《代码大全 2》纪念版

[美] 史蒂夫·麦康奈尔（Steve McConnell)著

周靖 译（https://bookzhou.com）(注：仅翻译了部分内容)

中文试读版第 5、6、9 章，第 14～19 章，第 24～28 章，翻译原稿，仅供参考，

更多精彩内容，请购买正式版：京东 淘宝 | 美国 Amazon 台湾博客来

配套资源和试读下载： ys168 网盘>> 百度网盘>>

访问中文版博客，获取最新资讯

清华大学出版社

北 京

译者说明

《代码大全 2》是著名科技类作家史蒂夫·麦康奈尔的经典著作，是一本完整的软件构建手册，涵盖了软件构建过程中的所有细节。它从软件质量和编程思想等方面论述了软件构建的各个问题，并详细论述了紧跟时代潮流的新技术、高屋建瓴的观点、通用的概念，还含有丰富而典型的程序示例。本书所论述的技术不仅填补了初级与中高级编程技术之间的空白，而且也为程序员提供了一个有关编程技巧的信息来源。

这本书特别强调了在编写代码之前彻底规划和设计的重要性。这包括理解需求，创建详细的设计，并考虑潜在的问题，以尽量减少在实施过程中遇到的问题。具体来说，这意味着在开始编码之前，要完全掌握项目的目标，细化设计文档，并预见到可能发生的挑战，从而确保软件开发过程更加顺利，避免或减少后期可能出现的修改和调试工作。

经典书自然有经典的用法。刚开始可以粗读，对作者的思路有一个基本的了解。遇到自己感兴趣的点，可以精读。在觉得自己收获满满后（这并没有一个具体的度量），就可以把它放在一边。以后，一旦在实际的项目中遇到问题，或者在完成某个项目后需要做“事后回顾”，就可以重拾本书，找到自己知识的薄弱这处，趁着“新鲜劲儿”还没有过去，赶快看看自己是否有改进之处。如此反复，你也能成为一名“大师”。

2021 年，有幸受出版社之邀，充当救火队员，参与重译了本书约 1/4 的内容。所有原始译文将在这里一并放出。是的，这是一个“重译”项目。虽然原来的中译版也不错，但存在不少瑕疵。此次重译，一并解决了这些问题，并以译者注的形式增补了不少内容。

最后说一下书名，这并不是一本所谓的“大全”，里面并没有大量枯燥的代码。相反，所有代码“片断”都是为当前主题服务的。之所以叫“代码大全”，只能说是源自一个美丽的“错误”。不多说了，说多了都是泪。

PS：本人重译的章：第 5、6、9 章，第 14 ~ 19 章，第 24 ~ 28 章，覆盖全书四分之一的内容，约 18 万字。PDF 文档包含目录大纲（书签），请在你的 PDF 阅读器中开启书签后阅读。

第 5 章　软件构建设计

内容

- 5.1 设计中的挑战
- 5.2 关键设计概念
- 5.3 设计构建基块：启发式方法
- 5.4 设计实践
- 5.5 点评各种流行的方法论

相关章节

- 软件架构：第 3.5 节
- 可以工作的类：第 6 章
- 高质量子程序的特征：第 7 章
- 防御式编程：第 8 章
- 重构：第 24 章
- 程序规模对构建的影响：第 27 章

一些人可能会说，设计并不是真正的构建活动，但在小型项目中，许多活动都被认为是构建，其中通常包括设计。在一些较大的项目中，正式架构可能只关注系统级的问题，许多设计工作则可能有意留给构建。在其他大型项目中，设计可能有意详细到足以使编码变成一种机械劳动，但设计很少有那么完整——程序员通常会正式或非正式地设计程序的一部分。

> 关联参考　要详细了解大项目和小项目所要求的不同正式程度，请参见第 27 章。

在小的、非正式的项目中，很多设计是在程序员坐在键盘前完成的。“设计”可能只是在详细编码之前用伪代码写一个类接口。可能是在编码前画一些类关系图。可能是问另一个程序员哪种设计模式更好。不管以什么方式设计，小项目和大项目一样，都能从精心的设计中受益，而将设计作为一项明确的活动来认识，可使你从中获得最大收益。

设计是一个庞大的话题，所以本章只考虑了它的几个方面。好的类或子程序设计的很大一部分由系统架构决定，所以要确保第 3.5 节讨论的架构先决条件已得到满足。更多设计工作是在单个类和子程序的层次上完成的，具体在第 6 章和第 7 章讲述。

如果你已熟悉了软件设计的主题，本章可以挑一些重点来看，包括第 5.1 节的设计挑战和第 5.3 节的关键启发式方法。

5.1　设计中的挑战

> 关联参考　关于启发式过程和确定性过程的差异，请参见第 2 章。

“软件设计”（software design）是指构思、发明或设计将计算机软件规范变成可工作的软件的一种方案。“设计”是将需求与编码和调试联系起来的活动。好的顶层设计提供了可以安全地包含多个低层设计的结构。好的设计在小项目中是有用的，在大项目中更是不可或缺。

设计也存在许多挑战，本节对它们进行了概括。

设计是一个棘手问题

> 期望软件设计人员以合理的、无错误的方式从需求说明中直接推导出设计，这非常不现实。从来没有一个系统是以这种方式开发的，以后也可能不会有。即使是教科书和论文中显示的小程序开发也是不现实的。作者其实一直在修改和完善这些程序，你以为他们是直接拿出一个设计，实则不然。——David Parnas 和 Paul Clements

Horst Rittel 和 Melvin Webber 将“棘手”（wicked）问题定义为只有通过解决它或解决它的一部分才能明确定义的问题（1973）。这个悖论意味着，本质上你必须“解决”一次问题才能清晰地定义它，然后再解决一次才能创建有效的解决方案。几十年来，这个过程一直是软件开发的核心（Peters and Tripp 1976）。

在我所在的地区，这种棘手问题的一个戏剧性的例子是最初的塔科马海峡吊桥的设计。当时建桥的主要设计考虑是要有足够的强度来支持计划的负载。在塔科马海峡吊桥的情况下，风产生了意想不到的侧向谐波。1940 年的一个大风天，谐波不受控制地增长，直到大桥倒塌，如图 5-1 所示。

这是一个很好的棘手问题的例子，因为直到大桥倒塌，它的工程师都不知道空气动力学需要考虑到如此程度。只有通过建桥（解决问题），他们才能了解到问题中的额外考虑，从而使他们能够建造另一座至今仍然屹立不倒的桥梁。.

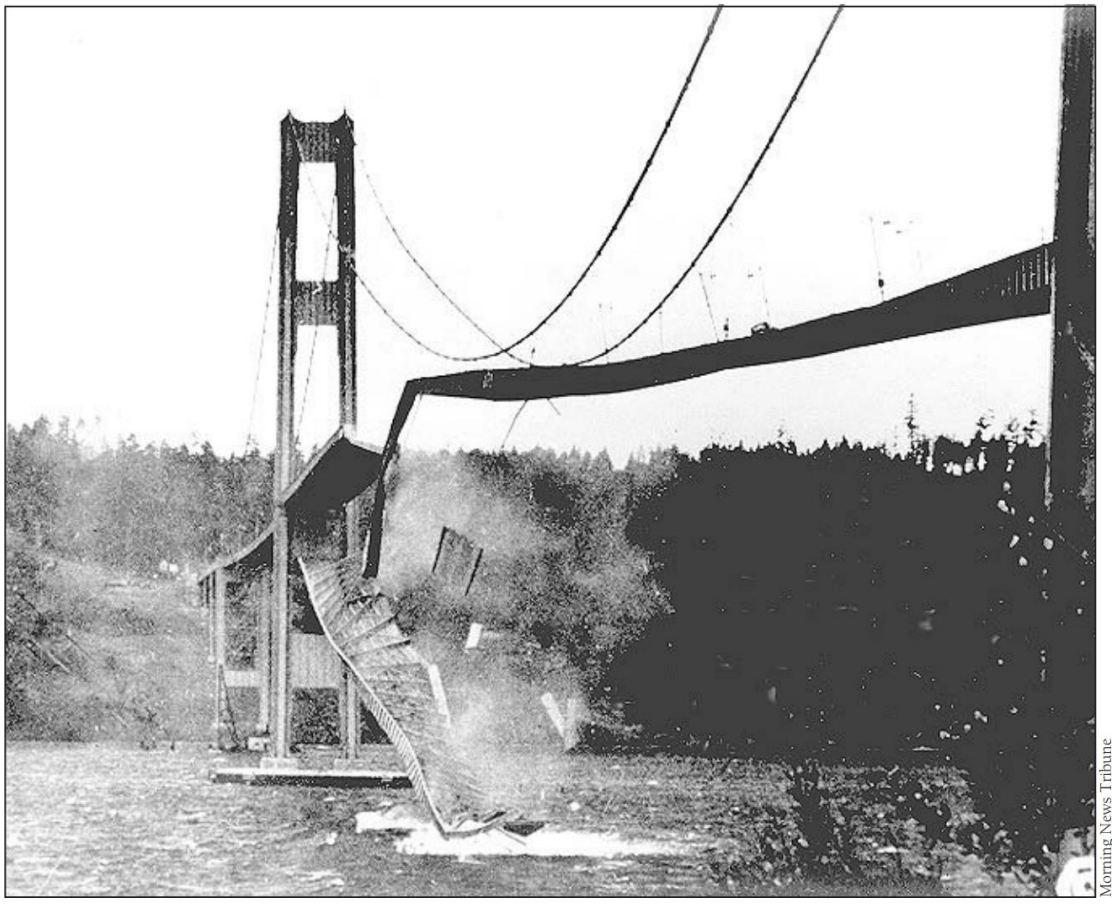


图 5-1　塔科马海峡吊桥——棘手问题的一个例子

你在学校开发的程序和你作为专家开发的程序的主要区别之一在于，学校程序解决的设计问题很少（甚至根本没有）是棘手的。学校的编程作业是为了让你从头到尾直线完成。如果老师布置了一个编程作业，然后在你刚完成设计时就修改了作业，在你准备交作业时又改了一次，那么你可能会想给他涂上柏油，粘上羽毛。但是，这一过程正是专业编程工作的日常。

设计没有章法（即使它产生了有条理的结果）

完成的软件设计应该看起来井井有条、清楚明了，但用于开发设计的过程却远不如最终结果这般有条理。

> 深入阅读　关于此观点更全面的探讨，请参见“A Rational Design Process: How and Why to Fake It”一文(Parnas and Clements 1986)。

设计之所以“没有章法”（sloppy），因为你会采取许多不恰当的步骤，会走许多死胡同——总之，会犯许多错。事实上，犯错是设计的重点——犯错并予以纠正设计，比犯同样的错误，在编码后才意识到它们，并不得不纠正都要写完的代码要便宜得多。设计没有章法，是因为一个好的解决方案与一个差的解决方案往往只有微妙的区别。

> 关联参考　对这个问题更好的回答请参见本章稍后 5.4 节中的“多少设计才够？”。

设计之所以没有章法，还因为很难知道自己的设计何时“足够好”。多少细节才够？有多少设计应该用正式的设计表示法（design notation）来完成，又有多少应该留到键盘上完成？什么时候才算完成？由于设计永无止境，对这一问题最常见的回答是“在你时间不足的时候”。

设计关乎取舍和优先级

在理想世界中，每个系统都能立即运行，消耗零存储空间，使用零网络带宽，永远没有任何错误，而且不用花费任何成本来构建。但在现实世界中，设计工作的一个关键部分是权衡相互竞争的设计特性，并从中取得平衡。如果快速响应速度比最小化开发时间更重要，设计师会选择一种设计。如果最小化开发时间更重要，好的设计师会精心打造另一个不同的设计。

设计涉及限制

设计的要点在于，它的一部分是在创造可能性，另一部分是在限制可能性。如果人们有无限的时间、资源和空间来建造实体结构，你会看到令人难以置信的、无边无际的建筑，有数百个房间，甚至每只鞋都有自己的房间。这就是软件在没有刻意施加限制的情况下的结果。由于建造建筑物的资源有限，人们才迫不得已简化解决方案，并最终改善解决方案。软件设计的目标亦是如此。

设计是不确定的

派三个人去设计同一个程序，他们很容易带着三种截然不同的设计回来，而且每种都完全能接受。做某些事情或许有不止一种方法，但设计计算机程序一般都是几十种方法起步。

设计是启发式过程

KEY POINT 由于设计是不确定的，所以设计技术往往是启发式（探索式）的，即需要用到“经验法则”或者“试试说不定能行”的策略，而不是使用保证能产生可预测结果的一种可重复的过程。设计涉及试验和错误。一个设计工具或技术在一项工作或工作的一个方面做得很好，但在下一个项目中就可能没那么有效了。没有一种工具是万能的。

设计是水到渠成的

总结上述设计特性，我们可以说设计是“水到渠成”（emergent）的一个过程。设计不会从人的大脑一出来便是完整形式。它们需通过设计评审、非正式讨论、写代码的过程以及修改代码的过程来发展和改进。

深入阅读　软件不是唯一会随时间而变的结构。物理结构也在演进，详情请参见《*How Buildings Learn*》一书（Brand 1995）。

几乎所有系统在最初的开发过程中都经历了某种程度的设计变化，在演进成后来的版本时，它们通常会发生更大的变化。变化在多大程度上有益或者可以接受，具体要取决于所构建软件的性质。

5.2 关键设计概念

好的设计取决于对少数几个关键概念的理解。本节讨论了复杂性在其中扮演的角色、理想的设计特征以及设计的层次。

软件的首要技术使命：管理复杂性

关联参考　第 34.1 节讨论了复杂性如何影响设计以外的编程问题。

为了理解管理复杂性的重要性，请参考 Fred Brooks 里程碑式的论文“No Silver Bullets: Essence and Accidents of Software Engineering”（没有银弹：软件工程的本质与偶然）（1987）。

偶然性和本质性困难

Brooks 认为，软件开发之所以困难，是因为存在两类不同的问题：本质的（essential）和偶然的（accidental）。Brooks 借鉴了源自亚里士多德的一个哲学传统来提出这两个术语。在哲学体系中，“本质的属性”是指一个事物为了成为该事物而必须具备的属性。汽车必须有发动机、车轮和车门才能成为汽车。缺少其中任何一个基本属性，就不成其为汽车。

“偶然的属性”是指一个事物碰巧拥有的属性，这些属性并不影响该事物是否能成为该事物。汽车可以装一台 V8 发动机、涡轮增压 4 缸发动机或其他类型的发动机。不管这个细节如何，它都是一辆汽车。汽车可以双门或四门；可以用普通轮毂或者镁合金轮毂。所有这些细节都是偶然的属性。也可以将偶然的属性看成是附带的、随意的、可选的和意外的。

关联参考　相较于后期开发，偶然性的困难在早期开发中更为突出。这方面的详情请参见第 4.3 节。

Brooks 观察到，软件中的主要偶然性困难在很久以前就已经解决了。例如，与笨拙的语言语法相关的偶然性困难在从汇编语言到第三代语言的演变过程中已基本消除，且其重要性从那时起逐渐下降。与非交互式计算机有关的偶然性困难在分时操作系统取代批处理模式系统时得到了解决。集成编程环境进一步消除了因工具配合不力而导致的编程工作低效率。

Brooks 认为，在软件剩下的本质性困难方面，进展必然会比较缓慢。原因是，就其本质而言，软件开发是为了解决一套高度错综复杂、环环相扣的概念的全部细节问题。之所以存在本质上的困难，是因为要和复杂、无序的现实世界对接；要准确、完整地识别依赖关系和例外情况；要设计出完全正确而不能只是大致正确的解决方案；等等。即使能发明出一种编程语言，使用与我们试图解决的现实世界问题相同的术语，编程仍然是困难的，因为在精确判断现实世界如何运作方面依然存在挑战。随着软件解决越来越大的现实世界问题，现实世界实体之间的相互作用变得越来越复杂，这反过来又增加了软件解决方案的本质性困难。

所有这些本质性困难的根源在于复杂性——无论偶然的还是本质的。

管理复杂性的重要性

> 有两种构建软件设计的方法：一种是使其简单到明显没有缺陷，另一种是使其复杂到没有明显的缺陷。——C. A. R. Hoare

软件项目调查在报告项目失败的原因时，它们很少将技术原因作为项目失败的主要原因。项目失败最常见的原因是需求不到位、计划不周或管理不善。但是，当项目确实主要是因为技术而失败时，原因往往是失控的复杂性。软件变得如此复杂，以至于没人真正知道它在做什么。一旦没人完全能够理解一个区域的代码变化对其他区域造成的影响，项目就再也发展不动了。

KEY POINT 管理复杂性是软件开发最重要的技术课题。在我看来，它是如此重要，以至于软件的首要技术使命（Software's Primary Technical Imperative）必须是管理复杂性。

复杂性并不是软件开发的一个新特征。计算先驱 Edsger Dijkstra 指出，计算是唯一一种一个人的思维必须跨越从一个比特到几百兆字节的距离，比例为 $1{:}10^9$，即九个数量级的职业（Dijkstra 1989）。这个巨大的比例非常惊人。Dijkstra 对此是这样说的："与这个语义层级数量相比，一般的数学理论几乎是平的。通过唤起对深层概念层次的需求，自动计算机使我们面临着一个全新的、史无先例的智力挑战。" 当然，和 1989 年相比，软件又变得复杂了许多，Dijkstra 的 $1{:}10^9$ 的比例在今天很容易变成 $1{:}10^{15}$。

> 陷入复杂性过载的一个症状是，你发现自己在顽固地运用一种明显不相干的方法（至少对任何旁观者来说是如此）。这就像一个不懂机械的人，他的车子坏了，于是他向电池里倒水，并把烟灰缸倒掉。——P.J. Plauger

Dijkstra 指出，没有谁的大脑真正大到足以容下一个现代计算机程序（Dijkstra 1972），这意味着作为软件开发人员，我们不应尝试将整个程序一次性塞进自己的大脑。相反，应该用一种安全的、一次只专注于其中一部分的方法来组织程序。目标是尽量减少任何时候都要考虑的程序的数量。可以把这看成是一种心智游戏（mental juggling，或心智杂耍）——程序要求你在空中同时保持的心智球越多，就越有可能丢掉其中一个球，从而导致设计或编码错误。

在软件架构层面，问题的复杂性通过将系统划分为子系统来降低。人类理解几小段简单的信息比理解一大段复杂的信息要容易得多。所有软件设计技术的目标都是将复杂问题分成简单的部分。子系统越是独立，就越能使其安全地专注于解决一个复杂的点。精心定义的对象将关注点分开，这样就可一次只专注于一件事。包（package）在更高的聚合层次上提供了类似的好处。

保持子程序简短，有助于减轻你的心智负担。从问题领域的角度来写程序，而不是从低层次的实现细节的角度来写，并在最高的抽象层次上工作，可以减少大脑的负担。

这里的要点在于，程序员只有认识到人类的天生不足，并对此予以弥补，写出来的代码对自己和其他人来说才更容易理解，错误也更少。

如何应对复杂性

高成本、低效率的设计有三个来源：

- 用复杂方案解决简单问题。
- 用简单的、不正确的方案解决复杂问题。
- 用不合适的、复杂的方案解决复杂问题。

正如 Dijkstra 所指出的，现代软件本质上是复杂的，无论如何努力，最终都会碰到现实世界问题本身所固有的某种程度的复杂性。所以，对复杂性的管理要双管齐下：

- 尽量减少任何人的大脑在任何时候都必须处理的*本质上的*复杂性的数量。
- 防止*偶然的*复杂性无谓地扩散。

一旦理解了软件其他所有技术目标对于复杂性的管理来说均为次要，许多设计上的考虑就会变得简单明了。

理想的设计特征

> 我处理问题时从不考虑美不美的问题。我只想着如何解决问题。但是，在我完成后，如果发现解决方案不漂亮，我就知道它是错的。——R. Buckminster Fuller

高质量设计有几个常规特征。如果能实现所有这些目标，设计确实会非常好。有的目标与其他目标相矛盾，但这就是设计的挑战——从相互竞争的目标中创造出一套好的折衷方案。设计质量的一些特征也是好程序的特征：可靠性、性能等等。其他则是设计的内部特征。

> 关联参考　这些特性与常规的软件质量特性有关。关于常规特性的细节，请参见第 20.1 节。

下面列出了内部设计特征：

最小化的复杂性　设计的首要目标是出于之前讲述的各种原因，将复杂性降至最低。避免做“聪明的”设计。聪明的设计通常难以理解。相反，要做“简单的”和“容易理解的”设计。如果设计不能让你在沉浸在一个特定的部分时安全地忽略程序的其他大多数部分，这个设计就不对。

易于维护　易于维护意味着要为负责维护的程序员而设计。不断想象维护程序员会对你所写的代码提出哪些问题。将维护程序员当作你的听众，然后把系统设计得不言而喻。

松散耦合　松散耦合是指在设计时，将程序不同部分之间的关联控制在最小范围内。通过遵循良好的类接口抽象、封装和信息隐藏等原则来设计相互关联尽可能少的类。最小的关联性能将集成、测试和维护时的工作量降至最低。

可扩展性　可扩展性意味着可在不破坏底层结构的前提下对系统进行增强。可在不影响其他部分的情况下改变系统的某一部分。而且最有可能的改变给系统带来的创伤最小。

可重用性　可重用性是指在设计系统时，可在其他系统中重用它的某些部分。

高扇入 高扇入（fan-in）是指有许多类在使用一个特定的类。高扇入意味着一个系统被设计成可以很好地利用系统中较低层次的实用类（utility classes，或工具类）。

低到中等扇出 低到中等的扇出（fan-out）意味着一个特定的类使用了低到中等数量的其他类。高扇出（超过约 7 个）表明一个类使用了其他大量类，因此可能过于复杂。研究人员发现，无论考虑在一个子程序内调用的子程序数量，还是在一个类内使用的类的数量，低扇出原则都是有益的（Card and Glass 1990; Basili, Briand, and Melo 1996）。

移植性 移植性是指设计的系统很容易转移到其他环境。

精简性 精简性（leanness）意味着设计系统，使其没有多余的部分（Wirth 1995, McConnell 1997）。伏尔泰说过，一本书什么时候才算完成？不是在增无可增的时候，而是在删无可删的时候。在软件中，这一点尤其正确，因为只要修改了其他代码，就必须开发、审查、测试和考虑额外的代码。软件的后续版本必须与额外的代码保持向后兼容。要回答一下严肃的问题："这很容易，所以把它放进去会造成什么伤害？"

层次性 层次性（stratification）是指尽量保持分层的层次，以便可以在任何一个层次上查看系统，并得到一致的见解。设计的系统应该在一个层次上就可以观察，而不必非要跑到其他层次。

关联参考 要更多地了解旧系统的使用，请参见 24.5 节。

例如，假设要写一个现代系统，但其中必须用到大量设计不良的旧代码，就可以在新系统中专门写一层来负责与旧代码的接口。这一层的设计宗旨是隐藏旧代码的不良质量，为较新的层提供一组一致的服务。然后，让系统其余部分使用这些类而不是旧代码。在这种情况下分层设计的好处在于：（1）它把不良代码的混乱分隔开；（2）如果以后可以抛弃旧代码或重构它，除了接口层，不需要修改任何新代码。

关联参考 一种尤其有价值的标准化是使用设计模式，详情参见 5.3 节中的"寻找常用设计模式"。

标准技术 系统越是依赖外来的部分，对于第一次试图理解它的人来说就越是令人生畏。尽量使用标准化的、常见的方法使整个系统具有熟悉的感觉。

设计的层次

软件系统需要在几个不同的细节层次上进行设计。有的设计技术适合所有层次，有些只适合一、两个层次。图 5-2 展示了这些层次。

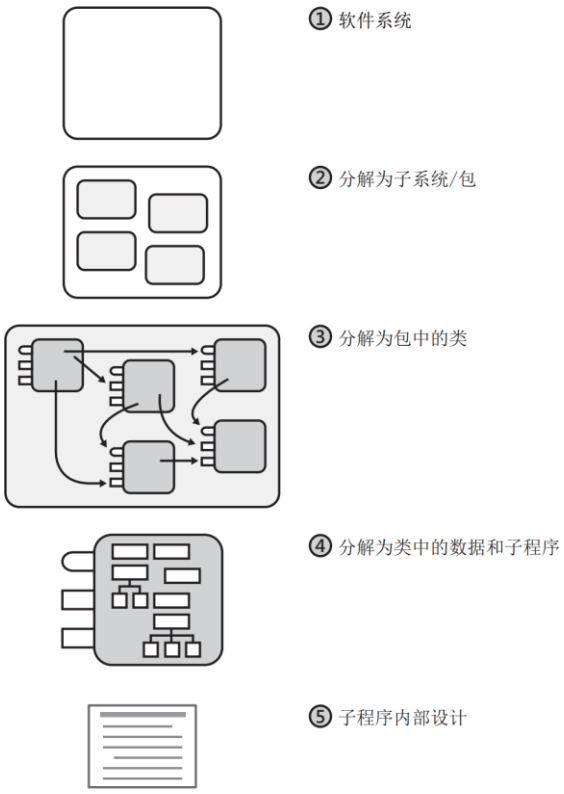


图 5-2　程序的设计层次。系统(1)首先分解为多个子系统(2)，子系统进一步分解为类(3)，类又分解为子程序和数据(4)，每个子程序的内部细节也需要设计(5)

> 换言之，其根本设计缺陷完全隐藏在表面设计缺陷之下，正是凭借着这条铁律，这家公司才在全银河系获得了巨大成功。——道格拉斯·亚当斯

第 1 层：软件系统

第 1 层是整个系统。有的程序员直接从系统层跳到类的设计。但是，从更高层次的类的组合（比如子系统或包）来思考问题是有益的。

第 2 层：分解为子系统或包

这一层设计的主要产出是识别出所有主要子系统。子系统可以很大：数据库、用户界面、业务规则、命令解释器、报表引擎等等。这一层次的主要设计活动是决定如何将程序划分为主要的子系统，并定义每个子系统如何使用其他子系统。任何耗时几周的项目都需要这一层次的分解。每个子系统可以使用不同的设计方法，系统的每一部分都要选择最合适的方法。在图 5-2 中，这一层的设计标注为 2。

在这个层次上，尤其重要的是各个子系统的通信规则。如果所有子系统都能与其他所有子系统通信，就丧失了将它们分开的好处。应该限制通信，使每个子系统变得有意义。

例如，假设要定义一个有 6 个子系统的系统，如图 5-3 所示。如果没有规则，热力学第二定律就会起作用，系统的熵会增加。熵增的一种方式是，如果对子系统之间的通信没有任何限制，通信将以不受限制的方式发生，如图 5-4 所示。

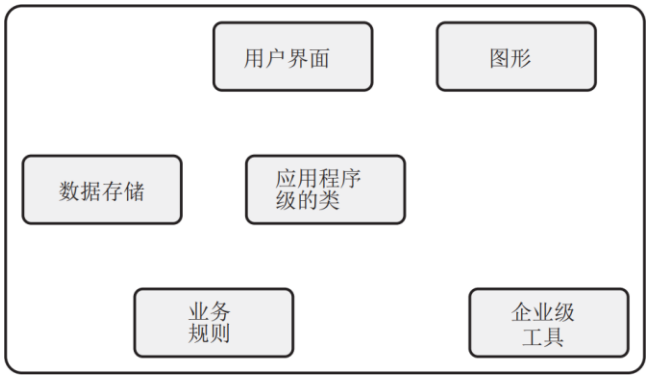


图 5-3　包含 6 个子系统的一个系统

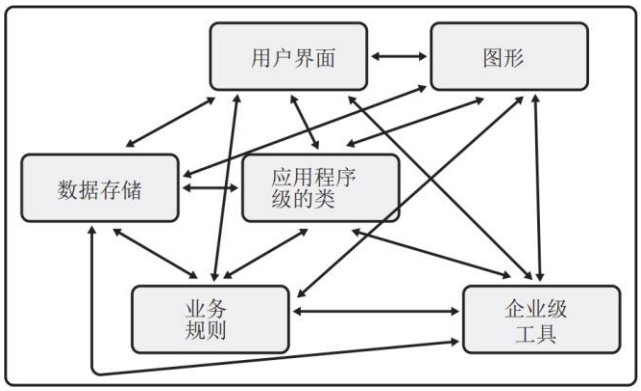


图 5-4　不限制子系统间通信会有什么后果

如你所见，每个子系统最终都会直接与其他每个子系统通信，这就提出了一些重要问题：

- 开发人员至少需要理解系统的多少个不同的部分，才能更改图形子系统中的一些东西？
- 试图在另一个系统中使用业务规则会发生什么？
- 如果给系统换一个用户界面（或许是用于测试的命令行 UI）会发生什么？
- 将数据存储放在一台远程机器上时会发生什么？

可将子系统之间的那些连线想象成水管，水在其中流淌。如果想伸手拉出一个子系统，这个子系统会有一些水管连接到它。必须断开和重新连接的水管越多，你会变得越湿。你要设计的系统应该是这个样子的：如果拉出一个子系统在其他地方使用，不会有很多水管需要重新连接，而且这些水管很容易重连。

只要做到未雨绸缪，解决所有这些问题都只需很少的额外工作。基于“需要知道”（need to know）原则来允许子系统之间的通信，而且最好有一个合适的理由。但凡有疑问，就尽早限制通信并在以后放宽。这比尽早放宽，然后在写了几百个子系统间的调用后再收紧要容易得多。图 5-5 说明了只需遵循几条通信原则，图 5-4 的系统就能有大幅改进。

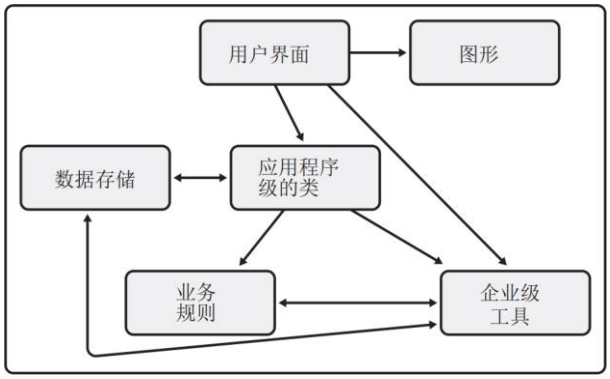


图 5-5　几条通信规则就能显著简化子系统的交互

为了使相互间的联系易于理解和维护，要在简单的子系统间的关系上多下功夫。最简单的关系是让一个子系统调用另一个子系统的子程序。更复杂的关系是让子系统包含另一个子系统的类。最复杂的关系是让一个子系统的类继承另一个子系统的类。

一条好的常规原则是，像图 5-5 这种系统层次的图应该是无环图（acyclic graph）。换言之，程序不应包含任何循环关系，即类 A 使用类 B，类 B 使用类 C，类 C 又使用类 A。

对于大型程序和程序家族，子系统层次上的设计至关重要。如果*认为*自己的程序不大，以至于可以跳过子系统层次的设计，至少要想好理由，不要任性而为。

常见的子系统　有些类型的子系统在不同的系统中反复出现，下面列出了一些常见的。

关联参考　要更多地了解如何用表格来表示业务逻辑，从而对其进行简化，请参见第 18 章。

业务规则　业务规则（business rules）是编码到计算机系统中的法律、法规、政策和过程。如果要写一个工资系统，可能需要对美国国税局（IRS）关于允许的预扣税优待项数和估计税率的规则进行编码。工资系统的其他规则可能来自工会合同，其中规定了加班费率、休假和节假日工资等等。如果写的是一个车保费率报价程序，规则可能来自政府对强险的要求、精算费率表或承保限制。

用户界面　创建一个子系统来隔离 UI 组件，这样 UI 就可以在不破坏程序其他部分的情况下演进。大多数时候，UI 子系统会使用几个附属子系统或类来处理 GUI 界面、命令行界面、菜单操作、窗口管理、帮助系统等。

数据库访问　可将数据库访问的实现细节隐藏起来。这样一来，程序的大多数部分就不必关注对低级结构进行处理时的混乱细节。相反，只需关注数据在业务问题的层次上如何使用。隐藏实现细节的子系统提供了一个宝贵的抽象层次，降低了程序的复杂性。它们将数据库操作集中在一处，减少了处理数据时出错的几率。现在不需要对程序动大手术，即可轻松完成对数据库设计结构的更改。

系统依赖项　出于和打包硬件依赖项一样的原因，将操作系统依赖项打包成一个子系统。例如，如果是为 Microsoft Windows 开发程序，为什么非要将自己限制在 Windows 环境中呢？用一个 Windows 接口子系统隔离所有 Windows 调用。以后若要将程序移植到 Mac OS 或 Linux，所要改变的只有接口子系统。接口子系统的覆盖面可能太广，你自己无法实现，但这样的子系统在一些商业代码库中很容易获得。

第 3 层：分解为类

深入阅读　关于数据库设计的一个很好的讨论，请参见《*Agile Database Techniques*》(Ambler 2003)。

这一层次的设计包括识别系统中的所有类。例如，一个数据库接口子系统可被进一步划分为数据访问类、持久化框架类以及数据库元数据。图 5-2 的第 3 层展示了将第 2 层的一个子系统分解为类的样子，第 2 层其他三个子系统也会做类似分解。

指定了类之后，每个类与系统其他部分的交互方式细节也会被指定。特别是，类的接口会被定义。总的来说，这一层的主要设计活动是确保所有子系统都已分解得足够精细，可将它们的各个部分作为单独的类来实现。

关联参考　关于高质量的类的具体特征，请见第 6 章。

在任何需要超过几天时间的项目中，通常都需要将子系统分解为类。如果项目很大，这种分解会与第二层的程序划分有显著的不同。如果项目很小，可考虑直接从第 1 层的全系统视图跳到第 3 层的类视图。

类和对象　面向对象设计的一个关键概念是区分对象和类。对象是在运行时存在于程序中的任何特定实体。类是在程序的代码清单中看到的静态事物。对象是在运行程序时看到的具有特定值和属性的动态事物。例如，可声明一个具有姓名、年龄、性别等属性的 Person 类。在运行时，会出现 nancy、hank、diane、tony 等对象（均为人名），也就是该类的特定实例。如果熟悉数据库术语，这就类似于“模式”（schema）和“实例”（instance）之间的区别。可将类想象成饼干刀，将对象想象成饼干。本书非正式地使用这些术语，类和对象这两个术语多少会互换着使用。

第 4 层：分解为子程序

这一层的设计是将每个类分解为子程序。第 3 层定义的类接口会定义一些子程序。第 4 层设计的目的就是细化出类的私有子程序。如图 5-2 所示，查看类中的子程序的细节时，会发现许多子程序都是简单的方框，但也有几个是由以层次化的方式组织的更多子程序构成的，这就需要更多的设计。

完整定义了类的子程序后，往往会对类的接口有一个更好的理解，而这又会引起接口的变化——换言之，要回到第 3 层去修改。

这一层次的分解和设计通常由个人程序员决定，而且在任何需要超过几个小时的项目中都需要这样做。不需要很正式，但至少要在心头过一遍。

第 5 层：子程序内部设计

关联参考　要详细了解如何创建高质量 子程序，请参见第 7 章和第 8 章。

子程序层次的设计是详细布置各个子程序的功能。子程序内部设计通常由从事单个子程序的程序员来完成。设计活动包括写伪代码、在参考书上查找算法、决定如何组织子程序中的代

码段以及用编程语言写代码等。这一层次的设计是肯定要做的，尽管有时在无意识的状态下会做得很差。若有意识地去做，效果可能会好一些。在图 5-2 中，这一层的设计标注为 5。

5.3 设计构建基块：启发式方法

软件开发人员喜欢我们给出干脆利落的答案："做 A、B 和 C，每次都会得到 X、Y 和 Z"。我们以习得产生所需结果的一系列神秘步骤为荣。一旦这些指示不能像宣传的那样工作，就会感到恼火。这种对确定性行为的渴望非常适合*细节*程度上的计算机编程。在这种情况下，对细节的严格关注决定了程序的成败。但软件设计与此大相径庭。

由于设计是不确定的，所以熟练运用一套有效的启发式方法（heuristics）便成了优秀软件设计的核心活动。后面几个小节描述了一些启发式方法。我们用这些方法思考设计，而且有时会获得好的设计见解。可将启发式方法看成是一种"试错"指南。毫无疑问，你以前就用到过其中的一些方法。所以，以下小节将从软件的首要技术使命——管理复杂性——的角度来描述每一种启发式方法。

找出现实世界的对象

> 先别问系统做什么；先问它为什么而做！——Bertrand Meyer

关联参考　要更详细地了解如何使用类来进行设计，请参见第 6 章。

第一种也是最流行的确定设计方案的方法是"按部就班"（by the book）的面向对象方法，它侧重于确定现实世界对象和合成对象。

使用对象进行设计的步骤是：

- 确定对象及其属性（方法和数据）。
- 确定可以对每个对象做什么。
- 确定每个对象允许对其他对象做什么。
- 确定每个对象的哪些部分对其他对象可见——哪些部分公共，哪些私有。
- 定义每个对象的公共接口。

这些步骤不一定按此顺序进行，而且经常会重复。迭代很重要。下面总结了这些步骤中的每一步。

确定对象及其属性　计算机程序通常基于现实世界的实体。例如，一个时间计费系统可建立在真实世界的雇员（Employee）、客户（Client）、工时卡（Timecard）和账单（Bill）之上。图 5-6 展示了这样一个计费系统的面向对象视图。

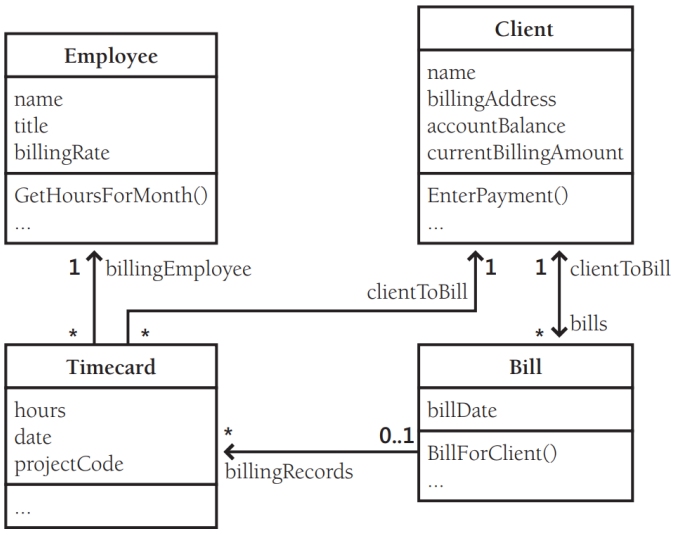


图 5-6　该计费系统由四个主要对象构成，本例对这些对象进行了简化

确定对象的属性并不比确定对象本身更复杂。每个对象都有与计算机程序相关的特征。例如，在时间计费系统中，雇员对象有名字、职务和费率。客户对象有名字、账单地址和账户余额。帐单对象有帐单金额、客户名称、帐单日期。

GUI 系统中的对象包括窗口、对话框、按钮、字体和绘图工具。相较于到现实世界对象的一对一映射，在进一步研究了问题领域之后，也许会产生更好的软件对象选择。但是，现实世界的对象是一个很好的开始。

确定可以对每个对象做什么　可对每个对象执行多种操作。在图 5-6 的计费系统中，雇员对象可以改变职务或费率，客户对象可改变名称或账单地址，等等。

确定每个对象允许对其他对象做什么　对象之间可以做的两种常规的事情是包含（containment）和继承（inheritance）。前者也称为“has a”关系，后者也称为“is a”关系。哪些对象可以包含其他哪些对象？哪些对象可以从其他哪些对象继承？在图 5-6 中，一个工时卡对象可以包含一个雇员对象和一个客户对象，而一个账单可以包含一张或多张工时卡。此外，一个账单可以指出已向客户开具账单，客户可为此账单付款（EnterPayment()）。更复杂的系统将包含更多的交互。

关联参考　要详细了解类和信息隐藏，请参见 5.3 节中的“隐藏秘密(信息隐藏)”。

确定每个对象的哪些部分对其他对象可见　一项关键的设计决定是确定对象的哪些部分应该公开，哪些部分应该保持私有。对数据和方法都必须做出这一决定。

定义每个对象的接口　为每个对象定义正式的、语法上的、编程语言级别的接口。对象向其他*所有*对象公开的数据和方法称为该对象的“公共接口”（public interface）。如果对象的某一部分通过继承向*派生*对象公开，则称为该对象的“受保护接口”（protected interface）。这两种接口都要考虑。

完成了面向对象系统的顶层组织后，将以两种方式进行迭代。第一种是对顶层系统组织进行迭代，以获得更好的类的组织方式。第二种是对定义的每个类进行迭代，将每个类的设计推向一个更详细的层次。

形成一致的抽象

抽象是指在使用一个概念时，安全忽略其部分细节的能力，也就是在不同层次处理不同细节的能力。任何时候只要使用一个集合体，就是在使用抽象。例如，假设将一个东西称为“房子”，而不是玻璃、木材和钉子的组合，就是在抽象。将房子的集合称为一个“镇”，也是在进行抽象。

基类是一种抽象，它允许你专注于一组派生类的共同属性，允许在处理基类时忽略具体类的细节。好的类接口也是一种抽象，它允许你专注于接口而不必关心类的内部工作方式。一个良好设计的子程序的接口在较低的细节层次上提供了同样的好处，而一个良好设计的包或子系统的接口在较高的细节层次上也提供了这样的好处。

从复杂性的角度看，抽象的主要好处在于，它允许你忽略不相关的细节。大多数现实世界中的物体已经是某种抽象了。如前所述，一栋房子是对窗户、门、壁板、电线、管道、绝缘材料以及组织它们的特殊方式的抽象。一扇门是对一块带有铰链和门把手的长方形材料的特殊形式的抽象。门把手又是对黄铜、镍、铁或钢等特定结构的抽象。

人们无时无刻不在使用抽象。如果每次推开前门时都要和单独的木纤维、油漆分子和钢分子打交道，每天就不用进出家门了。如图 5-7 所示，抽象是我们在现实世界中处理复杂性的一个重要部分。

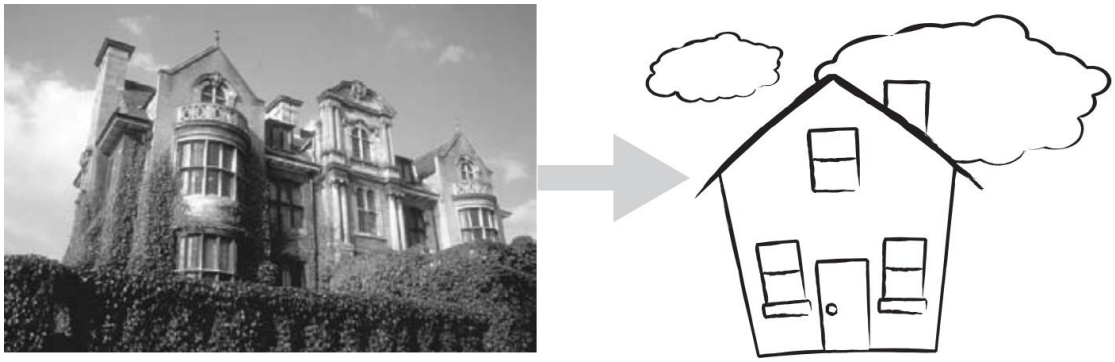


图 5-7　抽象使我们能以更简单的方式看待复杂概念

关联参考　要更详细地了解类设计中的抽象，请参见第 6.2 节中的“良好的抽象”。

软件开发人员有时会在木纤维、油漆分子和钢铁分子的层次上构建系统。这使系统过于复杂，在智力上难以管理。若程序员不能提供更大的编程抽象，系统本身有时就进不了门。

好的程序员会在路由接口层、类接口层和包接口层创建抽象，也就是门把手层、门层和房子层，这支持更快和更安全的编程。

封装实现细节

封装是抽象的延续。抽象说：“你允许从高的细节层次观察对象”。封装说：“此外，你不允许从其他任何细节层次观察对象。”

沿用房子建筑材料的比喻：封装是指你可以从外面看房子，但不能靠得足够近以看清楚门的细节。你可以知道有一扇门，也可以知道门是开着还是关着，但不知道这扇门是木头、玻璃纤维、钢还是材质。自然，更不可能看到每一根木头纤维。

如图 5-8 所示，封装通过禁止你看到复杂的细节来管理复杂性。第 6.2 节中的“良好的封装”介绍了更多关于封装之于类设计的背景知识。

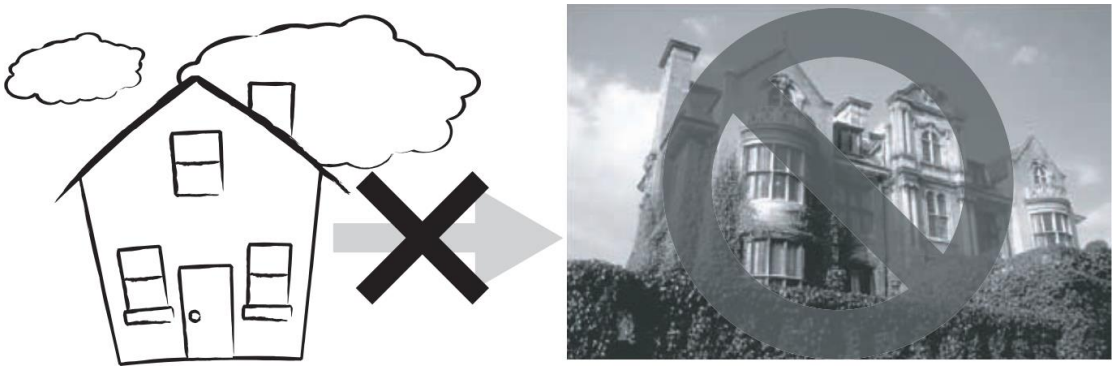


图 5-8　封装是说：不只是允许以更简单的方式看一个复杂概念，还不允许看到这个复杂概念的任何细节。所见即所得——这就是你得到的全部！

能简化设计就继承

设计软件系统时，经常会发现一些对象与其他对象很相似，只有些许区别。例如一个会计系统可能有全职和兼职雇员。与这两种雇员相关的大部分数据都是一样的，只有一些不同。使用面向对象的编程，可以首先定义一个常规雇员类型，然后将全职雇员定义为这种常规雇员，但引入一些差异；兼职雇员也定义成常规雇员，同样引入一些差异。如果对一个雇员的操作不依赖于雇员类型，就可将该雇员当作一个常规雇员来处理。如果操作需取决于雇员是全职还是兼职，处理方式就不同了。

定义这些对象之间的相似性和差异性就是所谓的“继承”，因为更具体的兼职和全职雇员*继承*了常规雇员类型的特征。

继承的好处在于，它能与抽象很好地协作。抽象处理不同细节层次的对象。还是以那扇门为例，它在一个层次上是某种分子的集合，在下个层次上是木质纤维的集合，而在下个层次上是防止小偷进入房子的东西。木材有一定的属性。例如，可以用锯子切割，或者用木胶粘合。无论标准木板还是雪松木片瓦，都具有木材的这些常规属性。但与此同时，它们还有自己的一些特殊属性。

继承简化了编程，因为可以写一个常规的子程序来处理任何依赖于“门的常规属性”的事情。然后，再写特定的子程序来处理对特定种类的门的特定操作。一些操作，如 Open()或 Close()，可能适用于各种门：实心门、内门、外门、纱门、法式门或滑动玻璃门。一种语言如果支持像 Open()或 Close()这样的操作，而且在运行时才知道具体处理的是哪种门，这种能力就称为“多态性”（polymorphism）。包括 C++、Java 和 Visual Basic（要较新的版本）在内的面向对象的语言都支持继承和多态性。

继承是面向对象编程最强大的工具之一。如果用得好，它可以提供巨大的好处；如使用不当，它也会造成巨大的损失。详情请见第 6.3 节中的“继承（‘is a’关系）”。

隐藏秘密（信息隐藏）

信息隐藏是结构化设计和面向对象设计的基础之一。结构化设计的“黑盒”概念就源自信息隐藏。在面向对象设计中，它产生了封装和模块化的概念，并与抽象概念密切关联。信息隐藏是软件开发的开创性思想之一，所以我们专门用一个小节来深入探讨。

信息隐藏（information hiding）的概念最早见于 David Parnas 于 1972 年发表的一篇名为“On the Criteria to Be Used in Decomposing Systems Into Modules”（论将系统分解为模块的准则）的论文，引起了公众的普遍关注。信息隐藏的特色概念是“秘密”（secrets），即软件开发者将设计和实现的决定隐藏在某个地方，与程序的其他部分分开。

在《*The Mythical Man Month*》（人月神话）20 周年纪念版中，Fred Brooks 总结说，他对信息隐藏的批评是他的书的第一版中为数不多的错误之一。他宣称：“Parnas 是对的，我对信息隐藏的看法才是错的”（Brooks 1995）。Barry Boehm 撰文称，信息隐藏是一种消除返工的强大技术。他指出，这种技术在增量、高度变化的环境中特别有效（Boehm 1987）。

对于软件的首要技术使命（管理复杂性）来说，信息隐藏是一种特别强大的启发式方法，因为从它的名字到所有细节，都在强调隐藏复杂性。

秘密和隐私权

在信息隐藏中，每个类（或者包、例程）都因为向其他所有类隐藏的“设计或构造决策”而具有了不同特征。这个隐藏起来秘密也许是一个容易变化的区域、文件格式、数据类型的实现方式或者需要与程序的其他部分隔离的区域（目的是最小化该区域的错误所造成的破坏）。类的职责是将这些信息隐藏起来，同时保护自身的隐私权。系统的微小变化可能影响类中的几个子程序，但不应波及到类的接口之外。

力求类的接口完整且最小。——Scott Meyers

设计类的时候，一项关键任务是决定哪些特性应向类的外部公开，哪些则应保密。类可以使用 25 个子程序，但只公开其中 5 个，另外 20 个仅限内部使用。类可以使用几种数据类型，但选择不公开关于它们的信息。类设计的这个方面也称为“可见性”，因其与类的哪些特性在类的外部“可见”（visible）或“公开”（exposed）有关。

类的接口应尽可能少地透露其内部运作情况。如图 5-9 所示，类很像一座冰山：八分之七在水下，你只能看到水面以上的八分之一。

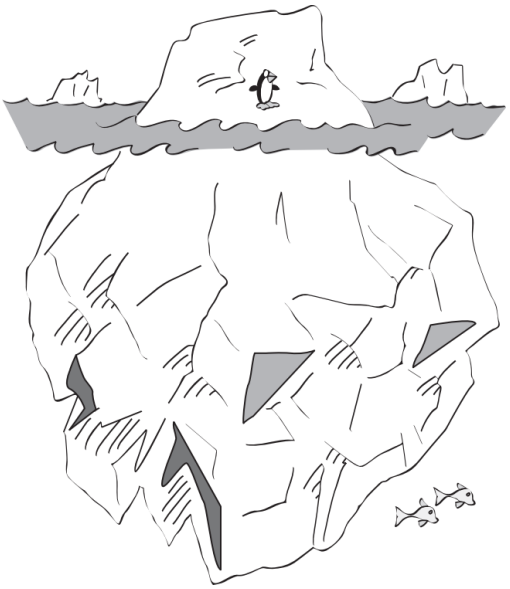


图 5-9　好的类接口正如冰山之一角，类的大多数部分都不暴露出来

类接口的设计是一个迭代过程，这和设计的其他方面是一样的。如果第一次没有把接口设计好，就多试几次，直到稳定下来。如果稳定不下来，就表明需要尝试一种不同的方法。

一个信息隐藏的例子

假设某个程序中的每个对象都应该有一个唯一 ID，存储在名为 id 的成员变量中。一个设计方法是使用整数作为 ID，并将迄今为止分配的最高 ID 存储在名为 g_maxId 的全局变量中。为每个新对象分配存储空间时（或许是通过每个对象的构造函数），可以简单地使用 id = ++g_maxId 语句，这保证会分配一个唯一 ID，而且在创建对象的每个地方，都只会增加绝对最少量的代码。这样做会出什么问题呢？

会出许多问题。如果想保留一个 ID 范围供特殊目的使用呢？如果想使用非连续的 ID 来提高安全性呢？如果想重用已被销毁的对象的 ID 呢？如果想添加一个断言，在分配的 ID 超过预期的最大数量时触发呢？如果用于分配 ID 的 id = ++g_maxId 语句在程序中散布得到处都是，就不得不修改与每一条语句相关的代码。另外，如果程序是多线程的，这种方法还不是线程安全的。

创建新 ID 的方式属于设计决策的范畴，应将其隐藏起来。如果在程序中到处使用++g_maxId，就暴露了创建新 ID 的方式，即简单地递增 g_maxId。相反，如果在程序中到处使用的是 id = NewId()语句，就可以将创建新 ID 的方式隐藏起来。在 NewId()子程序中，可以仍然只有一行代码，返回(++g_maxId)或其等价物。但是，如果以后决定为特殊目的保留某些范围的 ID，或重用旧 ID，可以在 NewId()子程序本身进行这些修改，而不必修改几十乃至几百条 id = NewId()语句。无论 NewId()内部的修改有多复杂，都不会影响程序其他任何部分。

现在，假设你决定将 ID 的类型从整数改为字符串。如果将 int id 这样的变量声明散布在整个程序中，那么即便使用了 NewId()子程序也无济于事。仍需在程序的几十乃至几百个地方进行修改。

所以，要隐藏的另一个秘密是 ID 的类型。通过公开 ID 是整数这一事实，你是在鼓励程序员对其进行整数操作，例如>、<和=。在 C++中，可用一个简单的 typedef 来声明你的 ID 是 IdType（可解析为 int 的用户定义类型），而不是直接声明为 int 类型。另外，在 C++和其他语言中，可以创建一个简单的 IdType 类。隐藏这个设计决策后，再一次极大减少了受变化影响的代码量。

KEY POINT 信息隐藏在设计的各个层次都很有用，从使用具名常量而不是字面量，到创建数据类型，再到类设计、子程序设计和子系统设计。

两种秘密

信息隐藏中的“秘密”主要有两种：

- 隐藏复杂性，这样你的大脑就不必处理它，除非你特别关注它。
- 隐藏变化源，这样当变化发生时，其影响被限制在局部。

复杂性的源头包括复杂的数据类型、文件结构、布尔测试、复杂的算法等等。本章稍后会全面介绍这些变化源。

信息隐藏的障碍

深入阅读　本节部分内容改编自“Designing Software for Ease of Extension and Contraction”（设计易于扩展和收缩的软件）一文（Parnas 1979）。

少数情况下，信息隐藏确实不可能，但信息隐藏的大多数障碍是在习惯性地使用其他技术的过程中建立起来的心理障碍。

信息过度分散　信息隐藏的一个常见障碍是信息在整个系统中的过度分散。你或许在系统的许多地方都硬编码了 100 这个字面量。将 100 作为字面量使用，会分散对它的引用。最好是将这种信息隐藏在一个地方（例如常量 MAX_EMPLOYEES）。这样要修改时，只需在一个地方修改。

另一个信息过度分散的例子是在系统中到处间插与人类用户的交互。如交互方式发生变化（例如从 GUI 界面改为命令行界面），几乎所有代码都要修改。最好将用户交互集中到单一的类、包或子系统中，这样在修改时就不至于影响整个系统。

关联参考　要更多地了解通过类接口来访问全局数据，请参见第 13.3 节中的“使用访问器子程序来取代全局数据”。

再举一个全局数据的例子，也许是最多 1000 个元素的一个雇员数据数组。程序的许多地方都访问了该全局数据。如程序直接使用全局数据，它的实现细节（它是一个数组，最多有 1000 个元素）将在整个程序中传播。相反，如果程序只通过访问器子程序来使用该数据，那么只有访问器子程序才知道这一实现细节。

循环依赖 信息隐藏的一个更微妙的障碍是循环依赖（circular dependencies）。例如，假设类 A 的一个子程序调用类 B 的一个子程序，类 B 的一个子程序又调用类 B 的一个子程序，就会发生循环依赖。

