TURING 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 lulach 著
王磊 朱兴 译

人民邮电出版社
北京

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 NetBeansIDE 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自己是否被他人所观察，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。

——译者注

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 也具有一致性。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 signaure，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 虚拟器来说，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 Hashell 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。每当我们需要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

第三部分 日常生活

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