is true", c.evaluate(true, true));
}

/**
 * 创建一个Circuit对象实例来计算表达式(x1 and x2)or x3时的输出值。
 * 当输入值为 (false, true, false) 的时候, 输出值是否为 false,
 * 而输入值为 (false, false, true) 的时候, 输出值是否为 true。
 */
public void testX1andX2orX3() throws Exception {
    Circuit c = Circuit.construct(
        Element.createOr(
            Element.createAnd(
                Element.createInput(0),
                Element.createInput(1)
            ),
            Element.createInput(2)
        )
    );

    assertFalse (
        "(false AND true) OR false is false",
        c.evaluate(false, true, false)
    );
    assertTrue (
        "(false AND false) OR true is true",
        c.evaluate(false, false, true)
    );
}

/**
 * 创建一个Circuit对象实例来计算表达式[x1 or not(x1)]or x3时的输出值。
 * 对于该表达式, 任何输入对应的输出值都应该是 true。
 */
public void testAlwaysTrue() throws Exception {
    Circuit c = Circuit.construct(
        Element.createOr(
            Element.createInput(0),
            Element.createNot(Element.createInput(0))
        )
    );

    assertTrue ("tautology is true", c.evaluate(false));
    assertTrue ("tautology is true", c.evaluate(true));
}

```

但其他的一些参与者则对该问题有着不同的理解,因此他们设计的 API 所解决的问题也就有所不同。这种情况是有可能发生的。有些人以前碰到过这种问题,知道这个问题是非常复杂的。对于其他第一次碰到这个问题的人来说,他们只考虑必须满足要求,而不越雷池半步。

这里可以给出一个简单的建议:一定要先把问题域给解释清楚,让所有参与者都很清楚这个问题,让他们来解决一个相同的问题。只是这条建议的用处也不大。经验告诉我们,不管你怎么努力去给他们解释,总会出现类似的问题。你可以将问题变小,但不可能完全避免。

考虑到上面的原因,所以我们还会在这个游戏中添加一轮。在这新的一轮中,你应该清楚地说明这个 API 还要满足哪些额外的需求。比如说,要能支持使用一个电路来执行多次运算操作。随后,留给参与者足够的时间来扩展他的方案。当然,如果在第一轮中就做对了的参与者就无须参加这一轮了。这样做的最重要原因就是想把问题的解决方案放在同一起跑线上,避免没有过此类经验的参与者有所吃亏。

17.2.3 遗漏实现的问题

解决布尔电路问题有一个新颖且有趣的方案,就是利用解析技术,参见 `day1/parsingsolution` 中的代码。

```
/**
 * 用法如下:
 * 第一个方法 parse, 在被调用时, 必须输入一个有效的表达式。
 * 如果返回值为 0, 则表示可以将一个数组作为参数来调用 evaluate 方法进行计算。
 * 这个 evaluate 方法可以调用任意次数来调用不同参数代入表达式后的输出结果。
 * 而 parse 方法则用来改变当前所用的表达式。
 */
public class Circuit {

    /** 解析一个表达式
     * @param 一个字符串形式的表达式
     * 有效的输入规则如下:
     * 可以用 x 加自然数来表达变量, 如 x1,
     * 另外, AND、NOT、OR 以及括号也可以用在表达式中。
     * 如 x1 AND x2 就是一个合法的表达式。
     * @return 如果返回值为 0, 则证明表达式合法, 且解析成功, 其他返回值表示不成功。
     */
    public int parse(String expression) {
        return 0;
    }

    /** 根据输入来进行计算
     * @param 输入参数是一个 boolean 的数组
     * 该数组的大小必须与表达式中的变量 N 一致。
     * 如果数组大小比表达式中的变量数 N 大, 则只有前面 N 个数据会用到, 其他的则忽略。
     * 如果数组大小比表达式中的变量数 N 小, 则会抛出一个 IllegalArgumentException 异常。
     * 如果调用该方法时, 还没有正确的表达式, 则会抛出一个 IllegalStateException 异常。
     */
}
```



```

    */
    public boolean evaluate(boolean [] x) {
        return true;
    }
}

```

这个解决方案则是使用了公式，用一个字符串来写一个公式，比如 `x1 AND NOT(x1)`，然后解析这个字符串来得到电路的表示，这样只要输入值不同，就可以重复地进行了。

这是一个非常漂亮而且灵活的解决方案，而且提供了很好的文档对功能进行说明，其他的解决方案相比之下就差了一些。但它也有一个问题——它没有提供一个实现。不必为此大吃一惊，因为想在一个小时内就写完一个表达式的解析器还是一件很有挑战性的任务。即使通过 Google 查找到一个现有的实现，参考一下，然后想在一个小时内就改头换面也仍然是件很困难的事。所以这个方案没有进入下一轮。

一定要向参赛者解释清楚，API 设计研讨会最关注的问题是要如何设计 API 才能便于在未来加以改进，还有功能向后兼容性问题。还得强调“API 包含用户依赖的任何东西”，也就是说也包含具体的实现代码！API 设计研讨会的主旨是要提供一个功能性的 API，它的几个版本要进行相互比较。所以这个 API 的具体实现必须是正确的，相比之下，Javadoc 就显得用处不太大了。

17.2.4 返回结果可能不正确的问题

在比赛的第一天结束以后，我最后要检查的一件事就是具体的功能实现是否正确，还有用户是否可能误用这个 API。比如说 `day1/stackbasedsolution` 中的代码就有这些问题。

```

/**
 * 创建一个 Circuit 对象实例来计算表达式 (x1 and x2) 时的输出值。
 * 对于该表达式，当输入为 (false, true) 时，对应的输出值应该是 false，
 * 当输入为 (true, true) 时，对应的输出值应该是 true。
 */
public void testX1andX2() {
    Stack<Character> s = new Stack<Character> ();
    s.addAll(Arrays.asList('1', '1'));
    assertEquals("'1' for '11' input.", '1',
        CircuitFactory.getBasicCircuit(Operation.AND).evaluate(s));
    s.addAll(Arrays.asList('1', '0'));
    assertEquals("'0' for '10' input.", '0',
        CircuitFactory.getBasicCircuit(Operation.AND).evaluate(s));
}

/**
 * 创建一个 Circuit 对象实例来计算表达式 (x1 and x2) or x3 时的输出值。
 * 当输入值为 (false, true, false) 的时候，输出值是否为 false，
 * 而输入值为 (false, false, true) 时，输出值是否为 true。
 */
public void testX1andX2orX3() {

```

```

Stack<Character> s = new Stack<Character> ();
s.addAll(Arrays.asList('0', '1', '0'));
assertEquals("'0' for '010' input.", '0',
    CircuitFactory.join(CircuitFactory.getTrivialCircuit(),
        CircuitFactory.getBasicCircuit(Operation.OR),
        Operation.AND).evaluate(s)
);
s.addAll(Arrays.asList('0', '0', '1'));
assertEquals("'1' for '001' input.", '1',
    CircuitFactory.join(CircuitFactory.getTrivialCircuit(),
        CircuitFactory.getBasicCircuit(Operation.OR),
        Operation.AND).evaluate(s)
);
}
/**
 * 创建一个Circuit对象实例来计算表达式[x1 or not(x1)]or x3时的输出值。
 * 对于该表达式，任何输入对应的输出值都应该是true。
 */
public void testAlwaysTrue() {
    Circuit alwaysTrue = CircuitFactory.join(
        CircuitFactory.getTrivialCircuit(),
        CircuitFactory.getBasicCircuit(Operation.NEG),
        Operation.OR
    );
    Stack<Character> s = new Stack<Character> ();
    s.addAll(Arrays.asList('0', '0'));
    assertEquals("'1' for '00'", '1', alwaysTrue.evaluate(s));
    s.addAll(Arrays.asList('1', '1'));
    assertEquals("'1' for '11'", '1', alwaysTrue.evaluate(s));
}

```

这个测试代码在执行的时候，对于每一个作为参数变量，每次用到的时候，都会加以计算。这样，像 $x1 \text{ OR } \text{NOT}(x1)$ 的求值就是一个重复变量了，因为这个表达式决定它的运算结果永远都是 true。但对于这个表达式，有一些 API 有时却可能得到 false，因为这个 API 的输入并不是一致的。

重复一次，只有这一个解决方案能完成指定的所有任务，进入了 API 的设计研讨会的下一轮。但这个方案与其他方案有所不同，是因为对于整个方案来说，下一轮的起点要重新调整，要对任何人都是一样的，所以在游戏中添加半轮，让参与者调整自己的 API，以便让所有参与者重新站到同一个新的起跑线上，公平地进入下一轮游戏。

17.2.5 第一天的解决方案

在这个 API 的设计研讨会的第一天，参与者设计出的布尔电路 API 可以分成几大类。有一类是最常见的，也是一个非常经典的防御型编程方式——只公布想要公布的内容。尽管有少许的不同，但 `day1/alwayscreatenewcircuit`、`day1/inputandoperation`、`day1/pinbasedsolution` 和 `day1/element-basedsolution` 都可以归为这一类，而 `day1/stackbasedsolution` 也有部分这方面的倾向。这几个设计都没有对外暴露构造函数，而是使用了工厂方法来创建对象。而他们在设计中提供的 `Circuit`

类是一个 `final` 类，不可被继承。他们都使用了 `package` 级别的私有方法来作内部沟通。换句话说，这些设计都是那种“少即是多”设计方式的典范。下面的代码来自于 `day1/elementbasedsolution`，我们还没有讨论过该设计。

```
public final class Circuit {
    private Circuit() {
    }

    public static Element and(final Element e1, final Element e2) {
        return new Element() {
            public boolean result() {
                return e1.result() && e2.result();
            }
        };
    }

    public static Element or(final Element e1, final Element e2) {
        return new Element() {
            public boolean result() {
                return e1.result() || e2.result();
            }
        };
    }

    public static Element not(final Element e1) {
        return new Element() {
            public boolean result() {
                return !e1.result();
            }
        };
    }

    public static Variable var() {
        return new Variable();
    }

    public static abstract class Element {
        private Element() {
        }

        public abstract boolean result();
    }

    public static final class Variable extends Element {
        private boolean value;

        public void assignValue(boolean b) {
            value = b;
        }
    }
}
```



```

        public boolean result() {
            return value;
        }
    }
}

```

day1/parsingsolution 的设计算是一个例外，因为它没有提供具体的实现，这主要是因为第一天中，只有一个小时的时间，很难写出一个表达式的解析器实现。下面的小错误在于 Circuit 这个类不是一个 final 类，也就是说在 API 的设计研讨会的第二轮中，它可能会引发问题。

但更要注意的设计其实是 day1/subclassingsolution，下面是该设计的代码。

```

/** 一个抽象类，避免从零开始设计 Circuit 类。
 *
 */
public abstract class Circuit extends Object {

    /** 随便写点什么吧 */
    public static Circuit AND = new Circuit() {

        @Override
        public boolean evaluate(boolean[] in) {
            if ( in.length != 2) {
                throw new IllegalArgumentException(
                    "Should have two parameters"
                );
            }
            return in[0] && in[1];
        }
    };

    public static Circuit OR = new Circuit() {

        @Override
        public boolean evaluate(boolean[] in) {
            if ( in.length != 2) {
                throw new IllegalArgumentException(
                    "Should have two parameters"
                );
            }
            return in[0] || in[1];
        }
    };

    public static Circuit NOT = new Circuit() {

        @Override
        public boolean evaluate(boolean[] in) {

```



```

        if ( in.length != 1) {
            throw new IllegalArgumentException(
                "Should have one parameter"
            );
        }
        return !in[0];
    }
};

/** 随便实现这个办法吧，如果有必要，也可以直接抛出 IllegalArgumentException 异常。
 */
public abstract boolean evaluate(boolean... in);
}

```

这个解决方案另辟蹊径，它没有像其他解决方案一样，从数据结构的层面来解决问题。而是从元数据结构的层面上来解决问题，所谓的元数据就是数据的模型。其他的解决方案都是在 Circuit 的对象实例中持有一个变量来保存真实的数据，而这个解决方案中的对象实例则是用来描述元素的类型。事实上，这里只有 AND、OR 和 NOT 这 3 种类型。这里有一个小的技巧，就是这个方案使用继承，但不允许使用组合。可以这样说，这个解决方案中使用的元数据的设计方式可谓是人意料。不过情理之中的却是，这个方案的设计者就是 NetBeans 中负责元模型框架（MOF，UML，MDR）方面的专家。

17.3 第二天

在 API 的生命周期中，总会有些需求改变或者有新的需求被提出。这个 API 的设计研讨会中的布尔电路也不例外。只不过它是将需求变化频度推向了一个极端。所以说这个游戏第二天的主要目标是收集需求方面的大量变化。在处理这些新需求的时候，需要以一种兼容的方式来对 API 加以调整，如果一个程序使用了第一天所设计的 API，那么这个程序必须能在第二天调整后的 API 上正常运行。为了模仿这种情况，所有的参与者都会收到一个新的测试类 RealTest.java，它会添加一些新的任务。参与者必须修改他们的 API，保证这些 API 在完成新功能的同时，还能保证旧功能的兼容。

```

/** 这个文件里描述了 API 设计竞赛第二天的需求。
 * 其实也很简单，就是原来的 Circuit 只支持 Boolean 量，现在要支持从 0 到 1 的浮点数。
 * <p>
 * 这个需求表明，原来使用 boolean 量作为输入和输出的地方，都可以使用介于 0 到 1 之间的浮点数来代替。
 * 但考虑到向下兼容性，原来对 boolean 量的操作仍然有效。
 * 事实上，true 可以用 1 来表示，而 false 则用 0 来表示。
 * <p>
 * 为了满足新的需求，计算规则调整如下：
 * <ul>
 * <li>非运算 neg(x)=1-x，这样，对于原先的表达式 neg(false)=neg(0)=1-0=1=true 是正确的。
 * <li>与运算 and(x, y)=x*y，这样，对于原先的表达式仍然是正确的。

```

```

* and(true,true)=1*1=true and also
* and(false,true)=0*1=0=false
* 仍然是正确的。
* </i>或运算  $or(x, y) = 1 - (1 - x) * (1 - y)$ , 这样, 对于原先的表达式
*  $or(false, false) = 1 - (1 - 0) * (1 - 0) = 1 - 1 = 0 = false$ 
*  $or(true, false) = 1 - (1 - 1) * (1 - 0) = 1 - 0 * 1 = 1 = true$ 
* 仍然是正确的。
* </ul>
* <p>
* 但支持浮点数的Circuit, 就比只支持布尔量的Circuit复杂多了。
* 对于使用这个API的用户来说, 会有更多的需求, 比如说定义自己的运算规则。
* 这个功能作为练习体现在下面的测试中, 并要求参与者实现该功能。
*/
public class RealTest extends TestCase {
    static {
        // 随意放置自己的代码
    }

    public RealTest(String testName) {
        super(testName);
    }

    /** 创建一个Circuit对象实例来计算表达式(x1 and x2)or not(x1)时的输出值。
     *
     * 当输入值 x1=true, x2=false, 则返回结果应该为 false。
     *
     * 当输入值 x1=false, x2=true, 则返回结果应该为 true。
     *
     * 当输入值 x1=0.0, x2=1.0, 则返回结果应该为 1.0。
     *
     * 当输入值 x1=0.5, x2=0.5, 则返回结果应该为 0.625。
     *
     * 当输入值 x1=0.0, x2=2.0, 则应该抛出一个异常。
     */
    public void testX1andX2orNotX1() {
        fail("testX1andX2orNotX1");
    }

    /** 请确保不能通过*的方式将两个不同的值运算得到一个新值。
     * 如创建一个Circuit对象实例来计算表达式(x1 and x1)时的输出值。
     * 不能通过(x1*x1)来进行计算, 而是应该抛出一个异常。
     * 比如说当输入值为 0.3 和 0.5 的时候, 就应该抛出一个异常表示, 当前用法不对。
     */
    public void testImproperUseOfTheCircuit() {
        fail("testImproperUseOfTheCircuit");
    }