这种循环依赖必须避免。它们造成难以进行系统测试，因为在另一个类的至少一部分准备好之前，类 A 和类 B 都无法单独测试。

类数据误作全局数据 如果你是那种勤勉的程序员，有效信息隐藏的障碍之一可能是将类数据当成了全局数据，并有意识地避免它，因为你想避免牵扯到全局数据的问题。虽然通往编程地狱的道路是由全局变量铺就的，但类数据带来的风险要小得多。

全局数据通常有两个问题：子程序对全局数据进行操作，却不知道其他子程序也在对其进行操作；子程序知道其他子程序正在操作全局数据，但不清楚进行的是什么操作。类数据则不受这两个问题的影响。对数据的直接访问被限制在用单一的类来组织的几个子程序中。这些子程序知道有其他子程序在操作数据，而且清楚是哪些子程序。

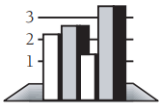
当然，上述讨论的前提是你的系统使用了良好设计的小类。如果程序被设计成使用巨大的类，其中包含数十个子程序，则类数据和全局数据之间的界限会开始变得模糊，类数据会遇到许多与全局数据一样的问题。

关联参考 代码级的性能优化在第 25 章和第 26 章讨论。

感觉到的性能损失 信息隐藏的最后一个障碍可能是试图避免在架构和编码层次的性能损失。但在这两个层次上都无需担心。在架构层次上，这种担心是不必要的，因为针对信息隐藏而设计的系统与针对性能而设计的系统并不冲突。如果兼顾信息隐藏和性能，这两个目标都可以实现。

更常见的担心是在编码层次。人们担心的是，间接访问数据项会因为额外的对象实例化、子程序调用等造成运行时的性能损失。这种担心为时过早。在能够度量系统的性能并找出瓶颈之前，为代码级性能工作做准备的最好方法是创建一个高度模块化的设计。以后检测到热点时，可在不影响系统其余部分的情况下对类和子程序进行单独优化。

信息隐藏的价值



HARD DATA 信息隐藏是为数不多在实践中无可争议证明了其价值的理论技术之一，且久经考验（Boehm 1987a）。人们多年前就发现使用信息隐藏的大型程序比不使用信息隐藏的程序更容易修改——程度能达到 4 倍之多（Korson and Vaishnavi 1986）。此外，信息隐藏是结构化设计和面向对象设计的基础的一部分。

信息隐藏具有独特的启发式力量，即一种能激励有效设计方案的独特能力。传统的面向对象设计提供了用对象建模世界的启发式力量，但对象思维不能帮你避免将 ID 声明为 int 而不是 IdType。面向对象的设计人员会问：“ID 应该被视为对象吗？”取决于项目的编码标准，若答案为“是”，程序员可能需要写一套构造函数、析构函数、拷贝操作符和赋值操作符；全都加上注释；并将其置于源代码管理之中。大多数程序员会决定：“不，不值得为一个 ID 创建完整的类。用 int 就好。”

注意刚才发生了什么事情。一个有用的设计方案（简单地隐藏 ID 的数据类型）甚至想都没有想到。相反，如果设计人员这样问自己：“ID 的什么应该隐藏？”他可能就会决定把它的类型隐藏在一个简单的类型声明背后，用 IdType 代替 int。在这个例子中，面向对象设计和信息隐藏之间的差异比明确的规则（rules）和条例（regulations）的冲突更微妙。其实，面向对象的设计会和信息隐藏一样赞同该设计决策。两者之间的区别更多地在于一种启发式方法：对信息隐藏的思考会激励并促进设计决策的形成，而习惯于用对象来思考则不会。

信息隐藏在设计类的公共接口时也很有用。在类的设计中，理论和实践之间的差距很大。而且在许多类的设计人员当中，本应决定将什么放到类的公共接口中，最后却变成了决定接口怎么使用最方便，这通常会导致类过多的部分被暴露出来。根据我的经验，有的程序员宁愿暴露类的所有私有数据，也不愿多写 10 行代码来保护类的秘密。

“这个类需要隐藏什么？”这个问题切中了接口设计问题的核心。如果能将函数或数据放到类的公共接口中而不影响它的秘密，那就去做。否则就不要做。

习惯于问自己需要隐藏什么，能在所有层次上支持好的设计决策。它在构建层次促进了具名常量的使用（而不是使用字面量）。它有助于在类中创建良好的子程序名和参数名。它还为类和子系统的分解以及系统级互连的决策提供了指引。

KEY POINT 养成问“我该隐藏什么”的习惯。你会惊讶发现许多困难的设计问题都迎刃而解。

确定容易改变的区域

深入阅读　本节描述的方法改编自“Designing Software for Ease of Extension and Contraction”（设计易于扩展和收缩的软件）一文（Parnas 1979）。

对一些优秀设计人员的研究发现，他们的一个共同属性是有能力预测变化（Glass 1995）。适应变化是优秀程序设计最具挑战性的方面之一。目标是隔离不稳定的区域，这样变化的影响就会被限制在一个子程序、类或者包中。以下是在准备应对这种扰动时应遵循的步骤。

1. **确定可能改变的项**。如需求做得好，就会有一份清单列举了潜在的变化以及每个变化的可能性。如果是这种情况，确定潜在的变化很容易。如需求中没有包括潜在的变化，请参考后面对可能发生变化的领域的讨论。

2. **分离(separate)可能发生变化的项**。将步骤 1 所确定的每个易变的组件单独划分为类，或与其他可能同步变化的易变组件划分为同一个类。

3. **隔离(isolate)可能发生变化的项**。设计类间的接口，使其对潜在的变化不敏感。设计接口，将变化限制在类的内部，确保不会影响到外部。任何使用被改变的类的其他类都不应该知道变化已经发生。类的接口应保护类的秘密。

关联参考　预测变化的最强大技术之一是使用表驱动法，详见第 18 章。

下面列举了可能发生变化的一些领域：

业务规则 业务规则是软件频繁变化的常见根源。美国国可能改变税收结构，工会可能重新谈判合同，而保险公司可能改变其费率表。如遵循信息隐藏的原则，基于这些规则的逻辑就不会在程序中散布得到处都是。这些逻辑会被隐藏在系统的某个黑暗角落里，直到它需要改变。

硬件依赖性 硬件依赖性的例子包括与屏幕、打印机、键盘、鼠标、磁盘驱动器、声音设备和通信设备的接口。在它们自己的子系统或类中隔离硬件依赖性。将程序转移到一个新的硬件环境时，对这种依赖关系的隔离会有所帮助。为不稳定的硬件开发程序时，它最初也有帮助。可以编写模拟与特定硬件交互的软件，只要硬件不稳定或不可用，就让硬件接口子系统使用模拟器。等硬件就绪后，将硬件接口子系统与模拟器脱钩，重新将新子系统与硬件挂钩。

输入和输出 在比原始硬件接口稍高的设计层次上，输入/输出是一个不稳定的领域。如果应用程序创建了自己的数据文件，文件格式可能随应用程序的进化而改变。用户层次的输入和输出格式也会发生变化——页面上字段的定位、每个页面上字段的数量、字段的顺序等等。一般来说，最好是检查所有外部接口可能的变化。

非标准的语言特性 大多数语言的实现都包含了方便的、非标准的扩展。这些扩展是一把双刃剑，因为它们到了一个不同的环境就可能无法使用，无论该环境使用的是不同的硬件，不同厂商的语言实现，还是同一厂商的语言新版本。

如果使用了编程语言的非标准扩展，将这些扩展隐藏在自己的类中，这样在转移到不同的环境时，就可以用自己的代码替换它们。类似地，如果使用的库子程序不是在所有的环境中都可用，就将实际的库子程序隐藏在一个在其他环境中能很好工作的接口背后。

困难的设计和构建区域 隐藏困难的设计和构建区域是个好主意，因为它们可能做得不好，以后需要返工。对它们进行划分，将不良设计或构建对系统其他部分可能造成的影响降到最小。

状态变量 状态变量表示程序的状态，往往比其他大多数数据更频繁地改变。在一个典型场景中，你最开始将一个错误状态变量定义为布尔变量，后来又觉得把它实现为具有 ErrorType_None、ErrorType_Warning 和 ErrorType_Fatal 值的枚举类型更佳。

可为状态变量的使用至少增加两个层次的灵活性和可读性：

不要用布尔变量作为状态变量。改为使用枚举类型。经常需要为状态变量添加一种新状态，为枚举类型添加一种新类型只需重新编译，不大费周章地修改检查了该变量的每一行代码。

使用访问器子程序（accessor）而不是直接检查变量。通过检查访问器子程序而不是变量，可以进行更复杂的状态检测。例如，假设要检查错误状态变量和当前功能状态变量的组合，如果测试隐藏在一个例程中，就很容易做到。相反，如果是在整个程序中硬编码的复杂测试，就很难做到。

数据大小限制 声明一个大小为 100 的数组，就向外部暴露了一些外部不需要看到的信息。捍卫你的隐私权！信息隐藏并不总是像一个完整的类那么复杂。有时只需简单地使用一个具名常量（如 MAX_EMPLOYEES）来隐藏 100。

预测变化的不同程度

关联参考　本节描述的预测变化的方法不涉及提前设计和提前编码。关于这两种实践方法，请参见 24.2 节中的“程序包含的代码似乎有一天会被需要”。

深入阅读　这里的讨论取材自“On the design and development of program families”一文所描述的方法（Parnas1976）。

考虑系统的潜在变化时，要设计系统使变化的影响或范围与变化发生的机率成正比。如果一个变化可能发生，就确保系统能很容易地适应它。只有极不可能的变化才应允许对系统的一个以上的类产生剧烈影响。好的设计人员还会考虑预测变化的成本。如某个变化的可能性不大，但很容易计划，就应更努力地去预测它。如可能性不大、难以计划，就不要多费功夫。

识别可能发生变化的领域的一个好办法是，首先识别可能对用户有用的程序最小子集。这个子集构成了系统的核心，不太可能会变。接着，定义系统的最小增量。它们可以小到看起来微不足道。考虑功能的变化时，一定要同时考虑质量的变化：使程序线程安全，使其可局部化（使变化只影响局部），等等。这些潜在的改进领域构成了系统的潜在变化；根据信息隐藏原则设计这些领域。通过首先确定核心，可以看出哪些组件是真正的附加组件，然后推断并隐藏对它们的改进。

保持松散耦合

耦合描述一个类或子程序与其他类或子程序的关系有多紧密。我们的目标是创建与其他类和子程序有小的、直接的、可见的和灵活的关系的类和子程序，这称为“松散耦合”（loose coupling）。类和子程序都存在耦合的概念，所以在接下来的讨论中，我将用“模块”这个词来指代类和子程序。

在模块之间，好的耦合是足够松散的，一个模块可以很容易地被其他模块使用。模型火车的车厢通过对向的一对车钩来耦合，碰到一起就会闭锁。所以连接两节车厢很容易，推到一起即可。想象一下，如果必须用螺丝来拧它们，或者要连接一组线，或者只能将某些种类的车连接到某些其他种类的车上，那会有多困难。模型火车的耦合之所以起作用，是因为它尽可能地简单。在软件中，也要使模块之间的连接尽可能简单。

试着创建很少依赖其他模块的模块。让它们像商业伙伴那样彼此分离，而不是像连体婴儿那样彼此相连。像 sin()这样的子程序是松散耦合的，因为要向其传递一个以度为单位的角度，而这是它唯一需要知道的东西。像 InitVars(var 1, var2, var3, ..., varN)这样的子程序则耦合得较紧密，作为调用方的模块根据必须传递的所有这些变量，实际上知道 InitVars()里面发生了什么。如两个类相互依赖对方对同一个全局数据的使用，它们耦合得更紧密。

耦合标准

根据以下标准评估模块之间的耦合：

规模 规模是指是模块之间的连接数。对于耦合，小即是美。这是因为模块的接口越小，其他模块连接它越容易。相较于带 6 个参数的子程序，带一个参数的子程序与调用它的模块的耦合更松散。相较于公开了 37 个公共方法的类，有 4 个良好定义的公共方法的类与使用它的模块的耦合更松散。

可见性 可见性是指两个模块之间的连接的显著程度。编程和中情局的工作不一样；你不会因为偷偷摸摸而获得荣誉。它更像是广告业；你会因为使连接更显眼而获得大量荣誉。通过参数列表传递数据就建立了一种明显的连接，所以是好的。修改全局数据使另一个模块可以使用这些数据，则是一种偷偷摸摸的连接，所以不好。如果用文档标注了全局数据的连接，使其更加明显，那么会稍微好一点。

灵活性 灵活性是指能多容易地改变模块之间的连接。理想情况下，你想要的是电脑 USB 连接器那样的东西，而不是裸线和焊枪。灵活性的一部分是其他耦合特征的产物，但它也有一点不同。假设有一个名为 LookupVacationBenefit()子程序能在给定雇用日期（hiring date）和职位分类（job classification）的前提下查询雇员每年获得的休假天数。假设另一个模块中有一个 employee 对象，该对象包含雇用日期、职位分类和其他东西，该模块将该对象传递给 LookupVacationBenefit()。

从其他标准的角度看，两个模块是松散耦合的。两个模块之间的雇员连接可见，且只有一个连接。现在，假设需要从第三个模块中使用 LookupVacationBenefit()模块，该模块没有 employee 对象，但有一个雇用日期和职位分类。LookupVacationBenefit()突然就不那么友好了，不愿意和新模块建立连接。

第三个模块要使用 LookupVacationBenefit()，就必须知道 Employee 类的情况。它可以伪造一个只有两个字段的 employee 对象，但这要求对 LookupVacationBenefit()内部情况的了解，而且要求那些是它唯一使用的字段。这样的解决方案显得非常笨拙和丑陋。第二个方案是修改 LookupVacationBenefit()，使其获取雇用日期和职位分类而不是 employee 对象。无论哪种情况，原来的模块都没有一开始看起来那么灵活。

这个故事的欢乐结局是，如果一个不友好的模块愿意变得灵活，它还是能交到朋友。在本例中，它改成专门接收雇用日期和职位分类而不是 employee。

简单地说，一个模块越容易被其他模块调用，它就越松散。这很好，因为它更灵活，更易维护。创建系统结构时，要沿着最小化相互连接的线来分解程序。搞定，将程序想象成一块木头，试着顺着纹理切割。

耦合种类

下面列出了最常见的耦合种类：

简单数据-参数耦合 如果两个模块之间传递的所有数据都是基本数据类型，而且所有数据都通过参数列表传递，这两个模块就是简单数据-参数（simple-data-parameter）耦合。这种耦合是正常的，可以接受。

简单对象耦合 如果一个模块实例化了一个对象，它和那个对象之间就是简单对象（simple-object）耦合。这种耦合没问题。

对象-参数耦合 如果 Object1 要求 Object2 向它传递一个 Object3，两个模块相互之间就是对象-参数（object-parameter）耦合。这种耦合比 Object1 要求 Object2 只向它传递基本数据类型更紧密，因为它要求 Object2 了解 Object3。

语义耦合 最隐蔽的一种耦合发生在一个模块不是利用另一个模块的某些语法元素，而是利用关于后者内部工作方式的某些语义知识。下面是一些例子：

- Module1 向 Module2 传递一个控制标志，告诉 Module2 要做什么。这种方法要求 Module1 知道 Module2 的内部工作方式，即 Module2 会用控制标志做什么。如果 Module2 为控制标志定义了一个特定的数据类型（枚举类型或对象），这种用法或许可行。
- Module2 使用被 Module1 修改之后的全局数据。这种方法要求 Module2 假设 Module1 是按 Module2 需要的方式修改数据，而且 Module1 是在正确的时间调用。
- Module 1 的接口规定，其 Module1.Initialize()子程序应在 Module1.Routine()之前调用。Module2 知道 Module1.Routine()无论如何都会调用 Module1.Initialize()，所以它直接实例化 Module1 并调用 Module1.Routine()，而没有先调用 Module1.Initialize()。
- Module1 将 Object 传递给 Module2。由于 Module1 知道 Module2 只使用了 Object 的 7 个方法中的 3 个，所以只对 Object 进行了部分初始化（使用这 3 个方法需要的特定数据）。
- Module1 将 BaseObject 传给 Module2。由于 Module2 知道 Module1 真正传给它的是 DerivedObject，所以将 BaseObject 转型为 DerivedObject，并调用 DerivedObject 特有的方法。

语义耦合很危险，因为一旦修改了被使用的模块中的代码，就会以编译器完全无法检测的方式破坏使用了该模块的那个模块中的代码。代码是一种很微妙的方式被破坏，似乎与被使用的模块中的改变无关，这就使调试变成了一项西西弗斯式的任务[1]。

松散耦合的意义在于，有效的模块提供了一个额外的抽象层次——写好就保证能用。这就降低了整个程序的复杂性，使开发者一次只需关注一件事。如果在使用一个模块的时候，还要同时关注好几件事情（内部工作方式、对全局数据的修改、不确定的功能），那么抽象的力量就会丧失，模块帮助管理复杂性的能力就会降低或消失殆尽。

KEY POINT 类和子程序是降低复杂性的首选智力工具。如果它们不能使工作变得简单，也就没有意义。

[1] 译注：西西弗斯(Sisyphus)就是希腊神话里那个不停推石头上山的人物。他因触怒众神而被罚将一块巨石推上山顶，但因为那巨石太重了，每每快成功时又滚下山去，由此前功尽弃、周而复始。现在，人们往往用“西西弗斯式的任务”形容那些没有尽头的徒劳之事。

寻找常用设计模式

设计模式（design patterns）精炼了众多现成的解决方案，可用于解决软件许多最常见的问题。有的软件问题需要从第一原理[2]中衍生出解决方案。但大多数问题其实都似曾相识，这些问题可用类似的解决方案——或者称为模式——来解决。常见模式包括 Adapter（适配器）、Bridge（桥接）、Decorator（装饰）、Façade（外观）、Factory Method（工厂方法）、Observer（观察者）、Singleton（单例）、Strategy（策略）和 Template Method（模板方法）。Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides 的《*Design Patterns*》（设计模式）一书（1995）对设计模式进行了权威论述。

与完全定制的设计相比，模式具有以下几方面的益处：

模式提供既有的抽象来降低复杂性 如果你说："这段代码使用工厂方法来创建派生类的实例"，你项目中的其他程序员立即明白你的代码会涉及相当丰富的相互关系和编程协议。只要提到了工厂方法设计模式，一切都不言而喻。

工厂方法模式允许实例化从特定基类派生的任何类，不需要在除了工厂方法之外的任何地方关注派生类的情况。《*Refactoring*》（Fowler 1999）一书的""（用工厂方法取代构造函数）很好地探讨了工厂方法模式。

不需要向其他程序员解释每一行代码，他们就能理解你的代码采用的设计方法。

模式通过将常见解决方案的细节制度化来减少错误 软件设计问题的细微之处只有在问题被解决了一两次（或三次，或四次，或.....）后才会完全显现。由于模式代表的是解决常见问题的标准化方法，所以体现了多年来尝试解决这些问题所积累的智慧，它们还体现了对人们在解决这些问题时所做的错误尝试的纠正。

使用设计模式在概念上类似于使用库代码而不是自己写。当然，每个人都写过几次自定义的 Quicksort（快速排序），但你的自定义版本第一次就完全正确的几率有多大？同样地，许多设计问题与过去的问题有足够的相似性，所以最好使用预先构建好的设计方案，而不是创建一个新的。

模式通过推荐替代设计方案来提供启发式的价值 熟悉常见模式的设计人员可以很轻松地过一遍模式清单，然后问自己："这些模式中哪一个适合我的设计问题？"从一组熟悉的替代方案中寻找，比从头创建一个自定义设计方案要容易得多。而且，从熟悉的模式中生成的代码也会比完全自定义的代码更容易被读者理解。

模式通过将设计对话转移到一个更高的层次来简化沟通 除了管理复杂性的好处，设计模式还通过允许设计者在更大粒度上思考和讨论来加速设计讨论。如果你说："我不好确定在这种情况下应该使用 Creator 还是 Factory Method"，那么只用几个字就传达出了许多信息。当然，前提是你和你的听众都熟悉这些模式。想象一下，如果还需要深入了解 Creator 模式和 Factory Method 模式的代码细节，再对这两种方法进行对比，又会花多少时间？

针对还不太熟悉设计模式的读者，表 5-1 总结了一些常见设计模式，希望能引起学习的兴趣。

---

[2] 译注：第一原理 (first principle)，哲学与逻辑名词，是一个最基本的命题或假设，不能被省略或删除，也不能被违反。第一原理相当于是在数学中的公理。最早由亚里斯多德提出。

表 5-1　流行的设计模式

模式	**说明**
Abstract Factory（抽象工厂）	指定集合类型而非单个对象的类型，从而支持创建相关对象的集合
Adapter（适配器）	将类的接口转换成为一个不同的接口
Bridge（桥接）	构建接口和实现，可独立于彼此改变
Composite（组合）	创建包含其他同类对象的对象，使客户端代码可与顶层对象交互而无需考虑所有的细节对象
Decorator（装饰）	直接为对象动态附加新职能，无需为每种可能的职能配置去创建专门的子类
Facade（外观）	为没有提供一致接口的代码提供一致的接口，也称为门面模式
Factory Method（工厂方法）	实例化特定基类的派生类时，除了在工厂方法内部，其他地方无须跟踪单独的派生类。换言之，工厂方法让类的实例化推迟到子类中进行
Iterator（迭代器）	提供遍历集合元素的统一接口，用一致的方法遍历集合元素，不需要知道集合对象的底层表示
Observer（观察者）	在一个由相关对象构成的集合中，任何成员发生改变时，都由一个对象将这种改变通知给集合中的所有对象，从而保持多个对象的同步
Singleton（单例）	为有且仅有一个实例的类提供全局访问功能
Strategy（策略）	定义一组可动态互换的算法或者行为
Template Method（模板方法）	定义算法的结构，但将部分实现细节留给子类

如果之前没有见过设计模式，对表 5-1 的反应可能是：“大多数概念我都懂”。这种反应是设计模式之所以有价值的重要原因。大多数有经验的程序员都很熟悉这些模式，为它们指定约定俗成的名称更有利于交流。

模式一个潜在的陷阱是让代码强行使用某模式。一些情况下，稍微修改代码以符合一个公认的模式会提高代码的可理解性。但是，如果代码必须大幅修改，强迫它看起来像一个标准模式，则可能增加复杂性。

模式另一个潜在的陷阱是特征强迫症：之所以使用一个模式，是因为想尝试这个模式，而不是因为该模式是合适的设计方案。

总的来说，设计模式是管理复杂性的一种强大工具。可利用本章末尾列出的任何一本好书深入学习。

其他启发式方法

之前的小节描述了主要的软件设计启发式方法。下面描述了其他一些启发式方法，它们可能不那么常用，但仍然值得一提。

力求强内聚性

内聚性（cohesion）是结构化设计的产物，通常与耦合性放到同一背景下讨论。内聚性指的是类中的所有子程序或者子程序中的所有代码对一个中心目的的支持有多紧密。换言之，类的目标有多集中。我们说包含强相关功能的类具有很强的内聚性，启发式方法的目标就是使内聚性尽可能地强。内聚性是管理复杂性的一种有用的工具，因为类中的代码越是支持一个中心目的，你的大脑越容易记住代码所做的一切。

几十年来，在子程序层次上思考内聚性一直是一种有用的启发式方法，今天依然有用。在类的层面上，内聚性的启发式方法在很大程度上已被归纳为良好的抽象，后者覆盖范围更广，这在本章之前已讨论过（第 6 章也会讨论）。抽象在子程序层次上也是有用的，但在细节层次上，它与内聚性的地位更平等。

建立层次结构

层次结构（hierarchy）是一种分层信息结构，最顶端包含的是概念最一般或最抽象的表示，往下的表示则越来越详细和具体。软件的层次结构存在于类层次结构中，也存在于子程序调用层次结构中（如图 5-2 中的第 4 层所示）。

用层次结构管理复杂信息集合至少有 2000 年的历史。亚里士多德曾用层次结构来组织动物王国。人类经常使用大纲要来组织复杂信息（如本书）。研究人员发现，人们普遍认为层次结构是组织复杂信息的一种自然方式。他们画一个复杂的物体（如房子）时，会分层次地画。首先画房子的轮廓，然后是窗户和门，最后是更多的细节。他们不会一砖一瓦地画，也不会一个钉子一个钉子地画（Simon 1996）。

层次结构是实现“软件的首要技术使命”（即管理复杂性）的有用工具，因为它允许你只关注当前要考虑的细节层次。细节并没有消失，它们只是被推到了另一个层次，这样就可以等以后想起的时候再去关注它们，而不必一直想着所有细节。

正式化类契约

> 关联参考　关于契约的更多讨论，请见第 8.2 节中的“采用断言来注解并验证前置条件和后置条件”。

在更详细的层次上，将每个类的接口视为与程序其余部分建立的契约可获得很好的见解。契约通常是这样的：“如果你承诺提供数据 x、y 和 z，并承诺它们具有特征 a、b 和 c，我就承诺基于约束 8、9 和 10 来执行操作 1、2 和 3。”类的客户向类做出的承诺通常称为“前置条件”（preconditions），对象向客户做出的承诺则称为“后置条件”（postconditions）。

契约对复杂性的管理很有用，因为至少从理论上讲，对象可以安全地忽略任何非契约行为。但在实践中，这个问题要困难得多。

分配职责

另一种启发式方法是思考应该如何将责任分配给对象。询问每个对象应负责什么，这类似于询问它应该隐藏什么信息。但是，我认为它可以产生更广泛的答案，这使该启发式方法具有独特的价值。

为测试而设计

一种可以产生有趣的设计见解的思考过程是，问一下如果设计系统时偏向测试的容易性，那么这个系统会是什么样子。是否需要将用户界面与代码的其他部分分开，以便独立测试它？是否需要组织每个子系统，使其尽量减少对其他子系统的依赖？为测试而设计往往会导致更正式的类接口，而这通常是有益的。

避免失败

土木工程教授 Henry Petroski 写了一本有趣的书《*Design Paradigms: Case Histories of Error and Judgment in Engineering*》(Petroski 1994)，书中记录了桥梁设计失败的历史。Petroski 认为，许多重大的桥梁失败案例都是因为专注于之前的成功而未充分考虑可能的失败模式。他的结论是，如果设计者仔细考虑桥梁可能失败的方式，而不只是照搬其他成功设计的属性，像塔科马海峡吊桥这样的失败是可以避免的。

过去几年，各种著名系统的高调安全失误使人很难不同意我们应该想办法将 Petroski 的设计失误见解应用于软件。

过去几年，许多知名的系统都出现了重大安全性事故，所以也应设法将 Petroski 关于设计失败的洞见应用于软件领域。

有意识地选择绑定时间

> 关联参考　要更多地了解绑定时间，请参见 10.6 节。

绑定时间（binding time）是指将特定值绑定到变量的时间。如采用早期绑定的形式，代码往往更简单，但也往往不那么灵活。有的时候，可通过问自己这样的问题来获得一个很好的设计见解：更早绑定这些值会怎样？晚点绑定呢？这张表就在代码的这个地方初始化会怎样？如果在运行时从用户那里读取这个变量的值会怎样？

建立中心控制点

P.J. Plauger 说他最关心的是“正确地方原则（The Principle of One Right Place）——任何重要的代码都只放在一个正确的地方，任何可能的维护更改过都只在一个正确的地方进行”（Plauger 1993）。控制可以集中在类、子程序、预处理宏、#include 文件中，甚至一个具名常量也是中心控制点的例子。

降低复杂性的好处在于，需要寻找的地方越少，修改起来越容易、越安全。

考虑使用蛮力

> 若无把握，就用蛮力（When in doubt, use brute force。另一种译法是“拿不准就穷举”）——Butler Lampson

一种强大的启发式工具是使用蛮力。不要轻视它。一个有效的蛮力解决方案比一个不起作用的优雅解决方案好。一个优雅的解决方案要发挥作用，可能需要很长的时间。例如，在描述查找算法的历史时，Donald Knuth 指出，尽管第一个关于二分查找算法的描述是在 1946 年发表的，但 16 年之后才有人发表了能正确查找各种规模的列表的算法（Knuth 1998）。二分查找自然更优雅，但使用蛮力的顺序搜索往往就足够了。

画图

画图是另一种强大的启发式工具。一图胜千言——某种程度上。实际上，这“千言”中的大部分你都想略去，因为使用图片的一个重要目的是在更高的抽象层次上表示问题。你有时希望在细节层次上处理问题。但其他时候，你希望从更一般的角度去处理。

保持设计的模块化

模块化（modularity）的目标是使每个子程序或类像一个“黑盒”。你知道进去的是什么，也知道出来的是什么，但不知道内部发生了什么。黑盒提供了如此简单的接口和良好定义的功能，以至于针对任何特定的输入，我们都能准确预测相应的输出。

模块化的概念与信息隐藏、封装和其他设计启发式方法有关。但有时想想如何基于一组黑盒来组装出一个系统，可获得信息隐藏和封装所不能提供的见解，所以这个概念值得你放在设计工具箱中。

对设计启发式方法的总结

> 更可怕的是，同一个程序员会用两种或三种方式完成同一任务。这有时是下意识的，但很多时候只是为了改变，或者为了提供优雅的变化形式。——A. R. Brown and W. A. Sampson

下面是对主要启发式设计方法的总结：

- 找出现实世界的对象
- 形成一致的抽象
- 封装实现细节
- 能简化设计就继承
- 隐藏秘密（信息隐藏）
- 确定容易改变的区域
- 预测变化的不同程度
- 保持松散耦合
- 寻找常用设计模式

以下启发式方法有时也很有用：

- 力求强内聚性
- 建立层次结构
- 正式化类契约
- 分配职责
- 为测试而设计
- 避免失败
- 有意识地选择绑定时间
- 建立中心控制点
- 考虑使用蛮力
- 画图
- 保持设计的模块化

使用启发式方法的原则

软件设计的方法可从其他领域的设计方法中借鉴。G. Polya 的《*How to Solve It*》（怎样解题）(1957)是关于问题求解过程中的启发式方法的早期著作之一。Polya 的广义问题求解方法侧重于数学。图 5-10 对他的方法进行了总结，改编自他的书中的类似总结。

1. 理解问题。你必须*理解*问题。

未知数是什么？*数据是什么*？*条件是什么*？条件可以满足吗？条件是否足以确定未知数？或者不充分？或者多余？或者矛盾？

画一个图。引入适当的符号。将条件的各个部分分开。能把它们写下来吗？

2. 制定计划。找到数据和未知数之间的联系。如果找不到中间的联系，可能不得不考虑辅助性问题。终究要拿出一个解决方案的*计划*。

以前见过这个问题吗？或者是否见过形式稍微不同的同样的问题？*知道一个相关的问题吗*？知道一个可能有用的定理吗？

注意这个未知数！试着去想有相同或类似未知数的一个熟悉的问题。*这里有一个与你的问题相关而且已被解决的问题。能使用它吗*？能使用它的结果吗？能使用它的方法吗？是否应该引入一些辅助元素，使它的使用成为可能？

能重述这个问题吗？而且能以不同的方式重述它吗？不行就回归定义。

如果不能解决所提出的问题，先试着先解决一些相关问题。能想出一个更容易获得的相关问题吗？一个更普遍的问题？一个更特殊的问题？一个类似的问题？能解决问题的一部分吗？只保留条件的一部分，放弃另一部分；离未知数的确定近了多少，它可以如何变化？能从数据中推导出有用的东西吗？能想到其他适合确定未知数的数据吗？能改变未知数或数据，或者必要时两者都改变，使新的未知数和新的数据更接近吗？

使用了所有的数据吗？使用了整个条件吗？是否考虑到了问题中涉及的所有基本概念？

3. 执行计划。*执行你的计划。*

执行你的计划，*检查每一步*。能清楚地看到这个步骤是正确的吗？能证明它是正确的吗？

4. 向后看。检查解决方案。

能核实结果吗？能核实论证吗？能以不同的方式推导出结果吗？能一目了然地看出来吗？

能在其他问题上使用这一结果或方法吗？

图 5-10　G. Polya 开发了一种数学解题方法，对解决软件设计中的问题也很有用(Polya 1957).

最有效的原则之一是不要拘泥于单一的方法。如果用 UML 绘制设计图不奏效，就用自然语言写。写一个简短的测试程序。尝试一种完全不同的方法。想一个使用蛮力的解决方案。一直用铅笔勾画，你的大脑就会跟上。如果其他所有方法都失败了，就放下这个问题。出去走走，或者在回到问题前先想点别的事情。如尽力而为之后却毫无进展，就把它从脑海中抛开一段时间，这往往比纯粹的坚持更快奏效。

不必一下子就解决整个设计问题。如果卡住了，就记着这个地方，同时认识到现在还没有充分的信息来解决这个具体的问题。本来下一次就能豁然开朗的事情，为何非要在设计最后 20% 的时候苦苦挣扎？本可在积累了更多经验后做出好的决定，为何非要在经验有限的时候做出错误的决定？在一个设计周期后，如果事情没有完成，有的人会觉得不舒服。但是，如果先把问题放在一边，并动手创建一些设计，会发现在没有充分的信息之前，本来就不适合强行解决那些问题（Zahniser 1992, Beck 2000）。

5.4 设计实践

上一节主要介绍了与设计特性有关的启发式方法——你希望完成的设计是什么样子。本节描述了设计实践的启发式方法，即可以采取的、通常能得到良好结果的步骤。

迭代

你可能有过这样的经历：从一个程序的编写中学到了很多东西，以至于希望带着第一次写程序所获得的见解再写一遍。设计也有同样的现象，但设计周期更短，对下游的影响也更大，所以只能负担不多的几次设计循环。

KEY POINT 设计是一个迭代的过程。通常不是只从 A 点到 B 点；而是从 A 点到 B 点，再回到 A 点。

在候选设计中循环往复并尝试不同的方法时，要同时从高层和低层的角度看问题。从高层问题中获得的大局观将帮助你正确看待低层细节。从低层问题中得到的细节则将为高层决策提供坚实的现实基础。顶层和底层考虑之间的拉锯战是一种健康的动态；它创造了一个应力结构，比一个完全自上而下或者自下而上建立的结构更稳定。

许多程序员（这个意义上说，就是许多人）在高低层视角之间转换存在一定的困难。从系统的一个角度转换到另一个角度，在心智上是很吃力的，但这对创造有效的设计至关重要。关于如何提高心智灵活性的一些有趣的练习，请参见本章末尾的“更多资源”描述的《*Conceptual Blockbusting*》一书（Adams 2001）。

关联参考　重构是一种在代码中尝试不同替代方案的安全方式。详情请参见第 24 章。

如果第一次设计尝试看起来不错，不要停止！第二次尝试几乎总是比第一次好，而且每次都能从中学到一些东西，从而改善你的整体设计。据报道，爱迪生在尝试了一千种不同的灯丝材料而没有成功之后，有人问他是否觉得时间被浪费了，因为他什么也没有发现。“胡说八道”，爱迪生应该是这样回答的：“我发现了一千种不起作用的东西”。许多时候，用一种方法解决问题会产生一些见解，使你能用另一种更好的方法解决问题。

分而治之

正如 Edsger Dijkstra 所指出的，没有人的大脑大到足以容纳一个复杂程序的全部细节，这同样适用于设计。将程序分解为不同的关注领域，再单独解决每个领域的问题。如果在其中一个领域进入了死胡同，那就迭代吧！

增量改进是管理复杂性的一个强有力的工具。正如 Polya 在解决数学问题时建议的那样，理解问题，制定计划，执行计划，再回头看看你做得怎么样(Polya 1957)。

自上而下和自下而上设计方法

“自上而下”和“自下而上”可能听起来有点老套，但它们为创建面向对象的设计提供了宝贵的见解。自上而下的设计开始于一个高层次的抽象。你定义基类或其他非具体的设计元素。随着设计的发展，你会增加细节的层次，确定派生类、协作类和其他详细设计元素。

自下而上的设计则是先具体再概括。它通常从确定具体的对象开始，然后从这些具体的对象中概括出对象的集合和基类。

有些人激烈地争论说，从一般情况开始，然后向具体事物发展是最好的，另一些人则认为，除非解决了重要的细节问题，否则无法真正确定一般的设计原则。下面是双方的论点。

自上而下的论点

自上而下方法背后的指导原则是，人类的大脑一次只能专注于一定数量的细节。如果从一般的类开始，一步一步把它们分解成更具体的类，大脑就不会被迫一次处理太多细节。

分而治之的过程从两种意义上说是迭代的。首先，通常不会只分解一层便停止。你会持续分解几层。其次，你通常不会满足于自己的第一次尝试。你会以一种方式分解程序。在分解的不同阶段，你会选择以何种方式来划分子系统，布置继承树，以及形成对象的组合。你做出选择，看看会发生什么。然后重新开始，以另一种方式分解，看看效果是否更好。经过几次尝试后，会对什么会起作用以及为什么起作用有一个很好的想法。

程序要分解到什么程度？持续分解，直到似乎为下一级直接写代码比分解还要容易。要到设计变得非常清晰和简单，你觉得再分解下去都有点不耐烦为止。这个时候，你就完成了。如果还不清晰，就再做一些。如果现在的解决方案对你来说都有点棘手，后人更会觉得无所适从。

自下而上的论点

有的时候，自上而下的方法是如此抽象，以至于无从下手。如果需要处理更具体的东西，可试试自下而上的设计方法。问问自己：“这个系统需要做什么？”无疑，这个问题你是可以回答的。你或许能确定一些低层次的职责，并将其分配给具体的类。例如，你可能知道系统需要格式化一个特定的报表，为该报表计算数据，标题居中，在屏幕上显示报表，在打印机上打印报表，等等。确定了几个低层次的职责后，通常会开始感到舒适，这时可以看高一些。

在其他一些情况下，设计问题的主要特性是由底层决定的。你可能需要与硬件设备对接，设计的一大块内容都要由这些设备的接口决定。

下面是在进行自下而上设计时需要记住的一些事情：

- 问自己对系统要做的事情有什么了解。
- 从对这个问题的回答中确定具体的对象和职责。
- 确定通用的对象，使用子系统组织、包、对象内的组合或者继承对它们进行分组，哪种合适就用哪种。
- 继续向上一层，或回到顶层并再次尝试自上而下

其实真的不存在争议

自上而下和自下而上策略的关键区别在于，一个是分解（decomposition）策略，另一个是合成（composition）策略。一个从一般性的问题出发，将其分解成可管理的部分；另一个从可管理的部分出发，建立起一般性的解决方案。这两种方法各有优点和缺点，将它们应用于自己的设计问题时，要考虑这些优缺点。

自上而下的设计的优点是很容易。人们善于把大的东西分解成小的部分，而程序员尤其善于此道。

自上而下设计的另一个优点是，可以推迟细节的构建。由于系统经常被构造细节的变化（例如，文件结构或报表格式的变化）所干扰，在早期就知道这些细节应隐藏在层次结构底部的类中是非常有用的。

自下而上的方法的一个优点是，它通常能及早识别出所需的实用功能，从而形成一个紧凑的、进行了良好分解的设计。如已经构建过类似的系统，自下而上的方法允许你观察旧系统的组成部分并问“我可以重用什么？”来开始新系统的设计。

自下而上合成方法的一个缺点是，它很难排他性地使用。大多数人更善于将一个大概念分解成小概念，而不是把小概念做成一个大概念。这就像一个古老的“自行组装”问题：我觉得都完成了，为什么盒子里还有零件？幸好，你并非只能单纯地使用自下而上组合方法。

自下而上设计策略的另一个缺点是，有时你会发现，无法从最初的那些部件中构建一个程序。用砖头造不了飞机，可能要先在顶层工作，然后才能知道底层需要什么样的部件。

总之，自上而下往往开始时很简单，但有时低级别的复杂性会波及到顶层，而这些会使事情变得不必要的复杂。自下而上往往开始时很复杂，但在早期识别这种复杂性会导致更好地设计更高层次的类——如果这种复杂性没有首先使整个系统遭受破坏的话！

归根结底，自上而下和自下而上的设计并不是相互竞争的策略，它们是相辅相成的。设计是一个启发式（探索式）的过程，这意味着没有任何解决方案可以保证每次都能成功。试验和错误都是正常的设计元素。尝试各种不同的方法，直到找到一种行之有效的。

实验性的原型设计

有的时候，除非更好地理解了一些实现细节，否则无法真正知道一个设计是否奏效。可能不知道一种特定的数据库组织方式是否可行，直到知道它是否能满足性能目标。可能不知道一个特定的子系统设计是否可行，直到选好了要使用的特定 GUI 库。这些都是软件设计中必不可少的“棘手”问题——除非至少解决了设计问题的一部分，否则无法完全定义这个问题。

以低成本解决这些问题的一般技术是实验性原型设计。原型设计（prototyping）一词对不同的人有许多不同的含义（McConnell 1996）。在当前的上下文中，原型设计意味着写最少的抛弃型代码（throwaway code）来回答一个特定的设计问题。

如果开发人员不自律，没有写回答问题所需的绝对最少的代码，原型设计的效果就很差。假设一个设计问题是：“我们选择的数据库框架能否支持需要的事务处理量？”不需要写任何生产代码来回答该问题。甚至不需要知道数据库的具体细节。只需要知道估算问题空间所需的就可以了，即表的数量、表中条目的数量，等等。接着，可以写非常简单的原型代码，表

使用 Table1、Table2、Column1 和 Column2 等“占位符”形式的名称，用垃圾数据填充这些表，然后进行性能测试。

若设计问题不够具体，原型设计的效果也很差。像“这个数据库框架能不能用？”这样的设计问题不能为原型设计提供足够的方向。而像“在假设 X、Y 和 Z 的条件下，这个数据库框架能否支持每秒 1000 个事务处理？”这样的设计问题能为原型设计提供更坚实的基础。

原型设计的最后一个风险发生在开发人员不将代码当成抛弃型代码的时候。我发现，只要人们觉得代码最终会出现在生产系统中，就不可能写出绝对最少的代码来回答问题。这就变得是在实现系统而非进行原型设计。一定要坚持这样的态度：一旦问题得到回答，代码就会被扔掉。这样才能将这种风险降至最低。避免这个问题的一个办法是用不同于生产代码的技术来创建原型。例如，可用 Python 创建一个 Java 设计的原型，或者用 Microsoft PowerPoint 模拟一个用户界面。如确实要用生产技术来创建原型，一个实用的标准也可以帮得到你，那就是要求原型代码的类或包的名称以 prototype 作为前缀。这至少能让程序员在试图扩展原型代码之前三思而行（Stephens 2003）。

在遵守纪律的情况下，原型设计是设计者对抗设计“棘手问题”的主要工具。如果不守纪律，原型设计本身就会带来一些“棘手问题”。

协作设计

关联参考　要更多地了解协作式开发，请参见第 21 章。

在设计中，两个人往往比一个好，无论这两个人是正式的还是非正式地组织在一起。协作可采取以下任何形式之一：

- 你非正式地走到一个同事的办公桌前，要求交换一些想法。
- 你和同事一起坐在会议室里，在白板上画设计方案。
- 你和同事一起坐在键盘前，用你们选择的编程语言进行详细设计。换言之，可使用第 21 章描述的结对编程。
- 你安排一场会议，和一个或多个同事探讨设计思路。
- 你安排对第 21 章描述的所有结构的一次正式检查（formal inspection）。
- 你找不到能对你的工作进行评审的人，所以你先做一些初步的工作，放到抽屉里，一周后再回来看它。这时已经忘得差不多了，所以能给自己一个中肯的评价。
- 你向公司外部的人寻求帮助：把问题发送到专门的论坛或新闻组。

如果目标是保证质量，我倾向于推荐高度结构化的评审实践，即正式检查，原因将在第 21 章描述。但是，如果目标是培养创造力和增加备选设计方案的数量，而不仅仅是为了发现错误，那么结构化程度较低的方法会更好。确定了具体设计方案之后，取决于项目的性质，可能需要改为一种更正式的检查。

多少设计才够？

> 我们试图通过匆忙的设计过程来解决问题，以便在项目结束时还有足够的时间来发现那些因为匆忙设计而产生的错误。——Glenford Myers

有的时候，在编码开始之前，只绘制了最基本的架构草图。其他时候，团队创造的设计是如此详细，以至于编码成了一项机械劳动。开始编码之前，应该做多少设计？

一个相关的问题是设计要变得多正规。需要正规的、精心设计的图表，还是对着白板上画的几张图拍几张数码照片就够了？

决定在开始全面编码之前要做多少设计，以及设计文档需要多正规，这几乎不可能有定论。团队的经验、系统的预期寿命、期望的可靠性水平以及项目和团队的规模都应考虑在内。表 5-2 总结了这些因素中的每一个是如何影响设计方法的。

表 5-2　设计的正规程度和所需的细节程度

因素	开始构建之前设计所需的细节程度	文档的正规程度
设计/构建团队在应用程序领域有很丰富的经验	低	低
设计或构建团队有很丰富的经验，但是在这个应用程序领域缺乏经验	中	中
设计或构建团队缺乏经验	中到高	低到中
设计或构建团队人员变动适中或者很频繁	中	—
应用程序对安全性要求很严格（safety-critical）	高	高
应用程序对任务的完成要求很严格（mission-critical）	中	中到高
小型项目	低	低
大型项目	中	中

软件预期的生命周期很短（数周或数月）	低	低
软件预期的生命周期很长（数月或数年）	中	中

其中两个或多个因素可能会在任何特定的项目中发挥作用，而且在某些时候，这些因素可能提出相互矛盾的建议。例如，你可能有一个经验丰富的团队在做对安全性要求很严格的软件。在这种情况下，你可能倾向于较高程度的设计细节和正规性。这时就需要权衡每个因素的重要性，并判断什么最重要。

如果设计层次交由每个人决定，那么一旦设计下沉至你以前完成过的任务的水平，或者下沉至只需对这样的任务做一次简单修改或扩展，那么就可能已准备好停止设计，开始编码。

如果我无法确定开始编码之前要对一个设计进行多深入的调研，我倾向于选择更多的细节。最大的设计错误来自于这样的情况：我认为已经走得够远了，但其实并没有走得多远，以至于没有意识到还存在其他设计挑战。换言之，最大的设计问题往往不是来自于我知道有困难而做了糟糕设计的领域，而是来自于我认为很容易而根本没有做任何设计的领域。我很少遇到因为做了太多的设计而受影响的项目。

> 我从来没有遇到过一个愿意阅读 17000 页文档的人，如果有的话，我会杀了他，让他离开基因库。——Joseph Costello

另一方面，我偶尔也会看到一些项目因为过多的设计文档而受到影响。葛兰辛法则（Gresham's Law，即“劣币驱逐良币”）指出：“程序化的活动往往会驱逐非程序化的活动”（Simon 1965）。急于润色设计描述就是该法则的一个很好的例子。我宁愿看到 80%的设计工作用于创建和探索多种备选设计方案，20%用于创造不太精炼的文档，也不愿意看到 20%用于创建平庸的设计方案，80%对不怎么好的设计的文档进行润色。

记录设计工作

记录设计工作的传统方法是将设计写在正式的设计文档中。但是，还可以用其他许多方法来记录设计，这些方法在小型项目、非正式项目或者需要以轻量级的方式记录设计的项目中效果很好：

> 坏消息是，在我们看来，我们永远找不到点金石。我们永远找不到一个能让我们以完全理性的方式设计软件的过程。好消息是，我们可以伪造它。——David Parnas 和 Paul Clements

在代码中插入设计文档 以代码注释的形式记录关键设计决策。一般将这些注释放在文件或类的头部。将这种方法与 JavaDoc 这样的文档提取器结合使用，就保证了设计文档可以随时提供给正在编写某段代码的程序员，并且更利于程序员保持设计文档的同步更新。

在 Wiki 上记录设计讨论和决策 在项目 Wiki（是一个网页的集合，项目中的任何人都可通过 Web 浏览器轻松编辑）上以书面形式记录你们的设计讨论。虽然打字比说话麻烦一些，但这将自动记录设计讨论和决策。还可以在 Wiki 上截取数字图片为文字讨论提供补充，提供支持设计决策的网站链接，展示白皮书和其他材料等。如开发团队分布于各地，这种技术特别有用。

写电子邮件汇总 在设计讨论之后，指定某人写一份讨论汇总（尤其是哪些已经决定下来的内容），并将其发送给项目组。在项目的公共电子邮件文件夹中存一个副本。

使用数码相机 记录设计时的一个常见障碍在于，用一些流行的绘图工具创建设计图时过于繁琐。但是，文档的选择并非只能是“用格式优美的正式符号来记录设计”和“没有任何设计文档”二选一。

用数码相机给白板上的图拍照，将图片嵌入传统文档。为此付出的精力相当少，只需大约 1%的工作，就可以获得用绘图工具来保存设计图 80%的好处。

保存设计挂图 没有任何法律规定你的设计文档只能使用标准打印纸。如果用大的挂图纸画设计图，就可以简单地将挂图存放在一个方便的地方。更好的做法是，将它们张贴在项目区周围的墙上，这样人们就可以在需要的时候方便地参考和更新它们。

使用 CRC(类、职责、合作者，Class, Responsibility, Collaborator)卡片 另一种记录设计的低技术含量的方法是使用索引卡。在每张卡片上，设计者写上类名、类的职责和合作者（与这个类合作的其他类）。然后，一个设计小组用这些卡片工作，直到他们满意于自己创建的设计。到那个时候，就可以简单地保存这些卡片以备参考。索引卡很便宜，不吓人，且便于携带，它们鼓励小组互动（Beck 1991）。

在适当的细节层次上创建 UML 图 一种流行的设计图表技术被称为统一建模语言（UML），它是由 Object Management Group 定义的（Fowler 2004）。本章前面的图 5-6 就是 UML 类图的一个例子。UML 为设计实体和关系提供了一套丰富的正规表示。可用 UML 的非正式版本来探索和讨论设计方法。从最基本的草图开始，只有在确定了最终设计方案后才可以增加细节。由于 UML 是标准化的，所以有利于在沟通设计思路时取得共识，而且在小组中工作时，它能加速考虑替代设计方案的过程。

以上技术可通过不同的组合来发挥作用，所以请在每个项目（甚至在一个项目的不同区域）中自由组合和匹配这些方法。

5.5 点评各种流行的方法论

在软件设计的历史中，出现过许多对自相矛盾的设计方法的狂热主张。我在上世纪 90 年代初出版《*代码大全*》的第一版时，设计狂热者主张在开始编码之前，设计的每一个细节都要搞得清清楚楚。这个建议毫无意义。

> 那些宣扬软件设计是一种有严明纪律的活动的人，他们花了如此多的精力来布道，使我们都觉得内疚了。我们永远不可能有足够的结构化或面向对象的能力在这一生中实现涅槃。我们都有一种原罪，那就是在可塑性很强的年龄学会了 Basic。但我敢打赌，我们中的大多数人都是比纯粹主义者所承认的更好的设计师。——P. J. Plauger

我在 2000 年代中期写本书这一版的时候，一些软件专家正在为不做任何设计而争论。“Big Design Up Front 就是 BDUF（预先做大设计）” 他们说：“BDUF 不好。最好在开始编码之前不做任何设计！”

就过了十年这么多时间，钟摆已经从“设计一切”摆到了“不设计”。但是，BDUF 的替代方案并不是不预先设计，而是少量预先设计（Little Design Up Front，LDUF）或者足够的预先设计（Enough Design Up Front，ENUF）。

你如何判断多少设计才够？这是一道主观判断题，没人能完美做出这个判断。但是，虽然无法判断准确的设计量，但有两种设计量保证是错误的：设计每一个细节和根本不做任何设计。天平两端的极端主义者所主张的两种立场，原来是唯一两种必然错误的立场。

正如 P.J. Plauger 所说的，“你越是教条地应用一种设计方法，要解决的现实问题就越少”（Plauger 1993）。将设计当作一个棘手的（wicked）、没有章法的（sloppy）、启发式的（heuristic）过程。不要满足于你想到的第一个设计。要协作。要努力追求简单。在需要的时候进行原型设计。迭代，迭代，再迭代。你最终会对自己的设计感到满意。

更多资源

软件设计是一个拥有丰富资源的领域。挑战在于确定哪些资源最有用。这里提供了一些建议。

软件设计的一般性问题

Weisfeld, Matt. 《*Object-Oriented Thought Process*》2d ed，SAMS，2004。这是一本容易理解的介绍面向对象编程的书。如果已经熟悉了面向对象编程，你可能想要一本更高阶的书，但如果是刚开始接触面向对象，这本书介绍了基本的面向对象概念，包括对象、类、接口、继承、多态性、重载、抽象类、聚合和关联、构造/析构函数、异常等。

Riel, Arthur J. 《*Object-Oriented Design Heuristics*》，Reading, MA: Addison-Wesley, 1996。这本书很容易懂，重点是在类的层次上进行设计。

Plauger, P.J. 《*Programming on Purpose: Essays on Software Design*》，Englewood Cliffs, NJ: PTR Prentice Hall, 1993。我从这本书学到的软件设计技巧和我从所有读过的其他书中学到的一样多。Plauger 精通大量设计方法，他很务实，他是一位了不起的作者。

Meyer, Bertrand. 《*Object-Oriented Software Construction*》2d ed，New York, NY: Prentice Hall PTR, 1997。Meyer 大力宣传了硬核的面向对象编程

Raymond, Eric S. 《*The Art of UNIX Programming*》，Boston, MA: Addison-Wesley. 2004。本书从 UNIX 的角度很好地研究了软件设计。1.6 节用 12 页的篇幅简明扼要地解释了 17 条关键 UNIX 设计原则。

Larman, Craig. 《*Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*》2d ed，Englewood Cliffs, NJ: Prentice Hall, 2001。这是一本很流行的基于统一过程介绍面向对象设计的著作。书中还讨论了面向对象分析。

软件设计理论

Parnas, David L., and Paul C. Clements.“A Rational Design Process: How and Why to Fake It.” *IEEE Transactions on Software Engineering* SE-12，no.2（February 1986）：251-257。这篇经典文章描述了在程序的设计理想和实际之间的巨大差距。其主旨在于，没人真正经历过理性的、有序的设计过程，但以此为目标，确实能在最后获得更好的设计方案。

我没有找到对信息隐藏进行全面论述的资料。大多数软件工程教科书都只是简单提了一下，通常都是在面向对象技术的语境中提及。下面列出的三篇 Parnas 论文是对这一观点的开创性介绍，可能仍然是信息隐藏方面最好的资源：

Parnas, David L.“On the Criteria to Be Used in Decomposing Systems into Modules.” *Communications of the ACM* 5，no. 12（December1972）：1053-1058.

Parnas, David L.“Designing Software for Ease of Extension and Contraction.”*IEEE Transactions on Software Engineering* SE-5，no.2（March1979）：128-138.

Parnas, David L., Paul C.Clements, and D.M.Weiss.“The Modular Structure of Complex Systems.” *IEEE Transactions on Software Engineering* SE-11，no.3（March 1985）：259-266.