    /** 想自定义一个运算规则, 称为“gte”, 它有两个输入和一个输出。

```

```

* 当  $x1 \geq x2$  的时候, 则输出值为 1, 反之为 0。
*
* 创建一个 Circuit, 对应的表达式为  $[x1 \text{ and not}(x1)] \text{ get } x1$ 。
*
* 当输入参数为 0.5 的时候, 输出结果应该为 0。
*
* 当输入参数为 1 的时候, 输出结果应该为 0。
*
* 当输入参数为 0 的时候, 输出结果应该为 1。
*/
public void testGreaterThanEqualElement() {
    fail("testGreaterThanEqualElement");
}
}

```

但考虑到第一天, 参与者给出的各种方案差别都非常大, 所以参与者首先需要调整他们的 API 设计, 使大家的设计尽量相似, 以便处于同一起跑线上。在某些情况下, 这个 Circuit 的对象实例不可以多次重复使用, 这没有达到创建一个 Circuit 对象实例的目的。所以在第二天中, 这些设计者就需要调整这个方案以解决该问题。另一个问题是有些 API 允许 Circuit 被不当地使用, 如恒为 true 的表达式得出 false, 这个问题也应该被修正, 而且还要考虑到向后兼容问题, 确实很有挑战性。

在理想的情况中, 为了公平起见, 应该让所有的参与者都站在同一起跑线上, 也就是说这两个任务其实应该分开做, 这样所有的参与者对游戏第二天中要做的事情才能真正地一无所知。但考虑到 API 设计竞赛只给设计和编码留出了两天的时间, 所以只能同时做这么多的内容。有一些参与者如果既要修正自己的问题, 又要实现第二天的功能, 实在是比较困难。何况第二天要做的新功能又很复杂, 难以实现。它主要涉及将布尔电路转换成“概率”电路, 原来的 Circuit 不是 0 (表示 false) 就是 1 (表示 true), 现在则是一个数字, 一个在 0.0 和 1.0 之间波动的值。

此前使用 boolean 变量作为输入或输出的地方, 现在就需要使用一个双精度数 x , 注意这个数应该大于等于 0 而小于等于 1。为了支持向后兼容, 仍然要支持原来的各种布尔操作运算。因此, 还将 false 当作 0, 而将 true 看作 1。

那么基本的元素就必须加以调整, 以支持在下列方式中使用双精度数。

- **Negation**: $\text{neg}(x) = 1 - x$ 。这是一个正确的扩展运算操作, 因为 $\text{neg}(\text{false}) = \text{neg}(0) = 1 - 0 = 1 = \text{true}$ 。
- **and**: $\text{and}(x, y) = x * y$ 。和上面一样, 这个运算操作也是正确的, 因为 $\text{and}(\text{true}, \text{true}) = 1 * 1 = \text{true}$, 同样 $\text{and}(\text{false}, \text{true}) = 0 * 1 = 0 = \text{false}$ 。
- **or**: $\text{or}(x, y) = 1 - (1 - x) * (1 - y)$ 。这个运算操作也是正确的, 因为 $\text{or}(\text{false}, \text{false}) = 1 - (1 - 0) * (1 - 0) = 1 - 1 = 0 = \text{false}$ 。再看一个例子, $\text{or}(\text{true}, \text{false}) = 1 - (1 - 1) * (1 - 0) = 1 - 0 * 1 = 1 = \text{true}$, 仍然是正确的。

为了防止某些开发人员在设计中进行欺骗, 也就是防止他完全重新设计一个新的 API, 与第一天设计的版本没有任何关系, 因此要求这个 API 必须能够创建一个电路, 先后接受双精度

数和布尔值的输入。而且不论如何创建这个对象，都必须保证创建出了与第一天的那个对象相同的对象，用第一天的 API 也能同样求值。这样就可避免参与者专为第二天从头重新设计新的 API。

还有一点就是，最好把一个任务留到 API 设计竞赛的第三天或者第四天去做。考虑到时间比较紧张，我们也把它放在第二天了。基于双精度数的 Circuit，相比之下，要比基于布尔量的 Circuit，具有更加丰富的语言，所以要求允许 API 的用户来写自己的“基本”类型。任务要求不仅仅能够组合元素了，而是要允许其他开发人员编写自己的电路元素插件。各项任务如下。

(1) 首先，创建一个 Circuit 对象实例，它可以用来计算 $(x1 \text{ and } x2) \text{ or } \text{not}(x1)$ 这个表达式，然后系统会通过一个变量来控制这个 Circuit 对象实例。

- a. 在 $x1=\text{true}$, $x2=\text{false}$ 的情况下，结果应该为 false 。
- b. 在 $x1=\text{false}$, $x2=\text{true}$ 的情况下，结果应该为 true 。
- c. 在 $x1=0.0$, $x2=1.0$ 的情况下，结果应该为 1.0 。
- d. 在 $x1=0.5$, $x2=0.5$ 的情况下，结果应该为 0.625 。
- e. 在 $x1=0.0$, $x2=2.0$ 的情况下，就会因为 $x2>1$ 而抛出一个异常。

(2) 要确保一个变量不可能接受两个不同的参数值。构造一个 Circuit 对象实例用来计算 $x1 \text{ and } x1$ 。要保证对于任何非 $x1*x1$ 的运算，都应该抛出一个异常。比如说，如果给电路两个不同值 0.3 和 0.5 ，那么就得抛出一个异常，表示 Circuit 的用法是不正确的。

(3) 再提供一个名为 `gte` 的操作，它有两个输入参数和一个返回值。如果输入参数 $x1 \geq x2$ ，那么返回值就会是 1 ，否则返回值就是 0 。现在有一个表达式 $(x1 \text{ and } \text{not}(x1)) \text{ gte } x1$ ，它应该满足下述要求。

- a. 在 $x1$ 值为 0.5 的时候，返回结果应该为 0 。
- b. 对于与上相同的 Circuit，在 $x1$ 值为 1 的时候，返回结果应该为 0 。
- c. 对于与上相同的 Circuit，在 $x1$ 值为 0 的时候，返回结果应该为 1 。

现在读者可以自己试着实现一下，能不能满足这些需求！下载 `RealTest.java` 文件，然后放到游戏第一天所建的项目中，然后重新写这个 API，看是否可以满足上面的需求。如果想做就得抓紧，因为下面的章节就会告诉大家解决该问题的最佳方案了！

有两种方式来解决这个问题。第一种方式是强化第一天写好的接口，对其进行调整，以适应“概率”电路上新的操作。第二种方式，则是重新写一个 API，提供个桥梁，允许布尔电路与“概率”电路相互转换。对于第二种方式，可以只构造一个 Circuit，然后再用布尔量和双精度数展开计算。因为第一天给出的所有方案都需要进行一些改进，于是所有的参与者都选择了第一种方式，就是强化原有的接口。没有人选择第二种桥接的方式。这个选择也许不错，因为第二种桥接方式需要的工作量肯定要大很多。

17.3.1 我想修正犯下的错误

比起第一天，第二天基本上没有人对要完成的任务有所误解。但出现这么一种认识：“我所做的方案不够漂亮，应该进行修正。”有一些参与者认为他们昨天提供的方案不够优雅，所以想

修正这些问题。但如果想对 API 中的一部分内容加以修正，那么也就是要承担可能引发兼容性问题风险。所以有必要向这些参与者解释，这个 API 设计竞赛其实就是在比试兼容性，至于漂亮与否完全可以无视。仅仅因为设计不够漂亮，就去修正它，只是在浪费时间。比赛的目标只是让你在保持向后兼容的同时，对 API 加以改进。

17.3.2 第二天的解决方案

那些原来不允许用一个 Circuit 对象执行多次操作的参与者，在第二天都遇到了些小问题。类似地，那些电路求值不一致的参与者碰到的问题更严重。他们有大量的问题要修复，才能赶上进度。像 `alwayscreatenewcircuit`, `inputandoperation`, `stackbasedsolution` 这几个方案都需要先修正存在的问题，才能进行下面的工作。

不管如何，参与者最终都要开始处理今天的具体任务：要对原先的电路功能进行扩展，使其可以处理 0.0 至 1.0 范围内的双精度数。扩展功能有两种方式。有一些参与者在游戏第一天就考虑到后面可能要提供扩展功能，所以在第一天的设计中，他们将支持运算操作的类设计成不可继承，这样他们可以向该类中添加新的方法以支持对双精度数的运算操作。`alwayscreatenewcircuit`, `elementbasedsolution`, `inputandoperation`, `pinbasedsolution`, `welltestedsolution` 这些方案就是这么做的。所有这些 API 都会充分利用一个不可继承的类的优势，向这些类中添加新的方法时，由于相应类不可继承，所以对 API 的改进不会破坏现有用户代码的兼容性。

还有一些方案，如 `subclassingsolution` 和 `stackbasedsolution`，它们把电路元素做成了可以继承的类，于是就不能保证兼容地去对类进行增强了。所以要提供新的类以支持新的功能。所以，`subclassingsolution` 方案引入了 `FuzzyCircuit`，而 `stackbasedsolution` 引入了 `Circuit2`。这两种方案的思路是类似的，都是给 API 的用户提供了一个新的方法，用来支持新功能，但同时也可以不破坏用户原来继承 `Circuit` 的代码。

```
public interface Circuit2 extends Circuit {
    public double evaluate (double ... input)
        throws IllegalArgumentException;
}
```

这两种方案都是可行的，但都有一定的危险。添加新的接口的危险之处在于，用户可能会觉得这个接口很难用，特别是他们搞不清楚什么时候使用这个简单的接口，什么时候使用这个增强类型的接口。用户会混淆方法的返回类型以及字段的类型。至于在一个不可继承的类中添加一个新的方法，对于用户来说比较好，但也同样存在相应的危险，因为新方法和老方法之间的协作存在一定的不确定性。参与者需要很小心地来保证其正确性。

API 设计竞赛第二天的所有参与者都完成了给定的任务，而且满足了新提出的需求。所有的解决方案都可以进入最后一轮了。如果读者有兴趣的话，可以看一下 `day2/alwayscreatenewcircuit`, `day2/inputandoperation`, `day2/elementbasedsolution`, `day2/subclassingsolution`, `day2/pinbasedsolution`, `day2/stackbasedsolution` 和 `day2/welltestedsolution` 这几个项目中的 `RealTest.java` 文件。下面只选取了 `day2/welltestedsolution` 这个项目下面的代码作为示例。

```

public class RealTest extends TestCase {
    static {
        // 随便放自己的代码
    }

    public RealTest(String testName) {
        super(testName);
    }

    protected void setUp() throws Exception {
    }

    protected void tearDown() throws Exception {
    }

    /** 创建一个Circuit对象实例来计算表达式(x1 and x2)or not(x1)时的输出值。
     *
     * 当输入值 x1=true, x2=false, 则返回结果应该为 false。
     *
     * 当输入值 x1=false, x2=true, 则返回结果应该为 true。
     *
     * 当输入值 x1=0.0, x2=1.0, 则返回结果应该为 1.0。
     *
     * 当输入值 x1=0.5, x2=0.5, 则返回结果应该为 0.625。
     *
     * 当输入值 x1=0.0, x2=2.0, 则应该抛出一个异常。
     */
    public void testX1andX2orNotX1() {
        Circuit c = Circuit.createOrCircuit(
            Circuit.createAndCircuit(Circuit.input(0),
                Circuit.input(1)),
            Circuit.createNotCircuit(Circuit.input(0))
        );
        assertFalse("true, false", c.evaluate(true, false));
        assertTrue("false, true", c.evaluate(false, true));
        assertEquals("0.0, 1.0", 1.0, c.evaluateFuzzy(0.0, 1.0), 0.0);
    }

    /** 请确保不能通过*的方式将两个不同的值运算得到一个新值。
     * 如创建一个Circuit实例来计算表达式(x1 and x1)时的输出值,
     * 不能通过(x1*x1)来进行计算, 而是应该抛出一个异常
     * 比如说当输入值为 0.3 和 0.5 的时候, 就应该抛出一个异常表示, 当前用法不对。
     */
    public void testImproperUseOfTheCircuit() {
        // 没有什么用处

        Circuit x1 = Circuit.input(0);
        Circuit c = Circuit.createOrCircuit(x1, x1);
    }
}

```

```

    assertTrue("x1 or x1", c.evaluate(true));
    assertFalse("x1 or x1", c.evaluate(false));
    try {
        c.evaluate();
        fail("x1 or x1 with wrong params");
    } catch (IllegalArgumentException iea) {
        // 应该对异常加以处理
    }
    // 两个相同的实例
    c = Circuit.createOrCircuit(Circuit.input(0), Circuit.input(0));
    assertTrue("x1 or x1", c.evaluate(true));
    assertTrue("x1 or x1", c.evaluate(true, false));
    assertTrue("x1 or x1", c.evaluate(true, true));
    assertFalse("x1 or x1", c.evaluate(false));
    try {
        c.evaluate();
        fail("x1 or x1 with wrong params");
    } catch (IllegalArgumentException iea) {
        // 应该对异常加以处理
    }
}

/** 想自定义一个运算规则，称为“gte”，它有两个输入和一个输出。
 * 当  $x_1 \geq x_2$  的时候，则输出值为 1，反之为 0。
 *
 * 创建一个 Circuit，对应的表达式为  $[x_1 \text{ and not}(x_1)] \text{gte } x_1$ 。
 *
 * 当输入参数为 0.5 的时候，输出结果应该为 0。
 *
 * 当输入参数为 1 的时候，输出结果应该为 0。
 *
 * 当输入参数为 0 的时候，输出结果应该为 1。
 */
public void testGreaterThanEqualElement() {
    Circuit gte = new Gte(Circuit.createAndCircuit(
        Circuit.input(0),
        Circuit.createNotCircuit(Circuit.input(0))),
        Circuit.input(0)
    );
    assertEquals("0.5", 0.0, gte.evaluateFuzzy(0.5), 0.0);
    assertEquals("1.0", 0.0, gte.evaluateFuzzy(1.0), 0.0);
    assertEquals("0.0", 1.0, gte.evaluateFuzzy(0.0), 0.0);
}

public void testSilly() {
    // (x1 and not x2) or x3
    Circuit c = Circuit.createOrCircuit(
        Circuit.createAndCircuit(
            null,

```

```

        Circuit.createNotCircuit(null)),
        null
    );
    assertEquals("1 1 1", 1.0, c.evaluateFuzzy(1.0, 1.0, 1.0), 0.0);
    assertEquals("1 1 0", 0.0, c.evaluateFuzzy(1.0, 1.0, 0.0), 0.0);
    assertEquals("1 0 1", 1.0, c.evaluateFuzzy(1.0, 0.0, 1.0), 0.0);
    assertEquals("1 0 0", 1.0, c.evaluateFuzzy(1.0, 0.0, 0.0), 0.0);
    assertEquals("0 1 1", 1.0, c.evaluateFuzzy(0.0, 1.0, 1.0), 0.0);
    assertEquals("0 1 0", 0.0, c.evaluateFuzzy(0.0, 1.0, 0.0), 0.0);
    assertEquals("0 0 1", 1.0, c.evaluateFuzzy(0.0, 0.0, 1.0), 0.0);
    assertEquals("0 0 0", 0.0, c.evaluateFuzzy(0.0, 0.0, 0.0), 0.0);
    }
}

```

17.4 第三天：评判日

到目前为止，所有的参与人员都还只是闷着头在自己的一亩三分地里玩。说实话，他们做得非常好！只不过现在需要对他人的工作加以评判了。

这个 API 的设计研讨会毕竟还是一个竞争性的游戏，只不过这个比赛非常精彩。所以“喜欢”、“爱”和“讨厌”这些词都不适用于评判这个游戏。不需要某个裁判来判断哪个 API 是最优秀的 API，而是每个参与者在一系列的改进中，自己来检查他人设计的 API，找出其中的问题。每找到一处问题，他们就会得到 1 分，同时他们自己的 API 如果不存在问题，那么就可以得到 5 分。每一个 API 的分数最终将由所有参与者给出。

如何编写一个测试来证明存在不兼容问题呢？读者可以把 `apifest1/day3-intermezzo/jtulach/against-pinbasedsolution/` 这个样例项目复制过来，它只是一个空架子，读者可以编写代码查找一下不兼容性问题，不管这个不兼容问题是源代码级别还是二进制级别。我给参与者的建议就是把源代码放置到一个“参赛者名字/against-API 名字”的目录下面。每一个目录下面都必须有一个用来支持 ANT 的 `build.xml` 文件，来验证是否存在改进方面的问题。发现的问题越多，得到的分数就越高。

预先准备好的一个 `project.properties` 文件要进行修改。先将 API 的名字改成你想要的名字，然后将文件移到合适的相对位置（如果没有问题，就不用这一步了）。接下来，你可以用 NetBeans IDE 来编译代码、运行程序、进行测试。和 NetBeans IDE 中的普通项目一样，示例项目也是完全基于 ANT 的，所以你也可以通过命令行直接使用脚本来做这一切。最后执行相应的 `build.xml` 文件。

ANT 在执行 `build.xml` 文件时，会基于第一天所设计的 API 来编译测试代码，并确认所写的测试代码能成功验证第一天的实现是否正确。接下来使用这个测试代码来验证第二天的实现是否正确。如果测试失败了，那么表示你发现了一个不兼容问题。这样做就能发现二进制和功能方面的不兼容性问题。除此以外，API 还要基于第二天所设计的 API 进行编译，如果编译失败，那么测试也是成功的，因为你发现了一个源代码级别的不兼容问题。

现在我会花点时间来讨论一下查找不兼容性问题的一些规则。首先要知道，定义这样的规

则是非常困难的，因为很多开发人员都是优秀的黑客，他们会在这些规则以外想出各种变通的方案。一般的原则就是：发现一个不兼容性问题时，使用的技巧越少越好！如果有两种方式都能证明同一个 API 中存在不兼容性问题，那么技巧性少的方式获胜。现在来看一下这里所说的“技巧”到底是什么东西。在代码运行的时候，所谓的“技巧”就是 Java 的访问许可，即 `java.security.Permission`。在运行测试代码时，用到的功能许可越少越好。比如说，不准用 `ReflectionPermission`，否则每一个方案都会出现一个不兼容问题。也不准使用 `Class.forName(...)`，否则就太简单了，因为几乎每一个 API 都在游戏第二天里添加了新的 `Class`。而 `Class.getName()` 也不太好，不过在某些情况下，还是可以考虑的。还要注意，你可以使用线程，但创建一个新的线程是需要许可的。如果有两个测试方案都发现了一个 API 中的不兼容性问题，但其中一个用了较多线程，那么哪一个会赢呢？肯定是用得少的那个了。而且在代码引入类的时候，不准使用 `*`，因为可能会因为这样而引起所有添加新类的 API 编译失败，所以一定要用类的全名。可见，究竟什么是技巧，是很宽泛的，规则只有一条：用的技巧越少越好。

想不想自己试试

想不想自己试一下？没有什么比这更简单的了，把解决方案和测试样例项目模板下载下来，然后试着找出 API 的两个版本间的不兼容问题！

结论

第三天的游戏是从周五下午开始的，到周日结束。也许就是因为这个原因^①，参与这一轮的人较少。也许还有一个原因，就是对测试架构的用法理解上出了一些问题，以及总看别人写的代码有些乏味。

还好我们有 Petr Nejedlý！他太强了，不但没有人能从他设计的 API 中找出问题，而且他还从所有其他人所写的 API 中都找出了问题。有时候，他也使用了一些不太好的方法，如 `getClass().getSuperclass()` 和 `getClass().getName()` 之类的。不过能从所有其他人的方案中都找出问题，仍然值得我们大家为他鼓掌。毫无意外，Petr 赢得了 API 设计竞赛的第一名。

1. subclassingsolution 和 stackbasedsolution^② 分别有什么问题

有一个参与者，他在设计 API 的时候，是采用“subclassingsolution”，他认为这种方案最大的问题在于，AND、OR 还有 NOT 这三项内容竟然不是 `final` 的。其实他错了，这个 API 最大的问题在于 `Circuit` 这个类，这个类具有两类特性。一是用户可以调用这个 API 来完成某些功能；二是用户可以继承和实现这个类以扩展功能。这样的设计其实是把自己放到了一个进退两难的困境。使用 `FuzzyCircuit` 的方式来设计 API 看来是比较正确的。只不过一定要记住，在 Java 中，只

^① 欧美的工作时间往往比较严格，周日一般不会有工作，所以作者说参与的人不多。——译者注

^② 这里使用了原文中的名称，因为这些名称都是当时研讨会的一个临时用语，用来表示某种具体方案，因此就不进行翻译了。而且象 `subclassingsolution` 这些名称都与具体的方案有关，很难有一个精确的中文与之对应。——译者注

要将一个字段、参数或者返回值换成另外一种类型，都会破坏二进制兼容性。Class 文件的格式中不仅仅包含了字段的名称，还有字段的类型。如果类型有所改变，那么分别在不同环境下编译的 Class 文件可能就无法正常连接了。所以下面这个测试代码对两个版本都进行编译。但如果针对第一天的 API 编译完成以后，再编译第二天的 API，就可能出现一个 `NoFieldFoundError` 的错误，导致运行失败。

```
public class CircuitTest extends TestCase {
    public CircuitTest(String n) {
        super(n);
    }

    public void testClass() throws Exception {
        Circuit c = Circuit.AND;
    }
}
```

还有一个参与者使用了“stackbasedsolution”这种方案，也存在类似的问题。这个方案会改变一个方法的返回类型。而这种改变也同样是二进制不兼容的，所以也会出现一个 `NoMethodFoundError` 的错误，程序仍然无法正常运行。

```
public class CircuitTest extends TestCase {
    public CircuitTest(String n) {
        super(n);
    }

    public void testClass() throws Exception {
        Circuit c = CircuitFactory.getBasicCircuit(null);
    }
}
```

2. inputandoperation 这种方案有什么问题

day1/inputandoperation 这是一个设计非常不错的方案，它快算得上无懈可击了。但这个方案仍然有一个问题：它的 `Circuit` 类不是 `final`。同样，可以在第二天里为它添加了一个新方法，代码如下。

```
public static double evaluateRealOperation(Operation op) {
    return op.performRealOperation();
}
```

尽管这个方法是 `static` 的，但仍然有问题。对于第一天的代码中，可以写一个类来继承 `Circuit`，然后添加一个同名的 `evaluateRealOperation` 的方法，此时编译是肯定没有问题的。但到了第二天，使用新的类库来编译，就无法通过编译了。

```
public class CircuitTest extends TestCase {
    public CircuitTest(String n) {
        super(n);
    }
}
```

```

    public void testSourceCompatibility() throws Exception {
    }

    // 有人忘记加一个 final 声明了，从而引发一场血案。
    class MyCircuit extends Circuit {
        public double evaluateRealOperation(Operation op) {
            return 0;
        }
    }
}

```

从上面问题中学到的一点就是，千万不要忘记向 API 公开的类中添加一个私有的构造函数或者一个 final 关键词，或者干脆两个都加上。否则 API 的用户的使用方法会出乎你的意料，所以说如果作者并不指望别人继承这个类，那么就不要允许他人继承。

3. `alwayscreatenewcircuit` 和 `welltestedsolution` 这两个方案又有什么问题

简单来说，这两个方案其实不存在什么问题。Petr 是想了不太正当的方法来破坏了这两个 API！其中一个方案，Petr 是发现了具体的实现类，代码如下。

```

public class CircuitTest extends TestCase {
    public CircuitTest(String n) {
        super(n);
    }

    public void testClass() throws Exception {
        // Okay, this is not fair as well.
        assertEquals("Created AND circuit", "AndCircuit",
            getName(Circuit.createAndCircuit(null, null))
        );
        assertEquals("Created OR circuit", "OrCircuit",
            getName(Circuit.createOrCircuit(null, null))
        );
    }

    private String getName(Object obj) {
        String base = obj.getClass().getName();
        int lastDot = base.lastIndexOf('.');
        int last = base.lastIndexOf('$');
        if (lastDot > last) last = lastDot;
        return base.substring(last+1);
    }
}

```

而对于 `alwayscreatenewcircuit` 方案，Petr 也是发现了其继承层次中的一个改变。

```

public class CircuitTest extends TestCase {
    public CircuitTest(String n) {
        super(n);
    }
}

```

```

    public void testReallyUnrealistic() throws Exception {
        // Okay, this is not fair, right?
        assertEquals(null, Circuit.or(false, false).getClass().
            getSuperclass().getSuperclass().getSuperclass());
    }
}

```

以上 Petr 所用的两个技巧其实与 API 对外的契约没有什么关系。大部分人会认为这些都只是一些实现细节。但最好还是牢记在心，像这种通过一些技巧也能从外部分析出内部细节的设计，也能越过 Java 的访问控制，而破坏整个 API。

这两个方案都是匆忙做出的，因为设计者在第二天开始的时候都要先完成一些额外的重写工作，所以他们直接让 Circuit 这个类可被继承，这样用户就可以提供自己的 Circuit 实现了。最后，搞得这个类扮演了两类角色：既是用户调用的 API 的一部分，又是用户实现的 API 的一部分。在后续 API 的改进中，这两种方案都会出现一些麻烦的事情。比如说，有个新需求想为每一个 Circuit 元素都添加一个唯一的 ID 号，这种需求引发的演进问题就和 inputandoperation 方案遇到的问题差不多。最好是添加一个额外的接口和一个工厂方法，就像方案 pinbasedsolution 添加一个 createGate 方法那样。

```

/**
 * 一个运算函数的声明
 */
public abstract class Function {
    public abstract double evaluate(double input1, double input2);
}
/**
 * 根据用户定义的运算函数来创建运算规则。
 */
public static Element createGate(
    final Element source1, final Element source2, final Function function
) {
    return new Element() {
        double evaluate(double[] inputs) {
            double x = source1.evaluate(inputs);
            double y = source2.evaluate(inputs);
            double result = function.evaluate(x, y);
            if (result < 0.0 || result > 1.0) {
                throw new InternalError("Illegal gate function");
            }
            return result;
        }

        int maxInput() {
            return Math.max(source1.maxInput(), source2.maxInput());
        }
    };
}

```

这个方法实现了同样的功能，但很清楚地将 API 的调用者和实现者的两个契约进行了分离。

4. elementbasedsolution方案有什么问题

下面是 elementbasedsolution 这个方案所作的改变，表面上看起来没有任何问题。

```
public final class Circuit {
    private Circuit() {
    }

    public static Element and(final Element e1, final Element e2) {
        return new Element() {
            public boolean result() {
                return e1.result() && e2.result();
            }
            public double doubleResult() {
                return e1.doubleResult() * e2.doubleResult();
            }
        };
    }

    public static Element or(final Element e1, final Element e2) {
        return new Element() {
            public boolean result() {
                return e1.result() || e2.result();
            }
            public double doubleResult() {
                return 1.0 -
                    (1.0 - e1.doubleResult()) * (1.0 - e2.doubleResult());
            }
        };
    }

    public static Element not(final Element e1) {
        return new Element() {
            public boolean result() {
                return !e1.result();
            }
            public double doubleResult() {
                return 1 - e1.doubleResult();
            }
        };
    }

    public static Element operation(
        final Operation op, final Element... elements
    ) {
        return new Element() {
            public boolean result() {
                return doubleResult() >= 1.0;
            }
            public double doubleResult() {
                double[] arr = new double[elements.length];
                for (int i = 0; i < arr.length; i++) {
```



```
        arr[i] = elements[i].doubleResult();
    }
    return op.computeResult(arr);
}
};

}

public static Variable var() {
    return new Variable();
}

public static abstract class Element {
    private Element() {
    }

    public abstract boolean result();
    public abstract double doubleResult();
}

public static final class Variable extends Element {
    private Boolean booleanValue;
    private Double doubleValue;

    public void assignValue(boolean b) {
        booleanValue = b;
    }
    public void assignValue(double d) {
        if (d < 0 || d > 1) {
            throw new IllegalArgumentException();
        }
        doubleValue = d;
    }

    public boolean result() {
        return booleanValue != null ?
            booleanValue : doubleValue >= 1.0;
    }

    public double doubleResult() {
        return doubleValue != null ?
            doubleValue : (booleanValue ? 1.0 : 0.0);
    }
}

public static interface Operation {
    public double computeResult(double... values);
}
}
```

但 Petr 还是想办法在测试代码中写了一个特定序列的操作,当使用第二天设计的类库来执行代码时,就会抛出一个 `NullPointerException` 的异常,但使用第一天设计的类库来执行这个测试代码时,却一切正常。

```
public class CircuitTest extends TestCase {
    public CircuitTest(String n) {
        super(n);
    }

    /**
     * ok, 这种方式不错, 简直无懈可击了, 和第一个版本保持一致, 但还有一个漏洞。
     */
    public void testEvaluateWithoutAssign() throws Exception {
        Circuit.Variable var = Circuit.var();
        Circuit.Element circuit = Circuit.not(var);

        assertTrue (circuit.result());
    }
}
```

API 改进时在源代码和二进制方面保持了兼容,但却没有在功能方面保持兼容。一个特定的方法调用序列在不同的版本上产生了不同的结果。从这里我们可以知道,对一个 API 进行任何改变都可能引发意想不到的副作用。不管你是否认真地评审这些变更,但总有些不起眼的地方会被落下,最终会在不同版本间体现出不兼容性。

17.5 也来玩下这个游戏吧

与 NetBeans 核心团队搞的这个 API 设计竞赛是一件非常有意思,也很有趣的事情。两天的游戏和一轮评估足够选出一个优胜者了。如果这个游戏中能再多几轮,每一轮能再多一点时间的话,就更好了。

这个竞赛的主要目的就是要学习如何对 API 加以改进。尽管向后兼容性不是评选优秀 API 的唯一标准,但肯定是一个首先要考虑的因素。同样,虽然在这次竞赛中还有很多内容没有涉及,比如正确地设计 API,让这个 API 易于理解,并提供清晰的文档加以说明,但是这几样都很重要。

第一届 API 的设计研讨会举办得非常成功。遗憾的是,第一期其实也是最后一期,我们也只做了这么一次,说起这事还是有点不好意思的。即使只看我们这个模仿 Boolean 的 Circuit 例子,那些能够支持改进的 API 就有多种方案,改进的方式也不尽相同。假设加个新任务,比如说支持多个输出,支持循环 Circuit,支持对未知 Circuit 的结构进行分析,大家可以继续来玩这个游戏。

对于设计 API 的人,对于学习向后兼容性的学生,还是说对这种设计难题饶有兴趣的程序员,API 的设计研讨会都是一个非常好的学习及培训方式。

通过游戏来学习 API 设计的技巧是一种不错的方式。还有一种学习方式则是完全可以只基于思考：先找到一个有意思的问题，对其进行分析，找出多种潜在的解决方案，然后选出一个比较好的，并给出相应的理由。在做这种思维锻炼的过程中，会有很多令你吃惊的收获，加深对问题的认识。

现在就来看一个有趣的例子。这个例子在我得知 JDK 6 中引入了一个新的 API 时就开始酝酿了。这个新加的 API 是把 Java 的源代码当作一个更完整、信息更加丰富的模型来使用。其实在 JDK 5 的 APT^①工具中已经有一个初始的版本，这个新 API 的功能在于：当源代码编写完成后，在后期可以用这个工具根据类，方法，字段上面的 Annotation 执行一些特定的功能，比如说生成配置文件之类^②。这个工具在运行时需要用户提供源代码模型，有点像反射的 API，只不过是针对源代码的。在 JDK 5，APT 工具类是放置在 `com.sun.mirror` 包下面的，所以不算是一个正式公开的 API。而到了 JDK 6，编译器团队已经通过了一个 JSR 标准^③，定义了正式的 API，放置在 `javax.lang.model`^④包下面。我于是着手去了解这个 API 及同类的 API，因为它们将定义今后方法体内部的模型。

这件事情引发了我的兴趣，因为我发现这个 API 在改进时，声称自己是二进制兼容，但源代码不兼容。确实，在 Java 中想保持源代码兼容是不太可能的，因为几乎 API 中的每一次修改都会使得现有代码无法正常编译，这个问题我在 4.2.1 节中已经说过了。况且，他们还想向接口中添加新的方法，这对兼容性是更大的威胁。但根据虚拟机的规范，这种行为在二进制方面却是兼容的，但从实用角度来说其实是完全不兼容了。所以在向接口中添加了方法以后，系统仍然是可以通过链接的，只不过如果有人未在实现的接口上调用了新添加的方法，那么系统就会抛出运行时错误。不用说，我对这种兼容肯定是没有兴趣的。

所以我开始思考是否能找到一个更好的方案。编译器的开发人员给了我一篇论文 *The*

① APT 就是 Annotation Processing Tool 的缩写，即处理 Annotation 的工具。——译者注

② 可以在 <http://java.sun.com/j2se/1.5.0/docs/guide/apt/GettingStarted.html> 这个网址上了解 JDK 5 更多关于 API 的信息。——译者注

③ 这个 JSR 是指 JSR269，关于 Java Annotation 处理工具的，可以通过 <http://www.jcp.org/en/jsr/detail?id=269> 这个地址了解更多这个 JSR 的信息。——译者注

④ 可以在 <http://java.sun.com/javase/6/docs/technotes/guides/apt/index.html> 这个网址上了解 JDK 6 中更多关于 API 的信息。

——译者注

Expression Problem Revisited^①，论文提出了一种方案可以在泛型的基础上做到各种类型的兼容。但他们并不打算采用这种方案，在研读过这篇论文以后，我也认同这一点。因为这个方案严重依赖于泛型技术，对于普通的开发人员来说，是难以理解的。如果想理解这些方案，那么需要在大学里花上几年的时间学习类型方面的理论知识才行。这正是一个过分正确的 API 的例子，你决不会去采纳这样的方案，因为用了该方案可能会吓走你的用户。这也就不难理解编译团队宁愿选择为接口添加方法。从实用角度考虑，这一点做起来比较容易。不过考虑到现在是做一个思考练习，所以我还是得继续看一下是不是有更好的解决方案呢。

尽管是因为 `javax.lang.model` 这个包而引发对这个问题的思考，但具体思考时却也不必将自己局限在这个问题上。事实上，想在一个 API 中使用 Visitor 模式时就会出现这种问题。对 Visitor 进行改进是一件非常困难的事情。Visitor 接口的方法只定义一次，但 Visitor 所用到的数据结构却可能会产生变化，问题是如果出现这种情况，如何来处理呢。向现有的 Visitor 方法中添加方法可以算是一种解决方案。但这种方式明显是一种笨方法，任何思路清晰的人都不会引以为荣。现在来开动脑筋看看能不能找到更漂亮、更准确、更优雅的解决方案！

Visitor 模式可谓众所皆知。它将操作（往往是基于层次数据结构之上）与数据表示本身相分离。它允许由一个团队来定义一种数据结构，然后提供一种遍历数据的方式，而其他的团队则只需要去构造数据即可。还有一些团队则负责编写处理数据的代码。这个设计模式非常简单，也很容易使用。但换个角度来看，背后隐藏了一些问题，这些问题由数据接口或遍历接口发进改变所引起。

还是先来看一个简单的例子吧，然后再慢慢地深入研究它的演进问题，找出解决问题的方案。最终，一个简单而且优雅的方案就可以解决所有的问题。但直接给出一个这样的方案显然不是我们唯一的目标，因为寻找这一方案的过程也极有意思，能展现我们环环相扣的推理。还能介绍很多有价值的思路，如果说不去苛求最完美的方案，那么其他的思路也会提供一些好方案。

思考如何来解决这个问题，第一步最好是先写个示例程序，让它出现问题，这样才好激励读者来解决这个问题。Visitor 模式可以用在很多场景下，不过最常用的还是用它来遍历异构数据层次。其中最深奥的一种就是基于语法树的表达式，如 $(1 + 2) * 3 - 8$ 。假设现在你要使用一个模型来处理这种表达式语言。

```
public abstract class Expression {
    Expression() {}
    public abstract void visit(Visitor v);
}
public final class Plus extends Expression {
    private final Expression first;
    private final Expression second;

    public Plus(Expression first, Expression second) {
        this.first = first;
        this.second = second;
    }
}
```

① Mads Torgersen, "The Expression Problem Revisited" in *ECOOP 2004 – Object Oriented Programming: 18th European Conference Oslo, Norway, June 14–18 2004, Proceedings*, ed. Martin Odersky, 123–146 (Berlin: Springer-Verlag, 2004).

```

    public Expression getFirst() { return first; }
    public Expression getSecond() { return second; }
    @Override
    public void visit(Visitor v) { v.visitPlus(this); }
}
public final class Number extends Expression {
    private final int value;
    public Number(int value) { this.value = value; }
    public int getValue() { return value; }
    @Override
    public void visit(Visitor v) { v.visitNumber(this); }
}

public interface Visitor {
    public void visitPlus(Plus s);
    public void visitNumber(Number n);
}

```

上面的代码给定了一个模型，可以让开发人员实现这个接口来处理数字而支持加法操作。如果说要处理 $1 + 1$ 或者 $1 + 2 + 3$ 这样的表达式，那么这个设计足矣。这种设计也可以用于处理写操作。可以看一下随后是怎么将表达式打印出来的。

```

public class PrintVisitor implements Visitor {
    StringBuffer sb = new StringBuffer();

    public void visitPlus(Plus s) {
        s.getFirst().visit(this);
        sb.append(" + ");
        s.getSecond().visit(this);
    }

    public void visitNumber(Number n) {
        sb.append(n.getValue());
    }
}

@Test public void printOnePlusOne() {
    Number one = new Number(1);
    Expression plus = new Plus(one, one);

    PrintVisitor print = new PrintVisitor();
    plus.visit(print);

    assertEquals("1 + 1", print.sb.toString());
}

```

这些都是很基础的内容，在任何一本编程的书里都可以找到。但现在来设计几个问题看一下如何解决。假想给表达式运算添加新的一个功能——支持减法。为了支持这个功能，很自然地要添加一个新的 Java 类。

```

/** @since 2.0 */
public final class Minus extends Expression {
    private final Expression first;
    private final Expression second;

    public Minus(Expression first, Expression second) {
        this.first = first;
        this.second = second;
    }
    public Expression getFirst() { return first; }
    public Expression getSecond() { return second; }
    public void visit(Visitor v) {
        /* 现在考虑一下如何向一个接口中添加新方法。*/
        v.visitMinus(this);
    }
}

```

但只是添加一个数据类还不够。还需要给 Visitor 这个接口添加一个新方法 visitMinus。但添加新方法这事说起来容易做起来难，因为向接口中添加 visitMinus 这个方法是一个不兼容的调整。也许在程序链接时不会有问题，但原有的代码在新版本的 Visitor 接口上是无法通过编译的，新版中增加了没有实现的方法。如果用新数据结构在老版本上执行代码，那么就会抛出一个 java.lang.AbstractMethodError 错误。

```

Number one = new Number(1);
Number two = new Number(2);
Expression plus = new Minus(one, two);

PrintVisitor print = new PrintVisitor();
plus.visit(print); // fails with AbstractMethodError

assertEquals("1 - 2", print.sb.toString());

```

看来在这个常见的 Visitor 设计模式背后还隐藏了一个无法逃避的问题，一旦有这类改进时，就会暴露出来。对于一个内部系统来说，这个问题可能还无关紧要，但如果它作为一个对外的 API，那么就意味着无法对其进行扩展了。所以说，这个模式并不适合解决广泛的问题，至少以目前的设计来说是做不到的。需要进一步处理 Visitor 来解决这类问题。接下来各节中就会对这个新的处理方案加以说明。

18.1 抽象类

首先看一个简单的方案，它多少还是可以解决这个问题的，那就是把 Visitor 设计成一个抽象类而不是一个接口。当然在 6.8 节中已经说过抽象类用处不大。但如果需要略作改进就解决问题，那么不能不考虑这种方案。在 Java 中，如果说需要提供继承方面的支持，而且希望添加新方法的风险尽量小，那么抽象类可以说是一个唾手可得的方案。现在和原先的设计不一样，不必在 1.0 版本中使用接口，而是使用一个抽象类，其所有方法全部都是抽象方法，那么就有可能通过新添加一个 Minus 类在 2.0 版中增强 Visitor 类的功能。

```

/** @since 2.0 */
public final class Minus extends Expression {
    private final Expression first;
    private final Expression second;

    public Minus(Expression first, Expression second) {
        this.first = first;
        this.second = second;
    }
    public Expression getFirst() { return first; }
    public Expression getSecond() { return second; }
    public void visit(Visitor v) {
        v.visitMinus(this);
    }
}

public abstract class Visitor {
    public abstract void visitPlus(Plus s);
    public abstract void visitNumber(Number n);
    /** @since 2.0 */
    public void visitMinus(Minus s) {
        throw new IllegalStateException(
            "Old visitor used on new exceptions"
        );
    }
}

```

对于这个新写的类 Minus 来说，它的 visit (Visitor) 方法里会调用 Visitor 新加的方法。对于基于 1.0 版本所编写的代码来说，当调到这个方法时，就会抛出一个异常，提醒开发人员要使用 2.0 版本的数据结构。因为开发人员并不知道 Visitor 这个抽象类中添加了这个方法，所以也不可能覆盖这个方法，因此抛出一个异常也可以算作一种非常自然的行为，而程序员也没有考虑过这种情况。如果执行中出现异常，引发程序中止，这样做总比出了问题还偷偷摸摸地藏起来好，否则出现问题就很难找到了。

读者也许会认为向类中添加新方法是一种不完全兼容的行为。因为这个类的子类有可能已经定义过一个同样的方法了，而且方法也不见得是公开的，即使是公开的，在 2.0 版中的行为也会与想象中的迥异。但对于 Visitor 这个案例来说，却自有其特别之处。添加一个新的方法就意味着要添加一个新的类。因为 Java 虚拟机中对方法的标识是包括其参数类型在内的，即使两个方法名称相同，但参数不同，那么虚拟机也认为这是两个不同的方法。

这种新的“抽象类”的解决方案与经典的 Visitor 设计模式相比，只有一个区别，就是前者用抽象类，后者用接口。但这种方案可以解决大部分对 API 进行改进时出现的问题。它可以保证源代码级别的兼容性，它同时也是二进制兼容的。当然，将新老版本混合使用的时候，可能会抛出异常。但相比之下，这只是一个缺点而已。我们已经取得了长足的进步，现在经过改进的 Visitor 模式已经可以用来开发广泛适用的 API 了。

18.2 为改进做好准备

在某些情况下，将默认的行为指定为抛出异常并不是一个很合适的方案。比如说，当你想验证对于 `Language` 这个类库 1.0 版本来说，一棵表达式语法树是否合法，代码就得这么写。