设计模式

Gamma, Erich, et al.《*Design Patterns*》，Reading，MA：Addison-Wesley，1995，四人帮（GoF）的这本著作是设计模式的开山之作。

Shalloway, Alan, and James R.Trott.《*Design Patterns Explained*》，Boston，MA：Addison-Wesley，2002。这本书对设计模式做了深入浅出的介绍。

广义设计

Adams, James L.《Conceptual Blockbusting: A Guide to Better Ideas》，4th ed。Cambridge，MA：Perseus Publishing，2001。虽然这本书不是专门讲软件设计的，但斯坦福大学用它来向工科学生讲解设计。哪怕从来不做任何的设计，这本书也对创新思维过程做了非常棒的描述。

书中包含很多有效设计所需的思维训练，还给出了一份详细注释的，关于设计和创造性思维的参考书目。如果读者喜欢解决问题，相信也会喜欢这本书的。

Polya, G.《*How to Solve It: A New Aspect of Mathematical Method*》，2d ed。Princeton，NJ：Princeton University Press, 1957。这本书讲解了数学领域中的启发式方法和问题求解，但同样适用于软件开发。Polya 的这本书首次在数学问题求解领域中引入启发式方法。他在书中清楚地区分了在探索问题时使用的杂乱无章的启发式方法，和一旦找到解决方案后用于呈现解法的更整洁的方法。这本书读起来并不容易，但如果读者对启发式方法感兴趣，那么不管想不想读，最终都会去读它。Polya 在书中明确说明，问题求解并不是一个确定性的活动，如固守于某一种方法，则无异于作茧自缚。曾有一段时间，微软把这本发给了每一名新入职的程序员。

Michalewicz, Zbigniew, and David B. Fogel.《*How to Solve It: Modern Heuristics*》，Berlin: Springer-Verlag, 2000。这本书对 Polya 的书做出了更新，相比之下更容易阅读，而且包含了一些非数学领域的例子。

Simon, Herbert.《*The Sciences of the Artificial*》, 3d ed。Cambridge, MA:MIT Press, 1996。这本书对与自然界相关的科学（生物学、地质学等）和与人造世界相关的科学（商业、建筑以及计算机科学）之间的差异做了非常精彩的描述。然后，这本书讨论了人工科学的特征，并着重强调了设计科学。对于那些渴望在软件开发或者任何“人工的”领域内工作的人员来说，这都是一本很好的学院派论著，值得一读。

Glass, Robert L.《*Software Creativity*》，Englewood Cliffs, NJ: Prentice Hall PTR, 1995。软件开发更多地是由理论指导还是由实践指导？软件开发从根本上而言是创造性的还是确定性

的？软件开发者需要什么样的智力素质？这本书针对软件开发的本质展开了有趣的讨论，并特别强调了设计。

Petroski, Henry. 《*Design Paradigms: Case Histories of Error and Judgment in Engineering*》, Cambridge: Cambridge University Press, 1994。这本书大量借鉴了土木工程领域（特别是桥梁设计）的设计案例，以诠释其主要观点：成功的设计至少同等地取决于从过去的失败中学习以及从过去的成功中学习。

标准

IEEE Std 1016-1998，*Recommended Practice for Software Design Descriptions*。这份文档包含了用于描述软件设计的 IEEE-ANSI 标准。其中描述了软件设计文档中应该包含哪些内容。

IEEE Std 1471-2000，*Recommended Practice for Architectural Description of Software Intensive Systems.* Los Alamitos, CA: IEEE Computer Society Press。这份文档是用于创建软件架构规范的 IEEE-ANSI 指南。

检查清单：软件构建中的设计

设计实践

- 是否已做过迭代，从多个结果中选择了最佳的一种，而不是简单地选择首次尝试的结果？
- 尝试过以多种方式分解系统以确定哪种最好吗？
- 同时采用了自上而下和自下而上的方法来解决设计问题吗？
- 针对系统中有风险或者不熟悉的部分进行过原型设计，写数量最少的抛弃型代码来回答特定问题吗？
- 自己的设计方案被其他人评审过吗？无论正式与否。
- 一直在推动设计，直至实现细节昭然若揭吗？
- 使用了某种适当的技术（例如 Wiki、电子邮件、挂图、数码照片、UML、CRC 卡片或者代码中内嵌的注释）来记录了你的设计吗？

设计目标

- 设计是否充分解决了在系统架构层次确定并决定推迟实现的问题？
- 设计是分层的吗？
- 对于程序分解为子系统、包和类的方式感到满意吗？
- 对于类分解为子程序的方式感到满意吗？
- 类的设计是否使它们之间的交互最小化？
- 类和子系统的设计是否方便你在其他系统中重用？
- 程序是否容易维护？

- 设计是否精简？它的所有部分都是绝对必要的吗？
- 设计是否使用了标准技术，避免了奇特的、难以理解的元素？
- 总的来说，这个设计是否有助于将偶然和本质的复杂性降至最低？

要点回顾

- 软件的首要技术使命是*管理复杂性*。以简单性作为目标的设计方案对此最有帮助。
- 简单性可通过两种方式来实现：一是尽量减少任何人的大脑在任何时候都必须处理的*本质上的*复杂性的数量；二是防止*偶然的*复杂性无谓地扩散。
- 设计是一种启发式过程。固守于某种单一的方法会损害创新能力，进而损害程序。
- 优秀的设计都是迭代而来的。尝试的设计可能性越多，最终的设计方案越好。
- 信息隐藏是一个非常有价值的概念。通过询问“我应该隐藏些什么？”能解决许多非常困难的设计问题。
- 在本书之外，还有其他许多有用和有趣的关于设计的资源。我们在这里提出的观点只是冰山之一角。

第 6 章　可以工作的类

内容

- 6.1 类的基础：抽象数据类型（ADT）
- 6.2 良好的类接口
- 6.3 设计和实现问题
- 6.4 创建类的理由
- 6.5 语言特定问题
- 6.6 超越类：包

相关章节

- 软件构建设计：第 5 章
- 软件架构：第 3.5 节
- 高质量的子程序：第 7 章
- 伪代码编程过程：第 9 章
- 重构：第 24 章

在计算的早期岁月，程序员基于语句思考编程问题。到上世纪七八十年代，程序员开始基于子程序去思考编程。进入 21 世纪，程序员基于类来思考编程。

KEY POINT 类是一组数据和子程序的集合，这些数据和子程序共享一组内聚的、良好定义的职责。类也可以是一组子程序的集合，这些子程序提供一组内聚的服务，即使其中并未涉及共用的数据。成为高效程序员的一个关键在于，在处理任何一段代码时，程序其余部分你能忽略的越多越好。类是实现这一目标的主要工具。

本章提炼了创建高质量类的一些建议。如果是刚开始接触面向对象的概念，本章可能显得过于高级。请确保你已读完了第 5 章，"构造中的设计"，再阅读第 6.1 节开始，这样其余小节读起来就比较轻松了。如果已经熟悉了类的基础知识，则可以粗略看一下第 6.1 节，然后直接看第 6.2 节关于类接口的讨论。本章末尾的“更多资源”小节列出了初级读物、高级读物以及与特定编程语言相关的资源。

6.1　类的基础：抽象数据类型（ADT）

抽象数据类型（Abstract Data Type，ADT）是数据和对这些数据进行的操作的一个集合。这些操作既向程序的其余部分描述数据，又允许程序的其余部分改变数据。“抽象数据类型”中的 "数据 "一词用得很宽泛。一个 ADT 可以是一个图形窗口和所有影响它的操作、一个文件和文件操作、一个保险费率表和对它的操作或者其他东西。

关联参考　首先考虑 ADT，然后才考虑类，这是“深入一种语言去编程”（programming into a language），而不是“在一种语言上编程”（programming in a language）的例子。详情请参见第 4.3 节和第 34.4 节。

理解 ADT 是理解面向对象编程的关键。不了解 ADT，程序员创建的类就只是名义上的“类”——实际只是塞满了各种松散相关的数据和子程序的一个行李箱。通过对 ADT 的理解，程序员可创造出最初更容易实现，以后也更容易修改的类。

传统的编程书籍在谈到抽象数据类型的话题时，喜欢以数学的方式来阐述。它们往往会说：“可以将抽象数据类型看成是定义了一系列操作的数据模型”。这样的书让人觉得，除了为自己催眠，似乎永远不会真正用到抽象数据类型。

这种对抽象数据类型的干巴巴的解释完全没抓住重点。抽象数据类型是令人振奋的，因为可用它操作真实世界的实体，而不是操作低级的实现实体。例如，你不是在一个链接列表中插入节点。相反，是直接在电子表格中添加一个单元格，在窗口类型列表中添加一种新的窗口类型，或者在火车模拟中添加另一节乘客车厢。能直接在问题领域中工作，而不是非要跑到低级的实现领域去干活儿，这是不是很酷？

需要用到 ADT 的例子

为展开讨论，先来看有一个 ADT 会很有用的例子。有了一个例子之后，再来讨论细节问题。

假设要写一个程序，用多种字体、字号和字体属性（如粗体和斜体）控制文本在屏幕上的输出。程序的一部分负责操作文本的字体。如果使用一个 ADT，会有一组捆绑了数据（字体名称、字号和字体属性）的字体子程序，子程序对这些数据进行操作。字体子程序和数据合起来就是一个 ADT。

如果不使用 ADT，就要用一种临时的方法来操作字体。例如，如需改为 12 磅字号，而这恰好为 16 像素高度，就会有这样的代码：

```
currentFont.size = 16
```

但是，如事先构建了一个子程序库，代码的可读性会好一些：

```
currentFont.size = PointsToPixels( 12 )
```

或者可以为属性提供一个更具体的名称，如下所示：

```
currentFont.sizeInPixels = PointsToPixels( 12 )
```

但是，不能同时拥有 currentFont.sizeInPixels 和 currentFont.sizeInPoints，因为如果这两个数据成员都在发挥作用，currentFont 就没办法知道它应该使用这两个中的哪一个。另外，如果程序的好几个地方都要修改字号，那么类似的代码行会散布于整个程序中。

如需将一种字体设为粗体，可能会写下面这样的代码，它用到了一个逻辑 or 和一个十六进制常量 0x02：

```
currentFont.attribute = currentFont.attribute or 0x02
```

如果运气好，可以写比这更清楚的代码。但是，只要使用的是临时性解决方案，得到的最好的代码也不过是这样：

```
currentFont.attribute = currentFont.attribute or BOLD
```

或者是这样：

```
currentFont.bold = True
```

和修改字号的情况一样，由于客户端代码需要直接控制数据成员，所以限制了 currentFont 的使用方式。

像这样编程，很可能程序中的许多地方都会出现类似的代码行。

ADT 的好处

问题不在于临时性方法是坏的编程实践。问题在于，完全可以用一种更好的编程实践来取代这种方法来获得以下好处：

可以隐藏实现细节 隐藏字体数据类型的信息意味着一旦数据类型发生变化，可以在一个地方修改而不会影响整个程序。例如，除非在 ADT 中隐藏了实现细节，否则一旦将数据类型从加粗的第一种表示法改为第二种表示法，就需要在设置了加粗的每个地方（而不是仅仅在一个地方）修改程序。如决定将数据存储在外部存储而不是内存中，或者用另一种语言重写所有字体操作子程序，隐藏这种信息也能保护程序的其余部分。

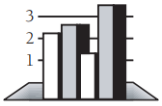
修改不会影响整个程序 如字体需要变得更丰富，支持更多操作（比如转换为小写、变成上标、加删除线等等），只需在一个地方修改。这种修改不会影响程序的其余部分。

可让接口提供更多信息 像 currentFont.size = 16 这样的代码存在歧义，因为 16 可能是一个以像素或磅数为单位的字号。上下文并未说明哪个是哪个。将所有类似的操作收集到一个 ADT 中，就可以明确以磅为单位定义整个接口，以像素为单位定义整个接口，或者明确区分两者。再也不会出现模凌两可的情况。

更容易提高性能 如需提高字体的性能，可重新编码几个进行了良好定义的子程序，而不必折腾整个程序。

更容易确定程序的正确性 不用麻烦地验证 currentFont.attribute = currentFont.attribute or 0x02 这样的语句是否正确。相反，简单地验证对 currentFont.SetBoldOn()的调用是否正确即可。在第一个语句的情况下，可能会使用错误的结构名称、错误的字段名称、错误的操作（比如写成 and 而不是 or）或者错误的属性值（写成 0x20 而不是 0x02）。在第二个语句的情况下，调用 currentFont.SetBoldOn()唯一可能出错的地方就是它调用了错误的子程序名称，所以更容易看出正确与否。

程序可读性更佳 虽然可将 0x02 替换为 BOLD 或任何 0x02 代表的东西，从而改善像 currentFont.attribute 或 0x02 这样的语句的可读性，但在可读性方面，无论如何都比不上 currentFont.SetBoldOn()这样的子程序调用。



HARD DATA Woodfield, Dunsmore 和 Shen 做了一项研究，他们让 CS 专业的研究生和高年级本科生回答了关于两个程序的问题：一个按功能分解为 8 个子程序，另一个分解为 8 个 ADT 子程序（1981）。结果，使用后者的学生比使用前者的学生得分高出 30%以上。

不必在程序中到处传递数据 在之前的例子中，必须直接修改 currentFont 或将其传给每个要处理字体的子程序。如使用抽象数据类型，就不必在程序中到处传递 currentFont，也不必把它变成全局数据。可在 ADT 中用一个结构来包含 currentFont 的数据。只有属于 ADT 一部分的子程序才可直接访问这些数据。不属于 ADT 的子程序则不必关心这些数据。

可直接操作现实世界的实体，不必操作低级的实现结构 可定义字体操作，使程序的大部分能直接操作字体，而不是搞什么数组访问、结构定义以及 True/False。

就本例来说，为了定义一个抽象数据类型，可以像下面这样定义几个用于控制字体的子程序：

```
currentFont.SetSizeInPoints( sizeInPoints )
currentFont.SetSizeInPixels( sizeInPixels )
currentFont.SetBoldOn()
currentFont.SetBoldOff()
currentFont.SetItalicOn()
currentFont.SetItalicOff()
currentFont.SetTypeFace( faceName )
```

KEY POINT 这些子程序中的代码可能很短——也许类似于在之前的临时字体解决方案中的看到的代码。不同的是，现在用一组子程序隔离了字体操作。这为程序的其余部分提供了一个更好的抽象层次来处理字体。它还提供了一层保护，防止因字体处理方式的变化而影响整个程序。

更多 ADT 示例

假设要写一个软件来控制核反应堆冷却系统。可将冷却系统视为一个抽象数据类型，为它定义以下操作：

```
coolingSystem.GetTemperature()
coolingSystem.SetCirculationRate( rate )
coolingSystem.OpenValve( valveNumber )
coolingSystem.CloseValve( valveNumber )
```

具体环境决定了为实现这些操作而编写的代码。程序其余部分可通过这些函数来操作冷却系统，同时不必担心数据结构的内部实现细节、数据结构的限制、未来可能的变化等等。

下面列举了更多抽象数据类型以及可能对它们的操作：

巡航控制	**搅拌机**	**油箱**
设置速度 获取当前设置 恢复之前的速度停止运行	开启 关闭 设置速度 启动“即时粉碎模式” 停止“即时粉碎模式”	填充油箱 排空油箱 获取油箱容积 获取油箱状态

列表	**灯**	**栈**
初始化列表 向列表中插入条目 从列表中删除条目 读取列表中的下一个条目	开 关	初始化栈 向栈中压入条目 从栈中弹出条目 读取栈顶条目
帮助屏幕	**菜单**	**文件**
添加帮助主题 删除帮助主题 设置当前帮助主题 显示帮助屏幕 关闭帮助屏幕 显示帮助索引 返回上个屏幕	开始新的菜单删除菜单 添加菜单项 删除菜单项 激活菜单项 禁用菜单项 显示菜单 隐藏菜单 获取菜单选项	打开文件 读取文件 写入文件 设置当前文件位置 关闭文件
指针		**电梯**
获取新分配内存的指针 清理(dispose)现有指针指向的内存 更改已分配内存大小		向上一层 向下一层 到指定层 报告当前楼层 回到底层

通过对这些例子的研究，可归纳出几个指导原则（后续小节会逐一解释）：

将典型的低级数据类型作为 ADT 来构建或使用，而不是作为低级数据类型 大多数关于 ADT 的讨论都集中在将典型的低级数据类型作为 ADT 来表示。从之前的例可以看出，可将栈、列表、队列和其他几乎任何典型数据类型都表示为 ADT。

你要问的问题是：“这个栈、列表或队列代表什么？”如栈代表一组雇员，就将 ADT 视为雇员而不是栈。如列表代表一组计费记录，就将其视为计费记录而不是列表。如队列代表电子表格中的单元格，就将其视为单元格的集合，而不是队列中的常规项。总之，要视为尽可能最高的抽象级别。

将文件等通用对象视为 ADT　大多数语言都包括一些你可能熟悉、但没有想到是 ADT 的抽象数据类型。文件操作就是一个很好的例子。向磁盘写入时，操作系统让你不必操心如何将读/写头定位到特定物理地址、旧的磁盘扇区满后后分配新的以及对神秘的错误代码进行解释。操作系统提供了第一级抽象和该级别的 ADT。高级语言提供了第二级抽象和该更高级别的 ADT。高级语言帮你避免了生成操作系统调用和操作数据缓冲区的繁琐细节，允许你将一大块磁盘空间当作一个“文件”。

自己可用类似的方式为 ADT 分级。例如，可在一个级别使用提供了数据结构级操作（例如入栈和出栈）的 ADT，然后在这一级的上方再创建另一级，在现实世界问题层面上工作。

简单的东西也能当作 ADT　不一定要用复杂的数据类型来证明使用 ADT 的合理性。在前面的示例列表中，有一个 ADT 是一盏灯，它只支持两个操作：开和关。你可能认为将简单的“开”和“关”操作隔离在它们自己的子程序中是一种浪费。但是，即使是简单的操作也能从使用 ADT 中受益。把灯和它的操作放到一个 ADT 中，可使代码更可读，更容易修改，修改也只会影响 TurnLightOn()和 TurnLightOff()子程序内部，并减少了必须传递的数据项的数量。

直接引用 ADT 而不必关心它的存储介质　假设一个保险费率表非常大，一直存储在磁盘上。最初可能会想把它称为“费率文件”（rate file），并创建诸如 RateFile.Read()这样的访问子程序。但是，一旦把它称为文件，就不必要地暴露了更多关于数据的情报。以后若修改程序，将表存储到内存而不是磁盘，以前把它作为文件来引用的代码就是不正确、误导性和混乱的。试着使类和访问子程序的名称和数据的存储方式无关，并改为引用抽象数据类型，例如“保险费率表”（insurance-rates table）。这样类名和访问子程序的名称就可变成 rateTable.Read()或更简单的 rate.Read()。

在非面向对象环境中用 ADT 处理多个数据实例

面向对象的语言为 ADT 多个实例的处理提供了自动化支持。如果只在面向对象的环境中工作过，而且完全不需要自己处理多实例的实现细节，那你应该感到庆幸！（可直接跳到下一节 “ADT 和类”）。

如果在非面向对象的环境（比如 C 语言）中工作，就不得不手动构建对多实例的支持。通常，这意味着要为 ADT 提供创建和删除实例的服务，还要设计 ADT 的其他服务，使它们能与多个实例一起工作。

本章前面讲过的字体 ADT 最初提供了以下服务：

```
currentFont.SetSize( sizeInPoints )
currentFont.SetBoldOn()
currentFont.SetBoldOff()
currentFont.SetItalicOn()
currentFont.SetItalicOff()
currentFont.SetTypeFace( faceName )
```

在非面向对象环境中，这些函数不附属于一个类，看起来更像下面这样：

```
SetCurrentFontSize( sizeInPoints )
SetCurrentFontBoldOn()
SetCurrentFontBoldOff()
SetCurrentFontItalicOn()
SetCurrentFontItalicOff()
```

```
SetCurrentFontTypeFace( faceName )
```

如果要同时处理多种字体，就需要添加创建和删除字体实例的服务，例如：

```
CreateFont( fontId )
DeleteFont( fontId )
SetCurrentFont( fontId )
```

这里添加 fontId 作为创建和使用多种字体时对它们进行跟踪的一个手段。对于其他操作，可从三种处理 ADT 接口的方式中选择：

- 选项 1：每次使用 ADT 服务时都显式标识实例。本例没有“当前字体”的表示法。相反，是将 fontId 传给每个操作字体的子程序。Font 的函数负责对所有底层数据的跟踪，而客户端代码只需跟踪 fontId。采用这种方式，需要添加 fontId 作为每个字体子程序的参数。

- 选项 2：显式提供 ADT 服务要使用的数据。采用这种方法，要在使用了 ADT 服务的每个子程序中声明 ADT 要使用的数据。换言之，要创建一个 Font 数据类型，并将其传给每个 ADT 服务子程序。必须设计 ADT 服务子程序，使它们在每次被调用时都使用传入的 Font 数据。采用这种方式，客户端代码就不需要一个字体 ID，因为它自己负责字体数据的跟踪。即使这些数据可直接从 Font 数据类型获得，也只应通过 ADT 服务子程序来访问。这就是所谓的保持结构的“闭合”（closed）。
  这个方式的优点在于，ADT 服务子程序不必根据一个字体 ID 来查找字体信息。缺点是它将字体数据暴露给了程序的其他部分，这增加了客户端代码会利用 ADT 的实现细节的可能性，而这些细节本应隐藏在 ADT 内部。

- 选项 3：使用隐式实例（要非常小心）。设计一个新服务，调用它将一个特定的字体实例变成当前字体，例如 SetCurrentFont(fontId)。设置当前字体会使其他所有服务在被调用时都使用当前字体。如采用这种方式，就不需要将 fontId 作为其他服务的参数。对于简单的应用程序，这能简化多个实例的使用。对于复杂的应用程序，这种系统范围内对状态的依赖意味着必须在使用了 Font 的各种函数的所有代码中跟踪当前字体实例。复杂性往往会激增。而且无论什么规模的应用程序，都存在更好的替代方案。

抽象数据类型内部有大量选项可供处理多个实例，但在外部，如果使用的是一种非面向对象的语言，选择就只有这么多了。

ADT 和类

抽象数据类型构成了类的概念的基础。在支持类的语言中，可将每个抽象数据类型作为它自己的类来实现。类通常涉及继承和多态性的额外概念。一种思考类的方式就是为抽象数据类型加上继承和多态性。

6.2 良好的类接口

创建高质量类的第一步（也可能是最重要的一步）是创建一个良好的接口。其中包括创建良好的抽象供接口呈现，并确保细节隐藏在抽象背后。

良好的抽象

正如第 5.3 节中的“形成一致的抽象”描述的那样，抽象是以简化形式看待一个复杂操作的能力。类的接口为隐藏在接口背后的实现提供了一个抽象。类的接口应提供一组明显应放在一起的子程序

假设用一个类来实现雇员。它包含对雇员的姓名、地址、电话号码等进行描述的数据。它还要提供一些服务，以便对雇员进行初始化和使用。如下所示：

关联参考　本书示例代码采用强调了多语言风格相似性的一种编码惯例。关于该惯例的细节（以及关于多种编码风格的讨论），请参见第 11.4 节中的“混合语言编程的注意事项”。

C++示例：呈现了良好抽象的类接口

```
class Employee {
public:
   // public constructors and destructors
   Employee();
   Employee(
      FullName name,
      String address,
      String workPhone,
      String homePhone,
      TaxId taxIdNumber,
      JobClassification jobClass
   );
   virtual ~Employee();

   // public routines
   FullName GetName() const;
   String GetAddress() const;
   String GetWorkPhone() const;
   String GetHomePhone() const;
   TaxId GetTaxIdNumber() const;
   JobClassification GetJobClassification() const;
   ...
private:
   ...
};
```

在内部，这个类可能有额外的子程序和数据来支持这些服务，但类的用户不需要知道这些。类的接口抽象非常好，因为接口中的每个子程序的都在朝一个方向努力。

如下例所示，抽象不好的类会是一个包含杂七杂八函数的集合：

C++示例：呈现了不良抽象的类接口

```
class Program {
public:
   ...
   // public routines
   void InitializeCommandStack();
   void PushCommand( Command command );
   Command PopCommand();
   void ShutdownCommandStack();
   void InitializeReportFormatting();
   void FormatReport( Report report );
   void PrintReport( Report report );
   void InitializeGlobalData();
   void ShutdownGlobalData();
   ...
private:
   ...
};
```

如果一个类同时包含了处理命令栈、格式化报告、打印报告和初始化全局数据的子程序，那么在命令栈（command stack）和报告子程序或全局数据之间，我们很难看到任何联系。类的接口没有呈现出一致的抽象，所以该类的内聚性（cohesion）很差（越明显相关，内聚性越好）。这些子程序应重新组织到职能更专一的类中，每个类都通过其接口提供更好的抽象。

如果这些子程序是 Program 类的一部分，可以像下面这样修订它们以呈现出一致的抽象：

C++示例：呈现了更好抽象的类接口

```
class Program {
public:
   ...
   // public routines
   void InitializeUserInterface();
   void ShutDownUserInterface();
   void InitializeReports();
   void ShutDownReports();
   ...
private:
   ...
};
```

对这个接口的清理假定原来的一些子程序已被转移到其他更合适的类中，另一些被转换为InitializeUserInterface()和其他子程序所使用的私有子程序。

这种对类抽象的好坏的评估是基于类的公共子程序集合——也就是类的接口提供的那些。不能认为整个类的抽象不错，类内部的单独子程序就肯定有良好的抽象。这些单独的个体也需要专门设计，使其呈现出良好的抽象。第 7.2 节提供了这方面的指导原则。

创建类的接口时，遵循以下指导原则实现良好的接口抽象：

在类的接口中呈现一致的抽象级别 可将类看成是实现第 6.1 节所描述的抽象数据类型（ADT）的一种机制。每个类都应实现一个且只有一个 ADT。如发现某个类实现了不止一个 ADT，或无法确定该类实现了什么 ADT，就应将该类重新组织成一个或多个良好定义的 ADT。

下面这个类呈现了不一致的接口，因其抽象级别不统一：

这些子程序的抽象在“雇员”(employee)级

这些子程序的抽象在“列表”(list)级

C++示例：抽象级别不一的类接口

```
class EmployeeCensus: public ListContainer {
public:
   ...
   // public routines
   void AddEmployee( Employee employee );
   void RemoveEmployee( Employee employee );

   Employee NextItemInList();
   Employee FirstItem();
   Employee LastItem();
   ...
private:
   ...
};
```

该类呈现了两个 ADT，即 EmployeeCensus 和 ListContainer。若程序员使用容器类或其他来自库的类进行实现，但又不隐藏使用了库中的类这一事实，通常就会出现这种混合抽象。要问问自己：使用容器类的事实是否应成为抽象的一部分？那通常属于实现细节的范畴，应该对程序的其余部分隐藏，如下所示：

所有这些子程序的抽象都在“雇员”(employee)级

类使用*ListContainer*库类的事实被隐藏起来了

C++示例：抽象级别一致的类接口

```
class EmployeeCensus {
public:
   ...
   // public routines
   void AddEmployee( Employee employee );
   void RemoveEmployee( Employee employee );
   Employee NextEmployee();
   Employee FirstEmployee();
   Employee LastEmployee();
   ...
private:
   ListContainer m_EmployeeList;
   ...
};
```

程序员可能会争辩说，从 ListContainer 继承很方便，因其支持多态性，允许外部搜索或排序函数获取一个 ListContainer 对象。但这一论调无法通过继承的主测试，即“是否只将继承用于 is a 关系？”从 ListContainer 继承，就意味着 EmployeeCensus“is a”ListContainer，而这显然不成立。如果 EmployeeCensus 对象的抽象是它可以搜索或排序，就应作为类接口的一个显式的、一致的部分被纳入。

如果将类的公共子程序想象成防止水进入潜艇的气闸（air lock），不一致的公共子程序就是这个类漏水的仪表盘。漏水的仪表盘可能不会像打开的气闸那样迅速让水进入，但如果给它们足够的时间，它们还是会把潜艇弄沉的。在实践中，这就是混合抽象级别时会发生的情况。随着程序的修改，混合的抽象级别使程序越来越难理解。而且它会逐渐退化，直到变得不可维护。

KEY POINT **一定要理解类实现的是什么抽象** 有的类非常相似，必须小心翼翼地理解类的接口应捕捉哪种抽象。我曾经做过一个程序，需要允许以表格形式编辑信息。我们想使用一个简单的网格控件，但现有的网格控件不允许为数据输入单元格着色，所以我们决定使用一个提供了这种功能的电子表格控件。

电子表格控件比网格控件复杂得多，它提供了 150 个子程序，而网格控件只提供了 15 个。由于目标是使用网格控件，而不是电子表格控件，所以我们指派了一个程序员来写一个包装类（wrapper class），以隐藏我们将电子表格控件作为网格控件使用这一事实。这个程序员对不必要的开销和官僚作风抱怨了一通，然后离开了。几天后，他带来了一个包装类，忠实暴露了电子表格控件的全部 150 个子程序！

这并不是我们想要的。我们想要的是一个网格控件接口，其中封装了“我们在幕后使用一个复杂得多的电子表格控件”的事实。程序员应只暴露 15 个网格控制子程序，另加第 16 个支持单元格着色的子程序。但是，如果公开全部 150 个子程序，程序员创造了这样一种可能性：如果以后想改变底层实现，就需要支持全部 150 个公共子程序。这个程序员没能实现我们所期望的封装，而且他的好多工作都白做了。

取决于具体情况，正确的抽象可能是一个电子表格控件，也可能是一个网格控件。若不得不从两个相似的抽象中选择，请确保你选择的是正确的那个。

成对提供服务并包含反向操作　大多数操作都有对应、等同和相反的操作。如果有一个开灯操作，可能还需要一个关灯操作。如果有一个向列表添加数据项的操作，可能还需要一个从列表中删除数据项的操作。如果有一个激活菜单项的操作，可能还需要一个禁用菜单项的操作。设计类时要检查每个公共子程序，确定是否需要它的互补操作。不要无脑地创建反向操作，但要核实是否需要一个。

将不相关的信息移到另一个类中　某些时候，你会发现类的一半子程序操作该类的一半数据，而一半子程序操作另一半数据。这其实是两个类伪装成了一个。把它们拆开吧!

尽可能使接口可编程而不是表达语义　每个接口都由一个可编程的部分和一个语义部分构成。可编程部分（programmatic part）由数据类型和接口的其他属性（attributes）构成，可由编译器予以强制（会在编译时检查错误）。接口的语义部分（semantic part）则由关于这个接口应该如何使用的假设构成，这些假设无法由编译器强制。语义接口包括诸如“必须在 RoutineB 之前调用 RoutineA”或者“如果 dataMember1 在传递给 RoutineA 之前没有被初始化，RoutineA 将崩溃”之类的考虑。语义接口要用注释来说明，但要尽量保证不看这些说明也能理解接口。接口的任何方面如果不能被编译器强制，这个方面就可能被误用。想一些办法，利用断言（assert）或其他技术，将接口的语义元素转换为接口的可编程元素。

关联参考　要更多地了解如何在修改代码时保持代码质量，请参见第 24 章“重构”。

小心接口的抽象在修改过程中被侵蚀　对类进行修改和扩展的时候，经常发现需要额外的功能，这些功能与原来的类接口不完全吻合，但似乎又很难用其他方式实现。例如，Employee 类可能演化成这样：

C++示例：类接口在维护过程中被不断侵蚀

```
class Employee {
public:
   ...
   // public routines
   FullName GetName() const;
   Address GetAddress() const;
   PhoneNumber GetWorkPhone() const;
   ...
   bool IsJobClassificationValid( JobClassification jobClass );
   bool IsZipCodeValid( Address address );
   bool IsPhoneNumberValid( PhoneNumber phoneNumber );

   SqlQuery GetQueryToCreateNewEmployee() const;
   SqlQuery GetQueryToModifyEmployee() const;
   SqlQuery GetQueryToRetrieveEmployee() const;
   ...
private:
   ...
};
```

起初清晰的抽象，现在已经演变成只有松散联系的一些函数的大杂烩了。在雇员和检查邮编、电话号码或职位的子程序之间没有逻辑联系。暴露 SQL 查询细节的子程序比 Employee 类的抽象级别要低得多，它们破坏了 Employee 的抽象。

不要添加与接口抽象不一致的公共成员　每当为类的接口添加子程序时，都要问：“这个子程序是否与现有接口提供的抽象一致？”如果不一致，就找一种不同的方式来修改，并保持抽象的完整性。

把抽象和内聚放在一起考虑　抽象和内聚的概念密切相关——呈现良好抽象的类接口通常有很强的内聚性。具有强大内聚性的类往往也会呈现出良好的抽象，虽然这种关系不如前者那么强。

我发现，相较于关注类的内聚性，若关注类的接口所呈现的抽象性，往往能为类的设计提供更多见解。如发现一个类的内聚性很弱，但又不知道如何去纠正，就换个角度问：这个类是否呈现了一致的抽象？

良好的封装

> 交叉参考　关于封装的更多内容，请参见第 5.3 节中的“封装实现细节”。

5.3 节讲过，封装是比抽象更强大的概念。抽象通过提供一个让你忽略实现细节的模型来管理复杂性。封装是一个执法官，它阻止你看到细节，想看也不行。

两个概念之所以相关，是因为如果没有封装，抽象往往会被打破。根据我的经验，要么同时拥有抽象和封装，要么两个都没有。没有中间地带。

> 区分设计良好和不良的模块最重要的因素在于，该模块在多大程度上对其他模块隐藏了它的内部数据和其他实现细节。　——Joshua Bloch

最小化类和成员的可访问性　最小化可访问性是旨在鼓励封装的若干规则之一。如果不确定一个特定的子程序应该公共、私有还是受保护，有一个学派认为应支持可行的、最严格的隐私级别（Meyers 1998, Bloch 2001）。我认为这是一个很好的指导原则，但我认为更重要的原

则是：“什么能最好地保护接口抽象的完整性？”若将子程序公开的做法是和抽象一致的，那么公开它可能是好的。如果不确定，一般多隐藏比少隐藏好。

不要公开成员数据　公开成员数据破坏了封装，而且限制了你对抽象的控制。正如 Arthur Riel 所指出的，若一个 Point 类公开了以下成员：

```
float x;
float y;
float z;
```

它就破坏了封装，因为客户端代码可自由操作 Point 的数据，而 Point 甚至不一定知道它的值何时被改变（Riel 1996）。但是，若 Point 类公开的是以下成员：

```
float GetX();
float GetY();
float GetZ();
void SetX( float x );
void SetY( float y );
void SetZ( float z );
```

则是在保持完美的封装。你不知道底层实现是否以 float x，y 和 z 为单位，不知道 Point 是否将这些数据项存储为 double 并将其转换为 float，也不知道 Point 是否将它们存储在月球上并从外层空间的卫星上检索它们。

避免将私有实现细节放到类的接口中　如果是真正的封装，程序员根本看不到实现细节。它们会隐藏起来——无论是形象地说，还是从字面意义上说。但在包括 C++在内的一些流行语言中，语言的结构要求程序员在类的接口中暴露实现细节，如下例所示：

C++示例：暴露类的实现细节

```
class Employee {
public:
   ...
   Employee(
      FullName name,
      String address,
      String workPhone,
      String homePhone,
      TaxId taxIdNumber,
      JobClassification jobClass
   );
   ...
   FullName GetName() const;
   String GetAddress() const;
   ...
private:
   String m_Name;
   String m_Address;
   int m_jobClass;
   ...
};
```

这里是暴露的实现细节

在类的头文件中包括 private 声明，这似乎只是小小地违反了封装，但它是在鼓励其他程序员检查实现细节。在本例中，客户端代码本应使用 JobClassification 类型表示职位（job class），但头文件暴露了职位是作为整数存储这一实现细节。

Scott Meyers 在《*Effective C++*》第 2 版(Meyers 1998)的第 34 条中描述了一个解决该问题的常规方法。将类的接口和类的实现分开，然后在类的声明中包含指向类的实现的一个指针，但不包含其他任何实现细节。

C++示例：隐藏类的实现细节

```
class Employee {
public:
   ...
   Employee( ... );
   ...
   FullName GetName() const;
   String GetAddress() const;
   ...
private:
   EmployeeImplementation *m_implementation;
};
```

实现细节隐藏在指针后面了

现在可以将实现细节放在 EmployeeImplementation 类中，它应该只对 Employee 类可见，而不是对使用 Employee 类的任何代码都可见。

如果已经为项目写了许多没有采用这种方法的代码，可能会觉得不值得为了采用这种方法而转换大量现有的代码。但是，在看公开了其实现细节的代码时，就要克制住梳理类接口的 private 部分寻找实现线索的冲动。

不要对类的用户做出预设 类的设计和实现应遵守类接口所隐含的契约。除了接口文档所提供的内容，不要对接口的使用方式做出其他任何预设。如果提供下面这样的注释，就表明类对用户做出了过多的预设：

```
-- 将 x、y 和 z 初始化为 1.0，因为如果初始化为 0.0，
-- DerivedClass 就会崩溃。
```

避免友元类 在少数情况下，例如 State 模式，可谨慎使用友元类（friend class）以管理复杂性（Gamma et al. 1995）。但一般情况下，友元类违反了封装。它们增大了你在任何时候要考虑的代码量，从而增加了复杂性。

不要因为子程序只使用了公共子程序就把它放到公共接口中 子程序是否只使用公共子程序这一事实并不是重要的考虑因素。相反，要问的是：公开该子程序是否与接口所呈现的抽象一致？

倾向于读代码方便而不是写代码方便 即使在最初的开发阶段，读代码的频率也要远远多于写代码的频率。牺牲读代码的方便性来换取写代码的方便性，这非常不可取。这一点尤其适合类接口的创建。有的时候，即使一个子程序不符合接口的抽象，你也会想在接口中加入它，以方便当时正在开发的类的特定客户端代码。但是，添加这个子程序是走下坡路的开始，还是不要迈出这一步为好。

> 如果必须看底层实现才能理解发生的事情，就没有抽象性可言。——P. J. Plauger

格外警惕在语义上破坏封装 我一度以为，学会如何避免语法错误就可以万事大吉。但我很快发现，学会如何避免语法错误，只是为我买了一张通往全新的编码错误剧场的门票，其中大多数错误比语法错误更难诊断和纠正。

语义的封装难度和语法的封装相近。在语法上，只需将类内部的子程序和数据声明为 private，就可以相对容易地避免窥探另一个类的内部工作情况。实现语义上的封装则完全是两码事。以下是类的用户在语义上破坏封装的一些例子：

- 不调用类 A 的 InitializeOperations()子程序，因为你知道类 A 的 PerformFirstOperation()子程序会自动调用它。
- 调用 employee.Retrieve(*database*)之前不调用 *database*.Connect()子程序，因为你知道 employee.Retrieve()函数会在没有建立连接的前提下自动连接数据库。
- 不调用类 A 的 Terminate()子程序，因为你知道类 A 的 PerformFinalOperation()子程序已经调用过它了。
- 即使在 ObjectA 离开作用域后，也使用由 ObjectA 创建的到 ObjectB 的指针或引用，因为你知道 ObjectA 将 ObjectB 放在静态存储中，ObjectB 依然有效。
- 使用类 B 的 MAXIMUM_ELEMENTS 常量，而不是使用 ClassA.MAXIMUM_ELEMENTS，因为你知道它们都等于同一个值。

KEY POINT 这些例子的问题在于，它们使客户端代码不是依赖于类的公共接口，而是依赖于其私有实现。任何时候注意到自己需要看一个类的实现才能弄清楚如何使用该类，就不是在为接口编程了；而是在“通过”接口为实现编程。如果是通过接口编程，封装就被打破了，一旦封装被打破，接着就轮到抽象了。

如果不能仅仅根据类的接口文档来搞清楚如何使用该类，正确反应不是调出源代码并查看实现。意图很好，方法却是错的。正确的反应是联系该类的作者，说：“我不知道如何使用这个类”。类作者的正确反应不是直接回答你的问题。相反，是签出类接口文件，修改类接口文档，签入文件，然后跟你说：“现在看看能不能理解它是怎么工作的。”像这样的对话你希望发生在接口代码本身，这样能留下来给未来的程序员看。你不希望这种对话只发生在你自己的脑海中，因为这样会将微妙的语义依赖植入使用该类的客户代码中。你也不希望这个对话发生在人和人之间，这样它就只适合你的代码，而不适合其他人的。

警惕过于紧密的“耦合” “耦合”（coupling）是指两个类的联系有多紧密。通常，联系越松（loose）越好。针对这个概念有几个常规的指导原则：

- 最小化减少类和成员的可访问性。
- 避免使用 friend（友元）类，因其紧密耦合。
- 将基类中的数据变成 private 而不是 protected，使派生类与基类的耦合不那么紧密。
- 避免在类的公共接口中公开成员数据。
- 警惕在语义上破坏封装。

- 遵守“得墨忒耳法则”（在本章第 6.3 节介绍）。

耦合与抽象/封装相辅相成。若抽象出现漏洞，或者封装被破坏，就会发生紧密耦合的情况。如果类提供了一套不完整的服务，其他子程序可能发现自己需要直接读取或写入该类的内部数据。这样类就被开了一个口子，使它变成了一个玻璃盒子，而不是一个黑盒子，而且类的封装性几乎被完全消除了。

6.3 设计和实现问题

定义良好的类接口对创建高质量的程序有很大的帮助。类内部的设计和实现也很重要。本节讨论了与包含、继承、成员函数和数据、类耦合、构造函数以及值对象/引用对象相关的问题。

包含(“has a”关系)

KEY POINT 包含（containment）是一个简单的概念，即类中包含一个基本数据元素或对象。

关于继承的文章比关于包含的文章多得多，但那是由于继承需要更多的技巧，更容易出错，而不是由于它更好。“包含”是面向对象编程的主力技术。

通过包含实现“has a”关系 可将“包含”想象成一种“has a”(有一个)关系。例如，雇员“有一个”姓名，“有一个”电话号码，“有一个”税号等等。通常可以使姓名、电话号码和税号成为 Employee 类的成员数据来建立这种关系。

除非万不得已，否则不要通过私有继承实现“has a”关系 有时可能无法通过使一个对象成为另一个对象的成员来实现包含。针对这种情况，一些专家建议以私有方式从被包含的对象继承（Meyers 1998, Sutter 2000）。这样做的主要原因是使负责包含的类能访问被包含类的受保护成员函数或受保护成员数据。但在实践中，这个方式与祖先类建立了一种过于舒适的关系，破坏了封装。应将这种方式视为设计上存在错误的一种警告信号，应通过私有继承以外的其他方式予以解决。

警告数据成员超过 7 个的类 “7±2”这个数字被认为是一个人在执行其他任务时能记住的离散项目的数量（Miller 1956）。如果一个类包含的数据成员超过 7 个，就要考虑该类是否应分解成多个小类（Riel 1996）。如果数据成员是整数和字符串这样的基本数据类型，“7±2”可以往多的算；如果数据成员是复杂的对象，则往低的算。

继承(“is a”关系)

继承是指一个类是另一个类的“特化”（specialization）。继承的目的是定义一个基类，在其中包含两个或多个派生类的通用元素，从而简化代码的编写。这些通用元素可以是子程序接口、实现、数据成员或数据类型。继承将代码和数据集中于基类，避免在多处重复。

决定使用继承时，必须做出几个决定：

- 每个成员子程序是否对派生类可见？是否有默认实现？默认实现是否可被覆盖（override，或称“重写”）？
- 每个数据成员（包括变量、具名常量、枚举等）是否对派生类可见？

下面几个小节将详细解释做这些决定时要考虑的东西。

> “用 C++语言进行面向对象编程时，最重要的法则就是：public 继承意味着‘is a’关系。将这一法则铭记于心。”——Scott Meyers

通过 public 继承实现“is a”关系 若程序员决定通过继承现有的类来创建一个新类，其实就是说新类是旧类的一个更具体的版本（特化）。基类设定了关于派生类如何运作的预期，并对派生类如何运作施加了限制（Meyers 1998）。

如派生类不打算完全遵守基类所定义的相同接口契约，继承就不是正确的实现技术。请考虑使用“包含”（has a），或在继承层次结构的更上层进行修改。

要么设计继承并提供文档说明，要么禁止继承 继承会增加程序的复杂性，所以是一种危险的技术。正如 Java 大师 Joshua Bloch 所说：“要么设计继承并提供文档说明，要么禁止继承”（Design and document for inheritance, or prohibit it）。如果一个类不是为了被继承而设计的，在 C++中就要把它的成员定义成非 virtual，在 Java 中定义成 final，在 Microsoft Visual Basic 中定义成 NotOverridable，从而禁止从该类继承。

遵循里氏替换原则（Liskov Substitution Principle，LSP） Barbara Liskov 在一篇面向对象编程的开创性论文中提出，除非派生类真的“is a”（是一个）基类更具体的版本，否则不应从基类继承（Liskov 1988）。Andy Hunt 和 Dave Thomas 这样总结 LSP：“子类必须能通过基类的接口来使用，使用者无需知道两者的差异。”（Hunt and Thomas 2000）。

换言之，基类中定义的所有子程序在任何派生类中使用时，其含义都应该相同。

假设 Account 基类有三个派生类：CheckingAccount、SavingsAccount 和 AutoLoanAccount。在 Account 的任何子类型上，程序员都可调用从 Account 继承的任何子程序，而无需关心一个特定的账户对象到底是什么子类型（无论是支票、储蓄还是汽车贷款账户）。

程序员应该能调用这三个派生类中从 Account 继承而来的任何一个子程序，而无须关心到底使用的是 Account 哪一个派生类的对象。

如程序遵循里氏替换原则，继承就能成为降低复杂性的一种强大工具，因为它能让程序员专注于对象的一般特性而不必关心细节。如程序员必须不断思考子类的实现在语义上的差异，继承会增大而不是降低复杂性。

假设程序员必须这样想：“如果在 CheckingAccount 或 SavingsAccount 上调用 InterestRate()例程，它返回银行支付给消费者的利息，但如果在 AutoLoanAccount 上调用 InterestRate()，就必须修改正负号，因为它返回消费者向银行支付的利息。”若遵循 LSP，本例的 AutoLoanAccount 就不应该从 Account 基类继承，因为 InterestRate()子程序的语义与基类的 InterestRate()子程序的语义不同。

确保只继承想要继承的东西 派生类可继承成员子程序接口、实现或同时继承两者。表 6-1 总结了对子程序进行实现和覆盖（override）的各种方式。

表 6-1　继承而来的子程序的各种变化

	可覆盖	不可覆盖
提供默认实现	可覆盖的子程序	不可覆盖的子程序
未提供默认实现	抽象且可覆盖的子程序	未使用（未定义又不让覆盖的子程序没有意义）

如表所示，继承而来的子程序有三种基本形式：

- 抽象且可覆盖的子程序（abstract overridable routine）是指派生类只继承子程序的接口，但不继承其实现。
- 可覆盖的子程序（overridable routine）是指派生类继承子程序的接口及其默认实现，并且可以覆盖该默认实现。
- 不可覆盖的子程序（non-overridable routine）是指派生类继承子程序的接口及其默认实现，但不允许覆盖该默认实现。

选择通过继承来实现一个新类时，需考虑每个成员子程序的继承方式。不要因为继承了接口就一定要继承实现，或者因为要继承实现就一定要继承接口。如果只想使用类的实现而不是它的接口，应使用“包含”（has a）而不是“继承”（is a）。

不要“覆盖”不可覆盖的成员函数　C++和 Java 都允许程序员以某种形式覆盖不可覆盖的成员子程序。如果函数在基类中是私有的，派生类就可创建一个同名函数。对于看派生类代码的程序员来说，这样的函数会造成混乱，因为它看起来应该是多态的，但实际又不是，只是恰好同名而已。这个指导原则还有另一种说法：“不要在派生类中重用不可覆盖的基类子程序名称”。

将通用接口、数据和行为移到继承树中尽可能高的位置　这些东西移得越高，派生类就越容易使用它们。到底要多高？以抽象性为准。如发现将一个子程序移到高处会破坏高处对象的抽象性，就应该停手了。

对仅一个实例的类持怀疑态度　单一实例可能表明设计中混淆了对象和类。考虑一下是否能直接创建一个对象而不是新类。派生类的差异是否可以用数据来表示，而不是非要作为一个不同的类？这个指导原则最引人注目的一个例外就是 Singleton 模式。

对仅一个派生类的基类持怀疑态度　当我看到某个基类只有一个派生类时，我会怀疑有的程序员做了“提前设计”——试图预测未来的需求，但通常没有完全理解这些未来的需求是什么。要为将来做准备，最好的办法不是设计“某天可能会要到”的额外基类层级，而是使当前的工作尽可能清晰、直接和简单。这意味着不要创建非绝对必要的继承结构。

对覆盖了子程序，但在子程序的派生版本中什么都不做的类持怀疑态度　这通常表明基类设计存在错误。例如，假设有一个 Cat 类和一个 Scratch()子程序，再假设最终发现有的猫已经去爪（declawed），不能做抓挠（scratch）这个动作。这个时候，你可能会想创建一个从 Cat 派生的 ScratchlessCat 类，并覆盖 Scratch()子程序，使其不做任何事情。这样做会带来几个问题：

- 它破坏了 Cat 类所呈现的抽象（接口契约），改变了其接口的语义。

- 若继续将这种做法扩展到其他派生类，局面很快就会失控。发现一只没尾巴的猫怎样办？一只不捉耗子的猫？一只不喝牛奶的猫？最终会得到像ScratchlessTaillessMicelessMilklessCat（不能抓挠、没尾巴、不捉耗子、不喝牛奶的猫）这样的派生类。
- 随着时间的推移，这种做法会导致代码的维护变得混乱，因为祖先类的接口和行为几乎没有为其后代的行为提供任何线索。

解决这个问题的地方不是在派生类，而是在最初的 Cat 类中。创建一个 Claw（爪子）类并将其“包含”在 Cat 类中。问题的根源是假设所有猫都会“抓挠”，所以要从源头上解决这个问题，而不是“头痛医头，脚痛医脚”。

避免过深的继承树 面向对象编程提供了许多技术来管理复杂性。但每个强大的工具都有其危害，一些面向对象技术甚至有增加而不是降低复杂性的倾向。

Arthur Riel 在他的优秀著作《*Object-Oriented Design Heuristics*》（1996）中建议将继承层级限制在最多 6 层。Riel 的建议建立在“神奇数字 7±2”的基础上，但我认为这过于乐观。依我的经验，大多数人都很难在大脑中同时处理超过两到三层的继承关系。“神奇数字 7±2”或许更适合限制基类的子类总数，而不是限制继承树的层级数。

人们发现，深的继承树与故障率的增加有很大的联系（Basili, Briand, and Melo 1996）。任何曾尝试过调试复杂继承层次结构的人都知道原因。深的继承树增加了复杂性，而这恰恰与继承所要达到的目的背道而驰。牢记自己的首要技术使命。确保自己是用继承来避免重复的代码，并尽量降低复杂性。

尽量利用多态而不是全面的类型检查 经常重复的 case 语句有时表明，继承可能是一个更好的设计选择——虽然这并非肯定成立。下面是应该采用面向对象方法的一个经典例子：

C++示例：这些case语句或许应该用多态性来替代

```
switch ( shape.type ) {
   case Shape_Circle:
      shape.DrawCircle();
      break;
   case Shape_Square:
      shape.DrawSquare();
      break;
   ...
}
```

在这个例子中，对 shape.DrawCircle()和 shape.DrawSquare()的调用应该被一个名为 shape.Draw()的子程序所取代；无论形状是圆是方，都可调用该子程序。

但有的时候，case 语句被用来分隔真正不同种类的对象或行为。下面是一个在面向对象程序中使用得当的 case 语句：

C++示例：这些 case语句或许不应该用多态性来替代

```
switch ( ui.Command() ) {
   case Command_OpenFile:
      OpenFile();
      break;
   case Command_Print:
      Print();
      break;
   case Command_Save:
      Save();
      break;
   case Command_Exit:
      ShutDown();
      break;
   ...
}
```

在这个例子中，可创建一个允许派生的基类，并为每个命令创建一个多态的 DoCommand()子程序（和 Command 模式一致）。但对于这个简单的例子，DoCommand()会被淡化至毫无意义，case 语句才是更容易理解的解决方案。

让所有数据 private 而不是 protected　正如 Joshua Bloch 所说的，“继承破坏了封装”（2001）。从对象继承，就获得了对该对象的 protected 子程序和数据的特许访问权。如派生类真的需要访问基类的属性，请改为在基类中提供 protected 访问函数（accessor）。

多重继承

> “C++多重继承一个毋庸置疑的事实就是，它打开了潘多拉的盒子，里面全都是单一继承所没有的复杂性。”——Scott Meyers

继承是一种电动工具（power tool）。这类似于用链锯而不是手锯伐木。如果小心地使用，它可能非常有用。但是，在不采取适当预防措施的人手中，它又显得很危险。

如果说继承是一把链锯，那么多重继承（multiple inheritance）就是一把 20 世纪 50 年代的链锯，没有护罩，不支持自动关闭，马达也难以伺候。这样的工具有时还是有价值的；但大多数时候，最好还是把它留在车库里以绝后患。

虽然一些专家建议广泛使用多重继承（Meyer 1997），但依我的经验，多重继承主要用于定义“mixins”（混合体），即用来给对象增加一组属性的简单类。之所以取“mixins”这个名字，是因为它们允许属性被“混入”（mix in）到派生类中。mixins 可以是像 Displayable（可显示）、Persistant（持久化）、Serializable（可序列化）或 Sortable（可排序）这样的类。mixins 几乎都是抽象的，不打算独立于其他对象进行实例化。

mixins 需要使用多重继承，但只要所有 mixins 都真正独立于彼此，就不会出现多重继承特有的经典钻石继承[3]问题。由于属性是“串”（chunking）到一起，它们还使设计更容易理解。如果对象使用了 Displayable 和 Persistent 这两个 mixins，那么相较于使用了 11 个更具体的子程序的对象（需要这些子程序来实现那两个属性），前者显得更容易理解。

Java 和 Visual Basic 已经意识到了 mixins 的价值。为此，它们允许了接口的多重继承，但类还是只允许单继承。C++则同时支持接口和实现的多重继承。程序员只有在仔细考虑了替代方案，并权衡了对系统复杂性和可理解性的影响之后，才应该使用多重继承。

继承的规则为何如此之多？

KEY POINT 本节介绍了许多避免继承出麻烦的规则。所有这些规则最基本的一点在于，继承往往与你作为程序员的首要技术使命（管理复杂性）相违背。为了控制复杂性，应对继承持十分警惕的态度。下面总结了何时使用“继承”（is a）以及何时使用“包含”（has a）：

关联参考　要更多地了解复杂性，请参见第 5.2 节中的“软件的首要技术使命：管理复杂性”。

- 如多个类有共同的数据，但没有共同的行为，就创建一个共同的对象供这些类包含。
- 如多个类有共同的行为，但没有共同的数据，就从定义了共同子程序的一个共同基类中派生出这些类。
- 如多个类有共同的数据和行为，就从定义了共同数据和子程序的一个共同基类中继承。
- 如果希望由基类控制你的接口，选择继承；如果想控制自己的接口，选择包含。

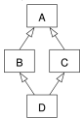
成员函数和数据

关联参考　第 7 章就子程序的常规主题进行了更多讨论。

下面是一些有效实现成员函数和成员数据的指导原则：

尽量减少类中的子程序数量　对 C++程序的研究发现，每个类中较多的子程序数量与较高的故障率有关（Basili, Briand, and Melo 1996）。但是，其他竞争因素更重要，包括深的继承树、类内调用的大量子程序以及类之间的强耦合。一边要最小化子程序数量，另一边也要照顾到其他这些因素。

[3] 译注：钻石继承（也称为菱形继承）是指当两个类 B 和 C 继承自 A，而类 D 同时继承自 B 和 C 时产生的一种歧义。如果 A 中有一个方法被 B 和 C 覆盖了，而 D 没有覆盖它，那么 D 继承哪个版本的方法：B 的，还是 C 的？



禁止隐式生成不需要的成员函数和操作符 有时需要禁止某些函数——可能想禁止赋值，或者不想让一个对象被构造。你可能以为，既然编译器自动生成了操作符，就只能允许访问。但在这些情况下，可通过将构造函数、赋值操作符或其他函数/操作符声明为 private 来阻止客户访问它。使构造函数 private 是定义单例（singleton）类的标准技术，本章后面会讨论该问题。

尽量减少类调用的不同子程序的数量 一项研究表明，类中的错误数量与类中所调用的子程序总数在统计学意义上相关（Basili, Briand, and Melo 1996）。同一项研究发现，类使用的类越多，其故障率往往越高。这些概念有时称为“扇出”（fan out）[4]。

深入阅读 关于得墨忒耳法则的更多内容，推荐阅读三本著作：*Pragmatic Programmer*（Hunt and Thomas2000）、*Applying UML and Patterns*（Larman 2001）和 *Fundamentals of Object-Oriented Design in UML*（Page-Jones 2000）。