```
private class Valid10Language extends Visitor/*version1.0*/ {
    public void visitPlus(Plus s) {
        s.getFirst().visit(this);
        s.getSecond().visit(this);
    }
    public void visitNumber(Number n) {
    }
}

public static boolean isValid10Language(Expression expression) {
    Valid10Language valid = new Valid10Language();
    try {
        expression.visit(valid);
        return true; // 在没有未知运算规则的情况下，返回 true。
    } catch (IllegalStateException ex) {
        return false; // 这里出错了，可能是因为调用了 visitMinus 方法/*2.0*/。
    }
}
```

代码可以这样写，但估计没有几个人会这样写就是了。首先，对一个通常情况下不会抛出的操作也会进行异常捕捉，不是正常的编码方式。第二个问题在于，在用第一个版本的 `Visitor` 来写代码的时候，就必须要知道所有未来采用的方法都会抛出异常，所以要把代码写成这个样子。当然可以在文档中指出，代码最好写成这个样子，但前提是 API 的设计者在写 1.0 版本的 `Visitor` 的时候，也要意识到这一点才成。但如果说 API 的设计者能提早意识到这一点的话，那么其实还可以找出更好的解决方式。其实还有第三个问题，那就是需要开发人员能正确地识别所抛出的异常，知道抛出异常的原因是语言中新加元素缺失处理程序。能做到这一点有点困难了，因为可能会有多个类似的异常从其他地方抛出。简而言之，这个 API 的易用性看起来实在不怎么样。

当然，我们一早就知道 API 的第一个版本绝非完善，但最好还是从一开始就为后续的演进做好准备。应该尽力去寻找一个方案，能够让开发人员自己来处理未知的元素。然后让开发人员选择是抛出一个异常，还是做其他更恰当的事情，比如说验证表达式语言的合法性。所以在 `Visitor` 的 1.0 版本最好定义一个回调的方法用来处理未知的元素，看下面代码中的 `visitUnknown(Expression)` 方法。

```
public abstract class Visitor/*1.0*/ {
    public void visitUnknown(Expression exp) {
        throw new IllegalStateException("Unknown element faced: " + exp);
    }
    public abstract void visitPlus(Plus s);
    public abstract void visitNumber(Number n);
}
```

在接口的 1.0 版本中，是不会调用 `visitUnknown` 这个方法的。但如果出现了 API 改进方面的需求，那么这个方法就用得上了。默认情况下，新添加的方法，如 `visitMinus` 就会调用这个 `visitUnknown` 方法。

```
public abstract class Visitor/*2.0*/ {
    public void visitUnknown(Expression exp) {
        throw new IllegalStateException("Unknown element faced: " + exp);
    }
    public abstract void visitPlus(Plus s);
    public abstract void visitNumber(Number n);
    /** @since 2.0 */
    public void visitMinus(Minus s) {
        visitUnknown(s);
    }
}
```

尽管添加一个 `visitUnknown` 方法不算是一个大的变化，而且默认的行为也没有任何改变，仍然是抛出一个异常 `IllegalStateException`，但是对于那些只想验证表达式树是否有效的 API 用户来说，却就友好多了，因为用户不需要加上一堆的异常捕捉代码了。

```
private class Valid10Language extends Visitor/*version1.0*/ {
    boolean invalid;

    @Override
    public void visitUnknown(Expression exp) {
        invalid = true;
    }
    public void visitPlus(Plus s) {
        s.getFirst().visit(this);
        s.getSecond().visit(this);
    }
    public void visitNumber(Number n) {
    }
}

public static boolean isValid10Language(Expression expression) {
    Valid10Language valid = new Valid10Language();
    expression.visit(valid);
    return !valid.invalid;
}
```

只要预先多多考虑一下演进问题，那么实现 Visitor 的人使用起 API 来就清楚多了，因为所有新添加的未知功能都转交给 `visitUnknown` 方法来处理。因此 `visitUnknown` 这个方法可以用来处理常见的情况。这样做，极大地提升了 API 处理此类未知情况的能力，而且这样做不会对其他已知方法的使用造成不良影响。只有两个抽象方法需要实现，同时以前的代码无须任何修改即可正常运行。

18.3 默认的遍历

一定要记住一点，用户们在使用 API 时，总是会将 API 所有的潜力都挖掘出来，使得 API

的用法远远超出设计者的想象。所以最好能告诉这些用户，哪些用法是错误的，API 设计出来并不是要处理他们那些特殊、古怪的需求。不过如果在没有将 API 复杂化的前提下，API 能提供更多的功能来满足用户的需求，那么越来越多 API 的用户也会满意此 API。如果对该 API 满意的用户越多，那么你所设计的 API 也就越发有用。所以公开地告诉用户，哪些功能已经超出了 API 的功能领域，是不可能通过正常使用 API 来实现的，是非常有意义的。前面的方案中，visitUnknown 这个方法并不能支持对一个表达式结构进行全面的遍历，它只针对部分结点，所以这个方案就不是一个纯粹的 Visitor 模式了，它更像是一个 Scanner。不管怎么叫，你恐怕还是想找个最优的 API 解决这个问题。对于一个静态语言来说，想写一个这样的 Scanner 其实是很容易的一件事，但如果如果说这种表达式的语法还会改变，在后续会添加更多元素的支持，那么就没办法了。有一个基于 Visitor 的方案，其代码如下。

```
private class CountNumbers extends Visitor/*version1.0*/ {
    int cnt;

    @Override
    public void visitUnknown(Expression exp) {
        // not a number
    }
    public void visitPlus(Plus s) {
        s.getFirst().visit(this);
        s.getSecond().visit(this);
    }
    public void visitNumber(Number n) {
        cnt++;
    }
}

public static int countNumbers(Expression expression) {
    CountNumbers counter = new CountNumbers();
    expression.visit(counter);
    return counter.cnt;
}
```

这个方案对于 1.0 版本的语言来说，或许可以正常运行，而一旦在表达式语言中加入了减法的支持，问题就浮出水面了。开发人员可以重载 visitUnknown 方法，但问题是方法内部怎么办。也许什么都不做，但还是会出现问题的，因为不支持减法，所以减法结点的数据就被忽略了，所以像 $1 + (3 - 4)$ 这个表达式，就认为其中只含有一个数字。

```
Number one = new Number(1);
Number three = new Number(3);
Number four = new Number(4);
Expression minus = new Plus(one, new Minus(three, four));

assertEquals(
    "Should have three numbers, but visitor does not " +
    "know how to go through minus",
    3, CountNumbersTest.countNumbers(minus))
```

```
);
```

虽然说让开发人员知道表达式中存在一个无法识别的元素是件好事,但对于一个想遍历特定元素的 Visitor 来说(比如上面的 CountNumbers 就想计算当前表达式有多少个元素),API 怎么也要支持一个“默认的遍历”吧。也就是说对于上面的 $1 + (3 - 4)$ 这个表达式来说,它只会调用 `getFirst().visit(this)` 和 `getSecond().visit(this)` 这两个方法。如果能支持此“默认的遍历”,即使代码是基于 API 1.0 版本所编写的,那么这个表达式中的计算结果也肯定是 3 这个正确的数字。现在唯一的问题在于如何调用“默认的遍历”。

一个幼稚的思路就是把这种“默认的遍历”作为每一个 Visitor 方法的默认实现。因为 Visitor 现在已经变成了一个类,所以这样做从表面上看是可行的。事实上,是行不通的,就像我上面所说的,这样做同样无法重写那个用户提供的 Valid10 类。因为根本没有任何地方或者场景会调用 `visitUnknown` 方法。所有方法都会有这个默认行为,这当然不是你想要的。必须给 API 用户提供一个合适的工具来处理各种情况。如果说在表达式树中出现了一个预期以外的元素,那么用户是可以收到这个消息的。这时可以由他们自行决定如何处理,比如说可以将这种情况看作是一种错误,通过返回一个 `false` 来中止继续遍历,或者返回 `true`,继续遍历,看一下到底都有什么东西。这两种需求都是正常的,所以应该支持,最好让 API 用户很清楚地知道如何来编写自己的代码来完成不同的功能,而且即使 API 添加了新的元素和功能,也同样可以正常运行。一种可能的增强方式是允许 visitor 实现通过 `visitUnknown` 方法的返回值,决定是否遍历未知元素。

```
public abstract class Visitor/*1.0*/ {
    public boolean visitUnknown(Expression e) {
        throw new IllegalStateException("Unknown element faced: " + e);
    }
    public void visitPlus(Plus s) {
        if (visitUnknown(s)) {
            s.getFirst().visit(this);
            s.getSecond().visit(this);
        }
    }
    public void visitNumber(Number n) {
        visitUnknown(n);
    }
}
```

上述方案,可以让用户定制未知元素的默认行为,自行选择中断程序或者继续遍历所有的内容。通过一个这样的 API,就可以完成前面提出的所有任务,就像下面代码是用来计算表达式树上元素的个数。

```
private class CountNumbers extends Visitor/*version1.0*/ {
    int cnt;

    @Override
    public boolean visitUnknown(Expression exp) {
        return true;
    }
    @Override
```



```

    public void visitNumber(Number n) {
        cnt++;
    }
}

public static int countNumbers(Expression expression) {
    CountNumbers counter = new CountNumbers();
    expression.visit(counter);
    return counter.cnt;
}

```

上面的方案不是只对某一个版本的 API 才有效，事实上，它可以用在所有的版本中。如果这个表达式树上添加了新的元素，只需要将 `visitUnknown` 方法的返回默认值设为 `true`，就可以保证所有的元素（不管认识与否）都会处理一次。将遍历的控制权交给了实现 API 的开发人员。这个开发人员也可以控制所有 API 的元素，从而保证了代码的一致性和合理性。

18.4 清楚地定义每个版本

上面的方案将 `Visitor` 设计成了一个抽象类，我也已经说到了该方案可能会出现的问题。虽然所有的功能都运行正常，但也要注意原来简单的抽象类已经不是只包含几个简单的抽象方法，而是变成了一个含有相对复杂的默认行为的组合类了。每当表达式语言出现了新的版本，这些方法就有可能随之增加。每一个方法都会带有一个 `@since` 的说明信息。可以想象一下，这样表达某个功能只有 7.0 版本才支持，而老的版本却不支持，用户用的时候就会比较困难。但这样的需求看起来却又非常合理。如果说我想写一个工具来处理表达式中的元素，最好还是先看一下这个表达式的版本号。对于这种方案来说，我不喜欢直接继承一个庞大的类，然后覆盖它的 `visit` 方法，特别是这个类已经有了多个版本以后，我要从 Javadoc 中找到哪些版本添加了哪些方法，相比之下，我宁愿要一个简单但清晰的接口，直接实现指定的方法，多省事啊。这样，编译器就可以告诉我说已经实现了所有需要实现的方法，可以正常编译了。这样做非常容易，而且事实上也更加“透明”一些，相比之下，抽象类的方式，需要实现一些方法，然后再执行程序，才发现原来还有一些方法要去实现，要去覆盖。从这个角度来看，如果对于语言的每个版本都能有一个接口，那么可能会好用一些。下面就是这种做法。

```

public interface Visitor {
    public void visitUnknown(Expression e);
}

public interface Visitor10 extends Visitor {
    public void visitPlus(Plus s);
    public void visitNumber(Number n);
}

/** @since 2.0 */
public interface Visitor20 extends Visitor10 {
    public void visitMinus(Minus s);
}

```

API 的 1.0 版本只用了一个 `Visitor`，而 2.0 版本却引入了一个新的元素，所以它也引入了一

一个新的 Visitor 类型。接下来对 API 的改变就会更多一些。当然，这样做可能会出现很多个接口，每一个接口都用来支持特定版本的表达式语言的遍历。但这样做有一个最大的好处：如果说你想使用表达式语言 7.0 版本，那么就只需要实现 Visitor70 这个接口就可以了。如果说你只想把表达式语言 2.0 版本中的元素打印出来，那么就只需要实现相应的接口，它可以保证表达式语言 2.0 版本中的元素都会被访问到。

```
class PrintVisitor20 implements Visitor20 {
    StringBuffer sb = new StringBuffer();

    public void visitUnknown(Expression exp) {
        sb.append("unknown");
    }

    public void visitPlus(Plus s) {
        s.getFirst().visit(this);
        sb.append(" + ");
        s.getSecond().visit(this);
    }

    public void visitNumber(Number n) {
        sb.append(n.getValue());
    }

    public void visitMinus(Minus m) {
        m.getFirst().visit(this);
        sb.append(" - ");
        m.getSecond().visit(this);
    }
}

Number one = new Number(1);
Number two = new Number(2);
Expression plus = new Minus(one, two);

PrintVisitor20 print = new PrintVisitor20();
plus.visit(print);

assertEquals("1 - 2", print.sb.toString());
```

相比之下，API 变得清爽了很多。只不过，现在要实现对 Visitor 接口调用的时候，就必须复杂一些了。现在每一个表达式中的元素都需要正确处理相应的逻辑，以保证调用相应的 visit 方法，参见下面的代码。

```
public void visit(Visitor v) {
    if (v instanceof Visitor20) {
        ((Visitor20)v).visitMinus(this);
    } else {
        v.visitUnknown(this);
    }
}
```

你必须在代码运行时通过 `instanceof` 这种判断来决定如何调用具体的代码。同样，这种方式放弃了编译时期的强检查机制。只有当运行到错误代码的时候，开发人员才会发现自己犯了某个错误。当然了，这个代码是语言结构实现的一部分，这里只由你自己写了一次，因此可以努力把这段代码写得正确。但是，如果系统有多个 `visit` 方法的话，那么这种难看而危险的代码会散落在很多地方，就很难跟踪和维护了。还有一个问题，就是如果在代码演进中一直坚持这种做法，那么每多出一种新的表达式语言，就意味着这些代码的长度又会增加了。

18.5 单向改进

通常来说，演化都是向着一个方向进行的，可以算是单向的。但假设有一种情况，已经设计了一个新的表达式语言，但该语言认为整数没有什么用处，而是把所有的数字都当成双精度数来处理。当然，这样还是可以支持整数的，只不过现在不需要通过 `Visitor` 来访问整数就是了。这样，在新的 3.0 版本中，其实就用不着 `Number` 这个类了。

```
/** @since 3.0 */
public final class Real extends Expression {
    private final double value;
    public Real(double value) {
        this.value = value;
    }
    public double getValue() {
        return value;
    }
    public void visit(Visitor v)
}

/** @since 3.0 */
public interface Visitor30 extends Visitor {
    public void visitPlus(Plus s);
    public void visitMinus(Minus s);
    public void visitReal(Real r);
}
```

要做到上面所说的内容，并不是简单地在模型中定义一个新的元素就可以了，还需要引入一个新的 `Visitor`。这个 `Visitor` 非常特殊，因为它并不是继承自 1.0 版本和 2.0 版本中定义的 `Visitor`。继承在这里没有任何意义：`Visitor30` 没有定义 `visitNumber` 方法，因为在 3.0 表达式版本中，根本不需要支持整数，只支持实数就已经够了。

本来代码在处理老版本的表达式语言时，可以正常运行。而一旦决定要使用 3.0 版本的表达式语言，就必须实现 `Visitor30` 这个接口了。但这样做，会把 `visit` 方法的实现变得越发复杂起来。现在所有的数据元素都需要在运行时多检查一个 `visitor` 类型。比如说，原来 `Minus` 类中已经很复杂的代码就变得更加复杂了，可以看下面的示例代码。

```
/** @since 2.0 */
public final class Minus/*3.0*/ extends Expression {
    private final Expression first;
    private final Expression second;
```

```

public Minus(Expression first, Expression second) {
    this.first = first;
    this.second = second;
}

public Expression getFirst() { return first; }
public Expression getSecond() { return second; }

public void visit(Visitor v) {
    if (v instanceof Visitor20) {
        ((Visitor20)v).visitMinus(this);
    } else if (v instanceof Visitor30) {
        ((Visitor30)v).visitMinus(this);
    } else {
        v.visitUnknown(this);
    }
}
}

```

现在看起来，3.0 版本内部具体实现的代码比起 2.0 版本来说，还要丑陋得多。每当出现一个这样的新的单向的版本，这样的代码就变得更差。

18.6 使用接口时的数据结构

关于 Visitor，已经讨论了这么多的内容，都一直有一个假设的前提：尽量要使用 final 类来描述一个数据结构。这样做的结果就是，对于一个类库来说，每一个元素都只有一个实现，在运行时要通过 instanceof 来进行检查。

但有时候，最好别使用类，前面已经说过，这个建议主要和性能有关。假设你创建了几千甚至数以百万计的对象实例，此时，每个对象节省一两个字节都很重要，如果每两个对象能少创建一个同样也会带来显著的性能区别。比如说，编译器和其他语言处理器所承担的工作，远不止单单为 Number 之类的模型类创建对象实例。它们需要维护额外的关联信息，例如元素的内存偏移信息。这样，为了增强现有类型而引入新的数据成员，同时避免代理的开销，应该允许现有类型被继承，由子类提供新的数据成员。特别是想使用多继承实现新接口时，比较有用。如果你需要优化内存占用，避免创建过多的对象实例，基于接口描述模型定义是比较合理的。

但是，由于模型代表了客户 API，所以把类转换成开发人员可以实现的东西，看起来好像已经违反了本章前面给出的所有建议。的确如此，如果想把模型定义为接口，那么在 1.0 版本中，就必须定义得非常正确。因为之后就不会再有改正的机会了。从版本对外发布以后，再向接口中添加新方法都会破坏现有的实现代码。不过让人吃惊的是，这样并不意味着没有办法对 API 加以改进了。Visitor 模式可以很好地解决这个演进问题。

首先，要在新的 API 中创建一个新的模型元素，就像 2.0 版本中新添加的那个 Minus 类一样。在 API 中添加新的类，完全是二进制兼容的，在大部分情况下对于源代码来说也是兼容的，所以它是一种完全正确的改进策略。如果新加的这些类扩展了 Expression 这个基类，那么总是可以将它们至少传递给 visitUnknown(Expression) 方法。接下来的问题就简单了，就是在 Visitor 中添加一

个新的 visit 方法。我们已经看到在两种情况下这都是可能的。如果 Visitor 是一个抽象类，添加新方法自然没有问题；如果是接口的话，就可以为表达式语言新版本定义新的 Visitor 接口。这就是说还得使用那种单向的改进方式，这种方式虽然有不少问题，但如果说碰到了模型接口上存在设计失误的情况，不需要支持现有模型了，那么这种方式还是很有效的，就像前面那个 Real 和 Visitor30 的例子一样，它就是不再需要支持 Number 类了。

最好在发布 1.0 版本的时候，就能把接口设计正确。但对于 Visitor 这种设计模式来说，如果出现了设计错误，还是有办法补救的。即使使用接口来描述表达式中的结点也是一种可以接受的方式，只要允许在后期对其进行改进，就可以把它看作一个合适的 API 设计模式。

18.7 针对用户和开发商的 Visitor 模式

使用接口来解决问题往往存在一个非常烦人的问题：当出现了多个表达式语言的版本以后，相应的 visit 方法会变得非常糟糕。如果说只有一个实现，那么正常情况下，基本不会出现什么问题，所以前面说过也是可以接受的。但对于这些接口来说，如果没有默认实现的话，那么就意味着每个接口的每个实现都得有这些糟糕的代码。而且，所有实现的代码还必须能够保持同步。考虑到每一个 API 的新版本都要定义一个新的 visitor，还必须更新所有的实现代码，来通过 instanceof VisitorXY 这样对相应的对象进行类型检查。这种做法很不好，因为同步的内容会越来越多，随时可能会落下一些内容。

为了解决这种同步带来的不便，需要使用一些 API 改进方面的技巧。对于本章来说，这个技巧非常重要，它会根据用户和开发商将设计的 API 进行分离，在第 8 章中我们已经讨论过这个问题了。前面所说的那几个使用了 VisitorXY 类型的例子，其目的都有两个。有些开发人员实现这个接口是为了遍历整个数据结构中的模型数据类或接口，而另一个目的则是想通过 visit 方法的实现来正确地派出模型类的实现。这一点才是目前为止出现问题的根源所在。还好解决方案比较简单，把这两个类一分为二就是了。下面是表达式语言 1.0 版本改进后的代码。

```
public interface Expression {
    public abstract void visit(Visitor v);
}

public interface Plus extends Expression {
    public Expression getFirst();
    public Expression getSecond();
}

public interface Number extends Expression {
    public int getValue();
}

public abstract class Visitor {
    Visitor() {}

    public static Visitor create(Version10 v) {
        return create10(v);
    }
}
```

```

public interface Version10 {
    public boolean visitUnknown(Expression e);
    public void visitPlus(Plus s);
    public void visitNumber(Number n);
}

public abstract void dispatchPlus(Plus p);
public abstract void dispatchNumber(Number n);
}

```

在这个例子中，模型类使用了接口，如果说不考虑性能优化的话，其实用 final 类也是不错的。Expression 这个类里定义了 visit 方法，可以把 Expression 看作是一种“客户 Visitor”，也许可以称为“调度者”。外部是不需要实现这个类的，而这个类则会负责调用相应的方法。如果想实现一个自己的 Visitor，那么 API 的用户可以实现 Visitor.Version10 这个“提供者 Visitor”接口，然后通过 Visitor.create 这个方法创建一个相应的 Visitor，最后再调用 expression.visit(v) 来遍历相应的数据。基于这个方案，就很容易对 API 进行演进了。假设在表达式语言 2.0 版本中，添加了对减法表达式的支持，那么代码可以写成下面这个样子。

```

/** @since 2.0 */
public interface Minus extends Expression {
    public Expression getFirst();
    public Expression getSecond();
}

public abstract class Visitor {
    Visitor() {}
    /** @since 2.0 */
    public static Visitor create(Version20 v) {
        return create20(v);
    }
    /** @since 2.0 */
    public interface Version20 extends Version10 {
        public void visitMinus(Minus m);
    }

    /** @since 2.0 */
    public abstract void dispatchNumber(Number n);
}

```

我们可以看到其他的方法仍然没有改变，只是添加了一个新的“提供者 Visitor”接口 Version20 来扩展了原来的 Version10 接口。一个工厂方法可以将这个 Visitor 转成带新方法的“客户 Visitor”，该新方法由 Minus.visit(Visitor) 的实现来调用，从而实现调度功能。

如果表达式语言升级到了 3.0 版本，使用实数来替代原有的整数。但 Version30 这个接口却不会继承现有的任何一个接口，因为 3.0 版本语言的变化与前面的调整已经不是一种类型的了，所以不再需要原有接口中的多数方法，只需要使用部分方法即可。现在来看下面的代码。

```

/** @since 3.0 */
public interface Real extends Expression {
    public double getValue();
}

public abstract class Visitor {
    Visitor() {}

    /** @since 3.0 */
    public static Visitor create(Version30 v) {
        return create30(v);
    }

    /** @since 3.0 */
    public interface Version30 {
        public boolean visitUnknown(Expression e);
        public void visitPlus(Plus s);
        public void visitMinus(Minus s);
        public void visitReal(Real r);
    }

    /** @since 3.0 */
    public abstract void dispatchReal(Real r);
}

```

18.8 三重调度

Visitor 模式又被称为“双重调度”，因为 `Expression.visit(visitor)` 这个方法在执行的时候，会调用 Visitor 中的方法，但具体如何调用，还要看 Expression 子类的具体实现，所以具体方法的执行既依赖于 Visitor，还依赖于 Expression 的子类，所以叫“双重调度”。而上面所说的“客户和提供者 Visitor”则可以称为“三重调度”，因为具体方法的调用会依赖于具体表达式、表达式语言的版本及 Visitor 子类的具体实现。先来看一下在表达式语言 3.0 版本中是如何实现 Visitor.create 方法的。下面的代码是基于 1.0 版本编写的。

```

static Visitor create10(final Visitor.Version10 v) {
    return new Visitor() {
        @Override
        public void dispatchPlus(Plus p) {
            v.visitPlus(p);
        }

        @Override
        public void dispatchNumber(Number n) {
            v.visitNumber(n);
        }

        @Override

```

```

        public void dispatchMinus(Minus m) {
            if (v.visitUnknown(m)) {
                m.getFirst().visit(this);
                m.getSecond().visit(this);
            }
        }

        @Override
        public void dispatchReal(Real r) {
            v.visitUnknown(r);
        }
    };
}

```

由于在表达式语言 1.0 版本中, 只支持 Plus 和 Number 这两种元素, 所以只有在遇到这两种元素的时候, 才会调用相应的方法, 如果是不认识的元素, 就会调用 visitUnknown 方法。而且, 在处理 Minus 元素的时候, 由于不认识该元素, 就会调用 visitUnknown 方法, 同时看这个方法的返回值, 如果为 true, 那么继续对元素进行遍历, 这样对于像前面说的那种 CountNumbers 这类计算 Visitor 就可以正常运行了。在表达式语言 2.0 版本下, 这种方案也可以很容易地就支持这种改进, 因为不需要为 Minus 提供回调了, 参见下面的代码。

```

static Visitor create20(final Visitor.Version20 v) {
    return new Visitor() {
        @Override
        public void dispatchPlus(Plus p) {
            v.visitPlus(p);
        }
        @Override
        public void dispatchNumber(Number n) {
            v.visitNumber(n);
        }

        @Override
        public void dispatchMinus(Minus m) {
            v.visitMinus(m);
        }

        @Override
        public void dispatchReal(Real r) {
            v.visitUnknown(r);
        }
    };
}

```

要支持表达式 3.0 就复杂了, 只不过还是可以做到的。因为对于原来的模型来说, 将整数转化成实数也是一个合理的处理方式, 只不过需要增加一点点额外的工作量而已, 代码如下。

```

static Visitor create30(final Visitor.Version30 v) {
    return new Visitor() {
        @Override

```



```

    public void dispatchReal(Real r) {
        v.visitReal(r);
    }

    @Override
    public void dispatchNumber(final Number n) {
        class RealWrapper implements Real {
            public double getValue() {
                return n.getValue();
            }
            public void visit(Visitor v) {
                n.visit(v);
            }
        }
        v.visitReal(new RealWrapper());
    }

    @Override
    public void dispatchPlus(Plus p) {
        v.visitPlus(p);
    }

    @Override
    public void dispatchMinus(Minus m) {
        v.visitMinus(m);
    }
};
}

```

在这个解决方案中引入了一个新的元素，就是 `RealWrapper` 这个类，它可以对一个整数进行封装，作为一个 `Real` 对象来使用。使用了这个转换类以后，`Version30` 就可以遍历那些使用整数的老版本模型了。

18.9 Visitor 模式的圆满结局

这种“客户和提供者 Visitor 模式”可以很好地解决前面谈到的各种问题。所以它是一种真正的“可扩展 Visitor 模式”。

- 可以向模型添加新的元素。
- 支持 `visitUnknown` 方法来访问未知元素。
- 对于未知的元素，也仍然提供了默认深度遍历的方式。
- 可以很清楚地分辨每一个不同的表达式语言的版本，因为每一个不同表达式语言的版本都提供自己特有的 Visitor 接口，这些接口可以将不同版本的模型和 Visitor 进行混用，可以在任何模型上使用任何的 Visitor。
- 可以支持那种单向演进，如 2.0 版本到 3.0 版本的改变。
- 解决方案是类型安全的，无须使用任何反射或者内省技术。
- 对于模型类来说，可以使用接口，同时也不妨碍演进是类型安全的。

对于上面这样一个设计，还能再要求什么呢？！结局可谓是皆大欢喜了。能有这样一个圆满

结局就是因为这个设计遵守了一个重要的原则：将用户和开发商分别要使用的功能分离清楚，并提供不同的接口予以支持。第 8 章中已经对这个话题有过很详细的说明了。在计算机科学中有一句古老的谚语，那就是：计算机科学中出现的所有问题都可以通过一个中间层来解决。看来这话也同样适用于 API 设计。

18.10 语法小技巧

尽管结局非常好，但还有一些东西需要进一步讨论。下面的代码是根据上面的方案来重写了一个 PrintVisitor 类，但你会发现原来 visitPlus 方法中的传统代码无法编译通过了。

```
public class PrintVisitor implements Visitor.Version10 {
    StringBuffer sb = new StringBuffer();

    final Visitor dispatch = Visitor.create(this);

    public void visitPlus(Plus s) {
        // s.getFirst().visit(this); // 无法编译通过，但需要留着给大家看一下：
        s.getFirst().visit(dispatch);
        sb.append(" + ");
        s.getSecond().visit(dispatch);
    }

    public void visitNumber(Number n) {
        sb.append(n.getValue());
    }

    public boolean visitUnknown(Expression e) {
        sb.append("unknown");
        return true;
    }
}
```

Expression.visit 这个方法要求 PrintVisitor 类只实现 Version10 这个接口。为了避免无谓的笔误，最好是创建一个新的调度者，或者通过一个对象引用来持有一个调度者的实例对象，就像上面代码中的 final 变量 dispatch 一样。一旦需要进行调度操作的时候，就可以把这个对象改成具体 Visitor 的实现，这样就可以在 visit 方法中对其进行访问了。但这样对于 Visitor 对象来说，可能会被引用两次，因为它是 Print 类中的一个实例变量。同样，想发现这样的小陷阱是有点困难的。不过换个角度看，只需要在 Javadoc 中提一下，或者通过一个小例子就可以解决这些问题。但这是一种传统的修复方式，不需要任何语法上的小技巧，也不需要 Visitor 模式进行任何调整。

还有一个方案，它可以对接口做一些增强，这样可以帮助用户轻而易举地找到实现一个 Visitor 的正确方法。在 VersionXY 接口中的每一个方法都有一个新的参数 Visitor self。

```
public abstract class Visitor {
    Visitor() {}

    public static Visitor create(Version10 v) {
        return create10(v);
    }
}
```

```

    }

    public interface Version10 {
        public boolean visitUnknown(Expression e, Visitor self);
        public void visitPlus(Plus s, Visitor self);
        public void visitNumber(Number n, Visitor self);
    }

    public abstract void dispatchPlus(Plus p);
    public abstract void dispatchNumber(Number n);
}

```

self 总是传递给 Expression.visit(Visitor) 方法的 Visitor，它可用在旧的 Print visitor 例子中替代 this。

```

public class PrintVisitor implements Visitor.Version10 {
    StringBuffer sb = new StringBuffer();

    public void visitPlus(Plus s, Visitor self) {
        s.getFirst().visit(self);
        sb.append(" + ");
        s.getSecond().visit(self);
    }

    public void visitNumber(Number n, Visitor self) {
        sb.append(n.getValue());
    }

    public boolean visitUnknown(Expression e, Visitor self) {
        sb.append("unknown");
        return true;
    }
}

@Test public void printOnePlusOne() {
    Number one = newNumber(1);
    Expression plus = newPlus(one, one);

    PrintVisitor print = new PrintVisitor();
    plus.visit(Visitor.create(print));

    assertEquals("1 + 1", print.sb.toString());
}

```

至于到底使用哪种处理方式，就要看用户自己的决定了。第一种处理方式虽然简单，但用起来却有点不方便，因为用户不会特别留意到 Visitor 10 这个接口；第二种则比较复杂。但如果想递归调用 Visitor 的话，还是可以考虑使用第二种方式。不管怎么说，也不管具体用哪种处理方式，Visitor 模式都是一个非常好的 API 设计模式，只要正确使用，是可以在 API 开发中大显身手的。

本书的大部分内容都在建议要保持兼容性，在设计类库和 API 的时候，要保持其兼容性，不管做什么事，都不应该破坏 API 用户现有的代码。这就是说，不能从 API 中移除任何一个方法、字段、类或包，因为这样做会给潜在客户带来不兼容性问题。只有这样，你才能维持现有类库 API 不变，或者引入新的内容。一个有趣的问题是，如果开发人员这样做的话结果会怎样？那是不是就意味着代码规模会无限制地增长呢？是不是说开发人员将会把大部分的精力都用来维护现有的类库呢？开发人员还有时间来编写新的代码吗？

经验告诉我们，事实不太可能如此。一方面，类库可能需要维护，可能存在大量 bug，而用户不可能使用 bug 丛生的类库。而另一方面，类库可能工作正常，那最好不要去修改它。有时候需要向类库添加新功能，或者需要修改现有 API 以适应新的需求。这可能来自市场竞争的压力，用这些改变来说服开发人员选择使用这个类库。无论如何，这可能意味着用户数量是非常大的。那么，投资于这样的改变还是值得的。然而，在有些情况下，整个类库还是可以接受并能够继续使用的，但某些部分却需要移除。那有什么可行的办法来部分移除现有的 API 吗？

前面曾多次提到，即使一个细微的不兼容问题都会让人感觉整个类库是在以一种不兼容的方式演变。“不兼容”三个字通常对于外部用户而言是一个坏消息，所以要尽可能地避免出现这种情况。Java 框架中就存在这样极端的例子，而且是在它的核心类库中。在 JDK 5 版本中，核心类被打包在 `rt.jar` 中。该 Jar 所包含内容是从 JDK 1.0 版本起所有内容的超集，最早要追溯到 1996 年。所有版本的 API 类都还存在，即使其中有些类十年前就已经被列为不推荐使用的，这意味着这些类不再被使用或者将被删除，但它们仍然被保留下来，因为相对于清理现有类库而言，对兼容性的保证更为重要。这里面不仅包含几个零落的方法或类，而且有时包含整个包，例如 `java.beans.beancontext`。它们之所以被保留下来，原因无它，唯兼容性而已。它们一直存在，只是为了履行之前对 Java 开发人员的兼容性承诺。

长期保持 API 兼容性确实是一件了不起的事情，必须投入大量的时间和金钱以履行这样的承诺。API 的维持工作往往成本非常高昂，而且，维持那些已经被证明是错误的或者已经在新版本中被替换的 API 是非常枯燥的，尤其是历经多年之后。必须有极富意义的愿景来激励我们对这种兼容性的追求，这种激励必须让人们相信唯有这样做才是可取的。他们得认识到要么保持兼容，要么违背承诺完全放弃兼容性，非此即彼。这样的观点会导向下面的主张：API 譬如恒星，一经

发现，永不消失。从某种角度讲，这样的观点无疑是正确的。然而，从更为灵活的观点看来，这样做很可能被认为是徒劳的。

这里要强调的是，通过一些框架的技术支持并配合细心处理，是可以应对某些类不再被使用的问题。在这点上，NetBeans 就提供了例证。很久之前，我们有一个基础的 API 类库，名为 OpenAPI，这些 API 全部都被打包在 openide.jar 文件中。经过若干年的维护，其中多数 API 已经被污染，好多已经被标记为不推荐使用。某些类最终被发现当初的设计极其糟糕，甚至是最坏的选择，这严重阻碍了整个 API 库的演进。我们在项目中通过模块化的办法来处理这些类。再者，我们也从 NetBeans Runtime Container 得到了一些支持，帮助我们将不再需要的类移入特定的 Jar 文件中。我们得以逐步不再向用户推荐使用这些类，进而将它们从产品中移除，同时又能对 API 用户保持向后兼容性^①。

19.1 明确版本的重要性

对 NetBeans 来说，我们之所以能够做到“兼容式移除”（compatible removals），是因为我们意识到仅仅指明对特定 API 的依赖是不够的，而且需要明确指明所依赖的具体版本。如下的依赖声明是不完备的。