尽量减少对其他类的间接子程序调用 直接连接就已经很危险了，而间接连接（例如 account.ContactPerson().DaytimeContactInfo().PhoneNumber()）往往更危险。研究人员总结出了一条“得墨忒耳法则[5]”（Law of Demeter）（Lieberherr and Holland 1989），意思是对象 A 可调用它自己的任何子程序。如对象 A 实例化了一个对象 B，它可以调用对象 B 的任何子程序。但是，它应避免在对象 B 提供的对象上调用子程序。在上面的账户例子中，这意味着调用 account.ContactPerson()可以，但不可以调用 account.ContactPerson().DaytimeContactInfo()。

这只是一个简化的解释。更多细节请参见本章末尾的“更多资源”。

通常要尽量减少一个类与其他类的协作程度 要尽量减少以下数值：

- 实例化的对象种类。
- 在实例的对象上进行的各种直接子程序调用的数量。
- 在其他实例化的对象所返回的对象上的子程序调用。

构造函数

以下是专门针对构造函数的一些指导原则。这些指导原则在不同语言（C++，Java 和 Visual Basic 等）中非常相似。析构函数的差别要大一些，所以请查看本章最后的“更多资源”了解关于析构函数的详情。

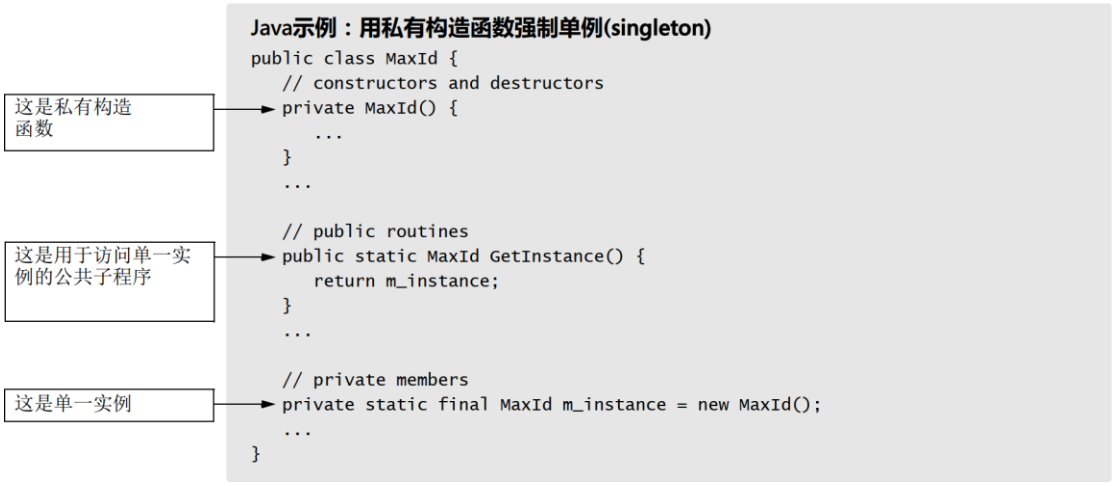
尽可能在所有构造函数中初始化所有成员数据 在所有构造函数中初始化所有数据成员是一种低成本的防御性编程实践。

深入阅读 在 C++语言中实现这个的代码十分类似。详情请参阅《*More Effective C++*》中的中的第 26 条（Meyers 1998）。

[4] 译注：类或方法的“扇出”是指该类所使用的其他类的数量或该方法所调用的其他方法的数量。

[5] 译注：得墨忒耳法则可简单地陈述为“只使用一个.操作符”。所以，a.b.Method()违反了此法则，而 a.Method()不违反。一个简单的例子是，人可以命令一条狗行走（walk），但是不应直接指挥狗的腿行走，应该由狗去指挥控制它的腿如何行走。

使用 private 构造函数强制单例（singleton）属性 为了强制类只能实例化一个对象，可隐藏该类的所有构造函数，再提供一个静态 GetInstance()子程序来访问该类的单一实例，如下例所示：



Java示例：用私有构造函数强制单例(singleton)

```
public class MaxId {
   // constructors and destructors
   private MaxId() {
      ...
   }
   ...

   // public routines
   public static MaxId GetInstance() {
      return m_instance;
   }
   ...

   // private members
   private static final MaxId m_instance = new MaxId();
   ...
}
```

只有初始化静态对象 m_instance 时才会调用该 private 构造函数。采用这种方法，如果想引用 MaxId 单例，引用 MaxId.GetInstance()即可。

除非论证可行，否则使用深拷贝而不是浅拷贝 你对复杂对象所做的主要决定之一是，是要实现对象的深拷贝（deep copy）还是浅拷贝（shallow copy）？对象的深拷贝是指成员数据也要逐一拷贝；浅拷贝则通常只拷贝引用（指针），不拷贝它们所指向的数据——虽然“深”和“浅”的具体含义有时会有区别。

创建浅拷贝的动机通常是为了提升性能。虽然创建大型对象的多个拷贝可能引起审美疲劳，但很少会引起任何可衡量的性能损失。少许对象可能出现性能问题，但在猜测是哪些代码真正导致了问题方面，程序员是出了名的差(详见第 25 章“代码调优策略”)。

既然糟糕的权衡反而会为可疑的性能提升带来复杂性，所以在深拷贝和浅拷贝的问题上，一个好的办法是除非证明浅拷贝更佳，否则无脑使用深拷贝。

深拷贝比浅拷贝更容易编码和维护。除了两种对象都要包含的代码之外，浅拷贝还增加了用于引用计数的代码，以确保安全拷贝对象、安全比较对象、安全删除对象等等。这种代码很容易出错，除非有令人信服的理由，否则应尽量避免。

如确实需要使用浅拷贝，Scott Meyers 的《*More Effective C++*》（1996）的第 29 条包含了对 C++中的这些问题的精辟讨论。Martin Fowler 的《*Refactoring*》（1999）描述了从浅拷贝到深拷贝以及从深拷贝到浅拷贝转换所需的具体步骤(Fowler 将浅拷贝称为“引用对象”，将深拷贝称为“值对象”)。

6.4 创建类的理由

关联参考 创建类的理由和创建子程序的理由有共同之处，详情参见第 7.1 节。

关联参考　关于识别现实世界对象的更多信息，请见第 5.3 节中的“找出现实世界中的对象”。

如果你看什么就相信什么，或许会以为创建类唯一的理由是建模现实世界的对象。实际上，创建类的理由远不止这个。下面列出了创建类的一些很好的理由。

建模现实世界的对象　对现实世界的对象进行建模或许不是创建类的唯一理由，但仍然是一个很好的理由！为程序建模的每种现实世界的对象都创建一个类。将对象所需的数据放入类中，然后构建服务子程序来模拟对象的行为。第 6.1 节在讨论 ADT 时提供了许多例子。

建模抽象对象　创建类的另一个很好的理由是建模抽象对象——一种不具体的、现实世界的对象，但它提供了对其他具体对象的抽象。一个很好的例子是经典的 Shape（形状）对象。Circle 和 Square 是具体存在的形状，而 Shape 是对其他具体形状的抽象。

在编程项目中，抽象并不像 Shape 那样是现成的，所以必须更努力地归纳出清晰的抽象。从现实世界的实体提炼抽象概念的过程是不确定的，不同设计人员会抽象出不同的共性。例如，如果我们不知道圆形、正方形和三角形等几何形状，我们可能会想出更多不寻常的形状，例如南瓜形、大头菜和 Pontiac Aztek（一种车型）形。想出合适的抽象对象是面向对象设计的主要挑战之一。

KEY POINT **降低复杂性**　创建类最重要的原因是降低程序的复杂性。创建类来隐藏信息，这样不用去考虑它们了。当然，写类的时候还是考虑这些信息。但写完之后，应该就能忘记这些细节，在不关心内部运作的情况下使用该类。创建类的其他理由——最小化代码规模、提高可维护性和改善正确性——也都是很好的理由，但如果没有类的抽象能力，复杂的程序就管理不过来。

隔离复杂性　所有形式的复杂性（复杂的算法、大的数据集、复杂的通信协议等等）都容易出错。如果一个错误真的发生了，如果它不是在代码中蔓延，而只是在一个类的局部出现，就更容易发现。修复错误所产生的变化不会影响其他代码，因为只有一个类需要修复，其他代码不会被波及。如果发现一个更好、更简单或更可靠的算法，如果它被隔离在一个类中，那么替换旧算法会更容易。在开发过程中，尝试几种设计并保留效果最好的那一种会更容易。

隐藏实现细节　隐藏实现细节是创建类的绝佳理由，无论这些细节是像数据库访问那么复杂，还是像数据成员是用数字还是字符串来存储那么简单。

限制变化造成的影响　隔离可能发生变化的区域，将变化的影响限制在一个或几个类的范围中。设计时要使最有可能改变的领域最容易改变。可能改变的领域包括硬件依赖性、输入/输出、复杂的数据类型以及业务规则。5.3 节中的“隐藏秘密（信息隐藏）”描述了几种常见的变化来源。

交叉参考　第 13.3 节讨论了使用全局数据时的一些问题。

隐藏全局数据　如需使用全局数据，可将其实现细节隐藏在一个类接口背后。与直接使用全局数据相比，通过专门的访问子程序（access routines）来使用全局数据有几个好处。可以改变数据的结构而不改变程序。可以监控对数据的访问。每次都要使用访问子程序，这也会鼓励你去思考数据是否真的是全局性的；经常都会发现所谓的“全局数据”实际只是对象数据。

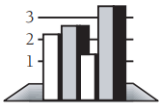
简化参数传递 如果需要在几个子程序之间传递一个参数，可能表明应将这些子程序纳入一个类，将参数作为对象数据来共享。简化参数传递本身并不是目标，但大量数据传来传去，表明换成一种不同的类组织方式可能更佳。

交叉参考 参见 5.3 节中的“隐藏秘密（信息隐藏）”详细了解信息隐藏。

建立中心控制点 每个任务都在一个地方控制是个好主意。控制有很多形式。对表中条目数量的了解是一种形式。对设备（文件、数据库连接、打印机等等）的控制是另一种形式。使用类来读写数据库也是一种集中控制形式。如果将数据库转换为一个平面文件或内存数据，这些变化只会对一个类产生影响。

集中控制的概念类似于信息隐藏，但它具有独特的启发式能力，值得加入你的编程工具箱。

使代码更容易重用 相较于将代码塞入一个大类，如果将相同的代码放入良好分解（well-factored）的类中，在其他程序中会更容易重用。即便一段代码仅在程序的一个地方调用，并且可以理解为大类的一部分，但只要这段代码可能在另一个程序中使用，把它放到自己的类中就是有意义的。



HARD DATA NASA 的“软件工程实验室”研究了十个积极追求重用的项目（McGarry, Waligora, and McDermott 1989）。不管采用的是面向对象的方法，还是面向功能的方法，最初的项目都不能从以前的项目中获取很多代码，因为以前的项目没有建立充分的代码库。后来，采用功能化设计的项目能从以前的项目中获取大约 35%的代码。使用面向对象方法的项目能从以前的项目中获得超过 70%的代码。如果能通过提前规划来避免写 70%的代码，就放手去做!

关联参考 关于实现最少所需功能的详情，请参见第 24.2 节中的“程序包含的代码似乎有一天会被需要”。

值得注意的是，NASA 创建可重用类的方法的核心并不涉及“为重用而设计”。相反，NASA 是在其项目结束时确定要重用的候选者。然后，他们进行必要的工作，使这些类在主项目的最后作为一个特殊项目来重用，或作为新项目的第一步来重用。这种方法有助于防止“镀金”——创建不需要的功能，从而不必要地增加复杂性。

为程序家族做计划 如预期一个程序会被修改，最好将你预期改变的部分隔离出来，把它们放到自己的类中。然后，就可以在不影响程序其余部分的情况下修改这些类，或者可以加入全新的类。不要只是想一个程序会是什么样子，还要想整个程序家族（family of programs）会是什么样子，这是一个强大的启发式方法，可以预测到完整的更改类别（entire categories of changes）（Parnas 1976）。

我几年前管理着一个团队，负责写一系列程序供我们的客户销售保险。我们必须根据特定客户的保险费率、报价-报表格式等来定制每个程序。但这些程序的许多部分是相似的，包括输入潜在客户信息的类、在客户数据库中存储信息的类、查询费率的类、计算团体总费率的类等等。团队对程序进行了良好的分解，使每个因客户而异的部分都在自己的类中。最初的编程可能花了三个月左右的时间，但当我们有了新的客户，就只需为新客户写几个新类，然后把它们放到其他代码中。只需干几天的活儿——哇！——定制软件！

打包相关操作 在不能隐藏信息、共享数据或为灵活性规划的情况下，仍然可将一组操作打包成合理的组，例如三角函数、统计函数、字符串处理子程序、位操作子程序、图形子程序等等。类是组合相关操作的一种手段。也可以使用包、命名空间或头文件，具体取决于所用的语言。

实现特定的重构 第 24 章“重构”描述的许多特定的重构都会产生新类，其中包括将一个类转换为两个，隐藏委托，去除中间人以及引入扩展类。为了更好地完成本节描述的任何目标，都可能需要创建新的类。

要避免的类

虽然一般来说类是好东西，但也可能会遇到一些麻烦。下面列举了一些需要避免的类。

避免创建万能类 避免创建无所不知、无所不能的类。如果一个类花了很多时间用 Get()和 Set()子程序从其他类获取数据（换言之，挖掘它们的内部业务，并告诉它们该怎么做），就问自己这些功能是否能更好地组织到其他类中而不是万能类（god class）中（Riel 1996）。

交叉参考 这种类通常称为结构或结构体，详情参见第 13.1 节。

消除无关紧要的类 如果一个类只有数据而无行为，想一想它是否真的是一个类，并考虑将其降级，使其成员数据成为一个或多个其他类的属性。

避免以动词命名的类 一个只有行为而无数据的类通常不是真正的类。考虑将负责 DatabaseInitialization()或 StringBuilder()等操作的类变成其他类的子程序。

总结：创建类的理由

下面总结了创建类的合理理由：

- 建模现实世界的对象
- 建模抽象对象
- 降低复杂性
- 隔离复杂性
- 隐藏实现细节
- 限制变化造成的影响
- 隐藏全局数据
- 简化参数传递
- 创建中心控制点
- 使代码更容易重用
- 为程序家族做计划
- 打包相关操作

- 实现特定的重构

6.5 语言特定问题

不同编程语言对类的处理方法不同，其中许多还很有趣。以如何覆盖（override）成员子程序，从而在派生类中实现多态性为例。在 Java 中，所有子程序默认可覆盖，子程序必须声明为 final 才能防止派生类覆盖它。在 C++中，子程序默认不可覆盖。要允许覆盖的子程序必须在基类中声明为 virtual。在 Visual Basic 中，要允许覆盖的子程序必须在基类中声明为 Overridable，派生类还必须使用 Overrides 关键字。

下面这些与类有关的领域因语言的不同而呈现出很大差异：

- 继承树中被覆盖的构造函数和析构函数的行为。
- 构造函数和析构函数在异常处理情况下的行为。
- 默认构造函数（无参构造函数）的重要性。
- 析构函数或终结器（finalizer）的调用时机。
- 对语言内置操作符（包括赋值和等于操作符）进行覆盖的智慧。
- 对象在创建和销毁时，或者在它们声明和超出作用域时，如何处理内存？

对这些问题的详细讨论超出了本书的范围，可参考“更多资源”进行延伸阅读。

6.6 超越类：包

关联参考　类和包的更多区别请参见第 5.2 节中的“设计的层次”。

类是程序员当前实现模块化的最佳方式。但模块化是一个很大的主题，它超出了类的范围。过去几十年，软件开发的进步在很大程度上是通过提高我们所要处理的聚合的粒度（granularity of the aggregations）来实现的。我们拥有的第一个聚合是语句，这在当时算得上机器指令的一个大进步。然后是子程序，再后来是类。

很明显，如果有好的工具来聚合对象组，就能更好地支持抽象和封装的目标。Ada 十多年前就支持包（package）的概念，Java 今天也支持包了。如果用一种不直接支持包的语言编程，可创建自己的可怜的程序员版本的包，并通过以下编程标准来强制它：

- 用于区分哪些类是 pubic 的，哪些类供包私用的命名约定。
- 使用命名约定，使用代码组织约定（项目结构），或同时使用两者，以确定每个类归属于哪个包。
- 定义哪些包允许使用其他哪些包的规则，包括是以继承、包含或同时以这两种方式使用。

这些变通方法很好地演示了“在一种语言上编程”（programming in a language）和“深入一种语言去编程”（programming into a language）的区别。关于这种区别的更多信息，请参见第 34.4 节“深入编程，不浮于语言表面”。

关联参考　这个检查清单关注的是类的质量。构建类的步骤列表请参见第 9 章的“检查清单：伪代码编程过程”。

检查清单：类的质量

抽象数据类型

- 是否将程序中的类视为抽象数据类型？是否从这个角度评估了它们的接口？

抽象

- 类是否有一个中心目的？
- 类的命名是否恰当？其名字是否表达了其中心目的？
- 类的接口是否呈现了一致的抽象？
- 类的接口是否让人一眼就知道应该如何使用这个类？
- 类的接口是否足够抽象，使开发者无需考虑它的服务具体是如何实现的？能将类看成是一个黑盒吗？
- 类的服务是否足够完整，使其他类无需摆弄其内部数据？
- 是否已从类中移除了无关信息？
- 是否考虑过把类进一步分解为组件类？是否已经尽可能地分解了？
- 修改类时是否保持了其接口的完整性？

封装

- 是否最小化了类成员的可访问性？
- 类是否避免了公开其成员数据？
- 在编程语言允许的范围内，类是否尽可能对其他类隐藏了其内部实现细节？
- 类的设计是否避免了对其用户（包括其派生类）做出预设。
- 类是否不依赖于其他类？是松耦合的吗？

继承

- 继承是否只用来建立“is a”关系？换言之，派生类是否遵循了里氏替换原则？
- 类的文档是否描述了其继承策略？
- 派生类是否避免了“覆盖”不可覆盖的方法？
- 是否将通用接口、数据和行为都放在继承树尽可能高的地方了？
- 继承树很浅吗？
- 基类中的所有数据成员是否都被定义为 private 而非 protected？

与实现相关的其他问题

- 类的数据成员是否只有 7 个或更少？
- 是否将类中直接和间接调用其他类的子程序的数量减到最少了？
- 类是否只在绝对必要时才与其他类协作？
- 是否所有数据成员都在构造函数中初始化了？
- 是否除非经过论证，否则类都被设计成深拷贝而不是浅拷贝来使用？

语言特定问题

- 针对你所用的编程语言，是否研究过语言特有的和类相关的问题？

更多资源

类的常规话题

Meyer, Bertrand. *Object-Oriented Software Construction*，2d ed. New York, NY: Prentice Hall PTR, 1997。这本书详细讨论了抽象数据类型，并解释了它如何构成类的基础。第 14 章~第 16 章深入讨论了继承。第 15 章提出了支持多重继承的正面论据。

Riel, Arthur J. *Object-Oriented Design Heuristics*. Reading, MA: Addison-Wesley, 1996。这本书就如何改善程序设计给出了大量建议，其中大多从类的角度出发。我有几年一直在回避这本书，因为它看上去实在太庞杂了，有时还在探讨“玻璃屋中的人”这样的主题（意思是五十步别笑百步）。不过，这本书的主体部分只有大约 200 页。作者的写作风格通俗易懂。内容重点突出，也很有实用性。

C++

Meyers, Scott. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*，2d ed. Reading, MA: Addision-Wesley, 1998。

Meyers, Scott, 1996, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Reading, MA: Addison-Wesley, 1996。这两本书都算得上 C++程序员的权威之作。在诙谐之中向我们传授了一位语言大师如何欣赏 C++语言的精妙。

Java

Bloch, Joshua. *Effective Java Programming Language Guide*. Boston, MA: Addison-Wesley, 2001。这本书给出了很多关于 Java 语言的实用建议，还介绍了一些更常规的、好的面向对象实践。

Visual Basic

下面几本书对 Visual Basic 语言中的类进行了非常好的介绍：

Foxall, James. *Practical Standards for Microsoft Visual Basic. NET*. Redmond, WA: Microsoft Press, 2003.

Cornell, Gary, and Jonathan Morrison. *Programming VB. NET: A Guide for Experienced Programmers*. Berkeley, CA: Apress, 2002.

Barwell, Fred, et al. *Professional VB. NET*, 2d ed. Wrox, 2002.

要点回顾

- 类的接口应提供一致的抽象。许多问题都是由于违反该原则而引起的。
- 类的接口应隐藏一些信息，包括系统接口、设计决策或实现细节。
- “包含”（has a）往往比“继承”（is a）更可取，除非必须建模“is a”关系。
- 继承是有用的工具，但它会增加复杂性，这违背了软件的首要技术使命，即“管理复杂性”。
- 类是管理复杂性的首选工具。只有在设计类时给予足够的关注，才能实现这一目标。

第 9 章　伪代码编程过程

内容

- 9.1 类和子程序构建步骤总结
- 9.2 面向专家的伪代码
- 9.3 使用 PPP 构建子程序
- 9.4 PPP 的替代方案

相关章节

- 创建高质量的类：第 6 章
- 高质量子程序的特征：第 7 章
- 软件构建设计：第 5 章
- 注释风格：第 32 章

虽然全书都可以看成是创建类和子程序的编程过程的一个扩展描述，但本章会从更宏观的层面看待这些步骤。本章关注小规模编程（programming in the small），即构建单独的类及其子程序的具体步骤，这些步骤在各种规模的项目中都至关重要。本章还描述了伪代码编程过程（Pseudocode Programming Process，PPP），它减少了设计和文档编写工作，并提高了两者的质量。

如果你是专家级程序员，可以只是粗读一下本章，但请看一下步骤总结，并注意第 9.3 节使用伪代码编程过程构建子程序的提示。很少有程序员充分发掘了该过程的全部潜力，它有许多好处。

PPP 并不是创建类和子程序的唯一过程。本章末尾的第 9.4 节介绍了最流行的替代方法，包括测试优先开发（test-first development）和契约式设计（design by contract）。

9.1　类和子程序构建步骤总结

类的构建可从多方面着手，但通常是一个迭代过程，其中包括为类创建常规设计，列出类中的具体子程序，构建具体的子程序，以及将类作为整体进行审查。如图 9-1 所示，类的创建可能是一个混乱的过程，原因和设计同样混乱一样（原因已在第 5.1 节解释）。

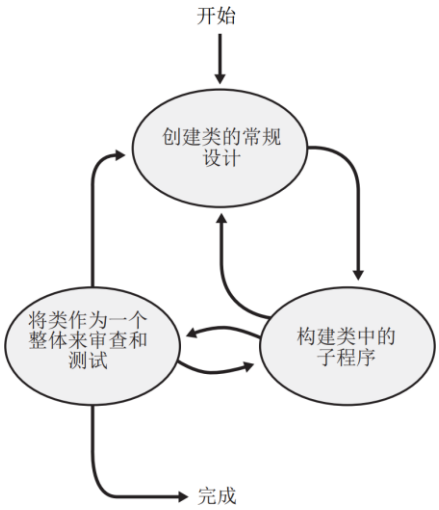


图 9-1　类的构建细节不一，但涉及的活动一般按此顺序进行

创建类的步骤

构建类的关键步骤如下所示：

创建类的常规设计　类的设计涉及许多具体问题。定义类的具体职责，定义类将隐藏哪些“秘密”，并准确定义类的接口将捕获的抽象。确定该类是否要从另一个类派生，以及是否允许其他类从该类派生。确定类的关键 public 方法，并确定和设计该类所用的任何重要的数据成员。根据需要多次重复这些任务，为子程序创建一个直接、易懂的设计。这些考虑因素和其他许多因素将在第 6 章“可以工作的类”详细讨论。

构建类中的每个子程序　在第一步确定了类的主要子程序后，必须构建每个具体的子程序。每个子程序的构建通常都会引出对额外子程序的需求，包括次要的和主要的，而创建这些额外子程序所引发的问题往往会反过来影响类的总体设计。

将类作为一个整体来审查和测试　通常，每个子程序在创建好后都要测试。在整个类可以运作后，应将类作为一个整体进行审查和测试，以发现在单个子程序的层次上无法测试的问题。

构建子程序的步骤

类的许多子程序实现起来都简单而直接：成员访问子程序、直通到其他对象的子程序等等。另一些子程序实现起来要复杂一些，创建这样的子程序可从一种系统化方法中受益。创建子程序所涉及的主要活动（设计子程序、检查设计、编码子程序和检查代码）一般按如图 9-2 所示的顺序进行。

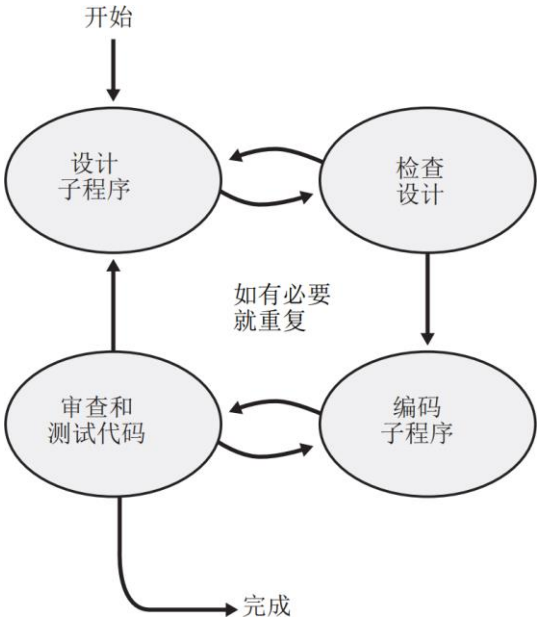


图 9-2　这些是构建子程序时的主要活动，一般按此顺序进行

专家们开发了许多创建子程序的方法，我最喜欢的是伪代码编程过程，在下一节中介绍。

9.2　面向专家的伪代码

“伪代码”（pseudocode）是指非正式的、和自然语言差不多的一种表示法，描述了算法、子程序、类或程序如何工作。伪代码编程过程（Pseudocode Programming Process，PPP）定义了使用伪代码来简化子程序内部代码创建的一种具体方法。

由于伪代码像自然语言，所以会很自然地以为，凡是收集了你的想法的任何像自然语言那样的描述都差不多。但在实践中，你会发现有些风格的伪代码比其他风格的伪代码更有用。以下是有效使用伪代码的指导原则：

- 使用自然语言风格的语句，精确描述具体的操作。
- 避免使用目标编程语言的语法元素。伪代码使你能在一个比代码本身稍高的层次上进行设计。一旦使用了编程语言结构，就会沉到一个较低的层次，会丧失在较高层次上进行设计的重要好处，而且会给自己带来不必要的语法限制。

关联参考　要详细了解意图层级的注释，请参见第 32.4 节中的“注释种类”。

- 在意图层级（level of intent）上写伪代码。描述方法的含义，而不要描述方法在目标语言中如何实现。
- 在足够低的层次上写伪代码，目的是几乎能直接转换成代码。如伪代码层次太高，它可能掩盖代码中的问题细节。用越来越多的细节完善伪代码，直到最后似乎能直接转换成代码。

写好伪代码后，就可以围绕它来写代码，伪代码就变成了编程语言的注释。这就省去了大部分注释的工作。如果按上述原则写伪代码，注释既完整，又有意义。

下面这个伪代码设计的例子几乎违反了刚才描述的所有原则。

糟糕伪代码的例子

```
increment resource number by 1
allocate a dlg struct using malloc
if malloc() returns NULL then return 1
invoke OSrsrc_init to initialize a resource for the operating system
*hRsrcPtr = resource number
return 0
```

这个伪代码块的意图是什么？由于写得很糟糕，所以很难厘清。这个所谓的伪代码之所以不好，是因为它包括了目标语言的编码细节，比如*hRsrcPtr（C 特有的指针表示法）和 malloc()（C 特有的函数）。这个伪代码块关注的是如何编写代码，而不是设计的含义。它深入到了编码细节，即子程序是返回 1 还是 0。只要想想它能不能转变成好的注释，就能理解它为什么没有太大帮助。

以下修改过的伪代码就好得多：

好的伪代码的例子

```
Keep track of current number of resources in use
If another resource is available
   Allocate a dialog box structure
   If a dialog box structure could be allocated
      Note that one more resource is in use
      Initialize the resource
      Store the resource number at the location provided by the caller
   Endif
Endif
Return true if a new resource was created; else return false
```

这个版本比第一个好，因为它完全是用自然语言写的；没有用到目标编程语言的任何语法元素。在第一个例子中，伪代码只能用 C 语言实现。而在第二个例子中，伪代码没有限制对语言的选择。第二个伪代码块还是在意图层级上写的。相较于原来的伪代码块，修改过的版本更容易理解其含义。

虽然是用清晰的自然语言写的，但修改过的版本足够精确和详细，可以很容易地用作编程语言代码的基础。将伪代码语句被转换成注释时，它们也能很好地解释代码的意图。

以下是使用这种风格的伪代码所带来的好处：

- 伪代码使审查更容易。可在不检查源代码的情况下审查详细的设计。伪代码使低级别的设计审查更容易，并减少审查代码本身的需要。
- 伪代码支持迭代细化（iterative refinement）的理念。从一个高层次的设计开始，将设计细化为伪代码，再将伪代码细化为源代码。通过这种小步骤的连续细化，在推动设计向更低层次的细节发展时，你可以方便地检查设计。结果就是，可在最高层次捕捉高级错误，在中间层次捕捉中级错误，在最低层次捕捉低级错误。这样可提前避免它们成为问题，或者对下个细节层次的工作造成污染。

深入阅读　要更多地了解在代价最小的阶段进行修改的好处，请参见 Andy Grove 的《*High Output Management*》（Grove1983）。

- 伪代码使修改更容易。几行伪代码比一页代码更容易修改。你是愿意修改蓝图上的一条线，还是愿意拆掉一堵墙，然后去别的地方钉木板呢？[6]软件中影响果虽然没有这么严重，但在产品最具可塑性的时候改动它的原则是一样的。项目取得成功的关键之一就是在“代价最小的阶段”捕捉错误，这个阶段投入的精力最少。在伪代码阶段投入的精力要比经过完整的编码、测试和调试之后少得多。因此，及早发现错误是有经济意义的。
- 伪代码使注释工作最小化。典型的编码场景是先写代码，再添加注释。而在 PPP 中，伪代码语句可直接成为注释，所以实际上删除注释还要比保留注释更花费精力。
- 伪代码比其他形式的设计文档更容易维护。在其他方法中，设计和代码是分开的。其中一个发生变化，两者就会不一致。而在 PPP 中，伪代码语句成为代码中的注释。只要维护好这些内联的注释，伪代码的设计文档就仍然是准确的。

KEY POINT 作为一种用于详细设计的工具，伪代码的地位很难动摇。一项调查表明，程序员们更喜欢伪代码是因为它能简化用编程语言进行的构建，能帮助他们发现不够充分的设计，而且简化了文档编制和修改（Ramsey, Atwood, and Van Doren 1983）。伪代码并不是用于详细设计的唯一工具，但伪代码和 PPP 是程序员工具箱中的有用工具。试试它们吧。下一节将解释具体如何做。

9.3　使用 PPP 构建子程序

本节描述了构建子程序所涉及到的活动，包括：

- 设计子程序
- 编码子程序
- 检查代码
- 收尾
- 根据需要重复

设计子程序

交叉参考　设计其他方面的详情请参见第 5 章~第 8 章。

[6] 译注：美国大多数房子的隔墙是木结构。

确定了类需要的子程序后，为了构造该类任何一个比较复杂的子程序，第一步就是设计它。假设要写一个子程序根据错误代码输出错误信息，并假设该子程序是 ReportErrorMessage()。以下是 ReportErrorMessage()的一个非正式规范：

“ReportErrorMessage()获取一个错误代码作为输入实参，输出与代码对应的错误消息。它应能处理无效的代码。如果程序以交互方式运行，ReportErrorMessage()向用户显示消息。如果以命令行模式运行，ReportErrorMessage()将消息记录到一个消息文件中。输出消息后，ReportErrorMessage()返回一个状态值，表明其操作是成功还是失败。”

本章剩余部分会将该子程序作为一个不断演化的运行实例。本节其余部分将介绍如何设计该子程序。

关联参考　关于检查先决条件方面的详情，请参见第 3 章和第 4 章。

检查先决条件　在对子程序本身进行任何工作之前，要检查该子程序的工作是否被很好地定义，并清晰地融入总体设计。核实该子程序确实是项目所要求的（至少是间接要求的）。

定义子程序要解决的问题　足够详细地说明子程序要解决的问题，为子程序的创建提供方便。如高级设计足够详细，这项工作可能已经完成了。高级设计至少应提供以下内容：

- 子程序将隐藏的信息。
- 子程序的输入。
- 子程序的输出。

关联参考　关于前置条件和后置条件的更多信息，请参见第 8.2 节中的“采用断言来注解并验证前置条件和后置条件”。

- 在调用该子程序之前保证成立的前置条件（输入值在特定范围内，流已初始化，文件已打开或关闭，缓冲区已填充或刷新，等等）。
- 子程序在将控制权传回调用者之前保证成立的后置条件（输出值在指定范围内，流已初始化，文件已打开或关闭，缓冲区已填充或刷新，等等）。

下面说明了在 ReportErrorMessage()的例子中如何解决这些问题：

- 该子程序隐藏了两个事实：错误信息文本和当前处理方法（交互式或命令行）。
- 该子程序没有任何前置条件
- 该子程序的输入是一个错误代码。
- 存在两种输出：第一种是错误信息，第二种是 ReportErrorMessage()返回给调用方的状态。
- 该子程序保证状态值要么是 Success，要么是 Failure。

关联参考　关于为子程序命名的详情，请参阅第 7.3 节。

命名子程序　为子程序命名表面上微不足道，但好的子程序名字是优秀程序的标志之一，而且不容易想出来。一般来说，子程序应该有一个清晰的、不含糊的名字。如果想不出一个好名字，通常表明该子程序的目的不明确。一个含糊不清的名字，就像政客在竞选时的表现。听起来好像说了什么，但当你集中注意力想去领会的时候，却搞不清楚意思。如果能使名字更清晰，就一定要这样做。如果模糊不清的名字是由模糊不清的设计造成的，请注意这个警告信号。退后并改进设计。

在本例中，ReportErrorMessage()是毫不含糊的。是个好名字。

深入阅读　一种完全不同的构建方法是先写测试用例，详情请参见《*Test-Driven Development: By Example*》（Beck 2003）。

决定如何测试子程序　写子程序时要想好如何测试它。无论是由你做单元测试，还是由测试人员独立测试你的子程序，这都很有用。

本例的输入很简单，所以可计划用所有有效的错误代码和多种无效代码测试 ReportErrorMessage()。

研究标准库中可用的功能　改进代码质量和工作效率最重要的一种方式就是重用好的代码。如发现自己正在努力设计一个似乎过于复杂的程序，问问这个程序的部分或全部功能是否已经在你所用的语言、平台或工具的代码库中存在了。问问该代码是否能在你的公司维护的代码库中找到。人们已经发明、测试、在文献中讨论、评审并改进了许多算法。与其花时间去发明已经有人写成了博士论文的东西，不如花几分钟检索人家已经写好的代码，一定不要做多余的工作。

考虑错误处理　考虑所有可能在子程序中出错的事情。想想糟糕的输入值、从其他子程序返回的无效值等。

子程序可通过许多方式处理错误，应该有意识地选择错误处理方式。如果程序的架构定义了程序的错误处理策略，可决定简单地遵循该策略。在其他情况下，则必须决定哪种方法对特定的子程序最好。

考虑效率问题　根据你的情况，可用两种方式之一处理效率问题。第一种情况针对的是绝大多数系统，它们对效率要求不高。在这种情况下，要确保子程序的接口进行了很好的抽象，而且代码良好的可读性，这样以后就可根据需要进行改进。只要有良好的封装，就可以在不影响其他子程序的情况下，用更好的算法或快速、节省资源的低级语言实现来取代一个缓慢、占用资源的高级语言实现。

交叉参考　有关效率的详情，请参见第 25 章和第 26 章。

第二种情况针对的是少数系统，性能对它们来说至关重要。性能问题可能和稀缺的数据库连接、有限的内存、很少的可用句柄、严格的时间限制或其他一些稀缺资源有关。架构应该说明每个子程序（或类）允许使用多少资源，以及应该以多快的速度执行其操作。

设计自己的子程序，使其能满足资源和速度目标。如资源或速度中的任何一个似乎更关键，设计时可考虑用资源换速度或相反。在程序的初始构建过程中，对其进行足够的调整以满足资源和速度预算是可以接受的。

除了针对这两种一般情况所推荐的方法，就单个子程序进行效率调整通常是对精力的一种浪费。大的优化来自对高层设计的完善，而不是来自单个子程序。通常，只有当高层设计被证明不能支持系统的性能目标时，才会考虑进行微优化，而在整个程序完成之前，你不会知道这一点。除非确定需要，否则不要浪费时间去搞增量改进。

研究算法和数据类型　如果所需的功能在可用的库中没有，就去算法书中找一找。开始从头写复杂的代码之前，检查一下算法书，看看有什么可用的资源。如果使用一种预定义的算法，一定要根据自己的编程语言正确改编它。

> 关联参考　这里的讨论假定是用好的技术编写子程序的伪代码版本。关于设计方面的详情，请参阅第 5 章。

写伪代码　完成前面的步骤后，你现在可能还没有写出什么东西。这些步骤的主要目的是建立一个心理定位，这在真正写子程序时是很有帮助的。

初始步骤完成后，就可着手用高层次的伪代码写这个子程序。使用你的代码编辑器或集成环境来写伪代码——它们不久将被用作编程语言代码的基础。

先从最常规的开始，然后越来越具体。子程序最常规的部分是描述该子程序作用的头部注释（header comment），所以先写一个关于该子程序作用的简要声明。写下这个声明将帮助你澄清自己对于该子程序的理解。如果写常规注释时遇到麻烦，表明你需要更好地理解这个子程序在整个程序中的作用。通常，如果很难归纳出子程序的作用，就可以认为某个环节出了问题。下面展示了对子程序进行描述的一段简明扼要的头部注释：

子程序头部注释(header comment)的例子

```
This routine outputs an error message based on an error code
supplied by the calling routine. The way it outputs the message
depends on the current processing state, which it retrieves
on its own. It returns a value indicating success or failure.
```

写好常规注释后，开始为该子程序写高层次的伪代码。本例的伪代码如下所示：

子程序伪代码示例

```
This routine outputs an error message based on an error code
supplied by the calling routine. The way it outputs the message
depends on the current processing state, which it retrieves
on its own. It returns a value indicating success or failure.

set the default status to "fail"
look up the message based on the error code

if the error code is valid
   if doing interactive processing, display the error message
   interactively and declare success

   if doing command line processing, log the error message to the
   command line and declare success

if the error code isn't valid, notify the user that an internal error
has been detected

return status information
```

注意，这些伪代码是在相当高的层次上写的。在其中找不到任何一种编程语言的影子。相反，它是用准确的自然语言表达了子程序需要做的事情。

交叉参考　关于有效使用变量的详情，请参见第 10 章~第 13 章。

思考数据　可在过程的几个不同的点设计子程序的数据。本例的数据很简单，而且数据处理并不是子程序一个重要部分。但是，如果数据处理对于子程序很重要，那么在考虑子程序的逻辑之前，值得先考虑一下主要数据。关键数据类型的定义对于子程序的逻辑定义非常有用。

交叉参考　关于评审技术的详情，请参见第 21 章。

评审伪代码　写好伪代码并设计好数据后，花点时间评审写好的伪代码。退后一步，想想如何向别人解释它。

请别人看一下，或听你解释。你可能以为让别人看 11 行伪代码很傻，但最后会惊讶地发现，相较于用编程语言写的代码，伪代码更能凸显你的预设和高层次错误。人们也更愿意评审几行伪代码，而不是审查长达 35 行的 C++或 Java 代码。

确定你自己能轻松自如地理解这个子程序的作用以及它是如何做到的。如果不能在伪代码的层次从概念上理解它，还有什么机会在编程语言的层次上理解它？如果自己都不理解，还有谁能理解呢？

交叉参考　关于迭代的详情，请参见第 34.8 节。

在伪代码中多尝试一些思路，保留其中最好的（这就是迭代）　开始编码之前，尽可能多地用伪代码尝试一些思路。一旦开始编码，就会对自己的代码产生某种感情，很难抛弃一个糟糕的设计并重新开始。

常规思路是在伪代码中迭代子程序，直到伪代码语句变得足够简单，以至于可在每个语句下面直接填上对应的代码，并保留原始伪代码作为文档。最开始尝试的一些伪代码可能过于高级，需进一步分解。请务必这么做！如果一样东西不确定应该如何编码，就继续使用伪代码，

直到最后能够确定。不断完善和分解伪代码，直到觉得写伪代码是在浪费时间，反而不如直接写实际的代码。

编码子程序

一旦设计好子程序，就开始构建。可按近乎标准的一个顺序执行构建步骤，但也可根据需要自由变化。图 9-3 展示了子程序的构建步骤。

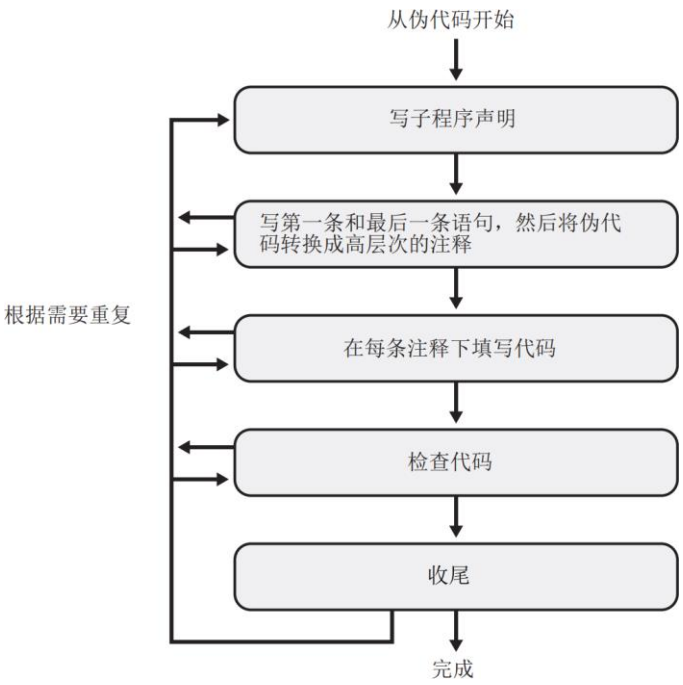


图 9-3　设计子程序时要执行所有这些步骤，但不一定按此顺序

写子程序声明　写子程序的接口声明（C++是函数声明，Java 是方法声明，Visual Basic 是函数或 Sub 过程声明，或者你的语言的对应物）。将原来的头部注释转变成编程语言的注释风格，并将其留在已经写好的伪代码上方。下面展示了 C++的子程序接口声明和头部注释：

用C++注释风格改写的头部注释

这里是接口语句

C++示例：在伪代码中添加了子程序的接口和头部注释

```
/* This routine outputs an error message based on an error code
supplied by the calling routine. The way it outputs the message
depends on the current processing state, which it retrieves
on its own. It returns a value indicating success or failure.
*/

Status ReportErrorMessage(
   ErrorCode errorToReport
   )
set the default status to "fail"
look up the message based on the error code

if the error code is valid
   if doing interactive processing, display the error message
   interactively and declare success

   if doing command line processing, log the error message to the
   command line and declare success

if the error code isn't valid, notify the user that an
internal error has been detected

return status information
```

这是将任何对接口的预设记录下来的好时机。本例的接口变量 errorToReport 非常直截了当，自己就能说明其具体的目的，所以无需另行记录。

将伪代码转换成高层次的注释　将第一条和最后一条语句写下来以界定子程序。在 C++中，就是写{和}。然后，将伪代码转换成注释，如下所示：

自此的伪代码语句已转换成C++注释。

C++示例：围绕伪代码写下第一条和最后一条语句

```
/* This routine outputs an error message based on an error code
supplied by the calling routine. The way it outputs the message
depends on the current processing state, which it retrieves
on its own. It returns a value indicating success or failure.
*/

Status ReportErrorMessage(
   ErrorCode errorToReport
   ) {
   // set the default status to "fail"
   // look up the message based on the error code
   // if the error code is valid
      // if doing interactive processing, display the error message
      // interactively and declare success

      // if doing command line processing, log the error message to the
      // command line and declare success

   // if the error code isn't valid, notify the user that an
   // internal error has been detected

   // return status information
}
```

到这个时候，子程序就已经尘埃落定了。设计工作已经完成，即使没有见到任何代码，也能感受到它是如何工作的。你应该能感受到，这时将伪代码转换为编程语言的代码将是一个机械、自然和容易的过程。如果没有感受到，就继续用伪代码进行设计，直至设计变得靠谱为止。

交叉参考　这是“写作”隐喻在小范围内运作良好的一个例子。要了解对于在大范围内应用“写作”隐喻的批评，请参见第 2.3 节中的“软件中的书法：写代码”。

在每条注释下填写代码　在每行伪代码注释下方填写代码。这个过程很像写一篇期末论文（term paper）。首先写一个大纲，再为大纲中的每个论点写一段话。每条伪代码注释都描述了一个代码块或段落。和文学作品中的段落一样，代码段落的长度也因其表达的思想而异，而段落的质量取决于思想的生动性和重点。

在本例中，前两条伪代码注释生成了两行代码：

C++示例：将伪代码注释转换成代码

```
/* This routine outputs an error message based on an error code
supplied by the calling routine. The way it outputs the message
depends on the current processing state, which it retrieves
on its own. It returns a value indicating success or failure.
*/

Status ReportErrorMessage(
   ErrorCode errorToReport
   ) {
   // set the default status to "fail"
   Status errorMessageStatus = Status_Failure;

   // look up the message based on the error code
   Message errorMessage = LookupErrorMessage( errorToReport );

   // if the error code is valid
      // if doing interactive processing, display the error message
      // interactively and declare success

      // if doing command line processing, log the error message to the
      // command line and declare success

   // if the error code isn't valid, notify the user that an
   // internal error has been detected

   // return status information
}
```

这是代码开始的地方。由于使用了 errorMessage 变量，所以需要声明它。如采用的是先编码再注释的方法，两行代码就有两行注释似乎有点用力过猛。但在这种方法中，重要的是注释的语义内容，而不是它们注释了多少行代码。注释本来就在那里，而且它们解释了代码的意图，所以把它们留在那里。

剩余每条注释下方的代码都需要填上：

C++示例：用伪代码编程过程(PPP)创建的一个完整子程序

```
/* This routine outputs an error message based on an error code
supplied by the calling routine. The way it outputs the message
depends on the current processing state, which it retrieves
on its own. It returns a value indicating success or failure.
*/

Status ReportErrorMessage(
   ErrorCode errorToReport
   ) {
   // set the default status to "fail"
   Status errorMessageStatus = Status_Failure;

   // look up the message based on the error code
   Message errorMessage = LookupErrorMessage( errorToReport );

   // if the error code is valid
   if ( errorMessage.ValidCode() ) {
      // determine the processing method
      ProcessingMethod errorProcessingMethod = CurrentProcessingMethod();

      // if doing interactive processing, display the error message
      // interactively and declare success
      if ( errorProcessingMethod == ProcessingMethod_Interactive ) {
         DisplayInteractiveMessage( errorMessage.Text() );
         errorMessageStatus = Status_Success;
      }

      // if doing command line processing, log the error message to the
      // command line and declare success
      else if ( errorProcessingMethod == ProcessingMethod_CommandLine ) {
         CommandLine messageLog;
         if ( messageLog.Status() == CommandLineStatus_Ok ) {
            messageLog.AddToMessageQueue( errorMessage.Text() );
            messageLog.FlushMessageQueue();
            errorMessageStatus = Status_Success;
         }
      else {
         // can't do anything because the routine is already error processing
      }
   else {
      // can't do anything because the routine is already error processing
   }
}

// if the error code isn't valid, notify the user that an
// internal error has been detected
else {
   DisplayInteractiveMessage(
      "Internal Error: Invalid error code in ReportErrorMessage()"
   );
}

// return status information
return errorMessageStatus;
}
```

此后每条注释都填上了代码。

这段代码适合分解成新的子程序：*DisplayCommandLine-Message()*。

这些代码和注释是新的，是充实*if*测试后的结果。

这些代码和注释也是新的。

每条注释都产生了一行或多行代码。每个代码块都在注释的基础上形成一个完整的思想。保留这些注释是为了对代码进行更高层次的解释。所有变量都在靠近它们第一次使用的地方声明和定义。每条注释通常应扩充为大约 2~10 行代码。(由于本例只是为了说明问题，所以代码的扩充程度只是实际开发时的下限。)

回头看看 9.3 节开头展示的规范和初始的伪代码。最初的五句话规范扩充为 15 行伪代码（取决于你如何统计行数），后者进而扩充为一页长的程序。虽然规范很详细，但创建该子程序还是需要对伪代码和代码进行大量设计工作。这种低层次的设计很好地解释了为什么说“编码”是一项不简单的任务，也很好地解释了本书的主题为什么重要。

检查代码是否要进一步分解 某些时候，某个初始伪代码行下方会出现代码爆炸的情况。在这种情况下，应从两种方案中选择一种：

> 关联参考 关于重构的详情，请参见第 24 章。

- 将注释下方的代码重构为新的子程序。如发现一行伪代码扩充出来的代码超出预期，请将代码重构为自己的子程序。写代码来调用该子程序，加入子程序的名字。如果一直在很好地运用 PPP，新子程序的名字应该很容易从伪代码中提取。完成最开始创建的子程序后，就可深入新的子程序中，再次对其应用 PPP。
- 递归地运用 PPP。与其在一行伪代码下面写几十行代码，不如花时间把原来的一行伪代码分解成几行伪代码。继续在每行新的伪代码下方填入代码。

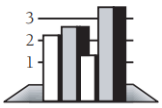
检查代码

设计并实现子程序之后，构建子程序的第三大步骤是进行检查，确保你所构建的东西是正确的。在这一阶段遗漏的任何错误只有在以后测试时才能发现。那时发现和纠正这些错误的成本会更高，所以应该在这一阶段尽可能地找出错误。

> 关联参考 关于检查架构和需求中存在的错误的详情，请参见第 3 章。

问题之所以在子程序完全编好码后才出现，是出于几方面的原因。伪代码中的错误可能在详细的实现逻辑中才变得明显。一个在伪代码中看起来很优雅的设计，在用于实现的语言中可能变得很笨拙。在详细的实现过程中，可能会发现架构、高层设计或需求中的错误。最后，代码中也可能出现一个老套的、乱七八糟的编码错误——人无完人！由于所有这些原因，继续之前要对代码进行评审。

心头检查子程序的错误 对子程序的第一次正式检查是在自己的心头进行。之前提到的清理和非正式检查步骤是心头检查的两种形式。还有一种是心头过一下每个执行路径。心头执行子程序很困难，而这种困难正是要保持子程序短小的原因之一。确保你检查了正常路径、端点以及所有异常情况。既要自己检查，这称为“桌面检查”（desk-check），也要和一个或多个同行一起检查，这称为“同行评审”（peer review）、“走查”（walk-through）或者“检查”（inspection），具体取决于你如何做。



HARD DATA 业余爱好者和专业程序员最大的区别之一是从迷信进入理解而产生的区别。这里的“迷信”并不是指程序在月圆之夜会令你感到毛骨悚然或者产生额外的错误。它是指凭对代码的感觉来代替理解。如经常发现自己怀疑编译器或硬件出了错误，表明你仍处于迷信的阶段。多年前的一项研究发现，所有错误中只有大约 5%是硬件、编译器或操作系统错误（Ostrand and Weyuker 1984）。如今这个比例甚至更低。相反，已进入理解阶段的程序员总是首先怀疑自己的工作，因为他们知道 95%的错误都是自己造成的。理解每一行代码的作用以及为什么需要它。没有什么只是因为它似乎能工作就一定正确。如果不知道它为什么能工作，它就可能不能工作——只是你还不知道而已。

KEY POINT 这里的要点在于，一个能工作的子程序还不够。如果不理解它为什么能工作，就研究它，讨论它，用替代设计进行实验，直到理解为止。

编译子程序 完成对子程序的评审后就编译它。由于代码好久之前就完成了，所以等这么长时间来编译似乎不划算。无可否认，可以早点编译子程序，让计算机检查一下未声明的变量、命名冲突等等，这样能省去一些工作。

但是，如果在程序后期才进行编译，会从几个方面受益。主要原因是，编译新代码时，一个内部秒表开始计时。首次编译后，你的压力会增大："只要再编译一次，我就能把它搞定"。这种"再编译一次"的综合症会导致仓促的、容易出错的修改，从长远来看会花费更多时间。避免急于求成，在确信程序是正确的前提下才进行编译。

本书的一个重点是告诉你如何跳出把东西拼凑在一起，跑一下看它是否能起作用的怪圈。在确定程序可以工作之前就进行编译，这是拼凑思维的一个典型症状。如果没有陷入"拼凑-编译"的怪圈，就会在感觉合适的时候才进行编译。但同时要意识到，其实大多数人都是以"拼凑-编译-修改"的方式开发出一个能工作的程序的。

以下指导原则可让你在编译程序时获得最大收益：

- 将编译器的警告级别设为最高。只要让编译器去做，它就能检测出数量令人震惊的细微错误。
- 使用校验工具。语言（例如 C）所进行的编译器检查可通过 lint 等工具来予以补充。即使是没有编译的代码（例如 HTML 和 JavaScript），也能通过校验工具进行检查。
- 消除所有错误信息和警告的根源。注意它们传送的关于你的代码的消息。大量警告往往意味着低质量的代码，应尝试理解显示的每一个警告。在实践中，反复出现的警告有两种可能的结果：你要么忽略它们，它们就会掩盖其他更重要的警告；要么它们只是变得越来越烦人。通常，重写代码以解决潜在的问题并消除警告是比较安全和省力的做法。

在调试器中逐行执行代码 一旦程序编译完成，就把它放到调试器中，逐行执行每一行代码。确保每一行都能按照你的期望执行。这一简单的做法能帮你发现许多错误。

关联参考 详情参见第 22 章。同时参见第 22.5 节中的"为测试各个类构造脚手架"。

测试代码 使用在开发程序时计划或创建的测试用例来测试代码。可能需要开发脚手架来支持你的测试用例——也就在仅在测试时用于支持子程序的代码，这些代码并不包括到最终产品中。脚手架可以是一个用测试数据调用你的子程序的测试工具，也可以是供你的子程序调用的存根（stub）。

关联参考 详情请参阅第 23 章。

消除子程序中的错误 检测到任何错误都必须消除。如果正在开发的子程序当前漏洞百出，它很可能会一直漏洞百出。发现一个子程序漏洞百出，就重新开始。不要再试图去拼拼凑凑，直接重写完事。但凡出现了需要拼凑的情况，就通常意味着对它的理解还不完整，现在和将来都肯定还会出错。为一个漏洞百出的子程序创建全新的设计是值得的。重写一个有问题的子程序，并确保以后再也挑不出它的任何错误，没有什么比这更让人有满足感了。

收尾

检查了代码的问题后，请核实它是否具有本书描述的常规特征。可采取几个清理步骤来确保该子程序的质量达到你的标准：

- 检查子程序的接口。确保所有输入和输出数据都有始有终，所有参数都得到了使用。详情请参见第 7.5 节。
- 检查常规设计质量。确保该子程序只做一件事并做好，它与其他子程序的耦合很松散，而且它的设计是防御性的。详情请参见第 7 章。
- 检查子程序的变量。检查是否存在不准确的变量名、未使用的对象、未声明的变量、初始化不当的对象等等。详情请参见关于变量使用的第 10 章~第 13 章。
- 检查子程序的语句和逻辑。检查是否有“相差 1”错误、无限循环、不正确的嵌套和资源泄漏。详情请参见关于语句的第 14 章~第 19 章。
- 检查子程序的布局。确保使用空白来澄清子程序、表达式和参数列表的逻辑结构。详情请参见第 31 章。
- 检查子程序的文档。确保转换为注释的伪代码仍然是准确的。检查算法描述，检查接口预设和非明显的依赖关系的文档，检查不明确的编码实践的理由，等等。详情请参见第 32 章。
- 删除多余注释。有的时候，一条伪代码注释对于注释所描述的代码来说确属多余，尤其是假如 PPP 已经被递归地运用，而注释恰好在对一个良好具名的子程序进行调用之前。

根据需要重复

如子程序质量堪忧，就退回到伪代码编程那一步。高质量编程是一个迭代过程，所以不要犹豫，再来一轮构建活动。

9.4 PPP 的替代方案

在我看来，PPP 是创建类和子程序的最佳方法。下面是其他专家推荐的一些不同方法。可将这些方法作为 PPP 的替代或补充：

测试优先开发(test-first development) 测试优先（或测试先行）是一种流行的开发风格，它是指在写任何代码之前，先写好测试用例。这种方法在第 22.2 节的“测试先行还是测试后行”中有更详细的描述。一本关于测试优先编程的好书是 Kent Beck 的《*Test-Driven Development: By Example*》(Beck 2003)。

重构 作为一种开发方法，重构是指通过一系列保留了语义的转换来改进代码。程序员根据不良代码的模式或“臭味”来识别需改进的部分。第 24 章详细讲述了这种方法，关于这个主题的一本好书是 Martin Fowler 的《*Refactoring: Improving the Design of Existing Code*》（Fowler 1999）。

契约式设计（design by contract） 作为一种开发方法，契约式设计中的每个子程序都被认为是有前置和后置条件的。这种方法在第 8.2 节中的“采用断言来注解并验证前置条件和后

置条件”进行了讲述。关于契约式设计的最佳信息来源是 Bertrand Meyers 的《*Object-Oriented Software Construction*》 (Meyer 1997)。

拼凑？ 一些程序员试图拼凑出能工作的代码，而不是采用像 PPP 这样的系统化方法。如果曾在编码一个子程序时陷入僵局，不得不重新开始，就表明 PPP 可能会更有效。如果发现自己在编码一个子程序的中途迷失，也表明 PPP 会有好处。是否经历过忘记写类或子程序一部分的情况？如果使用 PPP，这种情况几乎不会发生。如果发现自己盯着电脑屏幕不知道从何处下手，这就是 PPP 会使你的编程生活更容易的一个积极信号。

关联参考 此清单旨在核实创建子程序时是否遵循了一系列良好的步骤。有关专注于程序自身质量的检查清单，请参见第 7 章的“检查清单：高质量的子程序”。

检查清单：伪代码编程过程

- 确认已满足了所有先决条件吗？
- 定义好类要解决的问题了吗？
- 高层次设计足够清晰，能为类及其每个子程序起一个良好的名字吗？
- 想过应该如何测试类及其每个子程序吗？
- 主要是从可靠的接口和可读性好的实现，或者从满足资源和速度预算的角度去考虑效率吗？
- 在标准库或其他代码库中寻找过可用的子程序或组件了吗？
- 在参考书中查找过有用的算法了吗？
- 采用详细的伪代码去设计了每一个子程序吗？
- 已经在心头检查过伪代码吗？这些伪代码容易理解吗？
- 关注过那些可能会让自己重返设计的警告信息了吗（例如关于全局数据的使用、一些似乎更适合放在另一个类或子程序中的操作等）？
- 将伪代码准确转换成了实际代码吗？
- 以递归方式运用 PPP，并根据需要将一些子程序拆分成更小的子程序了吗？
- 在做出预设时对它们进行了说明吗？
- 删除了多余注释吗？
- 是从几次迭代中选择效果最好的那一个，而不是在第一次迭代之后就停止尝试了吗？
- 完全理解自己写的代码了吗？它们是否容易理解？

要点回顾

- 类和子程序的构建通常是一个迭代过程。构建子程序的过程中获得的认知常常会反过来影响类的设计。

- 编写好的伪代码需要使用容易理解的自然语言，要避免使用特定编程语言才有的特性，同时要在意图层级上编写伪代码（说明设计应该做什么，而不是具体怎么做）。
- 伪代码编程过程（PPP）是进行详细设计的一种有用的工具，它也使编码工作更容易。伪代码可直接转换为注释，从而确保了注释的准确性和实用性。
- 不要只停留在自己想到的第一个设计方案上。可以反复使用伪代码迭代出若干个方案，选出其中最好的一种再开始编码。
- 每一步完成后都检查自己的工作，并鼓励其他人帮忙检查。这样就能够在投入精力最少的时候，用最低的成本去发现错误。

第IV部分 语句

第 14 章　组织直线型代码

内容

- 14.1 顺序攸关的语句
- 14.2 顺序无关的语句

相关章节

- 常规控制问题：第 19 章
- 条件代码：第 15 章
- 循环代码：第 16 章
- 变量和对象的作用域：第 10.4 节“作用域”

本章从以数据为中心的编程视角转向以语句为中心的视角。介绍了最简单的控制流：将语句和语句块按顺序排列。

虽然组织直线型代码是一项相对简单的任务，但一些组织上的微妙之处影响了代码的质量、正确性、可读性和可维护性。

14.1　顺序攸关的语句

最容易排序的是那些必须按特定顺序的语句，如下例所示：

Java示例：语句顺序至关重要

```
data = ReadData();
results = CalculateResultsFromData( data );
PrintResults( results );
```

除非这个代码片段发生了什么神秘的事情，否则语句必须按所示顺序执行。计算结果之前必须先读取数据（data），打印结果之前则必须先计算结果（results）。

本例的基本概念是依赖关系。第三条语句依赖于第二条，第二条依赖于第一条。在这个例子中，一个语句依赖于另一个语句的事实从子程序名称（ReadData 等）就可以看出。在下面的代码片段中，依赖关系则不太明显：

Java示例：语句顺序仍然重要，但不太明显

```
revenue.ComputeMonthly();
revenue.ComputeQuarterly();
revenue.ComputeAnnual();
```

在本例中，季度收入的计算基于对月收入的计算。熟悉会计的人——甚至是常识——可能会告诉你，季度收入的计算必须先于年收入完成。这里存在一个隐性的依赖关系，但仅阅读代码并不明显。在下例中，依赖关系不仅不明显——它们实际还是隐藏的。

Visual Basic示例：语句对顺序的依赖被隐藏起来了

```
ComputeMarketingExpense
ComputeSalesExpense
ComputeTravelExpense
ComputePersonnelExpense
DisplayExpenseSummary
```

假设 ComputeMarketingExpense()负责初始化类成员变量，以便其他所有子程序在其中存储数据。所以，它需要先于其他子程序调用。但是，怎么通过阅读这段代码知道这一点呢？由于这些子程序的调用没有任何参数，你或许能猜到这些子程序中的每一个都是在访问类的数据。但是，无法通过阅读这段代码来确定。

若语句存在依赖性，必须以特定顺序排列，应采取措施使依赖性明确。这里有一些简单的语句排序指南：

组织代码，使依赖关系显而易见。在刚才的 Microsoft Visual Basic 例子中，ComputeMarketingExpense()不应初始化类成员变量。子程序的名称表明，ComputeMarketingExpense()与 ComputeSalesExpense()、ComputeTravelExpense()和其他子程序相似，只是它操作的是营销（marketing）数据而非销售数据或其他数据。让 ComputeMarketingExpense()负责成员变量的初始化过于任性，必须避免这种做法。为什么要在这个子程序中而不是其他两个子程序的任何一个中进行初始化？除非有一个很好的理由，否则应该写另一个子程序 InitializeExpenseData()来初始化成员变量。而且这个新的子程序的名称也清楚地表明，它应该在其他计算开销（expense）的子程序之前被调用。

子程序的命名要揭示依赖关系 在 Visual Basic 的例子中，ComputeMarketingExpense()的命名是错误的，因其不仅仅是计算营销费用；它还初始化成员数据。如果不想新建一个子程序来初始化数据，至少要把 ComputeMarketingExpense()换成一个描述它所执行的所有功能的名字。在本例中，ComputeMarketingExpenseAndInitializeMemberData()或许是一个合适的名字。你可能会说，这个名字真可怕，因为它太长了。但是，这个名字忠实描述了该子程序的作用，并不可怕。可怕的子程序本身！

关联参考 关于子程序及其参数的详情，请参见第 5 章。

使用子程序参数来揭示依赖关系 在 Visual Basic 的例子中，由于子程序之间没有传递数据，所以搞不清楚子程序是否使用了相同的数据。通过重写代码使数据在子程序之间传递，就可以提供一个线索，表明执行顺序的重要性。新的代码如下所示：

Visual Basic示例：数据揭示了对顺序的依赖

```
InitializeExpenseData( expenseData )
ComputeMarketingExpense( expenseData )
ComputeSalesExpense( expenseData )
ComputeTravelExpense( expenseData )
ComputePersonnelExpense( expenseData )
DisplayExpenseSummary( expenseData )
```

所有子程序都使用了 expenseData，由于处理的是相同的数据，所以语句的顺序也许很重要。

在这个特定的例子中，一个更好的方法可能是将这些子程序转换为函数，将 expenseData 作为输入，并将更新的 expenseData 作为输出返回，从而明确代码存在对顺序的依赖。

Visual Basic示例：数据和子程序调用明确了对顺序的依赖

```
expenseData = InitializeExpenseData( expenseData )
expenseData = ComputeMarketingExpense( expenseData )
expenseData = ComputeSalesExpense( expenseData )
expenseData = ComputeTravelExpense( expenseData )
expenseData = ComputePersonnelExpense( expenseData )
DisplayExpenseSummary( expenseData )
```

也可通过数据来表明执行顺序并不重要，如下例所示：

Visual Basic示例：从数据就看出不依赖顺序

```
ComputeMarketingExpense( marketingData )
ComputeSalesExpense( salesData )
ComputeTravelExpense( travelData )
ComputePersonnelExpense( personnelData )
DisplayExpenseSummary( marketingData, salesData, travelData, personnelData )
```

前四行的子程序没有任何共同的数据，暗示它们的调用的顺序并不重要。但是，由于第五行的子程序使用了来自之前四个子程序的数据，所以可认为它需要在前面四个子程序之后执行。

用注释记录不明确的依赖关系。首先尝试写不依赖顺序的代码。然后尝试写使依赖关系明显的代码。如果仍然担心顺序依赖不够明确，就把它记录下来。记录不明确的依赖关系是记录编码预设（coding assumptions）的一个方面，这对编写可维护、可修改的代码至关重要。在 Visual Basic 的例子中，像下面这样添加注释会很有帮助：

Visual Basic示例：用注释澄清隐蔽的顺序依赖

```
' 计算开销（expense）数据。每个子程序都访问
' 成员数据expenseData。DisplayExpenseSummary
' 应最后一个调用，因其依赖于其他子程序计算的数据。
InitializeExpenseData
ComputeMarketingExpense
ComputeSalesExpense
ComputeTravelExpense
ComputePersonnelExpense
DisplayExpenseSummary
```

这段代码没有使用让顺序依赖性变得明显的技术。平时最好依靠这种技术而非注释，但若在维护受到严格控制的代码，或由于其他原因不能对代码本身进行改进，就用文档来弥补代码的缺陷。

用断言或错误处理代码检查依赖关系　如代码非常关键，可用状态变量以及错误处理代码或断言来记录关键的顺序依赖关系。例如在类的构造函数中，可先将一个类成员变量 isExpenseDataInitialized 初始化为 false，再在 InitializeExpenseData() 中将 isExpenseDataInitialized 设为 true。每个要求 expenseData 已初始化的函数可在对 expenseData 进行其他操作之前检查 isExpenseDataInitialized 是否已被设为 true。取决于依赖关系的广泛程度，可能还需要诸如 isMarketingExpenseComputed、isSalesExpenseComputed 等变量。

这种技术创建了新的变量、新的初始化代码和新的错误检查代码，所有这些都带来了额外的出错可能性。技术是不错，但也要考虑额外的复杂性和增大的二次出错机率。

14.2 顺序无关的语句

你可能遇到这样的情况：几个语句或几个代码块的顺序似乎根本不重要。一个语句不依赖于另一个语句，或者非要在逻辑上紧随其后。但是，它们的排序会影响到可读性、性能和可维护性。若对执行顺序没有依赖，可依据次要标准来决定语句或代码块的顺序。这里用到的指导原则是“就近原则”（Principle of Proximity）：将相关的行动放在一起。

使代码自上而下阅读

一个常规的准则是使程序自上而下阅读，而不是跳来跳去。专家们同意，自上而下的顺序对可读性贡献最大。仅仅使控制在运行时自上而下流动还不够。如果别人在看你的代码时，不得不在整个程序中寻找需要的信息，就应考虑重新组织代码。如下例所示：

C++示例：跳来跳去糟糕的代码

```
MarketingData marketingData;
SalesData salesData;
TravelData travelData;

travelData.ComputeQuarterly();
salesData.ComputeQuarterly();
marketingData.ComputeQuarterly();

salesData.ComputeAnnual();
marketingData.ComputeAnnual();
travelData.ComputeAnnual();

salesData.Print();
travelData.Print();
marketingData.Print();
```

为确定 marketingData 是如何计算的，必须从最后一行开始，跟踪所有对 marketingData 的引用，并追溯到第一行。虽然只在其他几个地方用到了 marketingData，但不得不牢记 marketingData 在第一次和最后一次引用之间的所有地方是如何使用的。换言之，必须检查并思考这段代码中的每一行代码，以弄清楚 marketingData 是如何计算的。当然，这只是一个简化的例子，你在实际系统中看到的要复杂得多。下面是同样的代码，但进行了更好的组织：

C++示例：能自上而下阅读的、一气呵成的、好的、顺序的代码

```
MarketingData marketingData;
marketingData.ComputeQuarterly();
marketingData.ComputeAnnual();
marketingData.Print();
```

```
SalesData salesData;
salesData.ComputeQuarterly();
salesData.ComputeAnnual();
salesData.Print();

TravelData travelData;
travelData.ComputeQuarterly();
travelData.ComputeAnnual();
travelData.Print();
```

关联参考　第 10.4 节的“测量变量的生存时间”给出了变量“存活”（live）的一个更技术性的定义。

这段代码在几个方面更好。对每个对象的引用都保持在一起；它们是“局部”的。对象“存活”的代码行数很少。或许最重要的是，这段代码现在看起来可被分解为针对 marketing，sales 和 travel 数据的不同子程序。之前那段代码则看不出可以这样分解。

分组相关语句

将相关语句放到一起。若语句处理的是相同的数据，执行的是类似的任务，或要依赖于彼此的执行顺序，就可认为这些语句相关。

为测试相关语句是否被很好地分组，一个简单方法是打印出子程序的代码清单，将相关语句框到一起。如果这些语句的顺序很好，会得到一张如图 14-1 所示的图片，其中方框不会重叠。

关联参考　如遵循伪代码编程过程进行开发，代码会自动按相关的语句分组。参见第 9 章“伪代码编程过程”详细了解此过程。

图 14-1　如代码被很好地分组，围绕相关部分画的方框就不会重叠；嵌套是允许的

如语句排列不整齐，会得到类似图 14-2 所示的图片，其中的方框发生了重叠。如发现有方框重叠，请重新组织你的代码，使相关的语句更好地分组。

关联参考　第 10.4 节详细解释了如何将对变量的操作集中到一起。

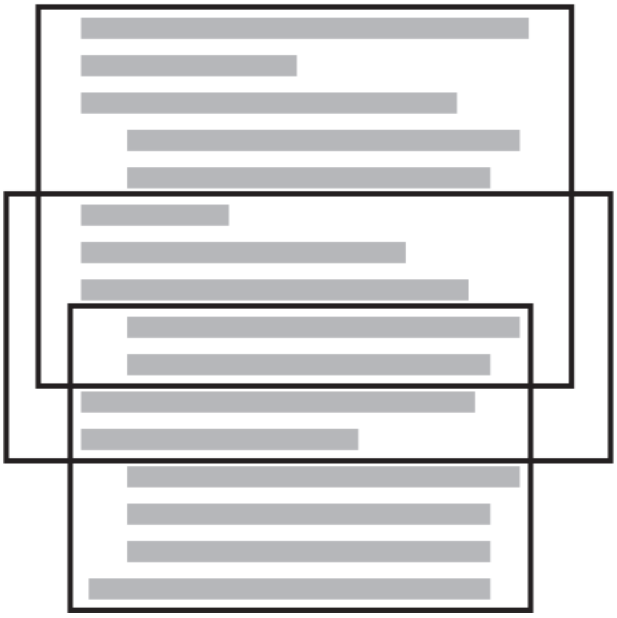


图 14-2　代码组织不善，围绕相关区域画的方框会发生重叠

对相关语句进行分组后，可能会发现它们是强相关的，与前后语句没有任何有意义的关系。在这种情况下，可将这些强相关的语句重构为自己的子程序。

检查清单：组织直线型代码

- 代码能明确语句之间的依赖关系吗？
- 子程序的名称能明确依赖关系吗？
- 子程序的参数能明确依赖关系吗？
- 用注释描述了不甚明确的依赖关系吗？
- 是否使用了内务处理（housekeeping）变量来核实关键代码中的顺序依赖？
- 代码能自上而下通畅阅读吗？
- 相关语句是否进行了分组？
- 相对独立的语句组是否转移到了各自的子程序中？

要点回顾

- 主要基于顺序依赖关系对直线型代码进行组织。
- 可使用良好的子程序名称、参数列表、注释以及（如果代码至关重要）用内务处理变量使依赖关系变得明显。
- 如代码之间没有顺序依赖关系，就设法使相关语句尽可能靠近。

第 15 章　使用条件语句

内容

- 15.1　if 语句
- 15.2　case 语句

相关章节

- 驾驭深层嵌套：第 19.4 节
- 常规控制问题：第 19 章
- 循环代码：第 16 章
- 直线型代码：第 14 章
- 数据类型和控制结构之间的关系：第 10.7 节

条件语句（conditional）是控制其他语句执行的语句；其他语句的执行要以 if、else、case 和 switch 等语句为“条件”。虽然从逻辑上讲，while 和 for 等循环控制也称为条件语句，但它们依据惯例区别对待。第 16 章“控制循环”将专门讲 while 和 for 语句。

15.1　if 语句

取决于你所用的语言，可使用几种 if 语句中的任何一种。最简单的是普通的 if 或 if-then 语句。if-then-else 要复杂一些，而 if-then-else-if 的链条最复杂。

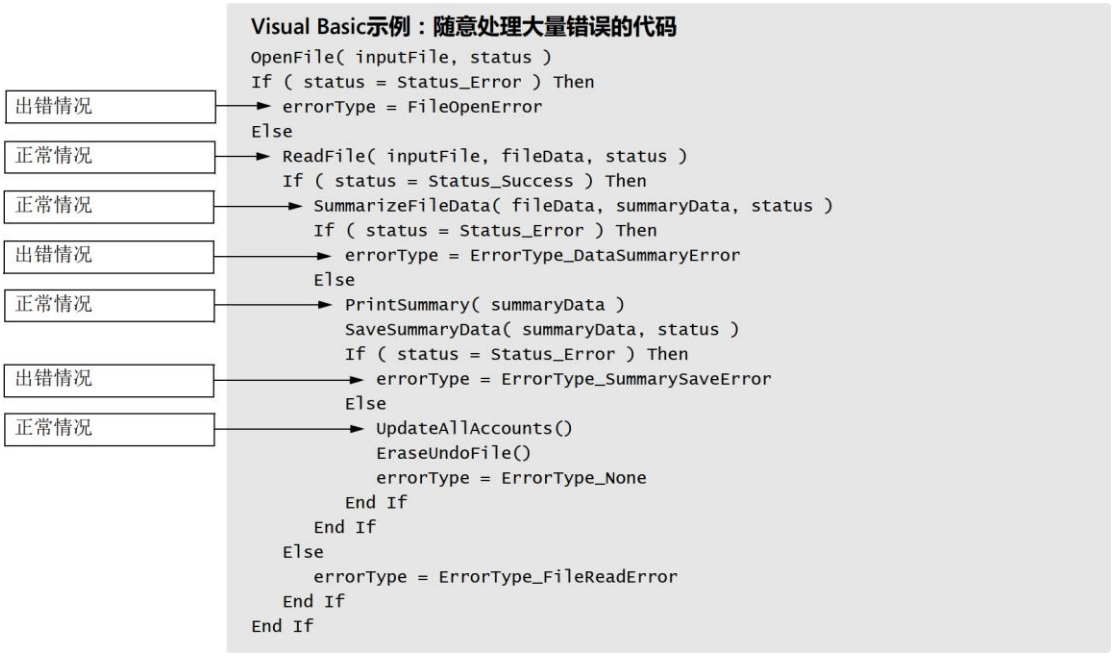
普通 if-then 语句

写 if 语句时遵循以下指导原则：

先写代码的正常路径；再写不寻常的情况　的代码要有清晰的正常执行路径。确保罕见情况不会掩盖正常执行路径。这对可读性和性能都很重要。

基于对相等性的测试正确地分支　使用>而非>=，或使用<而非<=，这类似于在访问数组或计算循环索引时犯下“相差 1”（off-by-one）错误。循环需考虑端点以避免这种错误。在条件语句中，也要考虑相同的情况以免犯错。

将正常情况放在 if 而非 else 之后　将通常期望处理的情况放在前面。这符合“将因决策而执行的代码尽可能放在决策附近”的一般原则。下例做了大量错误处理，一路随心所欲地检查错误：

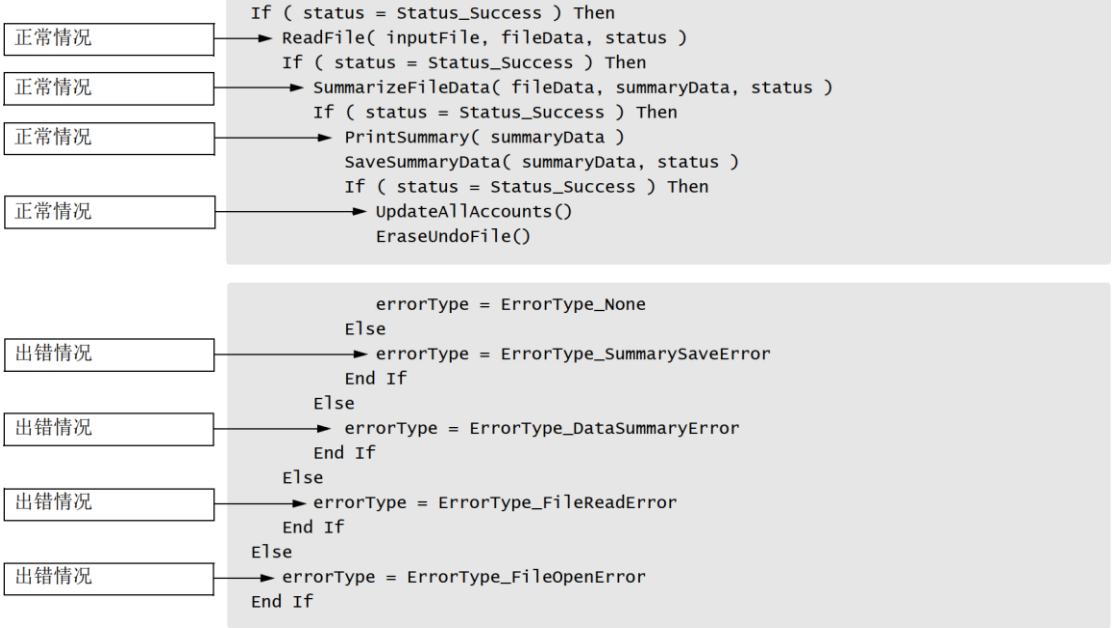


Visual Basic示例：随意处理大量错误的代码

```
OpenFile( inputFile, status )
If ( status = Status_Error ) Then
   errorType = FileOpenError
Else
   ReadFile( inputFile, fileData, status )
   If ( status = Status_Success ) Then
      SummarizeFileData( fileData, summaryData, status )
      If ( status = Status_Error ) Then
         errorType = ErrorType_DataSummaryError
      Else
         PrintSummary( summaryData )
         SaveSummaryData( summaryData, status )
         If ( status = Status_Error ) Then
            errorType = ErrorType_SummarySaveError
         Else
            UpdateAllAccounts()
            EraseUndoFile()
            errorType = ErrorType_None
         End If
      End If
   Else
      errorType = ErrorType_FileReadError
   End If
End If
```

关联参考　参见第 19.4 节了解其他编写错误处理代码的方法。

这段代码很难理清，因为正常情况和出错情况混在一起。很难找出正常的代码执行路径。此外，由于出错情况有时是在 if 子句中处理的，而不是在 else 子句中处理，所以很难判断正常情况跟哪个 if 测试对应。在下面重写的代码中，正常路径始终写在前面，所有出错情况都写在后面，从而可以轻松查找和阅读正常情况。



Visual Basic示例：井井有条地处理大量错误

```
OpenFile( inputFile, status )
If ( status = Status_Success ) Then
   ReadFile( inputFile, fileData, status )
   If ( status = Status_Success ) Then
      SummarizeFileData( fileData, summaryData, status )
      If ( status = Status_Success ) Then
         PrintSummary( summaryData )
         SaveSummaryData( summaryData, status )
         If ( status = Status_Success ) Then
            UpdateAllAccounts()
            EraseUndoFile()
            errorType = ErrorType_None
         Else
            errorType = ErrorType_SummarySaveError
         End If
      Else
         errorType = ErrorType_DataSummaryError
      End If
   Else
      errorType = ErrorType_FileReadError
   End If
Else
   errorType = ErrorType_FileOpenError
End If
```

在修订后的例子中，可通过阅读 if 测试的主流程来找到正常情况。修订后的例子将重点放在阅读主流程上，而不是还要被迫看一遍特殊情况，所以代码总体上更易阅读。一堆出错情况都集中在底部，这是错误处理代码写得好的一个标志。

这个例子说明了处理正常情况和出错情况的一种系统化方法。本书还讨论了其他解决方案，包括使用防卫子句（guard clauses），转换为多态分派（polymorphic dispatch），以及将测试的内部部分提取到一个单独的子程序中。关于可用方法的完整列表，请参见第 19.4 节中的“减少深层嵌套的技术总结”。

在 if 子句后面跟上一个有意义的语句　有时会看到像下例的代码，其中的 if 子句是空的：

Java示例：一个空的*if*子句

```
if ( SomeTest )
   ;
else {
   // do something
   ...
}
```

关联参考　构造有效 if 语句的关键是写出控制它的正确的布尔表达式。第 19.1 节详细讲述了如何有效使用布尔表达式。

仅仅是为了少写额外的空行和 else 行，大多数有经验的程序员也会避免这样写代码。这看起来很傻，对 if 语句中的谓词取反，将代码从 else 子句移到 if 子句中，再删除 else 子句，即可加以改进。下面是修改后的代码：

Java示例：转换后的空*if*子句

```
if ( ! someTest ) {
   // do something
   ...
}
```

考虑 else 子句　如果认为自己只需要一个普通 if 语句，就考虑是否真的不需要 if-then-else 语句。通用汽车所做的一项经典分析发现，有 50%到 80%的 if 语句都应配备一个 else 子句（Elshoff 1976）。

一个选择是写 else 子句（ 如有必要写成空语句），以表明 else 情况已经考虑过了。仅仅为了表明已考虑了相应的情况而写空的 else 语句似乎有点过犹不及，但这至少能促使程序员在写代码时考虑 else 情况。如果存在没有 else 的 if 测试，除非原因显而易见，否则需要通过注释来说明为什么这里不需要 else ，如下所示：

Java示例：一个有帮助的、添加了注释的*else*子句

```
// if color is valid
if ( COLOR_MIN <= color && color <= COLOR_MAX ) {
   // do something
   ...
}
else {
   // else color is invalid
   // screen not written to -- safely ignore command
}
```

测试 else 子句的正确性　测试代码时，可能以为主子句（即 if）是需要测试的全部内容。然而，如果可以测试 else 子句，一定要这样做。

检查 if 和 else 子句是否颠倒　编码 if-thens 时，一个常见的错误是把应该跟随在 if 子句后的代码和应该跟随在 else 子句后的代码颠倒过来，或者把 if 测试的逻辑倒过来。检查你的代码是否存在这种常见的错误。

if-then-else 语句链

在不支持 case 语句或者只部分支持 case 语句的语言中，经常需要写 if-then-else 测试链。例如，对一个字符进行分类的代码可能会使用下面这样的链：

C++示例：使用*if-then-else*链进行字符分类

```
if ( inputCharacter < SPACE ) {
   characterType = CharacterType_ControlCharacter;
}
else if (
   inputCharacter == ' ' ||
   inputCharacter == ',' ||
   inputCharacter == '.' ||
   inputCharacter == '!' ||
   inputCharacter == '(' ||
   inputCharacter == ')' ||
```

```
   inputCharacter == ':' ||
   inputCharacter == ';' ||
   inputCharacter == '?' ||
   inputCharacter == '-'
   ) {
   characterType = CharacterType_Punctuation;
}
else if ( '0' <= inputCharacter && inputCharacter <= '9' ) {
   characterType = CharacterType_Digit;
}
else if (
   ( 'a' <= inputCharacter && inputCharacter <= 'z' ) ||
   ( 'A' <= inputCharacter && inputCharacter <= 'Z' )
   ) {
   characterType = CharacterType_Letter;
}
```

关联参考　参见第 19.1 节详细了解复杂表达式的简化。

写这种 if-then-else 语句链时请遵循以下指导原则：

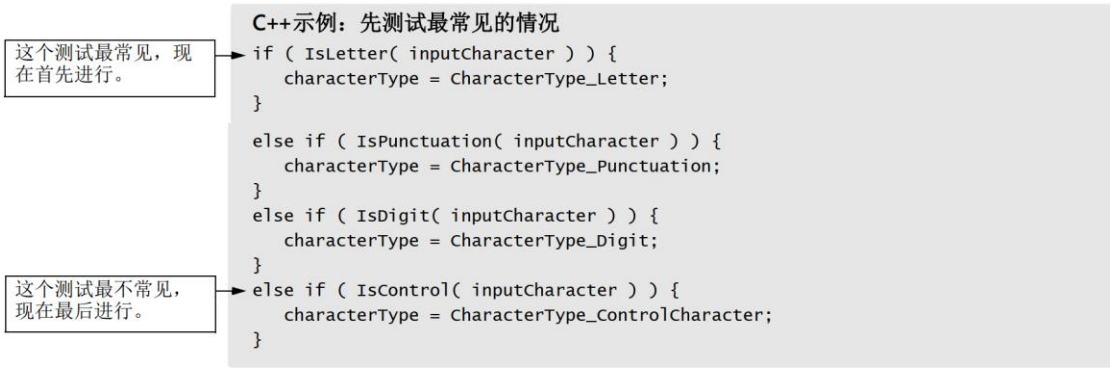
用布尔函数调用简化复杂测试　上例代码难以阅读的一个原因是，对字符进行分类的测试非常复杂。为提高可读性，可通过对布尔函数的调用来替换它们。

下例用布尔函数替换了测试：

C++示例：使用布尔函数调用的*if-then-else*链

```
if ( IsControl( inputCharacter ) ) {
   characterType = CharacterType_ControlCharacter;
}
else if ( IsPunctuation( inputCharacter ) ) {
   characterType = CharacterType_Punctuation;
}
else if ( IsDigit( inputCharacter ) ) {
   characterType = CharacterType_Digit;
}
else if ( IsLetter( inputCharacter ) ) {
   characterType = CharacterType_Letter;
}
```

常见情况放在前面　将最常见的情况放在前面，可最大限度地减少人们为寻找常见情况而必须阅读的异常情况处理代码的数量。效率也得到了提升，因为最大限度减少了代码为寻找最常见的情况所做的测试。在刚才的例子中，字母比标点符号更常见，但先测试的是标点符号。下面是修改后的代码，改为先测试字母：



C++示例：先测试最常见的情况

```
if ( IsLetter( inputCharacter ) ) {
   characterType = CharacterType_Letter;
}
else if ( IsPunctuation( inputCharacter ) ) {
   characterType = CharacterType_Punctuation;
}
else if ( IsDigit( inputCharacter ) ) {
   characterType = CharacterType_Digit;
}
else if ( IsControl( inputCharacter ) ) {
   characterType = CharacterType_ControlCharacter;
}
```

确保覆盖所有情况　最后编码一个 else 子句，用出错消息或者断言来捕捉你没有预计到的情况。错误消息是给你自己看的，而不是给用户看，所以措辞要恰当。下面展示了如何修改字符分类例子来执行对“其他情况”的测试：

关联参考　这个例子也很好地说明了可用 if-then-else 测试链来替换深度嵌套的代码。该技术的详情请参见第 19.4 节。

C++示例：用默认情况捕捉其他所有错误

```
if ( IsLetter( inputCharacter ) ) {
   characterType = CharacterType_Letter;
}
else if ( IsPunctuation( inputCharacter ) ) {
   characterType = CharacterType_Punctuation;
}
else if ( IsDigit( inputCharacter ) ) {
   characterType = CharacterType_Digit;
}
else if ( IsControl( inputCharacter ) ) {
   characterType = CharacterType_ControlCharacter;
}
else {
   DisplayInternalError( "Unexpected type of character detected." );
}
```

若语言支持，就用其他结构替换 if-then-else 链 一些语言（例如 Microsoft Visual Basic 和 Ada）支持使用字符串、枚举和逻辑函数的 case 语句。它们比 if-then-else 链更容易编码和阅读，所以尽量改为使用它们。在 Visual Basic 中，使用 case 语句对字符类型进行分类的代码可以这样写：

Visual Basic示例：使用*case*语句来替换 *if-then-else*链

```
Select Case inputCharacter
   Case "a" To "z"
      characterType = CharacterType_Letter
   Case " ", ",", ".", "!", "(", ")", ":", ";", "?", "-"
      characterType = CharacterType_Punctuation
   Case "0" To "9"
      characterType = CharacterType_Digit
   Case FIRST_CONTROL_CHARACTER To LAST_CONTROL_CHARACTER
      characterType = CharacterType_Control
   Case Else
      DisplayInternalError( "Unexpected type of character detected." )
End Select
```

15.2 case 语句

case 或 switch 语句的结构因语言不同而存在很大差异。C++和 Java 的 case 只支持一次取一个值的序数类型。Visual Basic 支持序数类型的 case，并有强大的速记符号来表达值的范围和组合。许多脚本语言根本就不支持 case 语句。

下面几节提供了有效使用 case 语句的指导原则。

选择最有效的 case 顺序

可选择多种方式组织 case 语句针对的不同情况。如果是一个小的 case 语句，仅三个选项和三行对应的代码，那么顺序并不重要。但是，如果是一个长的 case 语句（例如，在事件驱动的程序中用一个 case 语句处理几十个事件），那么顺序就很重要。下面列举了各种排序方案：

按字母或数字排列各种情况　如所有情况同等重要，把它们按 A-B-C 的顺序排列可提高可读性。这样可以从一组情况中轻松挑出一个特定的。

将正常情况放在前面　如果有一个正常情况和几个异常情况，将正常的放在最前面。用注释说明这是正常情况，其他的是异常。

按频率排列情况　将最经常执行的情况放在最前面，将最不常执行的放在最后。这种方法有两个好处。首先，人类读者可以很容易地发现最常见的情况。扫描列表来搜索一个特定的情况时，读者最有可能感兴趣的是其中一个最常见的情况。将常见情况放在代码顶部，可以加快搜索速度。

case 语句使用技巧

以下是使用 case 语句的技巧：

关联参考　参见第 24 章了解简化代码的其他技巧。

每个 case 的动作要简单　与每个 case 相关的代码要简短。在每个情况后面使用简短的代码，case 语句的结构显得更清晰。如果一个情况的操作很复杂，可以写一个子程序，并从 case 中调用它，而不是将完整代码放到 case 本身。

不要为了使用 case 语句而虚构变量　case 语句应该用于容易归类的简单数据。如数据不简单，可以使用 if-then-elses 链来代替。虚构的变量会令人困惑，应避免使用。例如，不要这样做：

Java示例：创建虚构的 *case*变量——这样不好

```
action = userCommand[ 0 ];
switch ( action ) {
   case 'c':
      Copy();
      break;
   case 'd':
      DeleteCharacter();
      break;
   case 'f':
      Format();
      break;
   case 'h':
      Help();
      break;
   ...
   default:
      HandleUserInputError( ErrorType.InvalidUserCommand );
}
```

关联参考 与这个建议相反，有时可将某个复杂的表达式赋值给一个命名良好的布尔变量或函数以提高代码可读性。详情参见第 19.1 节。

控制 case 语句的变量是 action。在这个 case 中，action 是通过截取 userCommand 字符串的第一个字符创建的，该字符串由用户输入。

这样写代码只会带来麻烦。通常，如果专门创建一个用于 case 语句的变量，真实数据可能不会以你想要的方式映射到 case 语句中。在本例中，如果用户输入 copy，case 语句会截取第一个'c'，并正确调用 Copy()子程序。但是，如果用户随便输入一个 c 开头的东西，例如 cement overshoes，clambake 或 cellulite， case 语句一样会截取'c'并调用 Copy() 。在 case 语句的 else 子句中对错误命令的测试不会有很好的效果，因为它只能检测到首字母有误的情况，检测不到命令有误的情况。

与其虚构变量，这段代码不如使用一个 if-then-else-if 测试链来检查整个字符串。下面是这段代码正确重写之后的样子：

Java示例：使用*if-then-elses*来替换虚构的*case*变量—这样很好

```
if ( UserCommand.equals( COMMAND_STRING_COPY ) ) {
   Copy();
}
else if ( UserCommand.equals( COMMAND_STRING_DELETE ) ) {
   DeleteCharacter();
}
else if ( UserCommand.equals( COMMAND_STRING_FORMAT ) ) {
   Format();
}
else if ( UserCommand.equals( COMMAND_STRING_HELP ) ) {
   Help();
}
...
else {
   HandleUserInputError( ErrorType_InvalidCommandInput );
}
```

只用 default 子句检测正当的默认情况 有时可能只剩一种情况，所以你决定将这种情况编码为 default 子句。看起来很诱人，但这很蠢。不仅失去了 case 语句标签所提供的自动注释，还失去了用 default 子句检测错误的能力。

以后想修改时，这样写的 case 语句会发生混乱。如果之前使用的就是一个正当的 default，那么添加新 case 会非常简单——添加 case 和相应的代码即可。但是，如果之前使用的是一个虚构的 default，修改起来就比较麻烦了。必须添加新 case，可能要把它变成新的 default，然后修改之前用作 default 的 case，使其成为一个正当的 case。所以，一开始就应该使用一个正当的 default。

使用 default 子句检测错误 如果 case 语句中的 default 子句没有用作其他处理，正常情况下也不应发生，就在其中放入一条诊断消息。

Java示例：使用 default情况检测错误—这样很好

```
switch ( commandShortcutLetter ) {
   case 'a':
      PrintAnnualReport();
      break;
   case 'p':
      // no action required, but case was considered
      break;
   case 'q':
      PrintQuarterlyReport();
      break;
   case 's':
      PrintSummaryReport();
      break;
   default:
      DisplayInternalError( "Internal Error 905: Call customer support." );
}
```

像这样的消息在调试和生产代码中都很有用。大多数用户宁愿收到类似“内部错误：请致电客户支持”这样的消息，也不愿看到系统崩溃，或者更糟，在老板检查之前，提供一个看起来正确、实则不然的结果。

如 default 子句被用作错误检测之外的目的，意味着其实每个 case 选择分支都是正确的。请反复确认可能进入 case 语句的每个值都是正当的。如果存在不正当的值，就重写语句，用 default 子句检查错误。

在 C++和 Java 中，避免直接通过 case 语句的结尾 C 风格的语言（C、C++和 Java）不会自动跳出每个 case。相反，必须明确编码每个 case 的结束，否则会直通（fall-through）到下个 case。这可能导致一些特别恶劣的结果，如下例所示：

关联参考 不要被这段代码优美的格式骗了。参见 31.3 节和第 31 章了解如何通过代码的格式化使好的代码看起来舒服，使不好的代码看起来也不舒服。

C++示例：滥用*case*语句

```
switch ( InputVar ) {
   case 'A': if ( test ) {
                  // statement 1
                  // statement 2
   case 'B':      // statement 3
                  // statement 4
                  ...
                  }
               ...
             break;
   ...
}
```

这种做法不好，因为它使控制结构混杂在一起。嵌套控制结构已经很难理解了；重叠的结构几乎不可能理解。修改 case 'A 或 case 'B'比做脑外科手术更难，而且任何修改要想生效，可能都需要对各种 case 进行清理。还不如一开始就做对。一般来说，避免在 case 语句的结尾处直通到下个 case 是个好主意。

在 C++中，要明确无误地在 case 语句的结尾处标识直通 如果是故意写代码在 case 的结尾处直通到下个 case，请清楚地加以注释，解释为什么需要这样编码。

C++示例：用注释说明要在*case*语句结尾处直通

```
switch ( errorDocumentationLevel ) {
   case DocumentationLevel_Full:
      DisplayErrorDetails( errorNumber );
      // FALLTHROUGH -- Full documentation also prints summary comments

   case DocumentationLevel_Summary:
      DisplayErrorSummary( errorNumber );
      // FALLTHROUGH -- Summary documentation also prints error number

   case DocumentationLevel_NumberOnly:
      DisplayErrorNumber( errorNumber );
      break;

   default:
      DisplayInternalError( "Internal Error 905: Call customer support." );
}
```

这个技术用到的时候不多。如果从一个 case 直通到下一个 case，以后修改代码时容易犯错，应尽量避免。

检查清单：使用条件语句

if-then 语句

- 代码的正常路径清晰吗？
- if-then 基于相等性测试正确分支了吗？
- else 子句是否使用并添加了注释？
- else 子句用得正确吗？
- 是否正确使用了 if 和 else 子句，它们没有用反吗？
- 正常情况是跟在 if 后面而不是 else 后面吗？

if-then-else-if 链

- 复杂测试封装到布尔函数调用里了吗？
- 最先测试的是最常见的情况吗？
- 覆盖了所有情况吗？
- if-then-else-if 链是最佳实现方式吗？比 case 语句还好？

case 语句

- 所有 case 都按有意义的方式排列吗？
- 每个 case 的行动都简单吗？必要时调用了其他子程序吗？

- case 语句判断的是一个真实变量，而非只为滥用 case 语句而虚构的一个变量吗？
- default 子句用得正当吗？
- default 子句是不是用来检测和报告出乎预料的情况？
- 在 C，C++或 Java 中，每个 case 的结尾处都有 break 吗？

要点回顾

- 对于简单 if-else 语句，要注意 if 和 else 子句的顺序，尤其是在它们处理了大量错误的时候。确保正常情况清晰可见。
- 对于 if-then-else 链和 case 语句，选择一个最有利于可读性的顺序。
- 捕捉错误用 case 语句中的 default 子句，或在 if-then-else 链中使用最后一个 else。
- 所有控制结构都不一样。为每部分代码选择最合适的。

第 16 章　控制循环

内容

- 16.1 选择循环类型
- 16.2 控制循环
- 16.3 轻松创建循环：由内而外
- 16.4 循环和数组的对应关系

相关章节

- 驾驭深层嵌套：第 19.4 节
- 常规控制问题：第 19 章
- 条件代码：第 15 章
- 直线型代码：第 14 章
- 数据类型和控制结构之间的关系：第 10.7 节

“循环”（loop）是一个非正式的术语，泛指任何类型的迭代控制结构——使程序重复执行一个代码块的任何结构。常见的循环类型有 C++和 Java 的 for、while 和 do-while，以及 Microsoft Visual Basic 的 For-Next、While-Wend 和 Do-Loop-While。循环的使用是编程中最复杂的方面之一；知道如何以及何时使用每种循环是构建高质量软件的决定性因素。

16.1　选择循环类型

大多数语言支持的循环只有以下几种：

- 计数循环（counted loop）执行特定次数；例如，为每个员工执行一次。
- 连续求值循环（iterator loop）事先不知道要执行多少次，而是每次迭代都测试是否已经完成。例如，只要账户上有钱就一直运行，直到用户选择退出或遇到错误。
- 无限循环（endless loop）一旦开始就一直执行。嵌入式系统喜欢用这个，例如心脏起搏器、微波炉和巡航控制。
- 迭代器循环（iterator loop）操作容器类中的每一个元素。

循环的类型首先由灵活度来区分，即循环是执行指定次数，还是每次迭代都测试是否已经完成。

循环的类型还由测试完成的位置来区分。可将测试放在循环的开始、中间或结尾处。根据测试的位置，可判断循环是否至少会执行一次。如循环一开始就测试，其主体不一定会执行。如循环在结束时测试，其主体至少执行一次。如循环在中间测试，则测试之前的循环部分至少执行一次，但测试之后的部分可能完全不会执行。

灵活度和测试位置决定了选择哪种类型的循环作为控制结构。表 16.1 总结了几种语言的循环类型，描述了每种循环的灵活度和测试位置。

表 16-1　循环类型

编程语言	循环的类型	灵活度	判断位置
Visual Basic	For-Next	严格	开始
	While-Wend	灵活	开始
	Do-Loop-While	灵活	开始或者结尾
	For-Each	严格	开始
C, C++, C#, Java	for	灵活	开始
	while	灵活	开始
	do-while	灵活	结尾
	foreach*	严格	开始
* 目前只在 C# 语言中可用。其他一些编程语言，包括 Java 语言，在本书写作的时候也有计划采用这种循环			

何时使用 while 循环

新手程序员可能以为 while 循环会不停地求值，一旦 while 条件的求值结果变成 false，无论执行到循环中的哪条语句，都会立即终止（Curtis et al. 1986）。虽然没有那么灵活，但 while 循环仍然是一种灵活的循环。如事先不知道循环要迭代多少次，就可考虑 while。与某些新手的想法相反，循环的退出测试每次迭代只进行一次。while 循环的主要问题在于决定是在循环开始处还是结尾处进行测试。

在开始处测试

若循环需要在开始处测试，可使用 C++、C#、Java、Visual Basic 和其他大多数语言的 while 循环。其他语言也可模拟 while 循环。

在结尾处测试

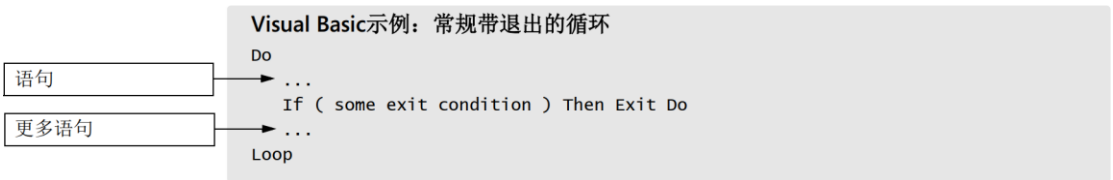
有时需要一个灵活的循环，但要保证至少执行一次。这时就可使用在结尾处测试的 while 循环。C++、C#和 Java 可使用 do-while，Visual Basic 可使用 Do-Loop-While，或者可以在其他语言中模拟在最后测试的循环。

何时使用带退出的循环

带退出的循环（loop-with-exit）的退出条件出现在循环中间而非首尾。Visual Basic 明确支持带退出的循环，C++、C 和 Java 可用结构化的 while 和 break 模拟，其他语言可用 goto 模拟。

正常的带退出的循环

带退出的循环由循环头、循环体（包括退出条件）和循环尾构成，下面是 Visual Basic 的一个例子：



Visual Basic示例：常规带退出的循环

```
Do
   ...
   If ( some exit condition ) Then Exit Do
   ...
Loop
```

若放在循环开始或结束处的测试要求编码一个 loop-and-a-half，通常就可以使用一个带退出的循环来避免。以下 C++示例就应该换成带退出的循环：

C++示例：重复的代码给维护带来麻烦

```
// Compute scores and ratings.
score  = 0;
GetNextRating( &ratingIncrement );
rating = rating + ratingIncrement;
while ( ( score < targetScore ) && ( ratingIncrement != 0 ) ) {
   GetNextScore( &scoreIncrement );
   score = score + scoreIncrement;
   GetNextRating( &ratingIncrement );
   rating = rating + ratingIncrement;
}
```

这些行在这里有…

…在这里也有

顶部两行代码在 while 循环的最后两行重复。以后若要修改，你很容易忘记保持这两组代码的一致。而负责修改代码的程序员可能意识不到这两组应平行修改。无论哪种情况，结果都是由于修改不完整而造成出错。下面更清楚地重写了代码：

C++示例：带退出的循环更容易维护

```
// Compute scores and ratings. The code uses an infinite loop
// and a break statement to emulate a loop-with-exit loop.
score = 0;
while ( true ) {
   GetNextRating( &ratingIncrement );
   rating = rating + ratingIncrement;

   if ( !( ( score < targetScore ) && ( ratingIncrement != 0 ) ) ) {
      break;
   }

   GetNextScore( &scoreIncrement );
   score = score + scoreIncrement;
}
```

这是循环退出条件(现在可用第19.1节描述的德摩根定律简化)

以下是这个例子的 Visual Basic 版本：

Visual Basic示例：一个带退出的循环

```
' Compute scores and ratings
score = 0
Do
   GetNextRating( ratingIncrement )
   rating = rating + ratingIncrement

   If ( not ( score < targetScore and ratingIncrement <> 0 ) ) Then Exit Do

   GetNextScore( ScoreIncrement )
   score = score + scoreIncrement
Loop
```

关联参考　本章稍后会详细讲解退出条件。参见第 32.5 节的“注释控制结构”了解如何在循环中合理使用注释。

使用这种循环时注意以下细节：

将所有退出条件都放在一个地方。若将其分散，调试、修改或测试期间很容易会忽略这个或那个退出条件。

使用注释来澄清。如语言不直接支持带退出的循环，使用这种技术时请用注释来说明你在做什么。

带退出的循环是一种单进单出的结构化控制结构，是首选的循环控制方式（Software Productivity Consortium 1989）。它已被证明比其他类型的循环更容易理解。一项对学生程序员的研究将这种循环与那些在顶部或底部退出的循环进行了比较（Soloway, Bonar, and Ehrlich 1983）。使用带退出的循环，学生的理解力测试得分高出 25%。研究的作者得出结论，带退出的循环结构比其他循环结构更接近人们对迭代控制的思考方式。

在通常的实践中，带退出的循环还没有被广泛使用。陪审团仍然被锁在一个烟雾缭绕的房间里，争论它是不是生产代码的一个好实践。在陪审团做出裁定之前，带退出的循环是程序员工具箱中的一样好东西——只要你小心地使用。

非正常的带退出的循环

下面是用于避免 loop-and-a-half 的另一种带退出的循环：

C++示例：用*goto*跑到循环中间—这样写不好

```
goto Start;
while ( expression ) {
   // do something
   ...

   Start:

   // do something else
   ...
}
```

乍一看，这似乎与前面带退出的循环示例相似。它用于模拟这样一种情况：// do something 不需要在第一次循环迭代时执行，但// do something else 需要。这是一个单入单出的控制结构：进入循环唯一的途径是通过顶部的 goto，而离开循环唯一的途径是通过 while 测试。这种方法有两个问题：它使用了一个 goto，而且它很不寻常，足以让人困惑。

在 C++中，可在不用 goto 的情况下实现同样的效果，如下例所示。如使用的语言不支持 break 命令，可用 goto 来模拟一个。

*break*前后的代码块被交换了

C++示例：重写的代码不用 *goto*—更好的写法

```
while ( true ) {
   // do something else
   ...

   if ( !( expression ) ) {
      break;
   }

   // do something
   ...
}
```

何时使用 for 循环

如果需要一个执行指定次数的循环，for 循环就是一个不错的选择。可在 C++、C、Java、Visual Basic 和其他大多数语言中使用 for。

为不需要内部循环控制的简单活动使用 for 循环。如果循环控制涉及简单的递增或递减，例如迭代一个容器中的元素，就使用 for 循环。for 循环的意义在于，在循环顶部设置好就可以把它忘了。无需在循环内部做任何事情来控制它。如存在一个必须跳出循环的条件，就用 while 循环来代替。

类似地，不要显式更改 for 循环的索引值来迫使其终止。这时应换用 while 循环。for 循环是用于简单的用途。大多数复杂的循环任务最好用 while 循环来处理。

深入阅读　参见《*Writing Solid Code*》一书(Maguire 1993)更多地了解使用 for 循环时的指导原则。

何时使用 foreach 循环

foreach 循环或其等价物（C#中的 foreach，Visual Basic 中的 For-Each，Python 中的 for-in）特别适合对数组或其他容器中的每个成员进行操作。其优点是避免了循环内务处理算术，进而消除了这种算术中出错的机会。

下面是这种循环的一个例子：

C#示例：一个*foreach*循环

```
int [] fibonacciSequence = new int [] { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
int oddFibonacciNumbers = 0;
int evenFibonacciNumbers = 0;

// count the number of odd and even numbers in a Fibonacci sequence
foreach ( int fibonacciNumber in fibonacciSequence ) {
   if ( fibonacciNumber % 2 ) == 0 ) {
      evenFibonacciNumbers++;
   }
   else {
      oddFibonacciNumbers++;
   }
}

Console.WriteLine( "Found {0} odd numbers and {1} even numbers.",
   oddFibonacciNumbers, evenFibonacciNumbers );
```

16.2 控制循环

循环会出什么问题？答案至少包括不正确或遗漏的循环初始化，遗漏循环变量或其他循环相关变量的初始化，不正确嵌套，不正确终止循环，忘记或不正确地递增循环变量，以及不正确地用循环索引来索引一个数组元素。

可通过遵循两个实践来预防这些问题。首先，尽量减少影响循环的因素的数量。简化！简化！简化！（重要的事情说三遍）。其次，将循环内部当作一个子程序来处理——尽可能多地将控制留在循环外部。明确说明循环体的执行条件。不要让读者通过查看循环内部来了解循环的控制。将循环想象成黑盒：外围程序知道控制条件，但不知道内容。

关联参考　如使用了之前描述的 while (true)-break 技术，退出条件就在黑盒内。即便只使用了一个退出条件，也失去了将循环视为黑盒的好处。

C++示例：将循环视为黑盒

```
while ( !inputFile.EndOfFile() && moreDataAvailable ) {
    [redacted]
}
```

这个循环的终止条件是什么？这很明显，要么 inputFile.EndOfFile() 为 true，要么 MoreDataAvailable 为 false。

进入循环

循环的进入要遵循以下指导原则：

只从一个位置进入循环　多种循环控制结构都允许在循环的开始、中间或结尾进行测试。这些结构足够丰富，允许每次都从顶部进入循环。你不需要在多个位置进入。

将初始化代码直接放在循环之前 就近原则主张将相关语句放在一起。如相关的语句散落在一个子程序中，以后修改时很容易找不齐，并做出不正确的修改。相关语句都在一起，修改时就不容易犯错。

> 关联参考　要更多地了解对循环变量作用域的限制，请参阅本章后面的“将循环索引的作用域限制在循环内部”。

将循环初始化语句和相关的循环放在一起。如果不这样做，将循环放到一个更大的循环而忘记修改初始化代码时，就很有可能出错。将循环代码移动或复制到一个不同的子程序时，如忘记移动或复制其初始化代码，也会发生同样的错误。将初始化放在远离循环的地方（例如数据声明部分，或循环所子程序顶部的内务处理部分），除了给初始化带来麻烦，看不出有什么好处。

将 while(true)用于无限循环 有的循环可能需要无休止地运行，例如心脏起搏器或微波炉中的循环。有的循环只在发生特定事件时终止，这称为“事件循环”。可用几种方式编码这样的无限循环。用 for i = 1 to 99999 这样的语句来伪造无限循环是一个糟糕的选择，因为具体的循环限制混淆了循环的意图——99999 可能是一个正当的值。这种假的无限循环在维护时也会出问题。