```
import java.awt.*;
```

仅仅指明哪个包是不够的，还必须明确包的具体版本。比如说，你需要明确需要的是 JDK 1.2 版本的 java.awt 包。每一个使用此包的应用程序，都应该包含这类版本信息。如果包含了这类信息，运行期容器或者平台就可以据此提供正确的运行环境，即 java.awt 版本 1.2 所需要的运行环境。

19.2 模块依赖的重要性

对 NetBeans 来说，达成“兼容式移除”的另一个前提是：不通过指定路径的方式声明对 Jar 文件的引用，而是通过依赖声明的方式。依赖声明需要一定的解释，而且解释过程中可以附加额外的逻辑，从而屏蔽那些潜在的由于类库版本不同而引起的不一致性和不兼容性问题。

如果使用 Java 原有的 -classpath 的方式，你只能告诉运行期容器：“我需要 rt.jar，因为它包含了我想要的 java.awt 包。”然而，这种方式暴露了太多实现方面的细节性内容。归根到底，你真正的需求是：我的程序是基于 JDK 1.2 编译的，需要 1.2 版本对应的 java.awt 包。

猛一看，这好像算不上是什么大的进步，但这样做确实开启了通向“兼容式移除”的大门。运行期容器可以清楚地知道特定组件的历史版本信息。在这个例子中，组件指的就是那些包，例如 java.awt。因此，在运行时，相应版本的类会被注入到运行时路径，这就使得后续版本的 java.awt 包可以不再包含 java.awt.Canvas 类，比如 1.3 版本。确实，基于 1.3 版本的 java.awt 编译的代码肯定是不大会用到这个类了，但是对于那些已经使用 1.2 版本编译，但同时又想在 1.3 版本之上运行的开发人员而言呢？一定会遇到 ClassNotFoundException 错误吗？不会，肯定不会。因为程序已经

^① 这种情况应该是作者所说的模块中的版本控制，可以根据不同的版本声明提供不同的版本，可以参见第 15 章。

——译者注

声明自己需要的版本号为 1.2，运行时系统就会自动找到相应版本，并注入对应的类，使得程序像在旧版本上一样运行于新版本之上。在后续的新版本中，这些旧版本的类可能已经被声明为不推荐使用，或者已经搁置不用，甚至已经被移除。然而，只要老版本还可用，那么系统仍旧能以透明的方式使用它们。在新版本中，`java.awt.Canvas` 仍然可以像在 1.2 版本中一样被正常使用（当然程序必须是声明依赖于 1.2 版本的）。这样整个程序链接和运行时都不会出现任何问题。这个例子说明 API 用户的开发能够保持兼容性，即使是在某些 API 在新版本中被移除的时候。

依赖关系的自动调整

NetBeans 其模块化系统允许每个模块都提供一些说明信息，这些信息可以被用来调整依赖关系。这些依赖声明信息会被系统使用，以创建模块所需要的运行环境。

比如说，NetBeans 之前打算移除 `SystemOption` 类，这个类用来负责用户配置信息的持久化管理，已经显得陈旧和多余，可以使用 `java.util.Preferences` 加以替换。弃用这个类并不困难，因为它是 `org.openide.options` 模块的一部分，而该模块基本不存在其他有用的元素，整个模块可能也已经弃用了。但问题是，有些其他模块对外暴露的 API 依赖于 `SystemOption` 类。比如说，编辑器功能模块提供了一个名为 `PrintSettings` 的共有类，这个类继承自 `SystemOption` 类。我们不可能在处理 `SystemOption` 类的时候，还同时保留 `PrintSettings` 类。因此，我们决定同时弃用 `PrintSettings` 类。图 19-1 描述了最初的情形：`org.openide.options` 模块 6.7 含有 `SystemOption` 类；`PrintSettings` 类则放置于 `org.openide.text` 模块中，该模块同时还含有其他类，例如 `CloneableEditorSupport`；`org.openide.text` 模块的版本号为 6.15，并声明该模块依赖不低于 6.7 版本的 `org.openide.options` 模块，因为它需要使用其中的类。

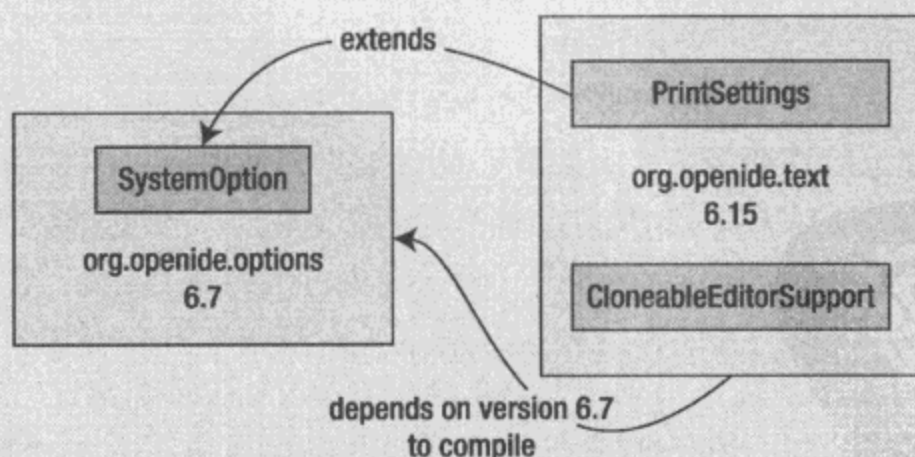


图 19-1 弃用处理之前的 `PrintSettings` 和 `SystemOption`

我们把 `PrintSettings` 这个类从 `org.openide.text` 模块移到了 `org.openide.options` 模块中，同时将其版本号提升至 6.8，如图 19-2 所示。`org.openide.text` 模块的版本号也相应升至 6.16。它根本不再需要对 `org.openide.options` 模块的依赖。现在 `org.openide.text` 模块仅仅包含了 `CloneableEditorSupport` 和其他一些类似功能的有用类。现在，需要被弃用的类已经放置在独立的模块中，系统中的其他模块就不需要依赖这个模块了。这样做有助于我们的改进计划，当 API 用户都不再使用 `SystemOption` 和 `PrintSettings` 这两个类之后，就可以彻底弃用这个模块了。

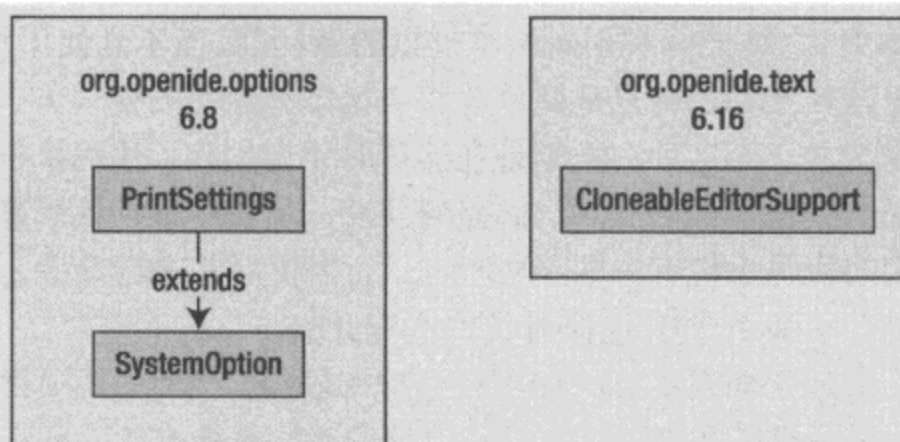


图 19-2 弃用处理之后的 PrintSettings 和 SystemOption

尽管如此，我们还是希望可以维持向后兼容性。将 PrintSettings 类直接从一个模块中移除肯定是一种完全不兼容的处理。但这个类还是存在的，只不过移到了其他的模块中。只需要指定一个模块依赖“自动触发装置”。如果需要使用版本号大于 6.16 的 org.openide.text 模块，就需要调整模块的依赖声明，添加对模块 org.openide.options > 6.8 的依赖。这意味着，对于 org.openide.text 的原有用户来说，他们仍旧可以在编译时使用所有的类，包括 PrintSettings。对于那些只需要 CloneableEditorSupport 类的用户来说，他们可以将依赖声明提升为 org.openide.text > 6.16，并促成对 org.openide.options 模块的弃用，直到再也不需要。

必须要指明一点，在新版本的正式类库中已经没有这个类，但其实老版本类库中还是保留了这个类，这主要是考虑到兼容性的缘故。对基于老版本进行代码编译的开发人员来说，仍然可以成功编译和运行，但对于切换至新版本的开发人员来说，就必须得更新他们的代码了。在 NetBeans 中有两种方式可以用来更新代码。一种是让开发人员自己更新代码，不再依赖于那些弃用的代码，另一种是开发人员调整程序的依赖关系，依赖于那个已经建议弃用的 java.awt。每人都可以有三个选择。

- 保证绝对的兼容性，基于老版本类库开发的代码仍然可以在新版本类库上编译和运行。
- 升级到新版本，但在模块的依赖关系中还是声明其要使用已经被声明为弃用的类库版本。
- 升级到新版本，但会更新代码，放弃使用那些古老且已经被移除的类。

NetBeans 项目告诉我们，开发人员往往更关心如何才能最简单地升级到新版本，所以他们会更希望选择方案 1。也就是说开发人员希望不做任何事情就可以让他们的程序能正常运行。我们可以告诉他们：只需要能保证前一个版本的兼容性，那么问题就可以解决了。

但经验告诉了我们，总有一些开发人员希望能使用最新的技术。他们想使用类库的最新版本，以使用其最新的功能，为了达到这个目标，他们不介意对其代码进行调整。这些人则倾向于方案 3。他们会修改代码，然后保证其代码不仅与最新版本类库兼容，同时还可以使最新版本的类库正常运行。

还有一些人则选择方案 2，就是去修改自己的代码，但同时使用那些被弃用的类库，当然这种人就是少数了。也许此时要找到一个简单的解决方案，保证要有一个完全兼容的情况，表面上看起来，这和上一个版本没有任何区别。或者说，如果选择了更新源代码，也就意味着有更充足

的理由来升级到新版本上,以便充分利用新版本提供的新功能,而不是使用那些已经被弃用的类。

一定要注意,这样做其实和那种只是简单地将类标识为弃用的处理方式是有所不同的。弃用只会在编译的时候给出警告。而这种方式则是在运行时才会用到。代码执行的时候,会因为发现开发人员使用了一些被弃用的类或者类库,而提醒开发人员说应该切换到新版本上,这样做可以让开发人员知道,他们的代码已经有一些不合时宜了,需要对其进行升级。NetBeans 项目的经验告诉我们,这种情况下,开发人员会选择对代码进行升级。

随着时间的推移,依赖于老版本 API 的项目会越来越少。当然,新项目根本看不到这些老版本的 API,但最终那些老项目还是需要被更新,以便升级到新版本上。所以前面说的“移除部分兼容性”这种事情是不会在一夜之间就实现的。开发人员可以自行决定升级或者继续使用老版本,但不管如何决定,都可以保证在一个合适的环境中来运行代码。

19.3 被移除的部分需要永久保留吗

你可能会认为,相对于用 deprecated 批注(annotation)来处理弃用类的方式,我会简单地建议将它们移到独自的 Jar 文件中。然而,考虑到向后兼容性,这些废弃类还是应该保留下来。虽然不是所有人都认同这种方式,但这话多少是有些道理的。

之所以这样说,是因为 NetBeans 团队确实会将弃用的 API 保留一段时间。然而,当 NetBeans IDE 中所有模块都不再需要这些 API 时,它们就只是被放置在硬盘上,不再被使用。它们既不会在运行时被载入内存,也不会在编译时用到,除非有人因为兼容性的原因才会用到。这样做确实也节省了计算机资源。然而,更大的好处恰恰出现在这种兼容性支持不再需要的时候,也就是当老版本的 API 用户数量降低到足够少时。这时,产品(例如 NetBeans IDE)的默认下载版本就不需要再包含那些老的 API 了。当然,这样做并不意味着那些老的 API 就彻底被删除了:用户仍然可以下载包含老 API 的特定版本,尽管在默认发布版本已经不需要再包含。对于那些还没有升级到新版本的来说,这正好再次提醒他们应该升级了。尽管如此,用户也并不是一定要被强制升级,他们仍旧可以选择兼容老 API 的下载版本。一个理想的运行期容器应该能够根据需要自动下载相应版本 API 库,这样一来,就可以完全避免由于产品不同版本引起程序不兼容的问题。

通过使用类似的技术,可以实现完全兼容,同时还给予 API 的改进以很大的自由度,包括从类库中移除部分 API。NetBeans 项目本身就采用了这种方式,事实证明这是成功的。我们可以像珍视钻石一样维护我们的 API 类库,同时也会弃用其中部分 API,就如同将钻石中的赝品丢进垃圾桶一样。

19.4 分解庞大的 API

正如 2.2 节中提到的,本书提倡模块化设计和基于组件的架构。一个理想的应用程序应该是基于模块化组件装配而成的。但要做到这一点,必须要有合适大小的组件。这里所说的“大小”,并不是说类库在下载时有多少字节。现今,至少对一台接入互联网的台式电脑来说,字节已经不是什么问题。同时,这个“大小”也不是指一个类库有多少个类,因为你不一定会用到其中所有

的类。这里的“大小”是用模块对外依赖的数量来衡量的。如果仅仅为了使用某个类库中的个别有用的功能就直接依赖该类库，而该类库又间接依赖了其他很多额外的类库（这些额外的依赖是为了满足类库中所有功能的需要），这样做是非常不合理的，因为你也必须要继承那些不必要的依赖关系。假如出了这种情况，最好的解决办法是把这个庞大的类库分解成更小的独立模块。

NetBeans 中原有的 OpenIDE 库和 Java 中的 rt.jar

尽管不应该，但这种庞大的类库还是会出现。不过，如果遵循本书中的建议，至少可以避免这种情况的发生。然而，因为庞大类库还是会不时地出现，所以应有所准备。众所周知，项目起初一般不大，但往往会逐渐膨胀，以至于发展到 JDK 中的 rt.jar 和 NetBeans 中原有的 openide.jar^①那样的规模。

之所以会出现这类膨胀的类库，往往是因为，在潜意识中我们认为这是全世界最重要的 API。因此，这些 API 看起来应该被归类为核心 API。如此一来，越来越多的内容都加入到这个所谓的核心类库中，类库的规模也越来越膨胀，以至于没人想去或者能去摆脱它。Java 中的 rt.jar 正是这样一个应该被分解的庞大类库，只不过没人知道 Java 开发团队什么时候才有勇气去做这件事。

在 NetBeans 以前的版本中，所有对外 API 都是放置在 openide.jar 这个文件里的。NetBeans IDE 中的所有模块都需要这个类库。因为这些模块需要很多服务，所以在这个 Jar 文件里就会包含大量的 API。这些 API 的功能涵盖范围很广，从管理窗口、视图和编辑器，到编译、执行以及调试。所有这些都放置在一个 Jar 文件中，一起打包和部署。哪怕你开发了一个应用程序，不需要上面提到的执行功能，无论如何你都需要引入整个 openide.jar。

我们花费了大量的时间来修正这些问题。经过多个阶段，历经从 NetBeans IDE 5.0 之后多个版本，我们终于弃用了这个庞大的 API 类库。我们用 15 到 20 个小的 API 类库来替换原来那个庞大的类库。这样一来，用户可以根据需要独立使用这些小的类库。而不是像原来一样，必须包含一个庞大的 OpenIDE 库。

怎么做才能把一个庞大的类库分解成多个更小的类库呢？你可以按包分解，也可以按类来分解。其他的分解方式就没有任何意义了。因为对于 Java 虚拟机来说，最小的装载单元就是类，想用一个更小的单元是做不到的，除非使用了某种二进制增强技术。现在让我们来看一下如何通过类和包来分解 Jar 的相关示例。

二进制代码增强

对于大部分开发人员来说，二进制代码超出了我们的理解范围。它不是无法理解，也不是不能用它来完成一些计算操作（如果你确实想那么做的话），我自己就做过几次。然而，这明显过于细节化了，超出了无绪的范畴。我不想去记住任何有关二进制代码的细节。如果确实需要，我也可以脱离自己现有的知识范畴进行一番探索，去浏览、查看和修改一些二进制代码。

^① NetBeans 的新版本中已经没有 openide.jar 了，只有老版本中还有。——译者注

只不过，我一直期望从二进制代码领域安全撤离，尽快忘记这种痛苦的经历。我希望多数 Java 开发人员也能够持类似的态度。

另一方面，时代在进步，二进制编码已经不再被认为是不可靠和危险的。尤其是因为面向对象编程相关技术被越来越广泛地被接受，开发人员已经开始将二进制代码看作是一个可以接受的选择。还有一个重要的原因是，现在有更高级的编程语言支持这些二进制操作。你不再需要深入到字节码序列，也不需要调用底层类库。相反，你只需要创建一个相应的 aspect。aspect 并不像纯二进制代码功能那么强大，但这不正是当初汇编程序员对 C 语言的评价：太高级了、太慢等等。

然而，针对特定的二进制代码操作，通过定义更高级的原语操作，就像 aspect 所做的那样，这样带来的益处要大于其缺点，而这些缺点也并不都是绝对的。这改变了 Java 开发人员对此的普遍看法。现在至少二进制代码修改已不再被视为绝对的禁忌。然而我对此类技术并没有足够的经验，关于该主题的探讨只能留给其他这方面的专家了。

类的拆分和包的拆分的唯一不同，就在于包是否为“密封的”。如果包是密封的，那么在两个不同的 Jar 文件中都存在的一个同名类^①。很难让类加载器和 Java 虚拟机去成功装载相应的两个类。当把包整体移到它们所属的 Jar 文件中时，就不会遇到这类问题。

现在问题是如何处理那些依赖众多 Jar 文件的类。如果这些类依赖于很多 Jar 文件，但同时它们又不适合待在其中任何一个 Jar 中，又该如何处理呢？这些类应该放到哪里去呢？例如，在 NetBeans 中我们就遇到过这样的类。