在 C++、Java、Visual Basic 和其他支持类似结构的语言中写无限循环时，while(true)被认为是一种标准方式。有的专家喜欢用 for(;;)，这也能接受。

如有可能就首选 for 循环 for 循环将循环控制代码集中在一处，这使循环易于阅读。程序员在修改软件时常犯的一个错误是修改了 while 循环顶部的循环初始化代码，却忘记修改底部的相关代码。而在 for 循环中，所有相关代码都集中在循环顶部，这使修改更容易正确。只要 for 循环比其他循环更合适，就用它。

若 while 循环更合适，就不要使用 for 循环 C++、C#和 Java 提供了灵活的 for 循环结构，但一个常见的滥用方式是将 while 循环的内容胡乱地塞进 for 循环头。下例显示了一个被塞进 for 循环头的 while 循环：

C++示例：*while*循环被胡乱塞入一个*for*循环头

```
// read all the records from a file
for ( inputFile.MoveToStart(), recordCount = 0; !inputFile.EndOfFile();
   recordCount++ ) {
   inputFile.GetRecord();
}
```

和其他语言的 for 循环相比，C++ for 循环的强项在于，它在初始化和终止信息的种类上更灵活。但这种灵活性所固有的弱点在于，可以把与控制循环无关的语句放入循环头。

只在 for 循环头中写循环控制语句，也就是初始化循环、终止循环或使其趋于终止的语句。在刚才的例子中，循环主体中的 inputFile.GetRecord()语句会使循环趋于终止，但 recordCount 语句不会；它们是和循环进度控制无关的内务处理语句。将 recordCount 语句放在循环头，

却将 inputFile.GetRecord()语句排除在外，这是一种误导；它造成了是 recordCount 在控制循环的错误印象。

如果想在这种情况下使用 for 循环而不是 while 循环，就将循环控制语句放在循环头，其他都不要。下面是使用循环头的正确方法：

C++示例：符合逻辑但非常规的*for*循环头用法

```
recordCount = 0;
for ( inputFile.MoveToStart(); !inputFile.EndOfFile(); inputFile.GetRecord() ) {
   recordCount++;
}
```

在这个例子中，循环头的内容全都与循环控制相关。inputFile.MoveToStart()语句初始化循环，!inputFile.EndOfFile()语句测试循环是否结束，而 inputFile.GetRecord()语句使循环趋于终止。影响 recordCount 的语句并不直接使循环趋于终止，所以从循环头中拿掉了。while 循环仍然可能更适合这项工作，但上述代码使用循环头的方式至少还算符合逻辑。为了内容的完整性，下面是使用 while 循环的版本：

C++示例：*while*循环的合适用法

```
// read all the records from a file
inputFile.MoveToStart();
recordCount = 0;
while ( !inputFile.EndOfFile() ) {
   inputFile.GetRecord();
   recordCount++;
}
```

处理循环体

下面几个小节描述了如何处理循环主体：

使用{和}封闭循环体 任何循环都要用花括号封闭循环体。它们在运行时不会造成任何速度或空间上的损失，它们有利于可读性，且有利于防止以后修改代码时出错，是一种良好的防御性编程实践。

避免空循环 C++和 Java 允许创建空循环，即循环所做的工作与检查工作是否完成的测试放在同一行中，如下例所示：

C++示例：空循环

```
while ( ( inputChar = dataFile.GetChar() ) != CharType_Eof ) {
   ;
}
```

在这个例子中，循环体是空的，因为 while 表达式做了两件事：循环的工作，即 inputChar = dataFile.GetChar()；以及测试循环是否应该终止，即 inputChar != CharType_Eof。应重新编码，使读者能轻松理解循环所做的工作，使其更清晰：

C++示例：将空循环转换为充实的循环

```
do {
   inputChar = dataFile.GetChar();
} while ( inputChar != CharType_Eof  );
```

新代码占用了完整的三行，而不是一行和一个分号，这很合适，因为它本来做的就是三行的工作，而不是一行和一个分号的工作。

只在循环开头或末尾进行循环内务处理　循环内务处理是指 i = i + 1 或 j++这样的表达式，这些表达式的主要目的不是做循环的工作，而是控制循环。下例的内务处理在循环末尾完成：

C++示例：循环末尾的内务处理语句

```
nameCount = 0;
totalLength = 0;
while ( !inputFile.EndOfFile() ) {
   // do the work of the loop
   inputFile >> inputString;
   names[ nameCount ] = inputString;
   ...

   // prepare for next pass through the loop--housekeeping
   nameCount++;
   totalLength = totalLength + inputString.length();
}
```

这些是内务处理语句

通常，在循环前初始化的变量就是要在循环的内务处理部分操作的变量。

每个循环只执行一个功能　虽然循环可同时做两件事，但这不足以成为非要一起做的理由。和子程序一样，每个循环都应只做一件事，而且要做得好。如果觉得在一个循环就够用的情况下使用两个循环效率不高，那么还是写成两个循环，注释说明它们也许能合并以提高效率。然后，等到基准测试表明程序的这一部分确实存在性能问题，再将两个改成一个。

关联参考　参见第 25 章和第 26 章更多地了解代码优化。

退出循环

下面几个小节描述了如何处理循环的结束：

确保循环会终止　这是最基本的要求。自己在心头模拟循环的执行，直到确信它无论如何都会终止。想想正常情况、端点以及每一种异常情况。

使循环结束条件显而易见　如果使用 for 循环，没有乱动循环索引，也没有使用 goto 或 break 来脱离循环，终止条件就会很明显。类似地，如果使用 while 或 repeat-until 循环，并将所有控制放在 while 或 repeat-until 子句中，终止条件也会很明显。关键是将控制放在一处。

不要乱动 for 循环索引来使循环终止　有的程序员对 for 循环索引值动手脚来使循环提前终止，如下例所示：

这里做了手脚

Java示例：乱动循环索引

```
for ( int i = 0; i < 100; i++ ) {
   // some code
   ...
   if ( ... ) {
      i = 100;
   }

   // more code
   ...
}
```

本例的目的是在符合某个条件时将 i 设为 100 来终止循环，该值比 for 循环的 0 到 99 的范围要大。几乎所有好的程序员都会避免这种做法；这显得很不专业。设置好 for 循环后，其循环计数器就不受你的控制了。可改用 while 循环对退出条件进行更多的控制。

避免依赖循环索引终值的代码 在循环之后使用循环索引的值不是一个好的做法。循环索引终值在不同语言和不同实现中是不相同的。循环正常终止和异常终止时，其值也是不同的。即便你碰巧想都不想就知道终值是什么，但下一个阅读代码的人可能不得不去思考它。在循环内部的适当位置将终值赋给一个变量，不仅代码的形式更好，在自我注释（self-documenting）方面也更佳。

以下代码滥用了索引终值：

这里滥用了循环索引的终值

C++示例：滥用循环索引的终值

```
for ( recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
   if ( entry[ recordCount ] == testValue ) {
      break;
   }
}
// lots of code
...
if ( recordCount < MAX_RECORDS ) {
   return( true );
}
else {
   return( false );
}
```

在这段代码中，对 recordCount < MaxRecords 的第二个测试使循环看起来像是要遍历 entry[] 数组中的所有值；找到与 testValue 相等的值就返回 true，否则返回 false。很难记住在越过循环尾后索引是否会递增，所以很容易出现“相差 1”错误。

最好写一些不依赖于索引终值的代码。下面重写了代码：

C++示例：没有滥用循环索引的终值

```
found = false;
for ( recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
   if ( entry[ recordCount ] == testValue ) {
      found = true;
      break;
   }
}
// lots of code
...
return( found );
```

这段重写的代码使用了一个额外的变量，并使对 recordCount 的相用更局部化。额外的布尔变量通常会使最终的代码更清晰。

考虑使用安全计数器 安全计数器（safety counter）是每完成一次循环迭代就递增的变量，目的是判断循环是否已执行太多次。在对安全性要求高的程序里，若错误会造成灾难性后果，就可用安全计数器确保所有循环都终止。以下 C++循环用安全计数器会很有好处：

C++示例：适合使用安全计数器的循环

```
do {
   node = node->Next;
   ...
} while ( node->Next != NULL );
```

以下是添加了安全计数器的版本：

这里是安全计数器代码

C++示例：使用安全计数器

```
safetyCounter = 0;
do {
   node = node->Next;
   ...
   safetyCounter++;
   if ( safetyCounter >= SAFETY_LIMIT ) {
      Assert( false, "Internal Error: Safety-Counter Violation." );
   }
   ...
} while ( node->Next != NULL );
```

安全计数器并非万能。每在代码中使用一次，都会增加一定的复杂性，并可能造成额外的错误。由于并不是每个循环中都会用它，所以以后修改程序中使用了安全计数器的那部分循环时，可能会忘记维护安全计数器代码。但是，如果安全计数器已被规定为项目范围内的关键循环的标准，以后就要习惯发现它们，安全计数器代码也不是说会比其他代码更容易造成错误。

提前退出循环

许多语言都提供了某种机制允许循环提前终止，而不必非要完成 for 或 while 条件。在我们的讨论中，会用 break 泛指 C++、C 和 Java 的 break；Visual Basic 的 Exit-Do 和 Exit-For；以及其他类似结构，包括在不直接支持 break 的语言中用 geto 来模拟的结构。break 语句（或等同形式）使一个循环通过正常的退出通道终止；程序将在循环后的第一个语句处恢复执行。

continue 语句与 break 相似，都是一种辅助性的循环控制语句。但是，continue 不会导致循环退出，而是使程序跳过当前迭代，开始循环的下一个迭代。contiune 语句是一个 if-then 子句的简写，用于阻止执行当前循环迭代剩余的部分。

考虑在 while 循环中使用 break 语句而不是布尔标志 在某些情况下，在 while 循环中加入布尔标志来模拟从循环体中退出，会使循环难以阅读。有的时候，可通过使用 break 语句而不是一系列的 if 测试来消除循环内部的几级缩进，并简化循环控制。将多个 break 条件放到单独的语句中，并使其位于产生中断的代码附近，可以减少嵌套，使循环更容易阅读。

警惕散布着大量 break 的循环 含有大量 break 的循环可能表明对循环的结构或者它在周边代码中的作用考虑得不清楚。大量的 break 意味着可能要将含有多个出口的一个循环拆分为多个循环，从而使代码更清晰。

根据 Software Engineering Notes 的一篇文章，1990 年 1 月 15 日导致纽约市电话系统停机 9 个小时的软件故障，其罪魁祸首就是多了一个 break 语句（SEN 1990）：

这个*break*原定用于*if*，但实际退出了整个*switch*

C++示例：在do-switch-if语句块中错误地使用了一个break语句

```
do {
   ...
   switch
      ...
      if () {
         ...
         break;
         ...
      }
      ...
} while ( ... );
```

多个 break 不一定意味着有错，但它们在循环中的存在是一个警告信号，是煤矿中的金丝雀，正在因为缺氧而窒息，而不是像本来应该的那样大声歌唱。

在循环顶部使用 continue 进行测试 continue 的一个很好的用途是在循环顶部测试了一个条件后，放弃此次对循环体的执行。例如，如循环的工作是读取记录，丢弃一种记录，并处理另一种记录，就可在循环顶部放一个如下例所示的测试：

伪代码示例：这样使用contiune相对安全

```
while ( not eof( file ) ) do
   read( record, file )
   if ( record.Type <> targetType ) then
      continue

   -- process record of targetType
   ...
end while
```

像这样使用 continue，可避免使用一个会造成整个循环体都缩进一级的 if 测试。另一方面，如 continue 出现在循环中间或末尾，就用 if 来代替。

如语言支持，就使用带标签的 break 结构 Java 支持带标签的 break，以防止出现纽约电话停机的那种问题。带标签的 break 可用来退出 for 循环、if 语句或任何用花括号封闭的代码块（Arnold, Gosling, and Holmes 2000）。

以下是纽约电话代码问题的一个可能的解决方案，编程语言从 C++改为 Java，以便用到带标签的 break：

Java示例：在do-switch-if代码块中，使用带标签的break语句更好

```
do {
   ...
   switch
      ...
      CALL_CENTER_DOWN:
      if () {
         ...

         break CALL_CENTER_DOWN;
         ...
      }
      ...
} while ( ... );
```

带标签的*break*使其目标一目了然

break 和 continue 使用需谨慎 一旦使用了 break，就没法子将循环体当作一个黑盒了。仅用一个语句来控制循环退出条件，是对循环进行简化的有力手段。只要用了 break，看你代码的人就会被迫查看循环内部以理解循环控制。这使循环变得更难以理解。

只有在考虑了其他所有选项之后，不得以才使用 break。你无法确定 continue 和 break 是良性还是恶性结构。一些计算机科学家认为它们是结构化编程中的合理技术；另一些人则不然。由于一般不知道 continue 和 break 是对是错，所以可以使用，但必须担心可能用错。这其实是一个简单的命题：只要无法证明 break 或 continue 的合理性，就不要用。

检查端点

循环通常只需关注三种情况：1.第一种情况；2. 随意选择的中间情况；以及最后一种情况。所以，在创建循环时，先在心头过一遍第一种、中间和最后一种情况，确保该循环没有任何“相差 1”错误。如存在任何有别于第一种和最后一种情况的特殊情况，也要检查那些情况。如循环中包含复杂的计算，拿出计算器手动核实。

愿不愿意进行这种检查，是高效和低效程序员的一个关键区别。高效的程序员会进行心头模拟和手动计算，因其知道这样做可帮助自己发现错误。

低效率的程序员倾向于随机试错，直到找到一个似乎有效的组合。如循环没有按设想的那样工作，低效的程序员会将<改为<=。如果还不行，低效的程序员又会对循环索引进行加 1 或减 1。这样做虽然可能碰巧发现正确的组合，但也可能用一个更不容易察觉的错误来取代原来的错误。即便这个瞎蒙的过程产生了一个正确的程序，程序员也不知道它为什么正确。

在心头模拟和手动计算有几方面的好处。在心头模拟，一开始的编码错误就会非常少，调试期间能更快地检测到错误，而且对程序的总体理解更佳。它还意味着你真的理解了代码，而不是纯粹靠蒙。

使用循环变量

以下是使用循环变量时的指导原则：

为数组和循环的限制使用序数或枚举类型　一般来说，循环计数器应该是整数值。浮点值不好递增。例如，26742897.0 加 1.0 得 26742897.0 而非 26742898.0。如递增的这个值是循环计数器，将得到一个无限循环。

> 关联参考　参见第 11.2 节详细了解如何为循环变量命名。

使用有意义的变量名改善嵌套循环的可读性　循环索引变量通常也用于索引数组。一维数组用什么变量名来索引都无所谓，i、j 或 k 都可以。但如果是二维或更多维度的数组，应该使用有意义的索引名称来明确你在做什么。有意义的数组索引名称既说明了循环的目的，也说明了你打算访问数组的哪一部分。

以下代码不符合这一指导原则，它使用了无意义的名称 i、j 和 k：

Java示例：糟糕的循环变量名称

```
for ( int i = 0; i < numPayCodes; i++ ) {
   for ( int j = 0; j < 12; j++ ) {
      for ( int k = 0; k < numDivisions; k++ ) {
         sum = sum + transaction[ j ][ i ][ k ];
      }
   }
}
```

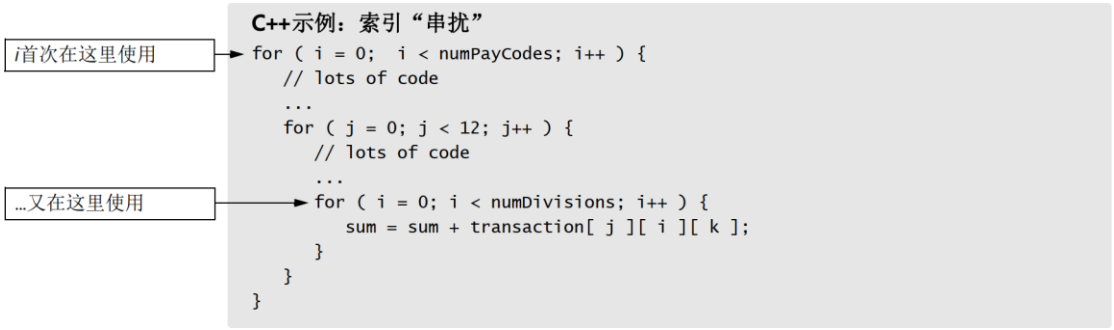
你认为 transcation 的数组索引是什么意思？i、j 和 k 能否告诉你关于 transaction 内容的任何事情？根据 transaction 的声明，能轻易确定索引顺序是否正确吗？下面是同一个循环，其循环变量名称更易读：

Java示例：好的循环变量名称

```
for ( int payCodeIdx = 0; payCodeIdx < numPayCodes; payCodeIdx++ ) {
   for (int month = 0; month < 12; month++ ) {
      for ( int divisionIdx = 0; divisionIdx < numDivisions; divisionIdx++ ) {
         sum = sum + transaction[ month ][ payCodeIdx ][ divisionIdx ];
      }
   }
}
```

这一次能明白 transcation 的数组索引的意思吗？修改之后就好懂多了，因为变量名 payCodeIdx、month 和 divisionIdx 比 i、j 和 k 更能说明问题。是的，这两个版本的循环在计算机眼中没有区别。但对人来说，第二个版本更容易阅读。而且第二个版本更好，因为读你代码的主要是人而非计算机。

使用有意义的名称来避免循环索引串扰　如习惯性地使用 i、j 和 k，容易造成索引发生“串扰”（cross-talk）——将同一个索引名称用于两个不同的目的。如下例所示：



i 用得是如此纯熟，以至于它在同一个嵌套结构中被使用了两次。由 i 控制的第二个 for 循环与第一个 for 循环发生了冲突，这就是所谓的索引“串扰”。使用比 i、j 和 k 更有意义的名称可以避免这个问题。通常，如循环主体不止两、三行，以后可能扩充，或者在一组嵌套循环中，就应避免 i、j 和 k。

将循环索引变量的作用域限制在循环内部 循环索引的串扰及其在循环外部的随意使用已造成了严重的问题，所以 Ada 的设计者决定使循环索引在其循环外部无效；试图在其所在的 for 循环外部使用会在编译时报错。

C++和 Java 在某种程度上实现了相同的理念——允许在循环内声明循环索引，只是不强求。在前几页的“C++示例：滥用循环索引的终值”中，recordCount 变量可在 for 语句中声明，这样即可将其作用域限制在 for 循环内，如下例所示：

C++示例：在*for*循环内部声明循环索引变量

```
for ( int recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
   // looping code that uses recordCount
}
```

原则上，这种技术应允许在多个循环中重新声明 recordCount，同时不会有滥用两个不同 recordCount 的风险，如此便可写出像下面这样的代码：

C++示例：在*for*循环内部声明循环索引并安全地重用—希望如此!

```
for ( int recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
   // looping code that uses recordCount
}
// intervening code
for ( int recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
   // additional looping code that uses a different recordCount
}
```

这个技术有助于为 recordCount 变量的用途添加注解；但是，不要依赖编译器来强制限定 recordCount 的作用域。《*The C++ Programming Language*》(Stroustrup 1997)一书第 6.3.3.1 节说，recordCount 的作用域应限于其循环。但当我用三个不同的 C++编译器检查这个功能时，却得到了三个不同的结果：

- 第一个编译器将第二个 for 循环中的 recordCount 标记为变量的重复声明并报错。
- 第二个编译器接受第二个 for 循环中的 recordCount，但允许在第一个 for 循环外部使用。
- 第三个编译器允许 recordCount 的两种用法，且不允许在各自 for 循环外部使用。

这很常见，和其他罕见的语言特性一样，编译器的实现各不相同。

多长的循环合适？

循环长度可用代码行数或嵌套深度来衡量。下面是一些指导原则：

循环要足够短，要让人一目了然 如一般在显示器上看循环，且一屏最多 50 行，这就给你带来了 50 行的限制。专家建议将循环长度限制为一页。但一旦你开始重视编写简单代码的原则，就很少会写超过 15 或 20 行的循环。

> 关联参考 参见第 19.4 节详细了解如何简化嵌套循环。

将嵌套限制在三层以内 研究表明，程序员对循环的理解能力在超过三层嵌套后会明显下降（Yourdon 1986a）。如果要超过这一层数，就将部分循环分解成子程序或简化控制结构，使循环（概念上）更短。

将长循环的内部代码移到子程序中 如果循环设计得好，循环内部的代码往往可以移到一个或多个子程序中，再从循环内部调用。

让长循环特别清晰 长度增加了复杂性。如果写的是短循环，可以使用风险较大的控制结构，如 break 和 continue、多个出口、复杂的终止条件等等。如果写的是较长的循环，并且怕你的读者搞不清楚，就限定该循环仅一个出口，并使退出条件明确无误。

16.3 轻松创建循环：由内而外

如果有时在写复杂循环时遇到困难（大多数程序员都会遇到），那么可以使用一个简单的技巧一开始就把它做对。下面是常规过程。先从一种情况开始。用字面量编码该情况。缩进它，用一个循环包围，然后将字面量替换成循环索引或计算的表达式。如有必要，再用一个循环包围它，再替换更多字面量。只要需要，就继续这一过程。完成后，添加所有必要的初始化代码。由于是从简单情况开始，再向外扩展，所以可认为这是由内而外的编码。

> 关联参考 由内而外编写循环的过程类似于第 9 章所描述的过程。

假设要为某保险公司写一个程序。它的人寿保险费率因人的年龄和性别而异。你的工作是写子程序来计算一组人员总的人寿保险费。需要一个循环，将列表中每个人的费率加到总额中。下面是你要做的事情。

首先，在注释中写出循环主体需要执行的步骤。在不考虑语法、循环索引、数组索引等细节时，写下需要做的事情会更容易。

步骤1：由内向外创建循环(伪代码示例)

```
-- 从表格中获取费率
-- 将费率加到总额上
```

然后，在不实际写整个循环的情况下，将循环主体中的注释尽可能多地转换成代码。本例是获取一个人的费率并加到总额。使用具体、特定的数据，而不要使用抽象的。

*table*还没有任何索引

步骤2：由内向外创建循环(伪代码示例)

```
rate = table[ ]
totalRate = totalRate + rate
```

本例假定 table 是容纳费率数据的数组。最开始不必关心数组索引。rate 变量容纳从费率表选择的费率数据。类似地，totalRate 变量容纳费率总额。

接着为 table 数组添加索引：

步骤3：由内向外创建循环(伪代码示例)

```
rate = table[ census.Age ][ census.Gender ]
totalRate = totalRate + rate
```

数组通过年龄和性别来访问，所以用 census.Age 和 census.Gender 来索引数组。本例假定 census 是一个容纳了计费组内人员信息的结构。

下一步是围绕现有语句构建循环。由于循环的工作是计算组内每个人的费率，所以循环应按人来索引。

步骤4: 由内向外创建循环(伪代码示例)

```
For person = firstPerson to lastPerson
   rate = table[ census.Age, census.Gender ]
   totalRate = totalRate + rate
End For
```

这里只需用 for 循环包围现有代码，然后缩进现有代码，并将其放到一个 begin-end 对中。最后，核实依赖于 person 循环索引的变量已进行了常规化（generalized）。在本例中，census 变量随 person 而变，所以要相应地常规化。

步骤5：由内向外创建循环(伪代码示例)

```
For person = firstPerson to lastPerson
   rate = table[ census[ person ].Age, census[ person ].Gender ]
   totalRate = totalRate + rate
End For
```

最后写必要的初始化代码。在本例中，totalRate 变量需要初始化。

最后一步：由内向外创建循环(伪代码示例)

```
totalRate = 0
For person = firstPerson to lastPerson
   rate = table[ census[ person ].Age, census[ person ].Gender ]
   totalRate = totalRate + rate
End For
```

如果要用另一个循环包围 person 循环，可采取相同的方式继续。不需要死板遵循这些步骤。我们的思路是，从具体的东西开始，一次只关注一件事，然后从简单的组件开始建立循环。在循环变得越来越常规和复杂的过程中，每次都采取小的、可理解的步骤。这样就能最大限度减少在任何时候都必须关注的代码量，从而最大限度减少出错的可能性。

16.4 循环和数组的对应关系

关联参考　参见第 10.7 节进一步了解循环和数组的对应关系。

循环和数组通常是相关的。许多时候需要创建循环来执行数组操作，而且循环计数器与数组索引一一对应。例如，以下 Java for 循环索引对应于数组索引：

Java示例：数组乘法

```
for ( int row = 0; row < maxRows; row++ ) {
   for ( int column = 0; column < maxCols; column++ ) {
      product[ row ][ column ] = a[ row ][ column ] * b[ row ][ column ];
   }
}
```

在 Java 中，这种数组操作确实需要一个循环。但要注意，循环结构和数组并不是天生就联系在一起的。某些语言（尤其是 APL 和 Fortran 90 及更高版本）支持强大的数组操作，从而消除了对上述循环的需要。以下 APL 代码执行相同的操作：

APL示例：数组乘法

```
product <- a x b
```

APL 更简单，更不易出错。它只使用了三个操作数，而上述 Java 代码使用了 17 个。它没有循环变量、数组索引或控制结构，能最大程度地避免错误编码。

这个例子的一个要点是，做一些编程来解决问题时，解决方案可能会依赖于特定的语言。用于解决问题的语言可能极大影响你的解决方案。

检查清单： 循环

循环的选择和创建

- 在合适的时候采用 while 循环取代 for 循环了吗？
- 循环是由内向外创建的吗？

进入循环

- 是从顶部进入循环的吗？
- 初始化代码直接放在循环前面了吗？
- 如果是无限循环或事件循环，其结构是否清晰，而不是采用类似 for i=1 to 9999 这样蹩脚的代码？
- 如果循环属于 C++、C 或者 Java 的 for 循环，循环控制代码都放在循环头里了吗？

循环内部

- 是否使用{和}或其等价形式来封闭循环体，以防止因修改不当而出错吗？
- 循环体里面有内容吗？它是非空的吗？
- 内务处理代码集中存放在循环开始或者循环结束的位置了吗？
- 循环就像定义良好的子程序那样，只执行一种功能吗？
- 循环是否足够短，让人一目了然吗？
- 循环的嵌套层数控制在三层或者更少层数以内吗？
- 长循环的内容转移到相应的子程序中了吗？
- 如果循环很长，它特别清晰吗？

循环索引

- for 循环体内的代码有没有随意改动循环索引值？
- 是否专门用变量保存重要的循环索引值，而不是在循环体外部使用循环索引？
- 循环索引是整数类型或者枚举类型，而不是浮点类型吗？
- 循环索引的名称有意义吗？
- 循环避免了索引串扰问题吗？

退出循环

- 循环在所有可能的情况下都能终止吗？
- 如已规定了安全计数器标准，循环使用安全计数器了吗？
- 循环的终止条件是显而易见的吗？
- 如果用到了 break 或者 continue 语句，它们的用法正确吗？

要点回顾

- 循环比较复杂。保持简单有助于别人看你的代码。
- 保持循环简单的技巧包括：避免使用怪异的循环、减少嵌套、提供清晰的入口和出口、将内务处理代码集中在一个地方。
- 循环索引很容易被滥用。要清楚地命名，并限定一种用途。
- 想清楚循环结构，确定在每种情况下都能正常运行，而且在所有可能的情况下都能终止。

第 17 章　不常见的控制结构

内容

- 17.1 子程序中的多处返回
- 17.2 递归
- 17.3 goto
- 17.4 针对不常见控制结构的观点

相关章节

- 常规控制问题：第 19 章
- 直线型代码：第 14 章
- 条件代码：第 15 章
- 循环代码：第 16 章
- 异常处理：第 8.4 节

有几个控制结构存在于一个朦胧的边缘地带，介于先进和被否定之间——而且往往是同时存在于这两个地方！这些结构并非所有语言都有，但在提供了这些结构的语言中，只要小心使用，就会很有用。

17.1　子程序中的多处返回

大多数语言都支持在子程序中途退出的一些方法。作为一种控制结构，return 和 exit 语句允许程序随意从子程序中退出。它们造成子程序不再走正常的退出通道，直接将控制权返回给发出调用的子程序。这里说的 return 泛指 C++和 Java 的 return、Microsoft Visual Basic 的 Exit Sub 和 Exit Function 以及其他类似结构。下面是使用 return 语句的指导原则：

KEY POINT **在有助于增强可读性时使用 return**　在某些子程序中，一旦获得了答案，就想立即把它返回给调用子程序。如果某个子程序被定义成在检测到错误时不需要执行进一步清理，不立即返回就意味着必须写更多的代码。

下例很好地说明了从一个子程序中的多个地方返回是有意义的：

该子程序返回一个 *Comparison*枚举类型

C++示例：从子程序多个地方返回的好例子

```
Comparison Compare( int value1, int value2 ) {
   if ( value1 < value2 ) {
      return Comparison_LessThan;
   }
   else if ( value1 > value2 ) {
      return Comparison_GreaterThan;
   }
   return Comparison_Equal;
}
```

如下一小节所述，其他例子则不那么明显。

使用防卫子句（提前返回或退出）来简化复杂的错误处理　执行正常行动之前必须检查许多错误情况的代码会导致代码深度缩进，并可能掩盖正常情况，如下例所示：

Visual Basic示例：掩盖了正常情况

```
If file.validName() Then
   If file.Open() Then
      If encryptionKey.valid() Then
         If file.Decrypt( encryptionKey ) Then
            ' lots of code
            ...
         End If
      End If
   End If
End If
```

这是用于正常情况的代码

将子程序的主体缩进到四个 if 语句里面不太雅观，尤其是假如最里面的 if 语句内还有很多代码。在这种情况下，先检查错误情况可能会使代码的流程更清晰，这样可以更清楚地看出代码的正常路径，如下所示：

Visual Basic示例：使用防卫子句澄清正常情况

```
' set up, bailing out if errors are found
If Not file.validName() Then Exit Sub
If Not file.Open() Then Exit Sub
If Not encryptionKey.valid() Then Exit Sub
If Not file.Decrypt( encryptionKey ) Then Exit Sub

' lots of code
...
```

当然，本例非常简单，使得这个技术看起来像一个整洁的解决方案。但是，生产代码往往需要在检测到错误情况时进行更全面的内务处理或清理。下面是一个更现实的例子：

更真实的Visual Basic示例：使用防卫子句澄清正常情况

```
' set up, bailing out if errors are found
If Not file.validName() Then
   errorStatus = FileError_InvalidFileName
   Exit Sub
End If

If Not file.Open() Then
   errorStatus = FileError_CantOpenFile
   Exit Sub
End If

If Not encryptionKey.valid() Then
   errorStatus = FileError_InvalidEncryptionKey
   Exit Sub
End If

If Not file.Decrypt( encryptionKey ) Then
   errorStatus = FileError_CantDecryptFile
   Exit Sub
End If

' lots of code
...
```

这是用于正常情况的代码

对于生产代码，Exit Sub 方法会在处理正常情况之前产生很大的代码量。但是，Exit Sub 方法确实避免了第一个例子中的深度嵌套。另外，若扩展第一个例子中的代码以加入对 errorStatus 变量的赋值，Exit Sub 方法还能更好地将相关语句保持在一处。无论如何，Exit Sub 方法在可读性和可维护性上确实有所提升，虽然幅度有限。

尽量减少每个子程序中的 return 数量 看子程序的底部时，如果不知道它可能在上面某个地方返回，就很难理解它。出于这个原因，只有在能提高可读性的情况下，才斟酌着使用return。

17.2 递归

在递归中，子程序解决问题的一小部分，将问题分解为更小的部分，再调用自身来解决每一个更小的部分。若问题的一小部分很容易解决，且一个大部分很容易分解成小块时，通常就可考虑使用递归。

递归并不是经常都有用，但用得合理的话会产生非常优雅的解决方案。在下例中，一个排序算法很好地利用了递归。

这里是递归调用

Java示例：使用递归的一个排序算法

```
void QuickSort( int firstIndex, int lastIndex, String [] names ) {
   if ( lastIndex > firstIndex ) {
      int midPoint = Partition( firstIndex, lastIndex, names );
      QuickSort( firstIndex, midPoint-1, names );
      QuickSort( midPoint+1, lastIndex, names )
   }
}
```

在这个例子中，排序算法将数组一分为二，然后调用自身对数组的每一半进行排序。一旦子数组太小以至于无法继续排序——例如(lastIndex <= firstIndex)——它就停止调用自身。

对于一小部分问题，递归可能产生简单、优雅的解决方案。对于稍大的一部分问题，它可能产生简单、优雅但难以理解的解决方案。对于大多数问题，它会产生非常复杂的解决方案——对于这些情况，简单的迭代通常更容易理解。选择性地使用递归。

递归的例子

假定有一个代表迷宫的数据类型。迷宫基本上是一个网格，在网格上的每个点，可以向左转、向右转、向上移动或向下移动。经常能朝一个以上的方向移动。

如图 17-1 所示，如何写一个程序来寻找走出迷宫的方法？如果使用递归，答案是相当直接的。从入口开始，尝试所有可能的路径，直到走出迷宫。首次到达一个点时，尝试向左移动。如果不能向左，就尝试向上或向下。如果不能向上或向下，就尝试向右。不必担心迷路，因为每次达到一个点，都会在这个点上撒下一些面包屑，从而避免访问同一个点两次。

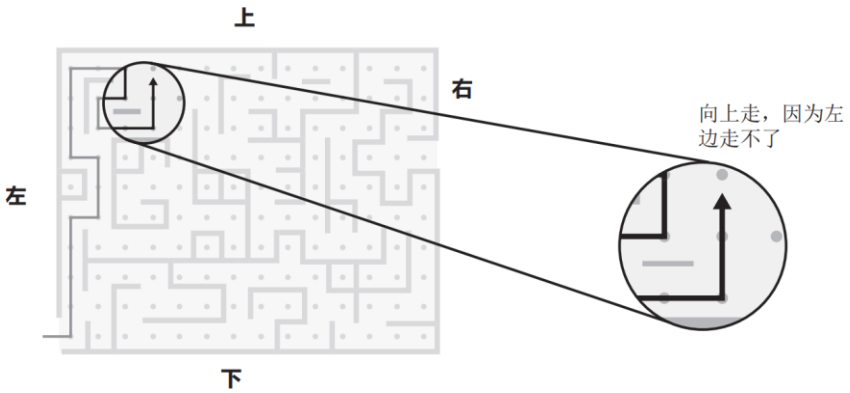


图 17-1　递归可以是对抗复杂性的一个有价值的工具——前提是问题要合适

以下是递归代码：

C++示例：用递归走迷宫

```
bool FindPathThroughMaze( Maze maze, Point position ) {
   // if the position has already been tried, don't try it again
   if ( AlreadyTried( maze, position ) ) {
      return false;
   }

   // if this position is the exit, declare success
   if ( ThisIsTheExit( maze, position ) ) {
      return true;
   }

   // remember that this position has been tried
   RememberPosition( maze, position );

   // check the paths to the left, up, down, and to the right; if
   // any path is successful, stop looking
   if ( MoveLeft( maze, position, &newPosition ) ) {
      if ( FindPathThroughMaze( maze, newPosition ) ) {
         return true;
      }
   }

   if ( MoveUp( maze, position, &newPosition ) ) {
      if ( FindPathThroughMaze( maze, newPosition ) ) {
         return true;
      }
   }

   if ( MoveDown( maze, position, &newPosition ) ) {
      if ( FindPathThroughMaze( maze, newPosition ) ) {
         return true;
      }
   }

   if ( MoveRight( maze, position, &newPosition ) ) {
      if ( FindPathThroughMaze( maze, newPosition ) ) {
         return true;
      }
   }
   return false;
}
```

第一行代码检查这个点是否已经尝试过。写递归程序的一个关键目标是防止无限递归。在本例中，如果不检查一个点是否已经尝试，就可能无限地尝试。

第二个语句检查这个点是不是迷宫出口。如 ThisIsTheExit()返回 true，则子程序本身返回 true。

第三个语句记住这个点是否已经访问过。这可以防止因为循环路径而造成无限递归。

子程序其余几行代码试图找到一条向左、向上、向下和向右的路径。只要子程序返回 true（换言之，子程序找到了通过迷宫的路径），代码就停止递归。

这个子程序所采用的逻辑是相当直观的。大多数人最初使用递归时都会有一些不适感，因为它在自己调用自己。但就本例来说，其他解决方案会复杂得多，而递归的效果很好。

递归使用技巧

使用递归时注意以下几个技巧：

确保递归停止 核实子程序包括一个非递归路径。这通常意味着该子程序要有一个测试，在不需要时停止进一步递归。在迷宫例子中，对 AlreadyTried()和 ThisIsTheExit()的测试确保了递归的停止。

使用安全计数器防止无限递归 如果无法进行上述简单测试，就用安全计数器来防止无限递归。安全计数器必须是一个每次调用子程序都不会重建的变量。可用类成员变量或将安全计数器作为参数传递，如下例所示：

递归子程序必须能更改*safetyCounter*变量的值，所以该变量在Visual Basic中是一个*ByRef*参数

Visual Basic示例：用安全计数器防止无限递归

```
Public Sub RecursiveProc( ByRef safetyCounter As Integer )
   If ( safetyCounter > SAFETY_LIMIT ) Then
      Exit Sub
   End If
   safetyCounter = safetyCounter + 1
   ...
   RecursiveProc( safetyCounter )
End Sub
```

在本例中，子程序超出安全限制（SAFETY_LIMIT）就停止递归。

如果不想将安全计数器作为一个显式的参数来传递，可使用 C++、Java 或 Visual Basic 中的成员变量，或使用其他语言提供的等价物。

将递归限制在一个子程序中 循环递归（A 调用 B，B 调用 C，C 调用 A）很危险，因其很难理顺。在心头管理一个子程序中的递归就已经很难了；理解跨子程序的递归就太难了。如果你有循环递归，通常可以重新设计子程序，使递归被限制在单个子程序中。如果做不到，同时仍然认为递归是最好的方案，就用安全计数器作为递归的保险。

留意栈 使用递归时无法保证程序用了多少栈空间，也很难提前预测程序在运行时的表现。但是，可采取几个步骤来控制其运行时的行为。

首先，如准备使用安全计数器，那么为其设置限制时，一个考虑因素是愿意为递归子程序分配多少栈空间。请将安全限制设得足够低，以防栈溢出。

其次，注意递归函数中局部变量的分配，尤其是那些内存消耗大的对象。换言之，使用 new 在堆上创建对象，不要让编译器在栈上创建自动对象。

计算阶乘或斐波那契数不要用递归 计算机科学教科书的一个问题是，它们经常展示一些愚蠢的递归例子。典型的例子是阶乘（或斐波那契数列）的计算。递归很强大，但在这两种情况下使用它很愚蠢。如果一个为我工作的程序员用递归来计算阶乘，我情愿雇别人。

下面是阶乘子程序的递归版本：

Java示例：用递归计算阶乘——不恰当的方案

```
int Factorial( int number ) {
   if ( number == 1 ) {
      return 1;
   }
   else {
      return number * Factorial( number - 1 );
   }
}
```

相较于它的迭代版本（如下所示），它除了速度慢，无法预测运行时内存的使用，还更难理解。

Java示例：用迭代计算阶乘——恰当的方案

```
int Factorial( int number ) {
   int intermediateResult = 1;
   for ( int factor = 2; factor <= number; factor++ ) {
      intermediateResult = intermediateResult * factor;
   }
   return intermediateResult;
}
```

从这个例子可得出三个教训。首先，计算机科学教科书中的递归例子并没有给世界带来任何好处。其次，更重要的是，在计算阶乘或斐波那契数时使用阶乘，并不能反映出阶乘真正的强大之处。第三，也是最重要的，决定使用递归之前，应考虑递归的替代方案。可以用栈和迭代做任何递归能做的事情。有时一种方法效果更好，有时另一种方法效果更好。做出决定之前，这两种方法都要考虑。

17.3 goto

你可能以为与 goto 相关的争论已经绝迹，但快速浏览一下现代的源代码库（如 GitHub）就会发现，goto 仍然活得很好，而且早已深深植入你公司的服务器中。此外，现代的 goto 争论仍以各种名义出现，其中涉及多个返回点、多个循环出口、具名循环出口、错误处理和异常处理。

反对 goto 的理由

反对 goto 的一般理由是，没有 goto 的代码是质量更高的代码。引发最初争议的著名信件是 Edsger Dijkstra 在 1968 年 3 月的《*Communications of the ACM*》上发表的“Go To Statement Considered Harmful”。Dijkstra 观察到，代码质量与程序员使用的 goto 数量成反比。在其随后的作品中，Dijkstra 认为，不含 goto 的代码更容易被证明是正确的。

含 goto 的代码很难格式化。缩进本应用于显示逻辑结构，而 goto 影响了逻辑结构。然而，用缩进来显示 goto 及其目标的逻辑结构非常困难，甚至是不可能的。

用 gotos 会使编译器的优化失效。一些优化依赖于程序的控制流程停留在几个语句中。一个无条件的 goto 使控制流程更难分析，降低了编译器优化代码的能力。所以，即使引入 goto 在源语言层面上产生了效率，也可能因为阻碍了编译器的优化而降低整体效率。

支持 goto 的人有时会争论说，它们使代码更快、更小。但是，包含 goto 的代码很少是最快或最小的。Donald Knuth 的经典文章“Structured Programming with go to Statements”（用 go to 语句进行结构化编程）给出了几个例子，说明使用 goto 会使代码变慢和变大（Knuth 1974）。

在实践中，goto 的使用违反了代码应严格自上而下流动的原则。即使 goto 在谨慎使用的前提下不会造成混乱，但一旦引入，它们就会像白蚁在腐烂的房子里一样在代码中传播。如果允许了任何 goto，局面就会变得不可控，好的和坏的都可能发生。所以，不如干脆完全禁止。

总之，在 Dijkstra 的信发表后的 20 年里，经验表明，编写充满 goto 的代码是愚蠢的。在一份文献调查（survey of the literature）中，Ben Shneiderman 总结说，证据支持 Dijkstra 的观点，即我们最好不要使用 goto（1980）。另外，许多现代语言，包括 Java，甚至根本没有 goto。

赞成 goto 的理由

赞成 goto 的理由主要是这个东西只要不滥用，在特定情况下谨慎使用就能发挥奇效。大多数人之所以反对 goto，是因为它很容易被无脑地使用。当 Fortran 是最流行的语言时，goto 的争论就爆发了。Fortran 没有现成的循环结构，当时也没人为用 goto 编码循环提出良好的建议，所以程序员写了许多意大利面条式的代码[7]。这样的代码无疑和低质量产品联系在一起，但它与谨慎使用 goto 来弥补现代语言能力的不足其实没有直接联系。

一个位置良好的 goto 可以消除对重复代码的需求。如果两套代码的修改方式不同，重复的代码就会导致问题。重复的代码会增加源文件和可执行文件的大小。在这种情况下，goto 的坏影响被重复代码的风险所抵消。

只要位置得当，goto 可消除重复代码的必要。重复的两套代码在修改时如果不一致，就会导致问题。重复的代码会增加源文件和可执行文件的大小。在这种情况下，goto 的坏影响被重复代码的风险所抵消。

在分配资源，对这些资源进行操作，然后释放资源的子程序中，goto 很有用。此时可用 goto 在一段代码里集中完成清理工作。goto 减少了在每个检测到错误的地方忘记释放资源的可能性。

关联参考　有关如何在分配资源的代码中使用 goto 语句的详情，请参阅本节的“错误处理和 goto”。同时参见第 8.4 节关于异常处理的讨论。

goto 有时能产生更快和更小的代码。Knuth 在 1974 年的那篇文章里也提到了 goto 能提供正面收益的例子。

不是说不用 goto 你的程序就写得好。大多数时候，对控制结构进行有条理的分解、提炼和选择，会自动导致无 goto 的程序。实现无 goto 的代码并非目的，而是结果。将重点放在避免 goto 上是缘木求鱼。

[7] 意大利面条式代码（spaghetti code）是软件工程中反面模式的一种，是指源代码的控制流程复杂、混乱而难以理解，尤其是用了很多 GOTO、例外、线程或其他无组织的分支。其命名的原因是因为程序的流向就像意面那样扭曲而纠结。

几十年来对 goto 的研究未能证明其危害性。在一次文献调查中，B.A. Sheil 总结说，不现实的测试条件、糟糕的数据分析以及不确定的结果都不能支持 Shneiderman 和其他人的说法，即代码中的 bug 数量与 goto 数量成正比（1981）。当然，Sheil 也不是说使用 goto 是一个好主意——只是是说反对 goto 的实验证据不确凿。

> “这些证据只能证明有意采用杂乱不堪的控制结构会降低(程序员)的效能，不能证明对控制流程进行结构化的任何一种方法能带来有益的影响。”
>
> —B. A. Sheil

最后，goto 已被纳入许多现代语言，包括 Visual Basic、C++ 和有史以来设计得最用心的 Ada 编程语言。Ada 是在 goto 争论双方的论点充分发展后很久才开发的，在考虑了问题的所有方面后，Ada 的工程师决定将 goto 包括在内。

没有抓住重点的 goto 争论

大多数关于 goto 的争论都没有抓住重点。“goto 坏得很”一方首先展示使用 goto 的一个无足轻重的代码片段，然后展示在不用 goto 的情况下重写有多容易。这主要证明的是无需 goto 即可轻松写出一些无足轻重的代码。

“goto 好得很”这一方通常会提出这样一种情况，即删除 goto 会导致额外的比较或一行代码的重复。这主要证明的是在某些时候，使用 goto 会导致更少的比较——这对今天的计算机来说并不是一个显著的增益。

大多数教科书在这方面也没有什么帮助。它们可能提供一个简单的例子，说明能在没有 goto 的情况下重写一些代码，好像这样就把整个事情交待清楚了一样。下面就是一个没有抓住重点的例子，它来自这种教科书的一段无足轻重的代码：

C++示例：试图证明不用*goto*也能轻松重写

```
do {
   GetData( inputFile, data );
   if ( eof( inputFile ) ) {
      goto LOOP_EXIT;
   }
   DoSomething( data );
} while ( data != -1 );
LOOP_EXIT:
```

书里面很快将其替换成无 goto 的版本：

C++示例：以为等价的、不用*goto*来重写的代码

```
GetData( inputFile, data );
while ( ( !eof( inputFile ) ) && ( ( data != -1 ) ) ) {
   DoSomething( data );
   GetData( inputFile, data )
}
```

这个所谓的“无足轻重”的例子包含一个错误。在 data 等于-1 的情况下进入循环，重写的代码检测到了-1 并在执行 DoSomething()之前就退出循环。而原代码在检测到-1 之前会执行 DoSomething()。那本试图展示不用 goto 也能轻松编程的书错误地翻译了自己的例子。但那

本书的作者也别往心里去，因为其他书也会犯类似的错误。即使是专家，在翻译使用 goto 的代码时也会有困难。

下面才是不用 goto 的忠实翻译：

C++示例：真正等价的、不用*goto*来重写的代码

```
do {
   GetData( inputFile, data );
   if ( !eof( inputFile )) {
      DoSomething( data );
   }
} while ( ( data != -1 ) && ( !eof( inputFile ) ) );
```

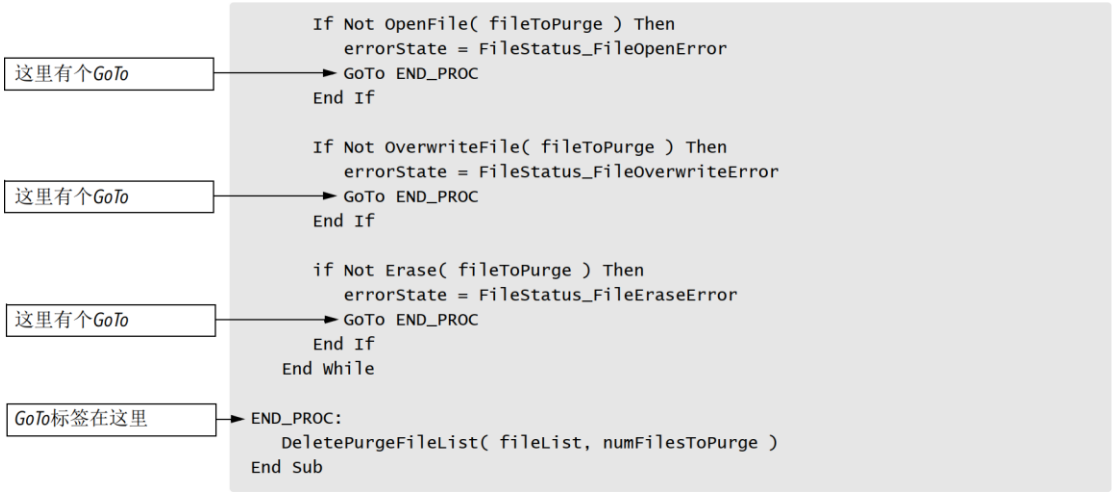
即使对代码进行了正确的翻译，这个例子仍然没有抓住重点，因其只展示了 goto 的一个无足轻重的使用情况。若深思熟虑的程序员选择 goto 作为其首选控制方式，针对的绝不是这种情况。

都这么晚了，我们很难为理论上的 goto 辩论添加任何有价值的东西。但有一个情况许多人都没有提到，即程序员可能在通盘考虑了 goto 的替代方案后，最终还是选择用 goto 改善可读性和可维护性。

以下小节介绍了一些有经验的程序员主张使用 goto 的案例。我们展示了用 goto 和不用 goto 的两个版本，并评估了两者的利与弊。

错误处理和 goto

写高度交互式的代码时，需要重点关注错误处理和错误发生时的资源清理。以下代码清除一组文件。子程序首先获得一组要清除的文件，然后找到每个文件，打开，重写，并删除。子程序在每一步都会检查是否出错。



这个子程序是有经验的程序员选择使用 goto 的典型情况。程序需要分配和清理资源（如数据库连接、内存或临时文件）时，也会出现类似的情况。在这些情况下，不用 goto 的替代方案通常都要求重复资源清理代码。所以，在权衡 goto 了的坏处和对重复代码进行维护的麻烦后，程序员会两害相权取其轻，最终选择 goto。

有几种方法都可重写前面的子程序以避免 goto，但均需权衡利弊。可能的重写策略包括：

用嵌套 if 语句重写　可将 if 语句嵌套起来，这样每个 if 语句只有在之前的测试成功的情况下才会执行。这是消除 goto 的标准的、教科书式的编程方法。下面是用标准方法重写的程序：

> 关联参考　该子程序也可用 break 重写，同样无需用到 goto。详情参见 16.2 节的“提前退出循环”。

Visual Basic示例：用嵌套if避免goto

```
' This routine purges a group of files.
Sub PurgeFiles( ByRef errorState As Error_Code )
   Dim fileIndex As Integer
   Dim fileToPurge As Data_File
   Dim fileList As File_List
   Dim numFilesToPurge As Integer

   MakePurgeFileList( fileList, numFilesToPurge )

   errorState = FileStatus_Success
   fileIndex = 0
   While ( fileIndex < numFilesToPurge And errorState = FileStatus_Success )

      fileIndex = fileIndex + 1

      If FindFile( fileList( fileIndex ), fileToPurge ) Then
         If OpenFile( fileToPurge ) Then
            If OverwriteFile( fileToPurge ) Then
               If Not Erase( fileToPurge ) Then
                  errorState = FileStatus_FileEraseError
               End If
            Else ' couldn't overwrite file
               errorState = FileStatus_FileOverwriteError
            End If
         Else ' couldn't open file
            errorState = FileStatus_FileOpenError
         End If
      Else ' couldn't find file
         errorState = FileStatus_FileFindError
      End If
   End While
   DeletePurgeFileList( fileList, numFilesToPurge )
End Sub
```

修改了*While*测试，添加了对*errorState*的测试

这一行距调用它的*If*语句有13行之远

对于习惯不用 goto 编程的人来说，这段代码比 goto 版本更容易阅读。采用这种方法，你将不必面对 goto 执法队的拷问。

> 关联参考　缩进和其他代码排版问题的详情请参见第 31 章。嵌套层级的详情请参见第 19.4 节。

这种嵌套 if 方法的主要缺点在于，它的嵌套层级很深，非常深。为了理解代码，必须将整套嵌套 if 同时记在心里。此外，错误处理代码和调用它的代码之间的距离太大。例如，将 errorState 设为 FileStatus_FileFindError 的代码距离调用它的 if 语句有 13 行之远。

而在 goto 版本中，任何语句都不会离调用它的条件超过 4 行。另外，不必将整个结构都记在心里。基本上可以忽略之前成功的任何条件，只需专注于下一个操作。在本例中，goto 版本比嵌套 if 版本更具可读性和可维护性。

用状态变量重写　为了用状态变量重写，需创建一个变量来指示子程序是否处于错误状态。在本例中，该子程序已经使用了 errorState 状态变量，所以可以直接用它。

Visual Basic示例：用状态变量避免*goto*

```
' This routine purges a group of files.
Sub PurgeFiles( ByRef errorState As Error_Code )
   Dim fileIndex As Integer
   Dim fileToPurge As Data_File
   Dim fileList As File_List
   Dim numFilesToPurge As Integer

   MakePurgeFileList( fileList, numFilesToPurge )

   errorState = FileStatus_Success
   fileIndex = 0
   While ( fileIndex < numFilesToPurge ) And ( errorState = FileStatus_Success )

      fileIndex = fileIndex + 1

      If Not FindFile( fileList( fileIndex ), fileToPurge ) Then
         errorState = FileStatus_FileFindError
      End If
      If ( errorState = FileStatus_Success ) Then
         If Not OpenFile( fileToPurge ) Then
            errorState = FileStatus_FileOpenError
         End If
      End If
      If ( errorState = FileStatus_Success ) Then
         If Not OverwriteFile( fileToPurge ) Then
            errorState = FileStatus_FileOverwriteError
         End If
      End If
      If ( errorState = FileStatus_Success ) Then
         If Not Erase( fileToPurge ) Then
            errorState = FileStatus_FileEraseError
         End If
      End If
   End While
   DeletePurgeFileList( fileList, numFilesToPurge )
End Sub
```

修改了*While*测试，添加了对*errorState*的测试

测试状态变量

测试状态变量

测试状态变量

状态变量方法的优点在于，它避免了第一个重写版本中深度嵌套的 if-then-else 结构，所以更容易理解。它还将 if-then-else 测试之后的行动放在比嵌套 if 方法更接近测试的地方，而且完全避免了 else 子句。

理解嵌套 if 版本需要费一些脑力。状态变量版本更易理解，因其更接近人思考问题的方式。首先找到文件。如一切正常，就打开该文件。如一切正常，就覆盖该文件。如一切还是正常……

这个方法的缺点在于，使用状态变量并不像它应该的那样普遍。要用注释解释它们的使用，否则一些程序员可能不明白你在做什么。在本例中，使用命名良好的枚举类型大有帮助。

用 try-finally 重写　包括 Visual Basic 和 Java 在内的一些语言提供了 try-finally 语句，可用它在出错时清理资源。

为了用 try-finally 方法重写，要将检查错误的代码放在 try 块中，将清理代码放在 finally 块中。try 块指定了异常处理的范围，而 finally 块负责清理资源。无论是否抛出异常，也无论 PurgeFiles()子程序是否捕捉到任何抛出的异常，finally 块都会被调用。

Visual Basic示例：用try-finally避免goto

```
' This routine purges a group of files. Exceptions are passed to the caller.
Sub PurgeFiles()
   Dim fileIndex As Integer
   Dim fileToPurge As Data_File
   Dim fileList As File_List
   Dim numFilesToPurge As Integer
   MakePurgeFileList( fileList, numFilesToPurge )
   Try
      fileIndex = 0
      While ( fileIndex < numFilesToPurge )
         fileIndex = fileIndex + 1
         FindFile( fileList( fileIndex ), fileToPurge )
         OpenFile( fileToPurge )
         OverwriteFile( fileToPurge )
         Erase( fileToPurge )
      End While
   Finally
      DeletePurgeFileList( fileList, numFilesToPurge )
   End Try
End Sub
```

这个方法假定所有函数调用在失败时都会抛出异常，而不是返回错误代码。

try-finally 方法的优点在于，它比 goto 方法更简单，而且用不到 goto。它还避免了深度嵌套的 if-then-else 结构。

try-finally 方法的局限在于，它必须在整个代码库中统一实现。如之前的代码是除了异常之外还使用了错误代码的代码库的一部分，就会要求异常代码为每个可能的错误设置错误代码。一旦存在这方面的要求，代码就会变得和其他方法一样复杂。

关联参考　参见第 19.4 节获得适用于这种情况的完整技术清单。

各种方法的对比

四种方法的每一种都有其可取之处。goto 方法避免了深度嵌套和不必要的测试，但当然要用到 goto。嵌套 if 方法避免了 gotos，但它深度嵌套，给人一种逻辑复杂性非常夸张的印象。状态变量方法避免了 goto 和深度嵌套，但又引入了额外的测试。try-finally 方法既避免了 goto，也避免了深度嵌套，但并非所有语言都适用。

若语言提供了 try-finally，且代码库尚未用另一种方法标准化，那么 try-finally 方法是显而易见的选择。如果用不了 try-finally，那么状态变量方法比 goto 和嵌套 if 方法略胜一筹，因其更易读，且能更好地建模问题。但是，这并不意味着它是所有情况下最佳的方法。

若对一个项目中的所有代码进行一致的应用，这些技术中的任何一种都能很好地工作。先考虑好所有方法的得失，再从整个项目的角度决定具体采用哪一种。

goto 和 else 子句中的共享代码

有的程序员会在一个颇具挑战性的情况下使用 goto，即当前有两个条件测试和一个 else 子句，而你既想执行其中一个条件的代码，也想执行 else 子句中的代码，如下例所示：

C++示例：用goto共享else子句中的代码

```
if ( statusOk ) {
   if ( dataAvailable ) {
      importantVariable = x;
      goto MID_LOOP;
   }
}
else {
   importantVariable = GetValue();

   MID_LOOP:

   // lots of code
   ...
}
```

这是一个很好的例子，因为它在逻辑上很曲折——几乎不可能顺着读下来，而且不用 goto 也很难正确重写。如果你是可以在没有 goto 的情况下轻松重写它，请别人审查一下你的代码吧！就连几个专家级的程序员也在改写时出错了。

可用几种方法重写。可复制代码，将共用的代码放到一个子程序中，然后从两个地方调用它，或重新测试条件。在大多数语言中，重写的代码会比原来的大一点，慢一点，但会非常接近。除非代码是在一个非常热的循环中，否则重写时不必考虑效率问题。

最好的重写是将// lots of code 部分放到它自己的子程序中。然后就可以从本来用作 goto 起点或终点的地方调用该子程序，同时保留条件的原始结构。如下所示：

C++示例：将共用代码放到子程序中以共享*else*子句中的代码

```
if ( statusOk ) {
   if ( dataAvailable ) {
      importantVariable = x;
      DoLotsOfCode( importantVariable );
   }
}
else {
   importantVariable = GetValue();
   DoLotsOfCode( importantVariable );
}
```

通常，写一个新的子程序是最好的方案。但有的时候，将重复代码放到自己的子程序中不现实。这时可通过重构条件来解决，使代码保留在同一个子程序中，而不是非要把它放到一个新的子程序中：

C++示例：不用goto共享else子句中的代码

```
if ( ( statusOk && dataAvailable ) || !statusOk ) {
   if ( statusOk && dataAvailable ) {
      importantVariable = x;
   }
   else {
      importantVariable = GetValue();
   }

   // lots of code
   ...
}
```

关联参考　解决该问题的另一个方案是使用决策表。详情参见第 18 章。

这是对 goto 版本中的逻辑的一个忠实和机械的翻译。它额外测试了 statusOK 两次，dataAvailable 一次，但代码是等同的。如重新测试条件让你感到困扰，注意 statusOK 的值不需要在第一个 if 测试中测试两次。另外，还可以在第二个 if 测试中放弃对 dataAvailable 的测试。

goto 的使用原则总结

goto 的使用关乎信仰。我的信条是，在现代语言中，可以很容易地用等价的顺序结构来取代 10 个 goto 中的 9 个。在这些简单情况下，应该想都不想地放弃 goto。在困难情况下，10 种情况中的 9 个中仍然可以放弃 goto。可将代码分解成更小的子程序，使用 try-finally，使用嵌套 if，测试并重新测试状态变量，或对条件进行重构。在这些情况下，消除 goto 是比较难，但这是很好的思维训练，本节讨论的技术为你提供了相应的工具。

最后，如果 goto 确实是解决问题的正当方案，请明确注释，然后使用它。雨靴都穿上了，就不值得为了躲避一个泥坑而绕着走。但是，要对其他程序员提出的不用 goto 的方案保持开放的心态。有时旁观者清。

下面总结了 goto 的使用原则：

- 若语言不直接支持结构化控制结构，就用 goto 来模拟。模拟需准确，不要滥用 goto 所带来的额外灵活性。
- 若有等同的内置结构，就不要用 goto。

关联参考　参见第 25 章和第 26 章更详细地了解如何提高效率。

- 用 goto 提高效率时，就实际测量其性能。大多数时候都可以在不用 goto 的情况下重新编码，在提高可读性的同时还不损失效率。如果你的情况是个例外，就将提高效率这一点记录下来。这样，反对 goto 的人以后看到时就不会随便删除。
- 除非是在模拟结构化结构，否则每个子程序只能有一个 goto 标签。
- 除非是在模拟结构化结构，否则限制 goto 只能向前，不要向后。
- 确保所有 goto 标签都被使用。未使用的标签表明缺少代码，即通往标签的代码。删除未使用的标签。
- 确保 goto 不会产生不可到达的代码。
- 如果你是经理，记得要有大局观，不值得为区区一个 goto 浪费太多时间。只要程序员意识到了有其他选择，并愿意为这个 goto 辩护，则该 goto 可能是没问题的。

17.4　针对不常见控制结构的观点

曾几何时，有人认为以下每个控制结构都是不错的：

- 不受限制地使用 goto。
- 能动态计算 goto 目标并跳转至计算的位置。

- 能用 goto 从一个程序的中间跳到另一个程序的中间。
- 调用子程序时传递一个行号或标签，以便从子程序中间的某个位置开始执行。
- 能让程序动态生成代码，然后执行它刚才写的代码。

这些想法中的每一个一度都被认为是可以接受的，甚至是可取的，虽然现在它们看起来都是那么的古板、过时或危险。软件开发领域的进步在很大程度上是通过限制程序员可以用他们的代码做什么。所以，我对非常规的控制结构持强烈怀疑态度。我猜想，本章描述的大多数结构最终都会和计算的 goto 标签、可变子程序入口、自修改代码以及其他倾向于灵活性和便利性的结构——而非管控复杂性的结构和能力——一起被程序员丢进废纸篓。

其他资源

以下资源也讲述了一些不常见的控制结构：

return

Martin Fowler 的《*Refactoring: Improving the Design of Existing Code*》，麻省雷丁市 Addison-Wesley 出版社，1999 年。本书在讲述“使用防卫子句替代嵌套条件”的重构技巧时，建议子程序使用多个 return 语句，从而减少一组 if 语句的嵌套层级。Fowler 认为，多个 return 能显著提升结构的清晰度，而一个子程序有多个 return 并不会带来危害。

goto

这些文章覆盖了完整的 goto 辩论。这种辩论时不时还在大多数工作场所、教科书和杂志中爆发，但不过都是在拾 20 年前那些人的牙慧罢了。

Dijkstra, Edsger。“Go To Statement Considered Harmful”（Go To 语句有害），Communications of the ACM 11，no. 3(1968 年 3 月)：147-48, www.cs.utexas.edu/users/EWD/。这是一封著名的信，Dijkstra 在信中把火柴放在纸上，点燃了软件开发中最长久的争议之一。

Wulf, W. A. “A Case Against the GOTO”（不适合 GOTO 的一种情况），Proceedings of the 25th National ACM Conference，1972 年 8 月：791–97。这篇论文也反对不分青红皂白地使用 goto。Wulf 认为，如果编程语言提供了足够的控制结构，goto 在很大程度上将成为不必要的。自 1972 年写这篇论文以来，C++、Java 和 Visual Basic 等语言已证明了 Wulf 的正确性。

Knuth, Donald，“Structured Programming with go to Statements”（使用 go to 语句的结构化编程），1974。见于 Edward Yourdon 编著的《Classics in Software Engineering》，Englewood Cliffs, NJ: Yourdon Press, 1979。这篇长文并不是完全关于 goto 的，但它包括了大量通过消除 gotos 而变得更有效率的代码实例，以及其他大量通过增加 goto 而变得更有效率的代码实例。

Rubin, Frank，“'GOTO Considered Harmful' Considered Harmful”（“GOTO 有害”才有害），Communications of the ACM 30，no. 3（1987 年 3 月）：195–96。在这封给编辑的信中，Rubin 断言，没有 goto 的程序设计使企业损失了“数亿美元”。然后，他提供了一个使用 goto 的简短代码片段，并论证了它比无 goto 的替代方案要好。

Rubin 的信所造成的反响比信本身更有趣。在五个月的时间里，Communications of the ACM(CACM)刊登了读者为 Rubin 原始七行程序写的不同版本。这些信在捍卫 goto 和指责

goto 的人之间平分秋色。读者们总共提出了大约 17 种不同的改写方法，这些代码覆盖了避免使用 goto 的全部方法。CACM 的编辑指出，这封信所造成的反响远远大于 CACM 历史刊发的文章。

这些后续信件请参见：

- Communications of the ACM 30, no. 5 (May 1987): 351 – 55.
- Communications of the ACM 30, no. 6 (June 1987): 475 – 78.
- Communications of the ACM 30, no. 7 (July 1987): 632 – 34.
- Communications of the ACM 30, no. 8 (August 1987): 659 – 62.
- Communications of the ACM 30, no. 12 (December 1987): 997, 1085.

Clark, R. Lawrence，“A Linguistic Contribution of GOTO-less Programming”（从语言学的角度看待无 goto 编程），Datamation, 1973 年 12 月。这篇经典论文幽默地论证了用“come from”语句代替“go to”的可行性。1974 年 4 月的 Communications of the ACM 也转载了此文。

检查清单： 不常见的控制结构

return

- 每个子程序都只在必要时才使用 return 吗？
- return 是否增强了可读性？

递归

- 递归子程序是否包含用于终止递归的代码？
- 子程序是否使用安全计数器保证自己终止？
- 递归是否只限于一个子程序（没有循环递归）？
- 子程序递归深度是否在栈的大小限制范围内？
- 递归是实现子程序最好的方式吗？比简单的迭代好？

goto

- goto 是否只是作为最后的手段使用，而且只是为了使代码更容易阅读和维护？
- 如果为了效率而使用 goto，是否对效率的提升进行了测量和注释？
- 每个子程序的 goto 是否只限于一个标签？
- 所有 goto 是否都是向前的，而不是向后的？
- 所有 goto 标签都用到了吗？

要点回顾

- 多处返回（多个 return）可提高子程序的可读性和可维护性，并有助于防止深度嵌套逻辑。尽管如此，还是要慎用。
- 递归为一小部分问题提供了优雅的解决方案。使用也需谨慎。
- 少数情况下，goto 是编写可读性和可维护性良好的代码的最佳方式。这种情况很少。只有万不得已才使用 goto。

第 18 章　表驱动法

内容

- 18.1 表驱动法使用总则
- 18.2 直接访问表
- 18.3 索引访问表
- 18.4 阶梯访问表
- 18.5 表查询的其他示例

相关章节

- 信息隐藏：5.3 节中的“隐藏秘密（信息隐藏）”
- 类的设计：第 6 章
- 采用决策表替代复杂的逻辑：第 19.1 节
- 采用查询表替代复杂的表达式：第 26.1 节

表驱动法（table-driven method）是一种允许在表中查询信息的方案，而不必用逻辑语句（if 和 case）来查询。几乎任何能用逻辑语句选择的东西，都可换用表来选择。在简单情况下，逻辑语句更容易、更直接。但是，随着逻辑链变得越来越复杂，表变得越来越有吸引力。

如果你已熟悉了表驱动法，这一章对你而言可能只是一个复习。如果是这种情况，可研究一下第 18.2 节中的“灵活消息格式示例”，它是一个很好的例子，说明了面向对象的设计并不因为它面向对象就一定比其他类型的设计好。然后，可转到第 19 章对常规控制问题的讨论。

18.1　表驱动法使用总则


KEY POINT 在适当的情况下使用，表驱动的代码比复杂的逻辑更简单，更容易修改，也更高效。

假设要将字符分类为字母、标点符号和数字，可能会使用如下所示的复杂逻辑链：

Java示例：用复杂逻辑进行字符分类

```
if ( ( ( 'a' <= inputChar ) && ( inputChar <= 'z' ) ) ||
   ( ( 'A' <= inputChar ) && ( inputChar <= 'Z' ) ) ) {
   charType = CharacterType.Letter;
}
else if ( ( inputChar == ' ' ) || ( inputChar == ',' ) ||
   ( inputChar == '.' ) || ( inputChar == '!' ) || ( inputChar == '(' ) ||
   ( inputChar == ')' ) || ( inputChar == ':' ) || ( inputChar == ';' ) ||
   ( inputChar == '?' ) || ( inputChar == '-' ) ) {
   charType = CharacterType.Punctuation;
}
else if ( ( '0' <= inputChar ) && ( inputChar <= '9' ) ) {
   charType = CharacterType.Digit;
}
```

换用查询表（lookup table），可将每个字符的类型存储在通过字符编码访问的一个数组中。上述复杂代码段可替换为：

Java示例：用查询表进行字符分类

```
charType = charTypeTable[ inputChar ];
```

这段代码假定已事先设置好了 charTypeTable 数组。总之，我们的目的是将程序已知的信息放到它的数据中，而不是放到逻辑中。换言之，是放到表中，而不是放到 if 测试中。

使用表驱动法的两个问题


KEY POINT 使用表驱动法需解决两个问题。首先，必须解决如何在表中查询条目的问题。可用一些数据直接访问表。例如，如需按月对数据进行分类，可直接创建一个以月份作为键的表。可使用一个索引为 1~12 的数组。

其他数据则有点麻烦，不好直接用来查询表项。例如，如需按社会安全号码对数据进行分类，就不能将社会安全号作为键，除非你能表中存储 999-99-9999 个条目。这时不得不使用一种更复杂的方法。下面列出了在表中查询条目的各种方式：

- 直接访问（Direct access）
- 索引访问（Indexed access）
- 阶梯访问（Stair-step access）

后续小节将详细探讨每一种访问方式。


KEY POINT 使用表驱动法的必须解决的第二个问题是应该在表中存储什么。某些时候，表查询的结果是数据。在这种情况下，可将数据存储到表中。在其他情况下，表查询的结果是一个动作。在这种情况下，可存储描述该动作的代码；或者在某些语言中，可存储对实现该动作的子程序的引用。在这两种情况下，表都会变得更复杂。

18.2 直接访问表

和所有查询表一样，直接访问表（direct-access tables）代替了更复杂的逻辑控制结构。之所以是“直接访问”，是因为不必费力讨好地查找表中的信息。如图 18-1 所示，可直接挑出想要的条目。

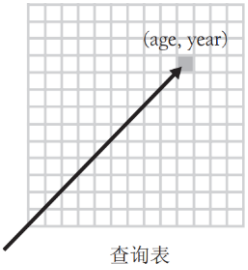


图 18-1　顾名思义，直接访问表允许直接访问需要的表元素

示例：一个月有多少天

假设需要确定每个月有多少天（为方便讨论，先忘记闰年）。当然，一个笨拙的方法是写一个大的 if 语句：

Visual Basic示例：判断一个月有多少天的笨办法

```
If ( month = 1 ) Then
   days = 31
ElseIf ( month = 2 ) Then
   days = 28
ElseIf ( month = 3 ) Then
   days = 31
ElseIf ( month = 4 ) Then
   days = 30
ElseIf ( month = 5 ) Then
   days = 31
ElseIf ( month = 6 ) Then
   days = 30
ElseIf ( month = 7 ) Then
   days = 31
ElseIf ( month = 8 ) Then
   days = 31
ElseIf ( month = 9 ) Then
   days = 30
ElseIf ( month = 10 ) Then
   days = 31
ElseIf ( month = 11 ) Then
   days = 30
ElseIf ( month = 12 ) Then
   days = 31
End If
```

更容易、也更容易修改的办法是将数据放到一个表中。以 Microsoft Visual Basic 为例先创建一个表：

Visual Basic示例：判断一个月有多少天的优雅方式

```
' Initialize Table of "Days Per Month" Data
Dim daysPerMonth() As Integer = _
   { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
```

现在，可以不使用长的 if 语句，只通过一个简单的数组访问来找出一个月的天数：

Visual Basic示例：判断一个月有多少天的优雅方式(续)

```
days = daysPerMonth( month-1 )
```

在表查询版本中如果想把闰年考虑在内，代码仍然很简单，假设 LeapYearIndex()的值为 0 或 1：

Visual Basic示例：判断一个月有多少天的优雅方式(续)

```
days = daysPerMonth( month-1, LeapYearIndex() )
```

在 if 语句版本中，如果要考虑闰年，一长串的 if 会变得更加复杂。

判断一个月的天数是一个简单的例子，因为可用 month 变量来查询表中的一个条目。一般都可以使用本来控制着大量 if 语句的数据来直接访问一个表。

示例：保险费率

假定要写一个计算医保费率的程序，费率因年龄、性别、婚姻状况和是否吸烟而不同。如必须为费率写一个逻辑控制结构，会得到这样的东西：

Java示例：判断保险费率的笨办法

```
if ( gender == Gender.Female ) {
   if ( maritalStatus == MaritalStatus.Single ) {
      if ( smokingStatus == SmokingStatus.NonSmoking ) {
         if ( age < 18 ) {
            rate = 200.00;
         }
         else if ( age == 18 ) {
            rate = 250.00;
         }
         else if ( age == 19 ) {
            rate = 300.00;
         }
         ...
         else if ( 65 < age ) {
            rate = 450.00;
      }
      else {
         if ( age < 18 ) {
            rate = 250.00;
         }
         else if ( age == 18 ) {
            rate = 300.00;
         }
         else if ( age == 19 ) {
            rate = 350.00;
         }
         ...
         else if ( 65 < age ) {
            rate = 575.00;
         }
      }
   else if ( maritalStatus == MaritalStatus.Married )
   ...
}
```

这个逻辑结构的简略版本应足以让你了解这种事情可以变得多么复杂。它还没有显示已婚女性、任何男性或 18~65 岁之间的大部分年龄。可以想象，完整的费率表编程会有多么复杂。

你可能会说：“好吧，但为什么每个年龄都要测试？为什么不直接将每个年龄的费率放到数组中？”这个问题提得很好，一个明显的改进就是将每个年龄的费率放到单独的数组中。

但是，一个更好的解决方案是将考虑了所有因素的费率放入数组，而非仅仅考虑年龄。下面是在 Visual Basic 中声明数组的方法：

Visual Basic示例：声明数组以建立保险费率表

```
Public Enum SmokingStatus
   SmokingStatus_First = 0
   SmokingStatus_Smoking = 0
   SmokingStatus_NonSmoking = 1
   SmokingStatus_Last = 1
End Enum

Public Enum Gender
   Gender_First = 0
   Gender_Male = 0
   Gender_Female = 1
   Gender_Last = 1
End Enum

Public Enum MaritalStatus
   MaritalStatus_First = 0
   MaritalStatus_Single = 0
   MaritalStatus_Married = 1
   MaritalStatus_Last = 1
End Enum                                                    形状对象

Const MAX_AGE As Integer = 125

Dim rateTable ( SmokingStatus_Last, Gender_Last, MaritalStatus_Last, _
   MAX_AGE ) As Double
```

关联参考　表驱动法的一个优势是能将表中的数据放到文件里，并在程序运行时读取。这样可在不必修改程序本身的前提下修改像保险费率这样的东西。10.6 节更多地讨论了这一思路。

声明好数组后，必须想一个办法将数据录入其中。可使用赋值语句，从磁盘文件读取数据，计算数据，或采取其他任何合适的方法。设置好数据后，需要计算一个费率时，直接取用即可。前面展示的复杂逻辑被像下面这样的简单语句取代：

Visual Basic示例：判断保险费率的优雅方法

```
rate = rateTable( smokingStatus, gender, maritalStatus, age )
```

这个方法具有将复杂逻辑替换成表查询的常规优势。表查询更易读，也更容易修改。

示例：灵活消息格式

可用表格描述那些过于动态而无法用代码表示的逻辑。在字符分类的例子中，在一月多少天的例子中，以及在保险费率的例子中，你至少知道在需要的时候可以写一长串 if 语句。但在某些情况下，数据过于复杂，硬编码的 if 语句也不好使。

如果你认为自己已经了解了直接访问表的工作方式，可能会想跳过下个例子。不过，这个例子比之前的例子要复杂一些，它进一步演示了表驱动法的威力。

假设要写一个子程序来打印存储在文件中的信息。文件通常含有约 500 条消息，而且每个文件有约 20 种消息。这些消息最初来自于一个浮标，描述了水温、浮标位置等等

每条消息都有几个字段，每条消息都从一个消息头（header）开始，这个消息头有一个 ID，让你知道处理的是 20 多种消息中的哪一种。图 18-2 说明了信息的存储方式。

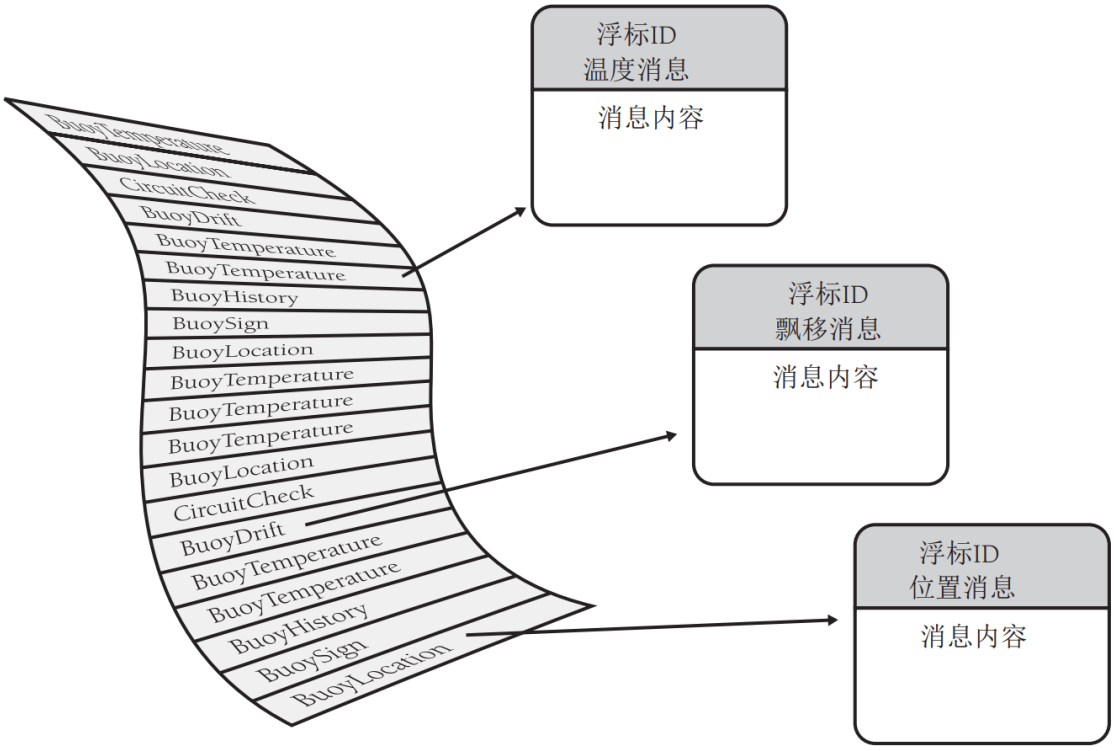


图 18-2　消息不按特定顺序存储，每条消息都用一个消息 ID 来标识

消息格式不固定，由你的客户决定，而你对客户的控制力不够，无法把它稳定下来。图 18-3 展示了几条消息的详情。

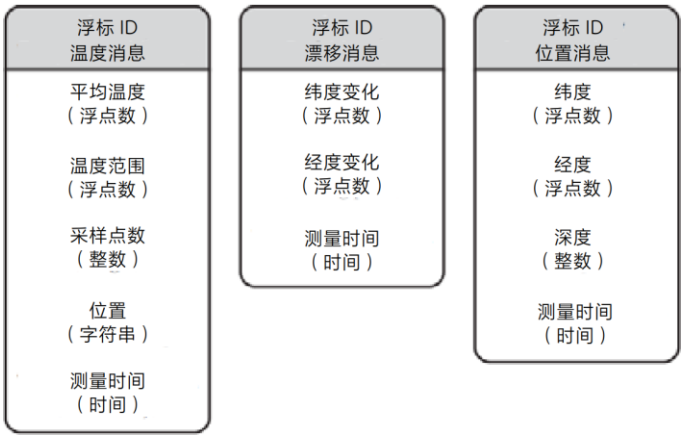


图 18-3　除了消息 ID，每种消息都有自己的格式

基于逻辑的方法

如使用基于逻辑的方法，你可能会读取每条消息，检查 ID，再调用一个设计用于读取、解释和打印每种消息的子程序。有 20 种消息就有 20 个子程序。另外，还有不知道多少个用于支持它们的低级子程序来支持它们。例如，可能有一个 PrintBuoyTemperatureMessage()子程序来打印浮标温度消息。面向对象的方法她好不到哪里去：通常要使用一个抽象消息对象，每种消息一个子类。

每当任何消息的格式发生变化，都必须更改负责该消息的子程序或类的逻辑。在前面的示例详细消息中，如“平均温度”字段从浮点变成了别的东西，就必须修改 PrintBuoyTemperatureMessage()的逻辑。（如果是浮标本身从“浮点”变成了别的东西，你恐怕只好换一个新浮标了！）

在基于逻辑的方法中，消息读取子程序由一个循环组成，它读取每条消息，解码 ID，然后根据消息 ID 调用 20 个子程序中的一个。以下是基于逻辑的方法的伪代码：

> 关联参考　这些伪代码有别于你平时用于子程序设计的伪代码，它们更低级，目的也不一样。参见第 9 章详细了解如何用伪代码进行设计。

```
While 有更多消息可供读取
    读取一个消息头（header）
    从消息头解码消息 ID
    If 消息头是类型 1，then
        打印类型 1 消息
    Else if 消息头是类型 2，then
        打印类型 2 消息
    ...
    Else if 消息头是类型 19，then
         打印类型 19 消息
    Else if 消息头是类型 20，then
         打印类型 20 消息
```

这些伪代码进行了简化，你无需看完全部 20 种情况就能心领神会。

面向对象的方法

如果只是生硬地使用面向对象的方法，逻辑会隐藏在对象的继承结构中，但基本结构一样复杂：

```
While 有更多消息可供读取
    读取一个消息头（header）
    从消息头解码消息 ID
    If 消息头是类型 1，then
        实例化一个类型 1 消息对象
    Else if 消息头是类型 2， then
        实例化一个类型 2 消息对象
    ...
    Else if 消息头是类型 19， then
        实例化一个类型 19 消息对象
    Else if 消息头是类型 20， then
        实例化一个类型 20 消息对象
    End if
```

```
End While
```

无论逻辑是直接写的，还是包含在专门的类中，20 种消息中的每一种都有自己的消息打印子程序。 每个子程序也可用伪代码表示。以下用于读取和打印浮标温度消息的子程序的伪代码：

```
打印 "浮标温度消息"

读取一个浮点值
打印"平均温度"
打印浮点值

读取一个浮点值
打印"温度范围"
打印浮点值

读取一个浮点值
打印"样本数"
打印整数值

读取一个浮点值
打印"位置"
打印字符串

读取一个时间
打印"测量时间"
打印时间
```

这还只是一种消息的代码。其他 19 种消息的每一种都需要类似的代码。如增加了第 21 种消息，就需要增加第 21 个子程序或第 21 个子类——无论如何，新的消息类型都要求对代码进行修改。

表驱动法

表驱动的方法比之前的方法更经济。读取消息的子程序由一个循环组成，它读取每个消息头，解码 ID，在 Message 数组中查询消息描述，然后每次都调用同一个子程序来解码消息。采用表驱动法，可在表中描述每种消息的格式，而不必在程序逻辑中硬编码。这使代码的编写一开始就很容易，代码量会少许多，而且以后维护时无需修改代码。

要使用这种方法，首先需列出消息种类和字段类型。在 C++中，可以像下面这样定义所有可能的字段的类型：

C++示例：定义消息中的数据的类型

```
enum FieldType {
   FieldType_FloatingPoint,
   FieldType_Integer,
   FieldType_String,
   FieldType_TimeOfDay,
   FieldType_Boolean,
   FieldType_BitField,
   FieldType_Last = FieldType_BitField
};
```

不需要为 20 种消息中的每一种都硬编码打印子程序。相反，可以创建少量的子程序来打印每一种主要数据类型：浮点、整数、字符串等等。可在一个表中描述每种消息的内容（包括每个字段的名称），然后根据表中的描述对每种消息进行解码。下面展示了对一种消息进行描述的表项：

消息表项定义示例

```
Message Begin
   NumFields 5
   MessageName "Buoy Temperature Message"
   Field 1, FloatingPoint, "Average Temperature"
   Field 2, FloatingPoint, "Temperature Range"
   Field 3, Integer, "Number of Samples"
   Field 4, String, "Location"
   Field 5, TimeOfDay, "Time of Measurement"
Message End
```

这个表可硬编码到程序中（在这种情况下，所显示的每个元素都要赋给变量），也可在程序启动时或之后从文件中读取。

程序一旦读取了消息定义，所有消息就不再是嵌入程序逻辑中，而是嵌入数据中。数据往往比逻辑更灵活。以后消息格式发生改变时，数据也很容易改变。如必须添加一种新消息，在数据表中添加另一个元素即可。

以下是表驱动法顶层循环的伪代码：

```
前三行和基于逻辑
的方法一样

While 有更多消息可供读取
    读取一个消息头（header）
    从消息头解码消息ID
    从消息描述表查询消息描述
    读取消息中的字段，并根据消息描述打印它们
End While
```

和基于逻辑的方法的伪代码不同，本例的伪代码无需为了节省篇幅而简化，因为它的逻辑是如此的简单。在这一层级往下的逻辑中，你会发现有一个子程序能解释来自消息描述表的消息描述，读取消息数据，并打印一条消息。该子程序比任何一个基于逻辑的消息打印子程序都要常规，而且简单许多。最重要的是，只有一个子程序，而不是 20 个。以下是该子程序的伪代码：

```
While 有更多字段需要打印
    从消息描述中获取字段类型
    case ( 字段类型 )
        of ( 浮点 )
            读取浮点值
            打印字段标签
            打印浮点值

        of ( 整数 )
            读取整数值
            打印字段标签
            打印整数值

        of ( 字符串 )
            读取字符串
```

```
            打印字段标签
            打印字符串

        of ( 时间 )
            读取时间
            打印字段标签
            打印时间

        of ( boolean )
            读取单一标志位(single flag)
            打印字段标签
            打印单一标志位

        of ( 位段 )
            读取位段(bit field)
            打印字段标签
            打印位段
    End Case
End While
```

诚然，这个子程序包含 6 种情况，要比打印浮标温度信息子程序长一些。但这是你唯一需要的子程序。不需要其他 19 个子程序来处理其他种类的信息。这个子程序处理 6 种字段类型，能照顾到所有种类的信息。

这个子程序还展示了实现这种表查询的最复杂的方法，因为它使用了 case 语句。另一个办法是创建抽象类 AbstractField，然后为每种字段类型创建子类。这样就不需要 case 语句，只需调用相应对象类型的成员子程序。

下面在 C++中设置对象类型：

C++示例：设置对象类型

```
class AbstractField {
   public:
   virtual void ReadAndPrint( string, FileStatus & ) = 0;
};

class FloatingPointField : public AbstractField {
   public:
   virtual void ReadAndPrint( string, FileStatus & ) {
   ...
   }
};

class IntegerField ...
class StringField ...
...
```

这段代码为每个类声明一个成员子程序，它有一个字符串参数和一个 FileStatus 参数。

下一步是声明一个数组来容纳对象集合。这个数组就是查询表，如下所示：

C++示例：设置对象列表

```
field[ Field_FloatingPoint ] = new FloatingPointField();
field[ Field_Integer ] = new IntegerField();
field[ Field_String ] = new StringField();
field[ Field_TimeOfDay ] = new TimeOfDayField();
field[ Field_Boolean ] = new BooleanField();
field[ Field_BitField ] = new BitFieldField();
```

设置对象表所需的最后一步是将特定对象的名称赋给 Field 数组：

C++示例：设置对象列表

```
field[ Field_FloatingPoint ] = new FloatingPointField();
field[ Field_Integer ] = new IntegerField();
field[ Field_String ] = new StringField();
field[ Field_TimeOfDay ] = new TimeOfDayField();
field[ Field_Boolean ] = new BooleanField();
field[ Field_BitField ] = new BitFieldField();
```

这段代码假定 FloatingPointField 和赋值语句右侧的其他标识符是 AbstractField 类型的对象的名称。将对象赋值给数组中的数组元素，意味着可通过引用数组元素来调用正确的 ReadAndPrint()子程序，而不必直接使用特定种类的对象。

一旦设置好子程序表，就可通过访问对象表并调用表中的一个成员子程序来处理消息中的一个字段。代码如下：

这些是针对消息中每个字段的内务处理代码

这是一个表查询，它根据字段类型调用一个子程序——在对象表中查找即可

C++示例：在表中查找对象和成员子程序

```
fieldIdx = 1;
while ( ( fieldIdx <= numFieldsInMessage ) && ( fileStatus == OK ) )
    { fieldType = fieldDescription[ fieldIdx ].FieldType;
    fieldName = fieldDescription[ fieldIdx ].FieldName;
    field[ fieldType ].ReadAndPrint( fieldName, fileStatus );
    fieldIdx++;
}
```

还记得当初含有 case 语句的 34 行表格查询伪代码吗？用一个对象表来代替 case 语句可实现相同的功能。令人难以置信的是，这也是代替基于逻辑的方法中所有 20 个单独子程序所需的全部代码。另外，如消息描述是从文件中读取的，除非有新的字段类型，否则新增消息类型将无需更改代码。

可在任何面向对象的语言中使用这种方法。与冗长的 if 语句、case 语句或大量子类相比，它更不容易出错，更容易维护，效率也更高。

不是说一个设计使用了继承和多态性就肯定一个好的设计。之前在“面向对象的方法”一节中描述的生硬的面向对象的设计，需要和生硬的功能（函数）式设计一样多的代码，甚至更多。这种方法使得解决方案的空间变得复杂化而不是简单化。在这种情况下，关键的设计理念既不是面向对象，也不是面向功能，而是使用一个经过深思熟虑的查询表。

编造查询键

之前的三个例子可直接使用数据来作为表键。换言之，可用 messageID 作为键，无需任何改动。在每月天数的例子中可使用 month，在保险费率的例子中可使用 gender（性别），maritalStatus（婚姻状态）和 smokingStatus（吸烟状态）。

谁都想直接访问键，因其简单而快速。但有的时候，数据并不配合。在保险费率的例子中，使用 Age 作为键并不好。逻辑是 18 岁以下的人有一个费率，18~65 每一个年龄都有单独的费率。65 岁以上的人则又是一个费率。这意味着对于 0 到 17 岁和 66 岁及以上的人来说，由于表中只为一些年龄存放了一组费率，所以不能直接将 Age 作为键来查询该表。

这就引出了需要编造表查询键的话题。可用几种方法来编造键：

复制信息，使键能直接使用 要使 Age 成为费率表的键，一个直接的方法是复制 0 至 17 岁费率中的每一个（费率相同），然后直接使用 Age 作为键。对 66 岁以上的人做同样的事情。这种方法的好处是，表结构本身是直接的，表的访问也是直接的。以后如果想更改 17 岁及以下的费率，为每个年龄都设置特定的费率，可直接修改表格。这样做的缺点在于，重复会浪费冗余信息的空间，还增大了表出错的可能性——还不仅仅是因为表中含有冗余数据。

转换键使其能直接作为表键 使 Age 直接成为键的另一个方法是向 Age 应用一个函数。在这种情况下，该函数必须将所有 0~17 岁的年龄更改为一个键（例如 17），而所有 66 岁以上的年龄更改为另一个键（例如 66）。这个特定的范围是良构的，可用 min()和 max()函数来进行转换。例如，可用以下表达式：

```
max( min( 66, Age ), 17 )
```

来创建 17~66 的表键。

创建转换函数要求在想作为键使用的数据中识别出一种模式，而这并不总是像使用 min()和 max()子程序那样简单。假设费率每 5 年一个年龄段而不是每 1 年一个年龄段。除非想把所有的数据复制五次，否则必须想出一个函数，将 Age 除以 5，再使用 min()和 max()子程序。

在自己的子程序中隔离键的转换 如必须编造数据使其作为表键使用，就将数据改为键的操作放到它自己的子程序中，从而避免在不同的地方使用不同的转换。以后修改转换时也会更容易。一个好的子程序名称，例如 KeyFromAge()，还可澄清并记录数学运算的目的。

如果你的开发环境提供了现成的键转换，请使用它们。例如，Java 提供的 HashMap 可用于联“键/值”对。

18.3 索引访问表

有的时候，一个简单的数学转换还不足以实现从 Age 这样的数据到表键的变迁。在这种情况下适合使用索引访问方案。

使用索引时，要用主数据查询索引表中的一个键，然后用索引表中的值来查询你感兴趣的主要数据。

假设你经营一个仓库，有大约 100 件物品的库存。再假设每件物品都有一个四位数零件编号（part number），范围从 0000 到 9999。在这种情况下，如果想用零件编号作为键来查询描述

了每件物品的某些方面的一个表格，就需建立一个有 10000 个条目（0~9999）的索引数组。该数组基本是空的，其中只有 100 个条目与仓库中 100 件物品的零件编号相对应。如图 18-4 所示，这些条目指向一个物品描述表，该表的条目数远远少于 10000。

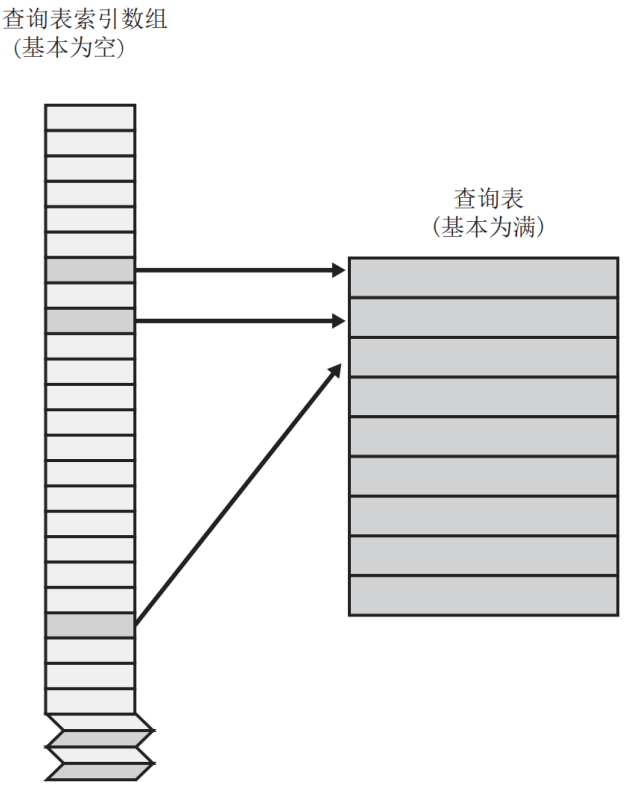


图 18-4　索引访问表不是直接访问，而是通过一个中间索引来访问

索引访问方案主要有两个优点。首先，如主查询表中的每个条目都很大，创建一个浪费了大量空间的索引数组比创建一个浪费了大量空间的主查询表要少很多空间。例如，假设主表每个条目需要 100 字节，索引数组每个条目只需 2 字节。假设主表有 100 个条目，用于访问它的数据有 10000 个可能的值。在这种情况下，要么选择有 10000 个条目的索引，要么选择有 10000 个条目的主数据成员。如使用索引，总的内存开销是 30000 字节。如放弃索引结构而在主表中浪费空间，总内存开销就变成了 1000000 字节。

第二个优点是，即使使用索引未能节省空间，有时操作索引中的条目比操作主表中的条目更便宜。例如，如果有一个包含员工姓名、招聘日期和工资的表，可创建一个索引，按员工姓名访问该表，创建另一个索引按招聘日期访问该表，再创建一个索引按工资访问该表。

索引访问方案的最后一个优点是表查询最基本的可维护性。编码到表中的数据比嵌入代码的数据更易维护。为最大限度提高灵活性，可将索引访问代码放在它自己的子程序中，需要从一个零件编号获得表键时，调用该子程序即可。需要修改表的时候，可切换索引访问方案或完全切换到另一个表查询方案。只要索引访问不要在整个程序中到处都是，访问方案会更容易更改。

18.4 阶梯访问表

另一种访问表的方法是阶梯访问。这种访问方法不像索引结构那样直接，但不会浪费那么多数据空间。如图 18-5 所示，阶梯结构的基本思路是，表中的条目对数据范围有效，而不是对不同数据点有效。

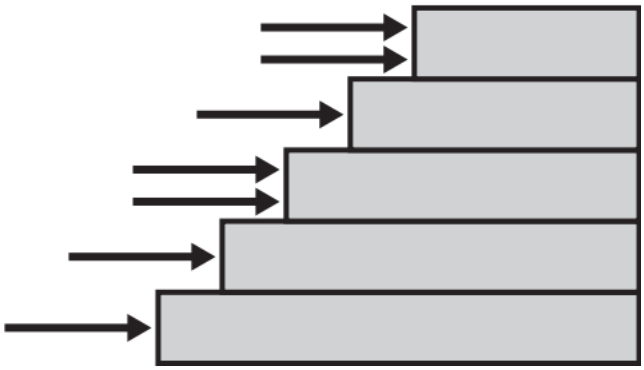


图 18-5　阶梯方法通过确定每个条目所处的“阶梯”（staircase）层级对其进行分类。它遇到的“台阶”（step）决定了它的类别

例如，假设要写一个成绩评定程序，“B”条目的范围从 75%到 90%，下面展示了一个可能的成绩范围：

≥ 90.0%	A
< 90.0%	B
< 75.0%	C
< 65.0%	D
< 50.0%	F

对于表查询来说，这是一个难看的区间，因为不能用一个简单的数据转换函数将字母 A~F 作为键。由于是浮点数，所以不好用索引方案。可考虑将浮点数转换成整数，在这种情况下，这确实一个有效的设计方案。但是，ff 是为了说明问题，这个例子将坚持使用浮点数。

为了使用阶梯法，需要将每个区间的上限放到一个表中，然后写一个循环，根据每个区间的上限检查一个分数。找到分数首次超过区间上限的那一点时，就知道字母成绩是多少了。使用阶梯技术，必须注意正确处理区间的端点。下面在 Visual Basic 中根据这个例子给一组学生分配成绩：

Visual Basic示例：阶梯表查询

```
' set up data for grading table
Dim rangeLimit() As Double = { 50.0, 65.0, 75.0, 90.0, 100.0 }
Dim grade() As String =      { "F",  "D",  "C",  "B",   "A"  }
maxGradeLevel = grade.Length - 1
...

' assign a grade to a student based on the student's score
gradeLevel = 0
studentGrade = "A"
While ( ( studentGrade = "A" ) and ( gradeLevel < maxGradeLevel ) )
   If ( studentScore < rangeLimit( gradeLevel ) ) Then
      studentGrade = grade( gradeLevel )
   End If
   gradeLevel = gradeLevel + 1
Wend
```

虽然这是一个简单的例子，但可以很容易地把它推广到处理多个学生、多个成绩评定方案（例如，不同的作业有不同的绩点区间，所以有不同的成绩）以及成绩评定方案的变化。

和其他表驱动方法相比，这种方法的优势在于它能很好地处理不规则数据。成绩评定的例子很简单，因为虽然成绩是以不规则的间隔分配的，但数字本身是“四舍五入”过的，以 5 和 0 结束。阶梯方法同样适合那些并非整齐地以 5 和 0 结束的数据。可在统计工作中使用阶梯方法处理像下面这样的概率分布：

概率	保险索赔额度
0.458747	$0.00
0.547651	$254.32
0.627764	$514.77
0.776883	$747.82
0.893211	$1,042.65
0.957665	$5,887.55
0.976544	$12,836.98
0.987889	$27,234.12
…	

对于如此丑陋的数字，实在无法想出一个函数将其整齐地转化为表键。这时，阶梯方法就是答案。

这个方法还享有表驱动法的常规优势：灵活且易修改。如成绩评定例子中的成绩区间发生变化，可通过修改 RangeLimit 数组中的条目来轻松调整程序。可以很容易地将程序的成绩分配部分进行常规化，使其接受一个成绩表和相应的截断分数。程序的成绩分配部分不必非要使用百分比分数；完全可以使用原始分数而不是百分比，而且程序不必做太多改动。

使用阶梯技术时，需注意几个容易被忽视的问题：

注意端点　确保覆盖了每个阶梯区间的上界（top end）。运行阶梯查询以找出映射到最上层区间以外的任何区间的项目，然后使其余项目落入最上层区间。有的时候，需要为最上层区间的上界创建一个人为的值。

注意不要误将<写成<=。确保循环在值落入最上层区间时正确终止，而且区间边界得到正确的处理。

考虑使用二分搜索而不是顺序搜索 在成绩评定的例子中，分配字母成绩的循环是在成绩限制列表中顺序搜索。但是，如果列表很大，顺序搜索的成本会变得很高。在这种情况下，可考虑用一个准二分搜索（quasi-binary search）来代替它。之所以是“准”二分搜索，是因为大多数二分搜索的重点是要找到一个值。但在本例中，你并不期望找到这个值；你期望找到的是这个值的正确类别。二分搜索算法必须正确判定该值的归属。另外，记住要将端点当作特例来处理。

考虑使用索引访问而不是阶梯技术 18.3 节描述的索引访问方案可能是阶梯技术的一个很好的替代方案。阶梯方法所需的搜索可能会累加，如果执行速度是一个问题，可考虑用额外索引结构所占用的空间来换取更直接的访问方法所带来的时间优势。

显然，该替代方案并非在所有情况下都是一个好的选择。成绩评定的例子或许可以使用它；如果只有 100 个离散的绩点，建立一个索引数组的内存成本不会太高。但另一方面，如果是之前列出的概率数据，就不能建立一个索引方案，因为不能用 0.458747 和 0.547651 这样的数字来作为键。

某些时候，这几种方案中的任何一种都可使用。设计的重点是为你的情况选择几个好的方案中的一个。不必太纠结于选出最佳方案。正如 Microsoft 的杰出工程师 Butler Lampson 所说，与其试图找到最佳方案，不如努力找一个好的方案，并避免灾难（Lamps 1984）。

将阶梯表查询放到它自己的子程序中 创建转换函数将 StudentGrade 这样的值转换为表键时，把它放到它自己的子程序中。

关联参考 参见第 5 章更多地了解如何选择替代的设计方案。

18.5 表查询的其他示例

本书其他章节还出现了其他一些表查询的例子。它们是在讨论其他技术的过程中使用的，当时的背景并没有强调表查询本身。下面列出了这些地方：

- 在保险表中查询费率：第 16.3 节
- 使用决策表来替代复杂逻辑：第 19.1 节
- 表查询期间的内存分页：第 25.3 节
- 布尔值的组合（A or B or C）：第 26.1 节
- 预先计算贷款还款表中的值：第 26.4 节。

检查清单：表驱动法

- 考虑过把表驱动法作为复杂逻辑的替代方案了吗？
- 考虑过把表驱动法作为复杂继承结构的替代方案了吗？
- 考虑过把表数据存储在程序外部并在运行期间读取，使数据可在不修改代码的前提下修改吗？

- 如无法采用一种直接的数组索引（像age例子那样）来直接访问表，那么将访问键（access-key）的计算功能提取为一个单独的子程序， 而不是在代码中复制这些计算吗？

要点回顾

- 表提供了复杂逻辑和继承结构的一种替代方案。如发现自己对某个应用程序的逻辑或继承树感到困惑，不妨好好想想是否可以采用查询表来加以简化。
- 如使用表，一项关键的考虑因素就是决定如何访问表。可采取直接访问、索引访问或阶梯访问的方式。
- 使用表时的另一个关键考虑因素是决定要将什么内容放入表中。

第 19 章　常规控制问题

内容

- 19.1 布尔表达式
- 19.2 复合语句（语句块）
- 19.3 空语句
- 19.4 驾驭深层嵌套
- 19.5 编程基础：结构化编程
- 19.6 控制结构与复杂度

相关章节

- 直线型代码：第 14 章
- 条件代码：第 15 章
- 循环代码：第 16 章
- 不常见的控制结构：第 17 章
- 软件开发的复杂度：第 5.2 节中的“软件的首要技术使命：管理复杂度”

思考控制结构时有几个常规问题需要注意，对控制的讨论不能没有它们。本章大部分信息都是详细而实用的。如果只是想了解控制结构的理论而不在意细节，可直接阅读 19.5 节了解结构化编程的历史观点以及 19.6 节了解控制结构之间的关系。

19.1 布尔表达式

除了最简单的控制结构，即顺序结构，所有控制结构都依赖于对布尔表达式的求值。

用 true 和 false 进行布尔测试

在布尔表达式中使用 true 和 false 标识符，而不是使用 0 和 1 这样的值。大多数现代语言都支持布尔数据类型，并为 true 和 false 提供预定义的标识符。它们使事情变得简单，甚至不允许向布尔变量赋除 true 或 false 之外的值。那些不支持布尔数据类型的语言则要求你自律，使布尔表达式更易读。下面是这个问题的一个例子：

Visual Basic示例：将有歧义的标志(flag)作为布尔值使用

```
Dim printerError As Integer
Dim reportSelected As Integer
Dim summarySelected As Integer
...
If printerError = 0 Then InitializePrinter()
If printerError = 1 Then NotifyUserOfError()

If reportSelected = 1 Then PrintReport()
If summarySelected = 1 Then PrintSummary()

If printerError = 0 Then CleanupPrinter()
```

既然大家都习惯了使用 0 和 1 这样的标志（flag），它有什么问题呢？通过阅读代码，我们并不清楚这些函数调用是在测试为 true 时还是在测试为 false 时执行。这段代码本身并没有告诉你 1 代表 true，0 代表 false，或是相反。甚至不清楚值 1 和 0 是否被用来代表 true 和 false。例如在 If reportSelected = 1 这一行中，1 很容易代表第一份报告，2 代表第二份，3 代表第三份；代码中没有任何东西告诉你，1 代表 true 或 false。当你的意思是 1 时，也很容易写成 0，反之亦然。

用布尔表达式测试时，使用名为 true 和 false 的关键字。如语言不直接支持这些关键字，可用预处理宏或全局变量来创建它们。下面使用 Microsoft Visual Basic 内置的 True 和 False 重写了前面的例子：

Visual Basic示例：好多了，但并不出彩；使用*True*和*False*而非数值来进行测试

```
Dim printerError As Boolean
Dim reportSelected As ReportType
Dim summarySelected As Boolean
...
If ( printerError = False ) Then InitializePrinter()
If ( printerError = True ) Then NotifyUserOfError()

If ( reportSelected = ReportType_First ) Then PrintReport()
If ( summarySelected = True ) Then PrintSummary()

If ( printerError = False ) Then CleanupPrinter()
```

关联参考　参见下个代码示例了解实现同一判断的更佳方法。

使用 True 和 False 常量使意图变得更明确。不必记住 1 和 0 代表什么，也不会意外地把它们颠倒过来。另外，在重写的代码中，现在很清楚在原始版本的代码中，一些 1 和 0 并没有被用作布尔标志。If reportSelected = 1 一行根本就不是布尔测试，它测试的是第一份报告是否被选中。

这种方法告诉看代码的人，你做的是一个布尔测试。本意是 false 时却写成 true，也比本意是 0 时写成 1 要难，而且可避免在代码中散布神秘的数字 0 和 1。以下是在布尔测试中定义 true 和 false 的技巧：

将布尔值隐式地和 true 和 false 比较 可将表达式视为布尔表达式来写更清晰的测试。例如，可以这样写：

```
while ( not done ) ...
while ( a > b ) ...
```

而不是这样写：

```
while ( done = false ) ...
while ( (a > b) = true ) ...
```

使用隐式比较可减少看你的代码的人必须记住的术语的数量，而且生成的表达式读起来更像是日常会话。前面的例子可以这样改写使其更易读：

Visual Basic示例：隐式测试*True*和*False*更佳

```
Dim printerError As Boolean
Dim reportSelected As ReportType
Dim summarySelected As Boolean
...
If ( Not printerError ) Then InitializePrinter()
If ( printerError ) Then NotifyUserOfError()

If ( reportSelected = ReportType_First ) Then PrintReport()
If ( summarySelected ) Then PrintSummary()

If ( Not printerError ) Then CleanupPrinter()
```

关联参考 布尔变量的详情请参见第 12.5 节。

如语言不支持布尔变量但又必须模拟它们，就可能无法使用这一技术，因为对 true 和 false 的模拟并非一定能使用像 while (not done)这样的语句来测试。

简化复杂的表达式

可采取几个步骤简化复杂的表达式：

用新的布尔变量将复杂的测试拆分为部分测试 与其用好几个术语创建一个巨大的测试，不如给术语分配中间值，以执行一个更简单的测试。

将复杂表达式转移到布尔函数中 如一个测试经常重复，或者从程序主要流程中分离，就将测试代码转移到一个函数中，并测试该函数的值。以下面这个复杂的测试为例：

Visual Basic示例：一个复杂测试

```
If ( ( document.AtEndOfStream ) And ( Not inputError ) ) And _
   ( ( MIN_LINES <= lineCount ) And ( lineCount <= MAX_LINES ) ) And _
   ( Not ErrorProcessing(  ) ) Then
   ' do something or other
   ...
End If
```

如果对测试本身不感兴趣，就没必要纠结于这个复杂的测试。把它放到一个布尔函数中，即可将测试隔离出来。读者若非觉得它很重要，否则可以完全忽视它。下例展示了如何将这个 if 测试放到自己的函数中：

Visual Basic示例：将复杂测试转移到布尔函数中，并用新的中间变量使测试更清晰

```
Function DocumentIsValid( _
   ByRef documentToCheck As Document, _
   lineCount As Integer, _
   inputError As Boolean _
   ) As Boolean

   Dim allDataRead As Boolean
   Dim legalLineCount As Boolean

   allDataRead = ( documentToCheck.AtEndOfStream ) And ( Not inputError )
   legalLineCount = ( MIN_LINES <= lineCount ) And ( lineCount <= MAX_LINES )
   DocumentIsValid = allDataRead And legalLineCount And ( Not ErrorProcessing() )

End Function
```

这里引入中间变量以澄清最后一行的测试

关联参考　要详细了解如何借助中间变量来澄清布尔测试，请参见第 12.5 节的“使用布尔变量对程序加以文档说明”。

本例假定 ErrorProcessing()是表示当前处理状态的布尔函数。现在，阅读代码的主要流程时，就不必再去深究复杂的测试：

Visual Basic示例：去掉复杂测试的代码主流程

```
If ( DocumentIsValid( document, lineCount, inputError ) ) Then
   ' do something or other
   ...
End If
```

KEY POINT 如测试只用一次，你可能觉得并不值得把它放到一个单独的子程序中。但是，把测试放到一个好名字的函数中可以改善可读性，能更清楚地理解代码所做的事情。仅此一个理由足矣。

新的函数名称在程序中引入了一个抽象的概念，在代码中说明了测试的目的。这甚至比用注释来说明测试更好，因为人们主要会看代码而非注释，也更容易保持更新。

使用决策表替代复杂条件　有的时候，一个复杂的测试会涉及几个变量。此时可考虑用决策表来完成测试，而不是使用大量 if 或 case。决策表查询最开始比较容易编码，只有几行代码，没有棘手的控制结构。由于尽可能降低了复杂性，所以不太容易出错。如数据发生变化，可修改决策表而不必修改代码；更新数据结构的内容即可。

关联参考　要详细了解如何使用表来替代复杂逻辑，请参见第 18 章。

构建肯定形式的布尔表达式

I ain’t not no undummy.（我又不是不显得不那么傻。）

——Homer Simpson

如果一个较长的表述采用了非肯定的形式，没有几个人在理解的时候没有一点困难——换言之，大多数人在理解大量否定表述时都有困难。可以做几件事情来避免程序中出现复杂的否定布尔表达式：

在 if 语句中，将否定转换为肯定，并在 if 和 else 子句中翻转代码 下面是一个对否定的表述进行测试的例子：

这里取反

Java示例：让人困惑的取反布尔测试

```
if ( !statusOK ) {
   // do something
   ...
}
else {
   // do something else
   ...
}
```

简单地修改一下，即可对肯定的表述进行测试：

这行的测试发生了反转

else块的代码交换到这里来...

if块的代码交换到这里来...