```
public abstract class TopManager {
    public abstract ExecutionEngine getExecutionEngine();
    public abstract CompilationEngine getCompilationEngine();
}
```

为了能够移除该类，我们另外创建了一个名为 openide-compat.jar 的 Jar 文件，用于放置那些放在哪个 Jar 里都不合适的类。比如说，我们最终将 TopManager 这个类放置在这个 Jar 中，就是基于这样的原因。最后一个要解决的问题就是，找到一个 API 放置这些 getter 方法。通常我们在 ExecutionEngine 类中引入一个名为 getDefault 的静态方法，用于获取 ExecutionEngine 对象实例，对 CompilationEngine 类做类似调整，然后让 TopManager 来调用这些新增的方法。我们将 TopManager 类设置为不推荐使用，并替换所有对它的引用。这样，我们就能让 openide-compat.jar 继续存在，外部的 API 用户仍然可以使用这个文件。对于那些依赖包含 TopManager 类的旧环境的用户而言，我们还使用了自动依赖调整技术，这在 19.2 节中有详细阐述。

底层细粒度 API VS. 上层粗粒度 API

我有个同事近来经常抱怨 NetBeans 中编辑器支持部分的实现过于模块化。我们有一个模块提供书签功能，另一个模块提供格式编排功能，还有其他的模块提供诸如代码折叠、代码提

^① 这个是指如果在同一个 ClassLoader 中，即使类路径上有两个 Jar 包中存在同名类，也不可能加载两个同名类，但对于两个不同的 ClassLoader，是可以分别存在于同名类中的。——译者注

示和参数设定等功能。在他看来,对用户而言,这会使编辑器 API 非常难用。因为所有的内容都分散在不同的地方,这样一来,查找某一个功能对应的 API 就非常困难。

该同事建议为整个编辑器功能部分提供一个统一的 API,而且他也确实这样尝试了。他希望用户只需要查找学习一个单一的 API,该 API 包含了创建一个编辑器所需要的所有功能,而不是许多小的 API。

我认为这样的组合能够在 API 技术细节和易用性之间找到很好的平衡。小的模块提供底层细粒度 API,包含全部功能细节,而上层粗粒度 API 支持在更高层次上的功能调用。用户可以从这样的上层粗粒度 API 开始很快着手开发。当用户需要用到底层细粒度 API 的时候,可以降低学习曲线。或者更好的情况是,将二者混合使用,这样既简单效果又好。从用户的角度来讲,这是非常不错的。

还有,不同模块的稳定性级别和生命周期有所不同。例如,上层包装模块可能很快弃用,而底层细节模块可能需要存在相当长时间,并保持其兼容性。在这种情况下,老的上层模块可以被简单地直接弃用,同时从头开始封装抽象出新的上层模块。底层细粒度 API 保持不变,为所有新旧上层模块提供支持,而上层抽象模块可以被快速创建出来。例如,像 Servlet 这样的底层细粒度 API 一直存在,而各种 Web 框架则不断涌现和消失。

这种高层模块其实可以解决一个问题:如果没有这种高层模块,那么 API 的用户都只能直接调用底层模块的 API,那么这些模块就会很快地扩散到各个程序、产品中,怎么看都不是一件好事。

这种将功能细节抽象封装成新的上层 API 的方法,在 API 拆分和应对老的 API 终结过程中是非常有用的。它以桥接的方式将对原有老方法的调用委派给新的更细粒度的 API 方法。通过这种类似于外科手术的调整,既可以保持现有类库的重要功能,同时又使得类库简洁和高效。

通过恰当的 API 改进规划与合理的 API 消亡过程管理,我们是可以做到在如下方面很好的平衡的:API 的向后兼容性,正确性、维护既有投资、其他改进;以及为实现 API 的简单、易懂、无绪(尽可能的对用户透明)、优雅、漂亮所涉及的技术方方面面。我们已经接近本次 API 设计之旅的结尾,当你真正懂得 API 的世界,通晓 API 设计的要求,其演进的规律,你会从中寻找到那些所有古老而完美的科学的创立者所一直追求的东西——美、真理、优雅。所以,设计良好的 API 世界还是值得向往的!

是时候对过去 10 年的开发做一个总结了。写到这里，我在前面的章节已经给出了很多建议和技巧，它们都来自于我设计和开发 NetBeans API 的经验。当然，这并不意味着 API 设计这个话题已经说清说透，没有什么需要再研究了。这本书也并不是是一本权威指南，我其实还可以继续就该话题来花个一年半载写下去。不过对于读者来说，还是应该早一点看到这本书。我和很多人谈过此书，他们都认为开发 API 的程序员需要阅读这样一本书，所以我就先写到这里。但在本书结束之前，我还想谈谈 API 设计和软件工程的未来。

本书第一部分是 API 设计作为一门科学而不是艺术来介绍的。API 设计作为科学有其牢固的理性逻辑基础，而艺术不是，尽管 API 的设计有时候看起来像艺术。预先明确相关术语和前置条件，才能客观评估一个 API 设计的好坏。这些规则尽量做到语言中立，这样就不仅适用于 Java，就还适用于其他编程语言了。本书给出的理论性内容也并不十分完善，相信存在其他的 API 设计原则，或者还有些原则有待我们去发现。尽管如此，我们也不必担心，因为本书的第一章已经给出了一个评判各种设计原则效果好坏的工具，判断特定的建议是否有助于我们设计出质量更好的通用类库和 API。它给出了一个极其重要的基本原则（元原则，meta-principle）：针对性无绪（selective cluelessness）。无绪是一柄利器，可以帮助我们衡量不同的建议或者规则是否有用。因为我们知道，在需要更少了解细节的前提下完成更多任务，并更加容易地构建出更好的软件系统，如果一个建议可以允许我们做到这一点，那么可以说它是一个好的建议。我们确实需要这样的好建议，尤其是在未来的时间里，因为那时软件系统的复杂性可能会给任何设计者带来极大的挑战。

本书的重头戏是第二部分的内容，主要介绍 API 设计理论在 Java 语言上的体现。有些经验可以广泛地用在各种编程语言上，还有一些则只适用于面向对象语言。有的似乎无关紧要，有的则很复杂，还有一部分则是有争议的。但我想再强调一次，这些经验并不全面，只是给总结 Java 的 API 设计模式起了抛砖引玉的作用。我们已经看到用 Java 语言设计 API 的各种场景，我们还有必要从演进的角度和 API 用户的立场理解各种语言结构的深意。

本书的第三部分则针对设计和维护 API 时的常见情况，来提供了各种技巧来解决相应的问题。其中最重要的内容就是关于“改进”的。随着工具的改进，人们更好地将其应用于实践。在任何情况下，未来的目标都是很清楚的：软件集成和 API 设计，其中软件集成是核心所在，它的

处理应该越来越简单，公众也更加容易使用。现在来看一些这方面的细节性内容，了解是如何来做到这一点的。

20.1 原则性内容

牛顿于 1687 年出版了《自然哲学之数学原理》一书，这个时间点可谓是科学史上一个非常重要的时刻。该书分为三大部分，被认为是一本划时代的科学巨著，它包含了非常重要的科学原则和结果，比如首次对力作出定义，力与力之间的相互作用，以及对物体运动法则的定义，还提供了微分学这样一种新的数学工具来对书中的内容加以支持。对于个人来说，这是一个无与伦比的成就。

但这本书最重要的地方不是它所包含的内容，而是它如何来对待那些未能包含在书中的内容。牛顿很清楚，尽管他为很多之前未被解释的事物提供了解释，但他所创建的力学世界并没有完全地反映全部现实世界。今天我们已经知道他所说的非常正确。因为他从一开始就考虑到了这一点，所以书中的内容至今仍然很有用，而且非常重要。对于众多规定条件下的应用，它给出了很好的结果，更重要的是，它还可以用作一个好的教学工具。对于初学者来说，这一理论非常容易学习和理解，是引导大家走向现代物理学的完美开端。

牛顿并不是这一理论的首创者。在他之前，就已经有很多智者也做了同样的事情，最为有名的当属笛卡儿。但他们的谦逊程度各有不同。笛卡儿认为他自己一个人可以就整个世界给出全面的解释，并在作品中努力做到这一点。而牛顿则坦然接受他的理论并不是最佳的。这样，不仅给后来者留出了空间，而且他自己可以只将精力集中在他所认识到的现象上，对其进行解释。笛卡儿则需要解释世界上的万事万物。显然，这是一个艰巨无比的任务，使得他往往未能深入具体领域的研究，甚至有时他给出的结论本身并不成立。例如他在解释运行物体的相互作用时。但据我所知，《自然哲学之数学原理》一书中的内容，至今还没有发现什么错误。

此外，《自然哲学之数学原理》一书将整个世界的复杂性留给他人来诠释，因此，它极大地促进了欧洲科学家在这方面的探索。他们采用并扩展了牛顿的理论和技術，而这些广阔的探索都已经远远超过牛顿当时的想象。各门学科间需要进行相互合作，而物理学就是合作的关键所在，它成为工业革命的推进力，促成了整个欧洲在 18 世纪和 19 世纪的发展。

《自然哲学之数学原理》^①一书的用处并不仅仅在于为每个人都提供了一项工具来描述或者说近似地描述真实世界的行为，它还提供了严格和稳定的理论来支持这一切，即使人们对现今世界的认知已经与书中不完全一致了，但书中的内容仍然有用。随时可以对其前置条件进行检验。只要理论的前置条件都满足了，那么就能把结论应用于真实物体，可以合理地用于现实世界。这种契合对我们身边的大多数情况都是成立的。因为我们生活的现实世界大部分既不依赖于高速运动，也不依赖于微观粒子，所以对于我们的日常生活来说，牛顿的理论就是普遍真理了。一本写于 300 多年前的书就能做到这一点，其成就可谓无与伦比。

在读完本书以后，读者也许可以看出，我十分尊重牛顿所做的工作，特别是牛顿做事的风格，尤其是他个人的谦卑。我想遵循与他一样的做事方式，使用与牛顿描述力学世界同样的风格来描

^① 该书英文名为 *Philosophiae Naturalis Principia Mathematica*。——译者注

述如何设计 API。我想建立一个论坛,使他人可以畅所欲言,表达自己的想法,而我也想与其中的一些朋友共同讨论和探索 API 设计的内容。然而,我一向认为,我的知识和意见都是有限的。还有许许多多工作等待着我们来完成,这本书只是迈出了第一步。事实上,在我的灵魂深处,我希望这本书能够像《自然哲学之数学原理》一样,成为 API 设计方面的杰作。但同时,我也深知:本书是绝对做不到的,因为本书只是收集了过去的十年我在 NetBeans 的实验室的笔记,只能算是一本对开发历程的记录,它与牛顿写作的《自然哲学之数学原理》一书相去甚远。那本书是多年研究之精华,集众多前人著作之精髓,且是牛顿和他人信件往来、思想碰撞的最终产物。尽管对牛顿的著作只能持高山仰止之态度,但我仍然决定采取类似的结构。本书的第一部分试图为 API 设计提供长久的架构示图,我尽可能客观地来建立这样的架构示图,不与任何编程语言具体绑定,因此,该章讲述的内容就可以普遍适用于各类 API 设计。就像牛顿把欧几里得空间的几何学结合上了力量的抽象,使其可以应用到现实世界中,本书第一部分中的理论也可以应用到任何编程语言上,不管是已经存在的编程语言,还是未来可能出现的编程语言。

在第一部分架构示图的基础上,我针对如何用 Java 正确设计 API 写下了很多建议。这些建议的有效期就没有理论那么久了。一旦有一天 Java 语言不复存在,或者说有了根本上的改变,那么这些建议就没有什么用处了。然而,在写作本书时,这些内容还是非常重要的。很多正在用 Java 编写自己类库和 API 的开发人员都会从这些建议中获益良多。此外,我并不是凭空来提出这些理论的,我从自己的 Java 开发经验中提炼出这些内容,并把这些经验都放在本书的第二部分中加以说明。读者可以把这些内容看作是通向理论的实践之路,也可直接把它们看作是理论的实践产出物。当然,最好是将这两者结合在一起来看。但本书不可能像《自然哲学之数学原理》一书那样严谨。本书没有给出真正的证明,虽然我试图做了一些,但也只能说是草图,谈不上是真正的证明。

无论如何,我还是为后来者留下了继续前进的路标。读者可以将书中给出的设计技巧应用到自己的项目中,扩展这些技巧,最终发表出来。本书只是学习 API 设计的起点,读者可以证明我的某些观点不正确,然后在此基础上提出新的建议。由于时间的原因,也受限于我个人的能力以及耐心,书中的内容不可能涵盖 API 设计的方方面面。我希望读者能对这些内容进行深入探究,并提供更多更好的建议。下面列出了一些我认为值得进一步研究的课题。

20.2 无绪长存

预测未来总是很难的,而且每个预测都会伴随着一些风险和不确定性。如果说我一定要对未来会发生什么事情加以预测的话,那么,我的首选必然是无绪。我想说,无绪注定要在软件工程的世界流行起来。在本书的第 1 章,我已经给出了一些理由,现在再补充一些理由,来告诉大家为什么在未来的软件工程中,无绪会有更好的发展。

对于程序员的大量需求,可能会一直甚至是永远持续下去。开发者使用的编程语言和工具会有所改变。但只要这个社会还需要信息,那么就需要有人来对这些信息进行整理、归纳。新的开发人员并不比我们现在的开发人员更聪明、教育程度更高,但他们所要面对的系统却比我们今天面对的系统大得多,得出的成果会更先进,而具体的表现形式也将更加多姿多彩。简而言之,社会期望他们要比我们做出更多的成绩。与此同时,他们的能力却并不比我们来得更高,而是仍然

处于同一水平线上，很明显，未来必然是这样的：对于一个系统来说，每一个开发人员所能理解的内容肯定只是其中的一小部分，而且占系统的比例会越来越小。对于无绪来说，这正是它用武之地，特别是我们希望一个大的系统能够稳定。即使我们能够理解的内容只是庞大系统中的一部分，我们仍然希望这个系统能够正常工作，在我们的掌控之下。针对性无绪对这个问题给出了一个答案：找出这些系统中的关键所在，通过检查和校验来确保这些关键点可以正常运转，整个系统就 OK 了。所以说“无绪长存”。

之所以说“无绪长存”，还有另外一个原因，就是以组装的方式来将大模块构建成应用系统是非常方便的。我在很多 Linux 论坛上看到了这一点。这些论坛上大部分的讨论内容是如何安装一些软件包和调整配置文件，怎样用一些 Shell 脚本实现一些非常强大的功能，比如说在系统休眠以后重新启动一个 X Window^① 界面之类。这类信息的数量非常多，相比之下，那种如何为一个 C 程序的源代码提供补丁，如何对其调试，这类问题就少得可怜了。这再次以铁一般的事实证明了无绪存在的价值。大多数 Linux 用户并不清楚他们所用的程序到底是怎么开发出来的，他们只知道程序的命令行选项以及程序执行时的输出内容。执行一个二进制的程序，然后使用管道技术和文本查找工具 grep 就可以找到各种重要的信息，然后根据这些信息就可以让整个系统正常运行了。这充分地证明了无绪的卓越之处。而且这也展示了 API 的重要性。命令行选项和这些输出的文本就是最重要的 API，所有高级 Unix 用户都会用到这些信息，这些 API 完全将一个模块的功能与其内部的实现分离清楚。经验告诉我：当我需要在电脑上解决一个问题时，我会基于现有的二进制类库来找一个解决方案。我甚至会花一天时间来写 shell 脚本，或者调整配置来找到一个解决方案，而不是说直接打开一个调试器，打开源代码，从中找到相应的 Bug。这两种方式差别很大，想跨越两者是一件非常艰难的事情，所以大部分情况下，我们宁愿使用脚本和命令行参数来解决问题。比如说，我已经尝试过很多次去解决 XServer 上的问题，但在过去这五年里，只有一个问题才需要我自己下载源代码来修正段错误，其他的问题只需要使用脚本和命令行参数就可以解决了。因为很容易就可以将很多模块的功能集成在一起来构建 Unix 应用程序。相比之下，普通用户的数量比要比深入 C 源代码的人多得多。这两者之间的比例之大，已经充分地说明了对于 API 来说，简单、一致、易用是非常重要的。我说的这一点，并不仅仅限于 Unix，而是通用的。API 越优秀，那么我们就可以在完全不了解该 API 的基础上来构建庞大的系统。所以说，无绪长存。

未来的任务是要为针对性无绪找出更多的使用场景以及正确的使用方法。本书中已经就优秀 API 及自动测试的重要性展开过深入的探讨，但还有很多其他的工具和做法，可以帮助我们在目前拥有的知识基础上设计出更可靠的系统，甚至对我们拥有的知识量要求得更少。最终，当设计一个系统的时候，我完全不需要去了解其细节性的内容。我的大脑不是用来记这些东西的，我更喜欢用大脑去记录朋友、爱好、家庭和生活。当然，我仍然要让整个系统可以正常运行，比如说

① X Window 系统（也常称为 X11 或 X）是一种以位图方式显示的软软件视窗系统。最初是 1984 年麻省理工学院的研究，之后变成 UNIX、类 UNIX，以及 OpenVMS 等操作系统所一致适用的标准化软件工具套件及显示架构的运作协定。X Window 系统透过软体工具及架构协定来建立作业系统所用的图形用户界面，此后则逐渐扩展适用到形形色色的其他作业系统上。现在几乎所有的作业系统都能支援与使用 X。更重要的是，今日知名的桌面环境 GNOME 和 KDE 也都是以 X Window 系统为基础建构成的。——译者注

有时候还需要修正系统的 Bug。我会不时地回头检查一下系统，然后对其进行调整。这时候，我才需要去回想一下与此有关的一些重要知识，应用这些知识，完事以后，就可以马上抛在脑后了。无绪是我的朋友。如果有谁能够找到任何能够增强无绪的方法，那么他会成为我心目中的英雄，而且也不仅是我的英雄，更是大家的英雄。无绪是我们所有人的朋友。

20.3 API 设计方法论

我的朋友 Tim Boudreau 在认真考虑如何帮我写好这本书的时候，总是这样说：“我们需要建立一套方法论。”

听了他的建议以后，我惊讶得下巴都掉了下来，并表示反对：“我们现在没有一套方法论，只有一大堆建议。建议不是一个检查表，可以让别人逐项对项目进行检查。”

“没关系，” Tim 回答说，“总还是要一张检查表的吧，这样就可以形成方法论了。没有检查表，那么所有的建议都会流于表面，没有人会真的听你的！”

对于他的观点，我觉得不爽，但总得做点事吧，因为我觉得脑子里还是能给出这样的东西。现在，既然已经谈到了 API 设计的未来，那么就就这个话题讨论一下吧。

这本书已经给出了很多建议，大部分都是用笔记的形式给出的。这些建议都来自我设计 Java 框架中的经验，所以它们更像是这么一个检查表，告诉开发人员哪些事情不应该做，这些事情会对你维护 API 的下一个版本带来一些不好的影响。在我刚开始做架构师的时候，我对正确开发软件的方法论几乎一无所知，所做的设计大多出于本能。随着时间的推移，开始出现各种错误，我为此很受伤，所以开始努力学习并总结出了一些方法论。

我仍然不愿意给出一个方法论。我宁愿让他人来做这件事，但我并不介意在这里为其做一个预测。本书第一部分已经指出：任何方法论都应该有一个理论背景。然而，就像牛顿定律中并不会规定如何才能设计一座不会坍塌的桥梁，理论的目的也不是给大家提供一本应用手册，因为这根本不是理论的目标。本书第二部分主要是讲述如何将理论的部分具体应用到某一特定的编程语言上，日常建议部分侧重于工具和流程，对方法论也没有涉及。不过，我想本书最后这一部分可以是一个很好的切入点，从这里来探索 API 设计的一个方法论。

开始找方法论之前，我们首先需要给它起一个适当的名字。世间的万事万物，不管是什么，名字都是最重要的。名正则言顺，如果我们给它起了一个很好的名称，相信大家听到后会产生共鸣，从而端正态度，会愉快地接受。因此，让我们来看一下有哪些好名字可供选择。这个名字应该要包括一个“高”。“高”是一个好词，怎么说也比“低”好吧。也许还应该给它加一个形容词，如“高效的”。听起来很不错吧，要知道软件开发和维护的费用总是太高，所以让我们用一个非常有效的方法论来降低这些费用。此外，还可再加一个“结构”之类的名词。“结构”这个词也不错，听起来就比“混乱”好得多。还可以考虑加个“合理”。如果觉得方法论可以让大家得到更好的软件，那么“强化”也是一种很不错的名字。还有“一致性”，软件的一致性也是非常重要的。我不喜欢无序的软件，所以“统一”这个词也应该考虑加进来。还有，“团结”是个好词，所有的开发人员所需要团结一致！现在，我想我们已经找到一堆好词，现在可以开始讨论方法论中应该带有什么内容。其实，我只是和大家开了一个玩笑，让我们现在开始真正地讨论方法论的内容。

理论及应用，还有本书给出的各种建议，都有一个假设的前提，那就是：事物是永远不会十全十美的，事物是会演化的。在一个完美、静态的世界里，我们可以利用各种理论来探讨所有可能的解决方案，然后选择最好的一个。这可能需要很长时间，但在某个时间点我们可能会获得最优的结果。这种处理方式，就是那种几何学中研究静态对象的方法论。然而，我们现实的世界并不是一成不变的，它一直在变化。如果说是探索多年后才能得到一个完美的解决方案，那时这样一个完美的解决方案很可能不再适用于我们变化后的世界了。当然，我们承认并不是在寻求一种完美的解决方案，也就表示我们并不是把优雅、漂亮放在第一位。这话不好听，但是很实在，并可能帮助我们简化寻求真理的过程。

软件工程始终在理性主义和经验主义两者之间摇摆不定。有时它更接近数学，努力寻找完美的解决方案。有时它更倾向于经验主义，接受能解决问题的方案，即使问题的解决并不完全满足我们的需求，也仍然是可以接受的。第一种思想要求有完美的方法论，提倡完美的计划和完美的文档，要在开始编码之前把一切都想清楚。有了良好的规划阶段，那么编码就只是一件小事了。如果说这个世界是静态的，或者说我们已经把这个世界认识得清清楚楚，那么这种方式是一个完美的工具。然而，我们所处的世界和我们的知识一直在发生变化。这就是为什么还有另外一套方法：极具创新性的极限编程（XP 方法论）。这些方法论在本质上就是承认：我们的知识永远都不会是完美的，在这种不利因素的影响下，如何才能尽可能地成功地开发软件。我认为，在过去十年里，这一思潮逐渐为众人所接受，并对软件开发有着强有力的影响。

在思考 API 设计的各种办法时，我也发现，其实这些方法论也是在理性主义和经验主义之间摇摆不定。我对理性主义和经验主义这两者都十分尊重。如果说真理同时也是漂亮的，我自己会非常喜欢它，因为这让我感觉它更优雅。但是，当你读完这本书后，你会发现漂亮不是真理的一个必要条件。即使我们没有对整个世界都有着清楚、全面的认识，只利用在现实世界的部分知识，也同样可以设计出优秀的 API，极限编程就是这样做的。现在，我觉得我已经为未来的方法论找到了一个好的名字。在极限编程中，有一个软件开发的方法论称为敏捷。我们也将我们的方法论称为敏捷 API 设计。

敏捷 API 设计是一个相当精巧的名字，全是响亮的词。它会提醒听众，这不是一个空泛的名字，它的背后是有很多内容作为支撑的。我发誓，在写到本书这一部分之前，我根本没有想到这个名字。但当我回头看书前面章节给出的各种建议时，我觉得这个名字真是太完美了。各章节中一再重复的那些话“第一个版本远非完美”、“了解你的用户”、“为日后的改进预作准备”……这些内容其实都与敏捷软件开发给出的建议不谋而合。说到这里，我仍然不愿意提供具体的方法论，但如果你在开发自己的类库或者框架时，采纳了本书的建议，那么请告诉大家，你遵守了敏捷 API 设计的原则。

20.4 编程语言的演变

本书中给出的建议原本只适用于 Java 语言，当然可以考虑将这些建议扩展到各种语言上，使 API 设计从原来的一种负担变成现在的一种享受。然而，这样做就像创建一个新的编程语言。这种语言可能会更容易使用和编程，但很难被软件业接受。这是因为在开发软件项目选择具体编

程语言进行编码时，软件业通常是保守的。这其实很正常，任何一种新的编程语言都要求开发人员掌握新的技能，需要时间对他们进行培训。这就是我选择 Java 语言来讲述 API 设计的原因，我想告诉大家即使只使用普通的 Java 语言，软件开发人员仍然可以实践敏捷 API 设计。然而，要想让一个 API 在后续改进时不出现问题，有时还是需要对编译器和虚拟机有一些深入的了解，以便利用正确的技巧把事情给做对。有时，这些技巧可能会极其复杂（如 5.5 节中的 Accessor 类，还有 10.4 节中的访问级别控制）。所以现在要来分析一下未来的编程语言或者软件构建系统（如 Maven^①）如何将技巧融为自然的编码结构。

这里，我有意使用了“未来”这个词，因为目前我还不知道有哪个语言或系统可以提供这种功能。现在能够解决 API 设计问题的方式大部分是由各个组织或者个人根据自己的实际情况来提供一些工具。但似乎没有任何其他通用的解决办法。

解决方案有一部分是取决于编译器，或者说，至少要在源代码中添加 Annotation，然后由一些 Annotation 的处理器随后对其进行分析处理。比如说，编译器在编译代码时，可以指定目标平台为 JDK 5，那么即使当前使用的 JDK 版本更高，但新版本引入的类和方法都不可用。这种情况应该可以用于所有的类库，而不仅仅是 `rt.jar`。只要我们能够知道某个特定版本中 API 的状态，那么这个解决方案还是比较容易实现的。比如说，我们在类库和 API 中对每一个类、方法及字段，都使用一个 `@Version(1.4)` 的 Annotation 来进行标示。如果编译器在编译代码的时候，留意一下这个信息，然后忽略所有比指定版本更新的类、方法和字段，这样在编译的时候，在版本使用不正确的时候，就会抛出一个“没有发现指定内容”的错误。当然，这个方案要求 API 中所有外部可用的元素都添加一个 `@Version` 的 Annotation。这也就是说包装和版本号都应该是编程语言的组成部分。这个改变之大，让很多编译器的开发人员望而却步，我和几个人谈过，没有人考虑过这样做。我认识一些编程语言的设计者，他们考虑过模块化。但是他们所想的模块其实是 20 世纪 70 年代所说的那种模块。这些模块只是解决了在编译时的分别编译处理，可以一个模块一个模块地对其进行编译处理，但没有说明，如果有一个模块有所调整，那么会发生什么事情。这只是静态形式的模块化，不是独立各块的动态汇编。这种模块化思想是三十年前的，但现今的世界已经改变了很多，需要越来越敏捷，所以我们也应该对编程语言加以改进以适应时代的要求了。

有更好的编译器来支持这些功能，肯定是一件好事，但编译器并不能解决所有的问题。系统会变得越来越复杂，它必须要清楚地了解自己的发展史。比如说，如果先前的版本已经提供了一个方法或者类，那么要移除这个方法或者类的话，就应该抛出一个错误。如果想做到这一点，那么就必须对先前的重要版本都做一个快照，这样才能对破坏二进制兼容性的行为进行检查。想做到这一点，还必须要清楚地知道版本号。同样还要提供一个策略让开发人员明确地声明：新版本和旧版本并不完全兼容。在有了这样的声明以后，编译器就可以忽略这种检查了。在许多情况下，上述的各种功能都有相应的工具来支持。但这些工具并不具有通用性，还需要人工来配置和干预

^① Maven 是一个项目管理工具，它包含了一个项目对象模型、一组标准集合、一个项目生命周期、一个依赖管理系统和用来运行定义在生命周期阶段中插件目标的逻辑。当你使用 Maven 的时候，你用一个明确定义的项目对象模型来描述你的项目，然后 Maven 可以应用横切的逻辑，这些逻辑来自一组共享的（或者自定义的）插件。它的官方网站是 <http://maven.apache.org/>。——译者注

才能正常运行。未来的编程语言和软件系统，应该在程序员准备开发一个新类库之前，就能做到前面所说的这些。

然而，另一件要考虑的事情就是关于访问级别控制，像这种 `public`、`protected` 和 `final` 的访问修饰符是否已经不合时宜了。在 10.4 节中，我认为的确存在这种可能性。为了设计一个 API，我必须要想办法知道某一个类或者接口是否允许继承，还要对继承者加一些限制。然后在设计每一个方法的时候，你都要考虑这个方法是让用户来调用以完成某些功能呢，还是说用户需要覆盖这个方法扩展功能呢。每个方法都不应该具有二义性。一旦出现了二义性，那么想清楚说明第二种含义就是一件非常困难的事情。比如说用 `public final` 来声明一个方法，显然这个方法是留给用户调用的，这样做要比只用一个 `public` 来得简单，因为 `public` 具有二义性。我并非主张取消访问级别控制。但是，我希望访问级别的设计能够更好地帮助开发人员设计 API。如果说访问级别控制的目的是为了表达设计人员对一个 API 元素访问级别的限制，那么臭名昭著和危险的“复用事件”在目前常见的主流面向对象语言中就会一直存在下去，所以说，目前普遍使用的访问级别控制的目标是不正确的，不应该再沿用下去。我虽然这样说了，但并不想对访问级别控制到底应该设计成什么样子给出一个定义，但我知道，在目前的情况下，设计 API 时，我们迫切需要更多的无绪：不需要花费太多的时间去考虑 API 中的某项内容到底应该用什么样的访问控制，而且也不会对 API 设计者的意图产生曲解。

尽管我在本节书中谈到编译器的时候，显得很没有礼貌，但这并不意味着我是在胡说八道。额外说一句，如果说已经有了一种系统和编程语言适合应用敏捷 API 设计，证明了我的观点是错误的，那么我反而会非常开心。如果不是，我想让编程语言的设计者从这个新的敏捷角度来思考一下他们的解决方案。在此期间，我也很想看一下是否可以从编程语言的外部做些什么工作。就像我们使用 NetBeans Runtime Container 一样，读者可以使用自己的运行平台和构建系统。

20.5 教育的作用

我们生活的四周到处都体现着“无绪”。但我们是否已经准备好接受这些了呢？我们是否教会别人这些内容呢？我们是否告诉过其他的开发人员如何从世界上数以万计的类库中找到合适的类库来构建一个庞大的系统呢？我想这些问题的答案都是否定的，我希望在不久的将来，事情会有所改观。

我不时地会去拜访各大学，并给当地学生做一些讲座。我认为，学校还是在教授基本的编码技术，而不是倡导和教授如何做到代码重用。不出所料，大学的教学方法似乎更倾向于理性主义，学习的内容充斥着美丽和优雅。当然，如果说程序员只是要写一个快速排序或者与图有关的算法，这种教学可以算是完美的。然而，学生应该了解的内容远不止这样。

教 滑 雪

一次偶然的机会，我参加了滑雪教练员培训班。只不过我没有时间去参加认证考试，所以也就没有资质来教新手如何滑雪。我的职业是一个软件设计师，并以此为生。但我对滑雪的了解非常多，我完全可以当个滑雪教练，并以此为生。

学习滑雪总是从基础开始,以确保学习者能够不从自己的滑雪板上掉下去,而且至少还要学习转弯的技巧。经过几个小时的练习后,教练通常会将学生分为两类:一类称为“生存者”,还有一类则可以称为“比赛者”。不管分到哪一类,这些人都可以从任何一座山的顶部滑到底部。但只有后一类人可以在滑雪时有风驰电掣的感觉,这也是人们喜欢骑摩托车、滑雪和滑雪板的最主要原因之一。

我相信,对于程序员来说,也应该采用类似的教学风格。我们需要每个人都了解一些基础知识,在我看来,这些基础知识应该包括针对性无绪的原则。只有这样,我们才能保证无论程序员多么优秀,他们构建的系统都是可靠的。在此处,也应该有一个岔路口。一个方向是更加实用,掌握更多的实用技术,如重用第三方组件和整合遗留代码。另外一个方向则更加“学术”一些,更好地应用无绪来提供更多的通用类库等。

现在准备结束这个滑雪的话题了,但有一个虽不太相关却很有意思的点:好的工具会引发非常好的效果。在我小时候,想试一下“比赛者”的滑雪方式是很难的,因为滑雪板不适合转弯。我首次在1996年试用单板滑雪的时候,几乎立刻迷上它,好几年都不用滑雪板了。然而,滑雪板的生产商立刻就吸引了我,原来,这些年,卡宾板已经使转弯变得非常容易了。因为工具的改善,所以出现了大量的滑雪“比赛者”。近年来,滑雪时能够拥有风驰电掣感觉的人已经远远高于10年前了。良好的工具,有助于大家更容易地感受高标准。这也就是我们需要有优秀的API设计工具的原因。

我们必须对两类人都进行教育。我们需要人来做“科学”类型的内容:发现新的原则和算法,并找到方法论、规则,以及无绪开发中的重点内容。然而,我们也迫切需要另一类的开发人员,他们能够在无绪的前提下很好地完成工作,而且工作成果非常可靠。然而,这一切需要经过学习的过程。否则,一旦学生离开大学后,他们就会发现,软件工程师的工作与他们原来的期望区别甚大。并不是每个人都能完成学业,然后十年来一直在创建和维护自己的框架。大多数学生开始工作的时候,他们要做的任务就是维护他人编写的代码。而这些内容大学里根本就没有教过。

害怕他人编写的代码

现在的学生们害怕那些不是他们自己写的代码,虽然我不想这么说,但这一点却是事实。去年我在奥地利林茨的约翰开普勒大学开了一门与NetBeans平台相关的课程。课程的一部分是要学生完成一个小项目。我给学生提供了3种选择:①在NetBeans平台的基础上创建一个新的模块,而且该模块能提供一些功能;②或者找到一个现有的模块,发现该模块缺失的功能,对程序进行修改以添加新的功能;③亦或修复三个Bug。在我看来,这三项任务中最简单的肯定就算是最后一个了。NetBeans有几千个未修复的bug,其中有很多Bug是很容易解决的,只是很多Bug不太重要,开发人员不需要花时间来解决这些小的问题。要修复这些Bug,往往可能只是简单地修改一行代码就可以。第二简单的任务是强化现有模块。现有模块的代码可以作为一个范例。只需要在其中插入一些新的代码就可以了,大多数情况下不需要开发人员有很多这方面的知识,只要参照例子就够了。相比从头开始写一个模块来说,这两个任务都容易得多。但是,猜一下这些学生都选择了哪个任务?没有任何人选择第三项任务,只有一个人选择了第

二项任务，其余所有人都选择了第一项任务——自己从头写代码。简而言之，学生都害怕他人编写的代码。这不是一个好消息，因为一旦离开大学，他们大部分的工作时间都是在处理他人所写代码中遗留的 bug。

还有一件事，说起来有点吹毛求疵了，那就是在计算机科学教育中，衡量学生代码质量的方式有问题。记得当年我入学的时候，我们需要写一个程序给老师看，然后教师来决定是否可以通过考试。然而，那个时候，能访问互联网的人很少，而且开源运动远不如现在这样普及。这就是为什么我希望学校在这方面应该有一些改进，尤其是考虑到因特网的力量和现有的很多开源项目寻求开发人员的加入。但学校对这方面毫不关注，学生还是需要从头开始创建自己的项目，然后把结果给他们的教授过一下眼，课程就算结束了。这个新做的项目随后就会被遗忘。如果说能指导学生如何在现有代码的基础上进行开发可能会更有价值。比如说，他们可以评估一下，是否将代码集成到现有的一些开源项目。如果某个学生能够让自己代码进入到开源项目的基础库中，那么他就应该得到最高分。得到最高分的原因，不仅仅是其高超的编码技能，更是该学生与其他开源社区沟通和协作的能力。如果说某个学生只能将代码和项目集成起来正常运行，但质量不够，不能被开源项目所接受，那么可以给他一个平均分。对教师来说，这是件好事，因为对学生评价大部分将由开源社区成员来给出。只不过，我还没有发现哪个学校想这样做。

教育的作用是非常重要的，因为它将培养出很多新的工程师来接替我们开发软件。我们需要他们提前就准备好来维护现有的代码，能够在“针对性无绪”的方式下工作，还能利用大量现有组件来构建自己的解决方案。每个人都应该可以有意识地工作于“无绪”状态。他们也都应该能够评估构建庞大软件的系统是否可行，比如说要判断类库或者框架是否可以复用。另外，这些未来的工程师们还应该知道他们的“无绪”是有针对性的。例如，他们应该明白，没有哪个人会在一个项目中永远地干下去，而且项目中的知识是隐藏在自动验证工具和测试背后的。虽然要求工程师尽量做到“无绪”，但这些工程师还是应当有自己的知识体系。比如说，他们应该不怕深入分析 Linux 内核中的代码，也不怕调试 NetBeans 平台的源代码来解决问题。

20.6 共享

敏捷 API 设计的时代才刚刚开始。这本书也仅仅是 API 设计的一个开始。如何正确设计 API 的相关知识也会有进一步的发展，我希望通过写作和出版这本书为大家提供一个起点，激励大家开始学习 API 设计。此外，我也期望有更多人在此基础上学习、工作，进而分享他们的成果。正如多个时代中的众多物理学家对牛顿所做的工作进行丰富、改进及扩展，相信还有更多的人需要通过分享相互的工作成果，使得敏捷 API 设计成为软件开发的未来。交流是非常重要的，因为今天交流合作要比牛顿所在的时候要容易得多，我已经注册了一个域名，并建立了网站，有兴趣的读者可以在这个网站上讨论并更正本书的内容。网站地址为 <http://agile.apidesign.org>，我希望可以有更多的人加入到社区中来，一并分享你们的建议和想法。我已经为这个域名付了今后 3 年的费用，保证 3 年内可用，如果这本书能够得到大家的关注，我愿意将这个域名再延长 300 年。让大家尽情地享受无绪和 API 设计。

参 考 书 目

- Bloch, Joshua. *Effective Java*. Upper Saddle River, NJ: Prentice Hall, 2001.
- Dijkstra, Edsger. "On the fact that the Atlantic Ocean has two sides." <http://www.cs.utexas.edu/~EWD/transcriptions/EWD06xx/EWD611.html>, 1976.
- Dijkstra, Edsger. *Selected Writings on Computing: A Personal Perspective*. New York: Springer-Verlag, 1982.
- Drepper, Ulrich. "How to Write Shared Libraries." <http://people.redhat.com/drepper/dsohowto.pdf>, 2006.
- Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Upper Saddle River, NJ: Addison-Wesley, 1995.
- Hansen, Per Brinch. "Java's Insecure Parallelism." <http://brinch-hansen.net/papers/1999b.pdf>, 1999.
- Hunt, Andy, and Dave Thomas. *Pragmatic Unit Testing in Java with JUnit*. Raleigh, NC and Dallas, TX: Pragmatic Bookshelf, 2003.
- Orwell, George. *Nineteen Eighty-Four*. London: Secker & Warburg, 1949.
- Rooney, Garrett. "Preserving Backward Compatibility." <http://www.onlamp.com/pub/a/onlamp/2005/02/17/backwardscompatibility.html>, 2005.
- Torgersen, Mads. "The Expression Problem Revisited." In *ECOOP 2004 – Object Oriented Programming: 18th European Conference Oslo, Norway, June 14–18 2004, Proceedings*, edited by Martin Odersky, 123–146. Berlin: Springer-Verlag, 2004.
- Vopěnka, Petr. *Úhelný kámen evropské vzdělanosti a moci*. Prague: Práh, 1999.



Jaroslav Tulach

NetBeans的创始人，也是NetBeans项目最初的架构师。有着丰富的项目开发经验，一直致力于如何提高开发人员的设计技巧。

王磊

工学硕士，自由软件Aquarius ORM Studio的作者。2006年起任职于普元公司，任Studio产品线主架构师。2010年加盟淘宝，任商户平台高级技术专家。

朱兴

一直从事软件研发工作，熟悉Java、Eclipse插件开发及API设计等相关技术。

“十多年难得一遇的佳作，通过阅读本书，Java程序员必将取得长足的进步。”

——亚马逊读者评论

“这绝对是一本不容错过的好书，据我所知，市场上还不曾有哪本书在框架设计领域有如此深刻的阐述。”

——亚马逊读者评论

“这本书是作者对自己十多年NetBeans开发的一个总结。他将自己的心路历程如实记下，见证了NetBeans从IDE走向平台、从混乱的代码走向清晰的模块化架构这一不平凡的历程。……与其他设计类图书相比，本书告诉读者的是一种大道，而非小技。”

——本书译者

Practical API Design Confessions of a Java Framework Architect

软件框架设计的艺术

敏捷API设计的时代才刚刚开始，这本书堪称是API设计的一部开风气之先的著作。

与枯燥乏味的理论性图书不同，本书从理论、实战及日常应用三个方面详细讲解了软件开发和框架设计的艺术，着眼于保证软件设计能够应对时刻变化的需求和技术。书中将理论与实践有机地结合在一起，对框架设计领域进行了深层次的阐释。

作为NetBeans框架的主架构师，作者在书中总结了自己多年的开发经验，与大家分享了API设计的技术细节、走过的弯路和教训。对于广大软件开发人员来说，这些都是不可多得的宝贵财富。本书就像一盏黑暗中燃起的明灯，为你照亮崎岖的开发之路，指明前进的方向。

Apress®

图灵网站: www.turingbook.com 热线: (010)51095186

反馈/投稿/推荐信箱: contact@turingbook.com

有奖勘误: debug@turingbook.com

分类建议 计算机/程序设计/API

人民邮电出版社网址: www.ptpress.com.cn



ISBN 978-7-115-24849-7



9 787115 248497 >

ISBN 978-7-115-24849-7

定价: 75.00元