Java示例：肯定形式的布尔测试更容易理解

```
if ( statusOK ) {
   // do something else
   ...
}
else {
   // do something
   ...
}
```

修改的版本和原始版本在逻辑上一致，但更容易阅读，因为否定的表述已被改为肯定。

另一个方案是选择一个不同的变量名，一个可以反转测试真值的变量。本例可用 ErrorDetected 代替 statusOK。statusOK 为 false 时它为 true。

关联参考　这里建议将布尔表达式换成肯定形式，但这和“在 if 后面而不是在 else 后面编码正常情况”的建议相矛盾(参见第 15.1 节“if 语句”)。此时，你必须考虑每种方法的好处，并决定哪种方法更适合自己的情况

用德摩根定理简化否定形式的布尔判断 德摩根定律（DeMorgan’s Theorems）让我们能充分利用一个表达式和另一个含义相同、但却采用双重否定形式的表达式之间的逻辑关系。例如，可能会遇到包含如下测试的代码片段：

Java示例：一个否定测试

```
if ( !displayOK || !printerOK ) ...
```

它在逻辑上等价于：

Java示例：应用了德摩根定律之后

```
if ( !( displayOK && printerOK ) ) ...
```

这里不需要翻转 if 和 else 子句；上述两个代码片段中的表达式在逻辑上等价。为了将德摩根定律应用于逻辑运算符 and/or 和一对操作数，需对每个操作数取反，将 and 和 or 互换，再对整个表达式进行取反。表 19-1 总结了基于德摩根定律定理所有可能的转换。

表 19-1　基于德摩根定律转换逻辑表达式

原始表达式	等价表达式
not A and not B	not (A or B)
not A and B	not (A or not B)
A and not B	not (not A or B)
A and B	not (not A or not B)
not A or not B*	not (A and B)
not A or B	not (A and not B)
A or not B	not (not A and B)
A or B	not (not A and not B)
* 这就是本例中所采用的表达式	

用圆括号澄清布尔表达式

> 关联参考　有关借助圆括号让其他类型的表达式更加清晰易懂的示例，请参见第 31.2 节中的“括号”。

对于复杂的布尔表达式，与其依赖语言的求值顺序，不如用圆括号来明确意图。使用圆括号使减轻读者的负担，因其可能不了解你的语言的布尔表达式求值细节。如果你足够聪明，就不会要求自己或读者牢记求值优先级——尤其是必须在两种或更多语言之间切换时。和发电报不一样，使用圆括号无需为每个新增的字符付费。

下面是一个圆括号用得太少的表达式：

Java示例：这个表达式的圆括号用得太少

```
if ( a < b == c == d ) ...
```

这个表达式乍一看就使人犯晕。更使人犯晕的是，搞不清楚编码者是想测试(a < b) == (c == d)还是((a < b) == c) == d。下面这个版本的表达式仍然有点令人困惑，但圆括号起了很大的帮助：

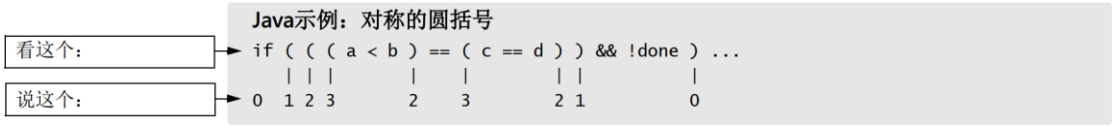
Java示例：多用圆括号后，表达式变得更好了

```
if ( ( a < b ) == ( c == d ) ) ...
```

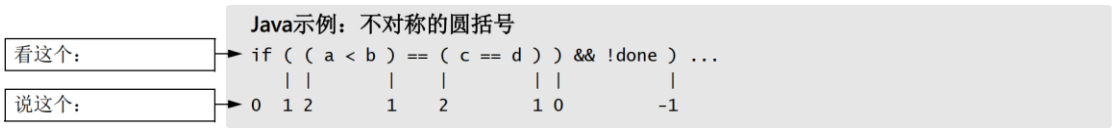
在这个例子中，圆括号有助于改善可读性和程序的正确性——编译器不会以这种方式解释第一个代码片断。但凡有疑问，就用圆括号。

> 关联参考　许多代码编辑器都提供了圆括号、方括号和花括号的配对功能。有关代码编辑器的详情，请参见第 30.2 节中的“编辑”。

使用一个简单的计数技巧使圆括号对称　如难以判断小括号是否对称，这里有一个简单的计数技巧可以帮到你。先说“零”。沿着表达式从左向右移动。遇到一个起始圆括号（(），说“一”。每遇到一个起始圆括号，都递增这个数字。每遇到一个结束圆括号（)），都递减这个数字。在表达式结束时归零，表明圆括号是对称的。



本例收尾于 0，表明圆括号对称。下例的圆括号则是不对称的：



到最后一个结束圆括号之前就归零，这表明在此之前缺少了一个圆括号。在表达式的最后一个圆括号之前不应归零。

用圆括号完整描述逻辑表达式　圆括号没什么代价，且有助于改善可读性。应习惯于完整地用圆括号描述逻辑表达式。

理解布尔表达式如何求值

对布尔表达式进行求值时，许多语言都支持一种隐含的控制形式。一些语言的编译器会先求值布尔表达式中的每一项，再合并这些项并求值整个表达式。其他语言的编译器支持“短路”（short-circuit）或称“懒惰”（lazy）求值，即求值必要的部分。如基于第一个测试的结果，第二个测试没必要继续，这个设计就特别好用。例如，假设要用以下测试对一个数组的元素进行检查：

```
伪代码示例：错误的测试
while ( i < MAX_ELEMENTS and item[ i ] <> 0 ) ...
```

如对整个表达式进行求值，会在循环的最后一次迭代时得到一个错误。变量 i 等于 maxElements，所以表达式 item[i] 等同于 item[maxElements]，这是一个数组索引错误。你可能会说，我只是在查看值，又不是更改它，所以这无所谓。但这是不严谨的编程实践，会使看代码的人感到困惑。在许多环境中，它还会引发运行时错误或内存保护违例。

可在伪代码中重构测试以避免错误：

```
伪代码示例：正确重构的测试
while ( i < MAX_ELEMENTS )
   if ( item[ i ] <> 0 ) then
      ...
```

这是正确的，因为除非 i 小于 maxElements，否则 item[i]不会被求值。

许多现代语言都提供了一些机制来防止这种错误的发生。例如，C++使用短路求值：如 and 的第一个操作数为 false，就不对第二个操作数进行求值，因为反正整个表达式都为 false。换言之，在 C++中，对于以下表达式：

```
if ( SomethingFalse && SomeCondition ) ...
```

一旦 SomethingFalse 求值 false，就停止对整个表达式的求值。

or 操作符也有类似的短路求值机制。在 C++和 Java 中，对于以下表达式：

```
if ( somethingTrue || someCondition ) ...
```

一旦 somethingTrue 求值为 true，对整个表达式的求值就会终止，因为任何一部分为 true，整个表达式必然为 true。

基于这种求值方法，以下语句不仅写好很好，还完全合法：

Java示例：利用短路求值

```
if ( ( denominator != 0 ) && ( ( item / denominator ) > MIN_VALUE ) ) ...
```

若整个表达式在分母为 0 时求值，第二个操作数的除法会造成除以 0 错误。但由于第二部分仅在第一部分为 true 时才会求值，所以只要分母为 0，第二部分永远不会求值，因此不会发生除以 0 错误。

另一方面，由于&&（and）从左向右求值，所以下面这个在逻辑上等价的语句无法工作：

Java示例：短路求值也救不了

```
if ( ( ( item / denominator ) > MIN_VALUE ) && ( denominator != 0 ) ) ...
```

在这种情况下，item / denominator 会先于 denominator != 0 求值。所以，这段代码会发生除以 0 错误。

Java 的“逻辑”操作符使这个情况进一步复杂化。Java 逻辑&和|操作符保证所有项都被完全求值，它们会忽略短路求值。换言之，在 Java 中，这样写很安全：

Java示例：因为有短路(条件)求值，所以这个测试能起作用

```
if ( ( denominator != 0 ) && ( ( item / denominator ) > MIN_VALUE ) ) ...
```

但这样写不安全：

Java示例：由于不保证短路求值，所以这个测试不起作用

```
if ( ( denominator != 0 ) & ( ( item / denominator ) > MIN_VALUE ) ) ...
```

不同语言使用不同的求值方式，而且语言的实现者往往对表达式的求值采取自由态度。所以，请检查语言相应版本的手册，了解具体使用的是哪种求值方式。更好的办法是，由于看你代码的人可能不像你那么敏锐，所以用嵌套测试来澄清你的意图，而不要单纯依赖求值顺序和短路求值。

按数轴顺序写数值表达式

组织数值测试，跟着数轴上的点走。通常，应组织数值测试来进行以下形式的比较：

```
MIN_ELEMENTS <= i and i <= MAX_ELEMENTS
i < MIN_ELEMENTS or MAX_ELEMENTS < i
```

我们的思路是使元素从左向右排序，从最小到最大。在第一行代码中，MIN_ELEMENTS 和 MAX_ELEMENTS 是两个端点，因此被放在两端。变量 *i* 应该在它们之间，所以放在中间。在第二行代码中，要测试 *i* 是否在范围之外，所以将 *i* 放在测试两端，MIN_ELEMENTS 和 MAX_ELEMENTS 则位于中间，如图图 19-1 所示。

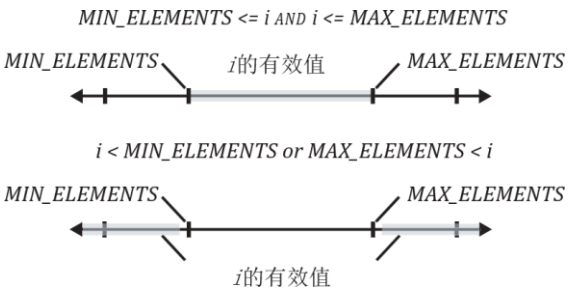


图 19-1　按数轴顺序进行布尔测试的例子

只是拿 *i* 和 MIN_ELEMENTS 测试时，*i* 的位置应因测试成功时 *i* 的位置而变化。如果 *i* 应该更小，应该这样写测试：

```
while ( i < MIN_ELEMENTS ) ...
```

但是，如果 *i* 应该更大，则应这样写：

```
while ( MIN_ELEMENTS < i ) ...
```

这种方法比以下测试更清晰：

```
( i > MIN_ELEMENTS ) and ( i < MAX_ELEMENTS )
```

上述测试不能帮助读者直观了解被测试的内容。

和 0 比较时的指导原则

编程语言中的 0 有多种用途。它是数值，是字符串中的空终止符，是空指针的值，是枚举中第一项的值，是逻辑表达式中的 false。由于有这么多用途，所以写代码时要强调 0 的具体用法。

隐式比较逻辑变量　如前所述，应该使用下面这样的逻辑表达式：

```
while ( !done ) ...
```

隐式和 0 比较很恰当，因为这个比较是在一个逻辑表达式中。

将数字和 0 比较　虽然隐式比较逻辑表达式是恰当的，但数值表达式应显式比较。对于数字，应该这样写：

```
while ( balance != 0 ) ...
```

而不是这样写：

```
while ( balance ) ...
```

在 C 中显式将字符和空终止符 (('\0')) 比较　字符和数字一样，都不是逻辑表达式。所以，对于字符要这样写：

```
while ( *charPtr != '\0' ) ...
```

而不要这样写：

```
while ( *charPtr ) ...
```

这个建议违背了处理字符数据的一般 C 规范（如第二个例子所示），但它强调了表达式处理的是字符数据而非逻辑数据。有些 C 语言规范并不是从最大化可读性或可维护性来出发的，这就是一个例子。

幸好，随着越来越多的代码使用 C++和 STL 字符串编写，这个问题正在逐渐变得不是问题。

将指针和 NULL 比较　对于指针，要这样写：

```
while ( bufferPtr != NULL ) ...
```

而不要这样写：

```
while ( bufferPtr ) ...
```

和字符的建议一样，它违背了约定俗成的 C 规范，但好处是可读性增加了。

布尔表达式的常见问题

特定语言存在的一些额外的陷阱会对布尔表达式产生影响：

在 C 家族语言中，将常量放在比较表达式的左侧　C 家族语言的布尔表达式存在一些特殊的问题。如害怕误将==写成=，可遵循将常量和字面量放在表达式左侧的编程惯例，例如：

C++示例：常量放在表达式左侧—出错时可由编译器捕获

```
if ( MIN_ELEMENTS = i ) ...
```

在这个表达式中，编译器应将单个=标记为错误，因为将任何东西赋给常量都是无效的。相反，如果是下面的表达式，编译器只会将其标记为警告，而且只有你完全打开编译器警告的情况下才看得见：

C++示例：常量放在表达式右侧—出错时编译器捕获不了

```
if ( i = MIN_ELEMENTS ) ...
```

这个建议与使用数轴顺序的建议相冲突。我个人倾向于使用数轴排序，并让编译器对非预期的赋值发出警告。

在 C++中，考虑为&&、||和==创建预处理宏替代物（但只作为最后的手段） 如果有这样的问题，可考虑为布尔 and 和 or 创建#define 宏，并用 AND 和 OR 替代&&和||。类似地，人们很容易在本该使用==时用了=。如经常被这个错误困扰，可为逻辑相等（==）创建一个类似 EQUALS 的宏。

许多有经验的程序员认为，这种方法对那些不能正确掌握编程语言细节的程序员来说有助于提高可读性，但对那些对语言比较精通的程序员来说则降低了可读性。此外，大多数编译器会对似乎有错的赋值和按位（bitwise）操作符使用发出警告。开启完整编译器警告通常比创建非标准宏更好。

在 Java 中，要知道 a==b 和 a.equals(b)的区别 在 Java 中，a==b 测试 a 和 b 是否引用同一个对象，而 a.equals(b)测试对象是否有相同的逻辑值。通常，Java 程序应使用 a.equals(b)而非 a==b。

19.2 复合语句（语句块）

“复合语句”或“代码块”（block）是一组语句的集合，它们出于程序流程控制的目的而被视为单个语句。在 C++、C#、C 和 Java 中，复合语句通过在一组语句周围添加{和}来创建。有的时候，它们会被一个命令的关键字所暗指，如 Visual Basic 中的 For 和 Next。下面提供了有效使用复合语句的指导原则：

> 关联参考 许多代码编辑器都提供了圆括号、方括号和花括号的配对功能。有关代码编辑器的详情，请参见第 30.2 节中的“编辑”。

先写好成对的花括号 写好代码块的起始和结束花括号后，再在中间填充内容。人们经常抱怨花括号的配对有多难。但是这完全是一个不必要的问题。只要遵循这一原则，就不会再有配对问题了。

先写循环头：

```
for ( i = 0; i < maxLines; i++ )
```

再这样写空白循环体：

```
for ( i = 0; i < maxLines; i++ ) {   }
```

最后填充内容：

```
for ( i = 0; i < maxLines; i++ ) {
   // whatever goes in here ...
}
```

所有成块的结构都适合这样做，包括 C++和 Java 中的 if、for 和 while，以及 Visual Basic 中的 If-Then-Else、For-Next 和 While-Wend 组合。

使用花括号来澄清条件 条件本身就够难读了，还要确定哪些语句与 if 测试配对。在 if 测试后面放单一的语句似乎很酷，但在后期维护时，单个语句往往需要扩充成更复杂的语句块，不加花括号的单一语句在这种情况下很容易出错。

无论块内有 1 行还是 20 行代码，都用语句块来明确你的意图。

19.3 空语句

C++支持空语句（null statement），即完全由一个分号构成的语句，如下所示：

C++示例：传统空语句

```
while ( recordArray.Read( index++ ) != recordArray.EmptyRecord() )
   ;
```

C++的 while 要求后跟一个语句，但它可以是一个空语句。单独占一行的分号就是空语句。下面列出了在 C++中处理空语句时的指导原则：

关联参考 处理空语句最好的方法或许就是避免使用。详情参见 16.2 节。

提醒注意空语句 空语句并不常见，所以要使其明显。一个方法是让空语句的分号单独占一行。并像其他语句一样缩进。这就是上例采用的方法。另外，也可用一组空的花括号来强调空语句。下面是两个例子：

用这个方法强调空语句 →
这是另一个方法 →

C++示例：对空语句予以强调

```
while ( recordArray.Read( index++ ) ) != recordArray.EmptyRecord() ) {}

while ( recordArray.Read( index++ ) != recordArray.EmptyRecord() ) {
   ;
}
```

为空语句创建预处理 DoNothing()宏或内联函数 空语句不做任何事，只是为了申明“什么都不做”这一事实。这相当于在空白页上标注“本页故意留白”。虽然这一页并不是真的空白，但你明白这一页上不应该有其他任何东西。

下面展示了如何在 C++中利用#define 制作自己的空语句。(也可将其创建为内联函数，这将有同样的效果)。

C++示例：用*DoNothing()*强调这是空语句

```
#define DoNothing()
...
while ( recordArray.Read( index++ ) != recordArray.EmptyRecord() ) {
   DoNothing();
}
```

除了在空的 while 和 for 循环中使用 DoNothing()外，还可在 switch 语句不重要的选择中使用它。故意包括 DoNothing()，可清楚表明该情况已经考虑过了，而且什么也不应该做。

如语言不支持预处理宏或内联函数，可创建一个 DoNothing()子程序，并在子程序中直接将控制返回给调用者。

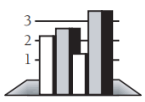
想想非空的循环体是否能让代码更清晰 大多数导致循环体为空的代码都依赖于循环控制代码的副作用。大多数时候，显式声明副作用会使代码更易读，如下例所示：

C++示例：用非空循环体重写的代码变得更清晰了

```
RecordType record = recordArray.Read( index );
index++;
while ( record != recordArray.EmptyRecord() ) {
   record = recordArray.Read( index );
   index++;
}
```

这种方法引入了一个额外的循环控制变量。虽然要写更多代码，但它强调了直接的编程实践，而不是取巧地利用副作用。生产代码尤其需要像这样进行强调。

19.4 驾驭深层嵌套



HARD DATA

过度缩进（或“嵌套”）在计算机文献中已被斥责了 25 年，并且仍然是造成代码混乱的主要原因之一。Noam Chomsky 和 Gerald Weinberg 的研究表明，很少有人能理解超过三层的嵌套 if(Yourdon 1986a)，许多研究人员建议嵌套不要超过三或四层(Myers 1976, Marca 1981, and Ledgard and Tauer 1987a)。深层嵌套与第 5 章描述的软件首要技术使命“管理复杂度”相悖。仅这一点理由就足以。


KEY POINT

要避免深层嵌套并不难。如果有深层嵌套，可重新设计在 if 和 else 子句中进行的测试，或者将代码重构为更简单的子程序。下面介绍了几种减少嵌套深度的方法：

通过重复测试部分条件来简化嵌套 if　如嵌套太深，可通过重复测试部分条件来减少嵌套层级。下例的嵌套深度足以让我们进行重构：


CODING HORROR

C++示例：糟糕的深层嵌套代码

```
if ( inputStatus == InputStatus_Success ) {
   // lots of code
   ...
   if ( printerRoutine != NULL ) {
      // lots of code
      ...
      if ( SetupPage() ) {
         // lots of code
         ...
         if ( AllocMem( &printData ) ) {
            // lots of code
            ...
         }
      }
   }
}
```

关联参考　重复测试部分条件以降低复杂度的方法类似于重复测试状态变量。该方法的详情请参见第 17.3 节的“错误处理和 goto”。

本例是为了演示嵌套层级而刻意设计的。// lots of code 部分表明该子程序有足够多的代码，会跨越几个屏幕或超出打印纸的边界。以下是修改后的代码，它利用了重复测试而不是嵌套：

C++示例：通过重复测试减少嵌套

```
if ( inputStatus == InputStatus_Success ) {
   // lots of code
   ...
   if ( printerRoutine != NULL ) {
      // lots of code
      ...
   }
}

if ( ( inputStatus == InputStatus_Success ) &&
   ( printerRoutine != NULL ) && SetupPage() ) {
   // lots of code
   ...
   if ( AllocMem( &printData ) ) {
      // lots of code
      ...
   }
}
```

这是一个特别现实的例子，因其表明不能免费减少嵌套层级；必须忍受更复杂的测试以换取嵌套层级的降低。但是，从四层减少到两层在可读性方面是一个很大的改进，值得考虑。

使用 break 块简化嵌套 if　除了刚才描述的方法，一个替代方法是定义一个代码块。如块中某些条件失败了，就通过 break 跳至这个块的末尾。

C++示例：利用*break*代码块

```
do {
   // begin break block
   if ( inputStatus != InputStatus_Success ) {
      break; // break out of block
   }
   // lots of code
   ...
   if ( printerRoutine == NULL ) {
      break; // break out of block
   }

   // lots of code
   ...
   if ( !SetupPage() ) {
      break; // break out of block
   }

   // lots of code
   ...
   if ( !AllocMem( &printData ) ) {
      break; // break out of block
   }

   // lots of code
   ...
} while (FALSE); // end break block
```

但这个技术并不常见，只有当整个团队都熟悉它，且已被团队采纳为公认的编码实践时，才可以使用。

将嵌套 if 转换为一组 if-then-else　慎重思考嵌套 if 测试，可能会发现能重新组织它，变成一组 if-then-else 而不是嵌套 if。以下面这个复杂的决策树为例：

Java示例：过犹不及的决策树

```
if ( 10 < quantity ) {
   if ( 100 < quantity ) {
      if ( 1000 < quantity ) {
         discount = 0.10;
      }
      else {
         discount = 0.05;
      }
   }
   else {
      discount = 0.025;
   }
}
else {
   discount = 0.0;
}
```

这个测试在几个方面组织得很差，其中之一存在多余的测试。测试数量是否大于 1000 时，不需要同时测试它是否大于 100 和大于 10。所以，可以重新组织代码：

Java示例：将嵌套*if*转换为一组*if-then-else*

```
if ( 1000 < quantity ) {
   discount = 0.10;
}
else if ( 100 < quantity ) {
   discount = 0.05;
}
else if ( 10 < quantity ) {
   discount = 0.025;
}
else {
   discount = 0;
}
```

这个方案比其他一些方案更容易懂，因为数字很整齐。如数字不是那么整齐，就像下面这样重新设计嵌套 if：

Java示例：若数字显得有点“乱”，将嵌套*if*转换为一组*if-then-else*

```
if ( 1000 < quantity ) {
   discount = 0.10;
}
else if ( ( 100 < quantity ) && ( quantity <= 1000 ) ) {
   discount = 0.05;
}
else if ( ( 10 < quantity ) && ( quantity <= 100 ) ) {
   discount = 0.025;
}
else if ( quantity <= 10 ) {
   discount = 0;
}
```

这段代码与之前的代码的主要区别在于，else-if 子句中的表达式不依赖于之前的测试。这段代码不需要 else 子句，而且测试实际上可按任意顺序进行。这段代码可由四个 if 构成，不需要使用 else。else 版本较好的唯一原因是，它避免了不必要的重复测试。

将嵌套 if 转换为 case 语句 可重新编码某些类型的测试（尤其是那些带整数的），使用 case 语句而不是 if 和 else 链。某些语言不支持这样的技术，但在那些支持的语言中，这是一种强大的技术。下面展示了如何在 Visual Basic 中重新编码这个例子：

Visual Basic示例：将嵌套*if*转换为*case*语句

```
Select Case quantity
   Case 0 To 10
      discount = 0.0
   Case 11 To 100
      discount = 0.025
   Case 101 To 1000
      discount = 0.05
   Case Else
      discount = 0.10
End Select
```

这个例子读起来像一本书。和前几页的两个多重缩进例子相比，它显得特别清爽。

将深层嵌套的代码纳入自己的子程序 如果深层嵌套发生在循环内部，通常可以将循环内部的代码纳入自己的子程序来改善这种情况。如嵌套是条件和迭代的结果，这样做尤其有效。将 if-then-else 分支保留在主循环中以显示决策分支，然后将分支内的语句移到自己的子程序中。以下代码就需要进行这样的改进：

C++示例：这些嵌套代码需要提取为单独的子程序

```
while ( !TransactionsComplete() ) {
   // read transaction record
   transaction = ReadTransaction();

   // process transaction depending on type of transaction
   if ( transaction.Type == TransactionType_Deposit ) {
      // process a deposit
      if ( transaction.AccountType == AccountType_Checking ) {
         if ( transaction.AccountSubType == AccountSubType_Business )
            MakeBusinessCheckDep( transaction.AccountNum, transaction.Amount );
         else if ( transaction.AccountSubType == AccountSubType_Personal )
            MakePersonalCheckDep( transaction.AccountNum, transaction.Amount );
         else if ( transaction.AccountSubType == AccountSubType_School )
            MakeSchoolCheckDep( transaction.AccountNum, transaction.Amount );
      }
      else if ( transaction.AccountType == AccountType_Savings )
         MakeSavingsDep( transaction.AccountNum, transaction.Amount );
      else if ( transaction.AccountType == AccountType_DebitCard )
         MakeDebitCardDep( transaction.AccountNum, transaction.Amount );
      else if ( transaction.AccountType == AccountType_MoneyMarket )
         MakeMoneyMarketDep( transaction.AccountNum, transaction.Amount );
      else if ( transaction.AccountType == AccountType_Cd )
         MakeCDDep( transaction.AccountNum, transaction.Amount );
   }
   else if ( transaction.Type == TransactionType_Withdrawal ) {
      // process a withdrawal
      if ( transaction.AccountType == AccountType_Checking )
         MakeCheckingWithdrawal( transaction.AccountNum, transaction.Amount );
      else if ( transaction.AccountType == AccountType_Savings )
         MakeSavingsWithdrawal( transaction.AccountNum, transaction.Amount );
      else if ( transaction.AccountType == AccountType_DebitCard )
         MakeDebitCardWithdrawal( transaction.AccountNum, transaction.Amount );
   }
   else if ( transaction.Type == TransactionType_Transfer ) {
      MakeFundsTransfer(
         transaction.SourceAccountType,
         transaction.TargetAccountType,
         transaction.AccountNum,
         transaction.Amount
      );
   }
   else {
      // process unknown kind of transaction
      LogTransactionError( "Unknown Transaction Type", transaction );
   }
}
```

这里是 *TransactionType_Transfer* 交易类型

虽然很复杂，但还不是你所见过的最糟糕的代码。它只嵌套了四层，有注释，有逻辑缩进，功能分解也很充分，特别是对于 TransactionType_Transfer 交易类型。虽然已经很充分，但还是可以通过将内层 if 测试的内容分解成它们自己的子程序来加以改进。

C++分解成子程序后好的、嵌套的代码

```
while ( !TransactionsComplete() ) {
   // read transaction record
   transaction = ReadTransaction();

   // process transaction depending on type of transaction
   if ( transaction.Type == TransactionType_Deposit ) {
      ProcessDeposit(
         transaction.AccountType,
         transaction.AccountSubType,
         transaction.AccountNum,
         transaction.Amount
      );
   }
   else if ( transaction.Type == TransactionType_Withdrawal ) {
      ProcessWithdrawal(
         transaction.AccountType,
         transaction.AccountNum,
         transaction.Amount
      );
   }
   else if ( transaction.Type == TransactionType_Transfer ) {
      MakeFundsTransfer(
         transaction.SourceAccountType,
         transaction.TargetAccountType,
         transaction.AccountNum,
         transaction.Amount
      );
   }
   else {
      // process unknown transaction type
      LogTransactionError("Unknown Transaction Type", transaction );
   }
}
```

关联参考　如果一开始就按照第 9 章描述的步骤来创建子程序，这种功能分解会相当容易。有关功能分解的指导原则，请参见第 5.4 节中的“分而治之”。

新的子程序中的代码只是从原来的子程序中提取出来，并形成了新的子程序。(这里没有显示新的子程序。)新的代码有几个优点。首先，两级嵌套使结构更简单，更容易理解。其次，可在一屏上阅读、修改和调试较短的 while 循环——不需要跨屏，也不会超出打印纸的边界。第三，将 ProcessDeposit()和 ProcessWithdrawal()的功能放到子程序中，可获得模块化的其他所有常规优势。第四，现在很容易看出代码能被分解成一个 case 语句，这会进一步提升它的可读性，如下所示：

C++示例：分解并使用*case*语句后好的、嵌套的代码

```
while ( !TransactionsComplete() ) {
   // read transaction record
   transaction = ReadTransaction();

   // process transaction depending on type of transaction
   switch ( transaction.Type ) {
      case ( TransactionType_Deposit ):
         ProcessDeposit(
            transaction.AccountType,
            transaction.AccountSubType,
            transaction.AccountNum,
            transaction.Amount
            );
         break;

      case ( TransactionType_Withdrawal ):
         ProcessWithdrawal(
            transaction.AccountType,
            transaction.AccountNum,
            transaction.Amount
            );
         break;

      case ( TransactionType_Transfer ):
         MakeFundsTransfer(
            transaction.SourceAccountType,
            transaction.TargetAccountType,
            transaction.AccountNum,
            transaction.Amount
            );
         break;

      default:
         // process unknown transaction type
         LogTransactionError("Unknown Transaction Type", transaction );
         break;
   }
}
```

使用更面向对象的方法 在面向对象的环境中，简化这种特殊代码的直接方法是创建一个抽象 Transaction 基类，再派生出 Deposit（存）、Withdrawal（取）和 Transfer（转）子类。

系统无论规模，都可将 switch 语句转换为一个工厂方法，以便在任何需要创建 Transaction 类型的对象的地方重复使用。如上述代码是在这样的系统中，这部分会变得更简单：

C++ Example of Good Code That Uses Polymorphism and an Object Factory

```
TransactionData transactionData;
Transaction *transaction;

while ( !TransactionsComplete() ) {
   // read transaction record and complete transaction
   transactionData = ReadTransaction();
   transaction = TransactionFactory.Create( transactionData );
   transaction->Complete();
   delete transaction;
}
```

关联参考　参见第 24 章“重构”更多地了解像这样的代码改进措施。

为了内容的完整，注意 TransactionFactory.Create()子程序中的代码从上例的 switch 语句中摘抄即可：

C++示例：一个对象工厂的好代码

```
Transaction *TransactionFactory::Create(
   TransactionData transactionData
   ) {

   // create transaction object, depending on type of transaction
   switch ( transactionData.Type ) {
      case ( TransactionType_Deposit ):
         return new Deposit( transactionData );
         break;

      case ( TransactionType_Withdrawal ):
         return new Withdrawal( transactionData );
         break;

      case ( TransactionType_Transfer ):
         return new Transfer( transactionData );
         break;

      default:
         // process unknown transaction type
         LogTransactionError( "Unknown Transaction Type", transactionData );
         return NULL;
   }
}
```

重新设计深层嵌套的代码

一些专家认为，在面向对象的程序设计中，case 语句几乎总是代表拙劣的代码。很少需要它（甚至根本不要用）（Meyer 1997）。从调用子程序的 case 语句到多态对象工厂方法的转换就属于这种情况。

更常规的是，复杂代码是一个信号，表明你对程序没有足够的了解来使它变得简单。深层嵌套是一个警告信号，表明需要分解成子程序或重新设计代码中复杂的部分。这并不是说必须修改子程序，但如果不这样做的话，应该有一个很好的理由。

总结减少深层嵌套的技术

下面列出了可用来减少深度嵌套的技术，还列出了本书讨论到这些技术的其他章节：

- 重复测试条件的一部分（本节）
- 转换为 if-then-else（本节）。
- 转换为 case 语句（本节）
- 将深度嵌套的代码提取为自己的子程序（本节）
- 使用对象和多态分派（本节）
- 重写代码以使用状态变量（第 17.3 节）
- 使用防卫子句退出子程序，使代码的正常路径更清晰（第 17.1 节）
- 使用异常（第 8.4 节）
- 完全重新设计深层嵌套的代码（本节）

19.5 编程基础：结构化编程

“结构化编程”（structured programming）一词起源于 Edsger Dijkstra 在 1969 年 NATO 软件工程会议上发表的一篇具有里程碑意义的论文“*Structured Programming*”（Dijkstra 1969）。自结构化编程这一概念被提出后，“结构化”一词被应用于每一个软件开发活动，包括结构化分析、结构化设计和结构化折腾（structured goofing off）。各种结构化的方法论并没有任何共同点，唯一共同的是它们都是在“结构化”这个词给它们带来额外魅力的时候产生的。

结构化编程的核心思想很简单，即程序应该只使用单进单出的控制结构。单进单出的控制结构是一个代码块，它只有一个地方可以开始，只有一个地方可以结束。没有其他入口或出口。结构化编程与结构化、自上而下的设计不一样。它只针对详细编码的层级。

一个结构化的程序以一种有序的、有规律的方式进行，不会不可预测地跳来跳去。可以自上而下阅读，其执行方式也基本是这样。由于方法不太规范，导致不好通过阅读源代码弄懂程序在机器中的执行方式。可读性差意味着不好理解，最终降低了程序的质量。

结构化编程的核心概念今天仍然有用，break、continue、throw、catch、return 的使用以及其他主题都需要考虑它。

结构化编程的三个组成部分

后续三个小节描述了构成结构化编程核心的三种结构。

关联参考　要详细了解顺序结构的使用，请参见第 14 章。

顺序

顺序结构是一组按顺序执行的语句。典型的顺序语句包括赋值和调用子程序。下面是两个例子：

Java示例：顺序代码

```
// a sequence of assignment statements
a = "1";
b = "2";
c = "3";

// a sequence of calls to routines
System.out.println( a );
System.out.println( b );
System.out.println( c );
```

选择

> 关联参考　要详细了解选择结构的使用，请参见第 15 章。

选择结构使语句有选择地执行。if-then-else 语句是一个常见的例子。要么执行 if-then 子句，要么执行 else 子句，但不能都执行。其中一个子句被“选择”执行。

case 语句是另一个选择控制的例子。C++和 Java 中的 switch 语句和 Visual Basic 中的 select 语句都是 case 的例子。在这些例子中，都是从几种 case 中选择一个执行。从概念上讲，if 语句和 case 语句是相似的。如语言不支持 case，可以用 if 语句来模拟。下面是两个例子：

Java示例：选择

```
// selection in an if statement
if ( totalAmount > 0.0 ) {
   // do something
   ...
}
else {
   // do something else
   ...
}

// selection in a case statement
switch ( commandShortcutLetter ) {
   case 'a':
      PrintAnnualReport();
      break;
   case 'q':
      PrintQuarterlyReport();
      break;
   case 's':
      PrintSummaryReport();
      break;
   default:
      DisplayInternalError( "Internal Error 905: Call customer support." );
}
```

迭代

> 关联参考　要详细了解迭代结构的使用，请参见第 16 章。

迭代结构使一组语句被多次执行。迭代通常被称为“循环”。迭代的种类包括 Visual Basic 中的 For-Next 以及 C++和 Java 中的 while 和 for。以下代码展示了 Visual Basic 中迭代的例子：

Visual Basic示例：迭代

```
' example of iteration using a For loop
For index = first To last
   DoSomething( index )
Next

' example of iteration using a while loop
index = first
While ( index <= last )
   DoSomething ( index )
   index = index + 1
Wend

' example of iteration using a loop-with-exit loop
index = first
Do
   If ( index > last ) Then Exit Do
   DoSomething ( index )
   index = index + 1
Loop
```

结构化编程的核心论点是，任何控制流程都可以从顺序、选择和迭代这三种结构中创建（Böhm Jacopini 1966）。程序员们有时会偏爱那些用起来方便的语言结构，但编程的进步似乎在很大程度是通过限制我们被允许用编程语言做的事情来实现的。在结构化编程之前，goto 的使用提供了控制流程的终极便利，但以这种方式写的代码最终被证明是不可理解和不可维护的。我认为，除了三种标准的结构化编程结构之外，任何控制结构的使用——也就是 break、continue、return、throw-catch 等——都应该以一种批判的眼光来看待。

19.6 控制结构与复杂度

对控制结构如此关注的一个原因是它们对整个程序的复杂度有很大的影响。控制结构用得不好会增大复杂度；用得好则会降低。

> 事情应该力求简单，不过不能过于简单。
>
> ——阿尔伯特·爱因斯坦

KEY POINT “编程复杂度”（programming complexity）的一个衡量标准是为了理解程序而必须在同时关注的心智对象（mental objects）的数量。这种心智游戏是编程中最困难的方面之一，

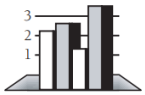
也是编程比其他活动需要更多注意力的原因。它是程序员对“不时被打断”感到不安的原因——这种打断无异于要求杂耍者同时保持三个球在空中，同时手上还要拿着杂货。

直观地说，程序的复杂度似乎在很大程度上决定了理解它所需的工作量。Tom McCabe 发表了一篇有影响力的论文，认为程序的复杂度是由其控制流程定义的（1976）。其他研究人员已确定了 McCabe 循环复杂度度量（cyclomatic complexity metric）之外的因素（例如子程序所用的变量数量）。但他们同意，控制流程至少是造成复杂性的最大因素之一，而且可能就是最大的那个。

复杂度有多重要？

关联参考　复杂度的详情请参见第 5.2 节中的“软件的首要技术使命：管理复杂度”。

计算机科学研究人员意识到复杂度的重要性至少已有二十年了。许多年前，Edsger Dijkstra 就对复杂度的危害提出警告：“称职的程序员会意识到自己大脑容量有限。所以，他以完全谦逊的态度处理编程任务”（Dijkstra 1972）。这并不是说应该增大你脑容量来处理巨大的复杂度。它的意思是说，你永远无法处理巨大的复杂度，必须尽可能采取措施降低它。



HARD DATA 与控制流程相关的复杂度很重要，因为它与低可靠性和频繁的错误相关（McCabe 1976, Shen et al. 1985）。William T. Ward 在 Hewlett-Packard（1989b）的一个报告指出，在使用了 McCabe 的复杂度度量后，软件的可靠性发生了显著提高。McCabe 的度量标准被应用于一个 77000 行的程序来识别有问题的区域。该程序在发布后的缺陷率为每千行代码 0.31 个缺陷。一个 125000 行的程序的发布后缺陷率为每千行代码 0.02 个缺陷。Ward 报告说，由于它们的复杂度较低，所以这两个程序的缺陷比 HP 公司的其他程序少得多。我自己的公司 Construx Software 在 2000 年代使用复杂度度量来识别有问题的子程序，也得出了类似的结果。

降低复杂度的常规指导原则

可通过以下两种方式之一更好地处理复杂度。首先，可进行心智锻炼来提高自己的心理游戏能力。但编程本身通常就能提供足够多的练习，而且人们似乎难以处理超过大约 5~9 个心智实体（Miller 1956）。所以，改进的空间很小。其次，可降低程序的复杂度，并减少理解它们所需的注意力。

如何度量复杂度

深入阅读　这里描述的方法源于 Tom McCabe 颇有影响的论文“A Complexity Measure”（一种复杂度度量方法）(1976)。

你可能有一种直觉，知道什么会使一个子程序变得更复杂或不复杂。研究人员试图将这种直觉正规化，并提出了几种衡量复杂度的方法。或许最有影响力的数字技术是 Tom McCabe 提出的。在他看来，复杂度是通过计算子程序中的“决策点”数量来衡量的。表 19-2 描述了用于统计决策点数量的方法。

表 19-2　用于统计子程序中的决策点数量的技术

1. 从 1 开始，沿子程序常规路径开始计数。
2. 遇到以下每个关键字或其等价物都递增 1：if while repeat for and or。
3. 为 case 语句中的每个 case 递增 1。

下面是一个例子：

```
if ( ( (status = Success) and done ) or
    ( not done and ( numLines >= maxLines ) ) ) then ...
```

对于这段代码，从 1 开始计数，遇到 if 变成 2，遇到 and 变成 3，遇到 or 变成 4，遇到 and 变成 5。所以，这段代码共有 5 个决策点。

如何处理复杂度度量

计算好决策点之后，可用这个数字来分析子程序的复杂度：

0~5	子程序可能还行。
6~10	开始考虑如何简化这个子程序。
10+	将子程序的一部分分解成第二个子程序，并从第一个调用。

将一个子程序的一部分移到另一个子程序中并不会降低程序的整体复杂度；它只是将决策点移动了一下。但它降低了你在任何时候都要应对的复杂度。由于重要的目标是尽量减少必须在心智上处理的项目的数量，所以降低一个特定子程序的复杂度是值得的。

最多 10 个决策点并不是一个绝对的限制。将决策点数量作为一种警告标志，表明一个程序可能需要重新设计。不要把它作为一个呆板的规则。一个有很多 case 的 case 语句可能含有超过 10 个元素；取决于 case 语句的目的，对它进行拆分可能是愚蠢的。

其他类型的复杂度

深入阅读　有关复杂度度量的精辟论述，请参阅(Conte, Dunsmore, andShen 1986)。

McCabe 的复杂度度量并不是唯一合理的度量，但却是计算文献中讨论得最多的。考察控制流程时，它特别有帮助。其他度量还有使用的数据量、控制结构中的嵌套层数、代码行数、对变量的连续引用之间的行数（“跨度”或 span）、一个变量被使用的行数（“生存时间”）以及输入和输出量。一些研究人员在这些较简单的指标的组合基础上开发了复合指标。

检查清单：控制结构问题

- 表达式中采用的是 true 和 false，而不是 1 和 0 吗？
- 布尔值是与 true 和 false 隐式比较吗？
- 数值是与它们的测试值显式比较吗？
- 通过新增布尔变量、使用布尔函数和决策表来简化表达式了吗？

- 采用的是肯定形式的布尔表达式吗？
- 花括号正确配对了吗？
- 在必要时使用花括号使语句更清晰了吗？
- 逻辑表达式都用圆括号括起来了吗？
- 测试是按数轴顺序写的吗？
- 在适当的时候，Java 测试采用的是 a.equals(b)形式，而不是 a==b 形式吗？
- 空语句明显吗？
- 通过重复测试条件的一部分、转换成 if-then-else 或 case 语句、提取到单独的子程序中、转换为更面向对象的设计或采用其他改进方法简化了嵌套语句了吗？
- 如一个子程序的决策点超过 10 个，有充分的理由不重新设计吗？

要点回顾

- 使布尔表达式简单易读将非常有助于提升代码质量。
- 深层嵌套导致子程序变得难以理解。幸好，可以相对容易地避免这种情况。
- 结构化编程是一种简单且仍然适用的理念：可通过组合顺序、选择和迭代这三种基本结构来构建出任何程序。
- 使复杂度最小化是写出高质量代码的关键。

第 24 章　重构

内容

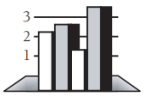
- 24.1 软件演化的类型
- 24.2 重构简介
- 24.3 特定的重构
- 24.4 安全地重构
- 24.5 重构策略

相关章节

- 修复缺陷的技巧：第 23.3 节
- 代码调优方法：第 25.6 节
- 软件构建设计：第 5 章
- 可以工作的类：第 6 章
- 高质量子程序：第 7 章
- 协同构建：第 21 章
- 开发者测试：第 22 章
- 可能改变的区域：第 5.3 节中的“找出容易改变的区域”

> “所有成功的软件都发生过变化。”
>
> ——Fred Brooks

理想世界：一个管理良好的软件项目会进行有条理的需求开发，并定义一个稳定的程序功能清单。设计按需求进行，而且仔细进行，使编码能从开始到结束线性完成。换言之，大多数代码都可以只写一次，测试，然后就忘记。在这个理想世界中，仅在软件维护阶段才需要对代码进行大幅修改，而这种情况只发生在系统的初始版本交付之后。



HARD DATA 现实世界：代码在最初的开发过程中会发生大幅演化。最初编码过程中的许多变化和维护过程中的变化最起码都一样大。编码、调试和单元测试在一个典型的项目中消耗了30%~65%的工作量，具体由项目规模而定(详见第 27 章)。如果像在理想世界中那样，编码和单元测试是一个直截了当的过程，它们则将消耗不超过项目总工作量的 20%~30%。但是，即便在管理良好的项目中，需求每个月也会有 1%~4%的变化（Jones 2000）。需求的变化必然导致代码发生相应的变化——甚至大幅变化。

KEY POINT 另一个现实：现代开发实践增大了在构建过程中代码发生变化的可能性。在传统的软件开发生命周期中，无论成功与否，人们的重点都是避免代码发生变化。但是，较现代的方法正在逐渐远离编码的可预测性。目前的方法更多以代码为中心，在项目的生命周期中，应期待代码比以往发生更多的变化。

24.1 软件演化的类型

软件的演化就像生物进化一样，有的突变有益，其他许多则无益。好的软件演化产生的代码的发展模仿了从猴子到尼安德特人，再到我们现在作为软件开发者的崇高地位的过程。但是，演化的力量有时会反过来打击程序，将其打入一个不断退化的螺旋。

KEY POINT 不同的软件演化类型有一个关键区别，即程序的质量在修改后是提高了还是降低了。如果使用逻辑胶带和迷信来修复错误，质量就会下降。如果将修改当作收紧程序原始设计的机会，质量就会提高。如发现程序质量在下降，就像之前提到的矿井里不再放声高歌的金丝雀。这是一个警告，说明程序正在向错误的方向演变。

区分不同软件演化类型的另一个依据是变化发生于构建期间还是维护期间。这两种演化在几个方面有所不同。构建期的修改通常由最初的开发者进行，而且一般在程序被完全遗忘之前。这个时候，系统还没有上线，所以完成修改的压力只是进度上的压力——而不是 500 个愤怒的用户想知道为什么他们的系统会瘫痪所带来的压力。出于同样的原因，构建期间的修改会更自由——系统处于更动态的状态，犯错的惩罚也很低。这些情况意味着一种不同于软件维护过程中的演化方式。

软件演化哲学

> “不管多庞大、怪异、复杂的代码，维护一下反正不会变得更差。”
>
> ——Gerald Weinberg

在程序员对软件进行演化的各种方法中，一个常见的弱点在于，它是作为一个无意识的过程进行的。如意识到开发过程中的演化是一个不可避免的重要现象，并对其进行规划，就可以反过来加以利用。

进化既是危险的，也是接近完美的机会。如不得不做出修改，要努力改进代码，使未来的修改更容易。刚开始写一个程序的时候，对它的了解绝对比不上你之后对它的了解。若有机会修改程序，一定要倾尽全力改进。无论是刚开始写代码，还是之后进行修改，都要注意方便以后进一步的改动。

软件进化的基本规则是，内部质量应随着代码的进化而提高。下面的章节描述了如何实现这一目标。

KEY POINT 软件演化准则（Cardinal Rule of Software Evolution）是它应改进程序内在的质量。后续小节描述了如何实现这一目标。

24.2 重构简介

为实现软件演化准则，关键策略是重构，Martin Fowler 将其定义为“在不改变其可观察的行为的情况下，对软件内部结构进行修改，使其更容易理解，修改起来代价更低”（Fowler 1999）。现代编程中的“重构”（refactoring）一词由 Larry Constantine 在结构化编程中最初使用的“分解”（factoring）一词发展而来，指的是将一个程序尽可能地分解成其组成部分（Yourdon and Constantine 1979）。

重构的理由

有的时候，代码在维护过程中会发生退化，而另一些时候，代码一开始就不是很好。无论哪种情况，这里有一些警告信号——有时被称为“臭味”（Fowler 1999）——表明哪里需要重构。

代码发生重复 重复的代码几乎总是意味着一开始就没有对设计进行充分的分解。重复的代码使你不得不进行平行的修改——每修改了一个地方，都必须在另一个地方进行平行的修改。这还违反了 Andrew Hunt 和 Dave Thomas 所说的“DRY 原则”：不要重复自己（Don’t Repeat Yourself）（2000）。我认为 David Parnas 说的最精辟：“复制和粘贴是一个设计错误”（McConnell 1998b）。

子程序太长 在面向对象的编程中，很少需要比一屏还要长的子程序，这通常意味着试图将结构化编程的脚强行塞进面向对象的鞋子里。

我有个客户接手的一个任务是分解某遗留系统中最长的子程序，它有 12000 多行。经过努力，他只能将其缩减至 4000 行左右。

改进系统的一个方法是提高其模块化程度——增加定义清晰、名称明确的子程序的数量，这些子程序只做一件事并做好。若变化导致必须重新审视一段代码，利用这个机会检查这一部分中的子程序模块化程度。如某个子程序的一部分提取成一个单独的子程序后会更干净，就创建一个单独的子程序。

循环太长或嵌套太深 循环内部往往适合转化成子程序，这有助于更好地分解代码，降低循环的复杂性。

类的内聚力很差 如发现一个类承担了太多不相关的职责，该类应被分解成多个类，每个类负责一组内聚（coherent，即相互关联）的职责。

类的接口不能提供一致的抽象层级 即使类一开始就提供了内聚性的接口，也可能失去其原有的一致性。类的接口往往会随时间的推移而发生变化，这可能是由于一时兴起而进行了修改，而这些修改偏向于方便而不是接口的完整性。最终，类的接口变成了一个弗兰肯斯坦式的维护怪物，对提高程序的可管理性没什么作用。

参数表有太多参数 分解得好的程序往往有许多小的、进行了良好定义的子程序，并不需要大的参数表。长的参数列表是一个警告，表明子程序接口的抽象化没有经过深思熟虑。

在类中进行的修改各自独立 有的时候，一个类有两个或多个不同的职责。当这种情况发生时，你会发现自己要么修改了类的一部分，要么改变了类的另一部分，但很少有修改会同时影响类的两个部分。这是一个信号，表明该类应按不同职责拆分为多个类。

必须平行修改多个类 我见过一个项目，它有大约 15 个类的一个核对清单。每增加一种新的输出，所有这些类都必须修改。若发现自己经常要对同一组类进行修改，表明可以重新安排这些类中的代码，使修改只影响一个类。根据我的经验，这是个很难实现的理想，但无论如何都是一个很好的目标。

必须平行修改继承层次结构 若发现自己每次创建一个类的子类，都必须创建另一个类的子类，这就属于一种特殊种类的平行修改，应予以解决。

必须平行修改 case 语句 虽然 case 语句本身并不坏，但如果发现自己要在程序的多个部分对类似的 case 语句进行平行修改，就应该问自己继承是不是更好。

一起使用的相关数据项没有被组织成类 如发现自己反复操作同一组数据项，应该问自己这些操作是否应合并到自己的类中。

一个子程序使用了另一个类（而非它自己的类）更多的特性 这表明该子程序应被转移到另一个类中，然后由它的旧类调用。

无脑使用基本数据类型 基本数据类型（primitive data types）可用来表示无限多的现实世界的实体。如程序使用像整数这样的基本数据类型表示一个常见实体，比如货币，就可考虑创建一个简单的 Money 类，这样编译器能对 Money 变量执行类型检查，方便你为 Money 的赋值以及其他操作添加安全检查。但是，如果 Money 和 Temperature 都用整数表示，编译器就不会对像 bankBalance = recordLowTemperature（银行余额 = 极低温度）这样的错误赋值发出警告。

类的作用不大 有的时候，对代码进行重构会造成一个旧类变得无所事事。如果一个类似乎没有发挥什么作用，就应该问自己是否应该把这个类的所有职责分配给其他类，并完全取消这个类。

子程序链传递流浪数据 如果发现将数据传递给一个子程序的目的只是为了让该子程序将其传给另一个子程序，这种数据就称为“垃圾数据”（tramp data）（Page-Jones 1988）。这也许是可以的，但要问一下自己，传递这种数据是否与每个子程序的接口所呈现的抽象一致。如果每个子程序的抽象没有问题，传递这种数据就没有问题。否则，想办法让每个子程序的接口更加一致。

一个中间对象没有做任何事情 如果发现一个类的大多数代码都只是在传递对其他类中子程序的调用，应考虑是否应该取消这个中间对象，直接调用其他类。

一个类与另一个类过于亲密 封装（信息隐藏）可能是你所拥有的最强大的工具，它使你的程序在脑力上易于管理，并使在修改代码时的连锁反应最小化。任何时候，一旦看到某个类对另一个类的了解超过了它应该了解的程度——包括派生类对其父类了解太多——就应选择更强的封装，而不是更弱的封装。

某个子程序的名字太差劲 如某个例程有一个糟糕的名字，就修改它在定义地方的名字。然后再重新编译。虽然现在做这件事很难，但以后会更难，所以一旦发现有问题就马上去做。

公共数据成员 在我看来，公共数据成员总是一个坏主意。它们模糊了接口和实现之间的界限，而且本质上违反了封装原则，限制了未来的灵活性。强烈推荐将公共数据成员隐藏在访问它子程序后面。

一个子类只使用了其父类的一小部分子程序 通常，这意味着该子类的创建是因为父类碰巧包含了它所需要的子程序，而不是因为该子类在逻辑上是其超类的后代。考虑将子类与超类的关系从 is-a（属于）关系转换为 has-a（拥有）关系来实现更好的封装；搞换言之，将超类转换为前子类的成员数据，并只公开前子类中真正需要的子程序。

用注释解释难以理解的代码 注释很重要，但不应用它解释坏的代码。古老的智慧是正确的。“不要记录糟糕的代码——重写它”（Kernighan 和 Plauger 1978）。

关联参考 参见 13.3 节“全局变量”了解全局变量的使用指导原则；参见 5.3 节了解全局数据和类数据的区别。

全局变量的使用 重新审视一段使用全局变量的代码时，请再三斟酌。自上次访问这部分代码以来，可能已经想到了一种避免使用全局变量的方法。由于刚开始写这段代码时还不熟悉，所以现在可能发现全局变量非常令人困惑，所以愿意开发一种更简洁的方法。另外，也可能对如何在访问子程序中隔离全局变量有了更好的认识，并深刻意识到了不这样做带来的痛苦。所以，咬紧牙关，做一些有益的修改。离最开始写这些代码已过了足够久的时间，现在能更客观地看待你的工作，但又足够近，你还记得进行正确修订所需的大多数内容。早期修订阶段最适合改进代码。

子程序在调用前使用了设置（setup）代码，或在调用后使用了收尾（takedown）代码 下面这样的代码应该是一种警告：

这种设置代码是一种警告

这种收尾代码是另一种警告

C++示例：一个子程序调用前后的设置和收尾代码——这样不好

```
WithdrawalTransaction withdrawal;
withdrawal.SetCustomerId( customerId );
withdrawal.SetBalance( balance );
withdrawal.SetWithdrawalAmount( withdrawalAmount );
withdrawal.SetWithdrawalDate( withdrawalDate );

ProcessWithdrawal( withdrawal );

customerId = withdrawal.GetCustomerId();
balance = withdrawal.GetBalance();
withdrawalAmount = withdrawal.GetWithdrawalAmount();
withdrawalDate = withdrawal.GetWithdrawalDate();
```

一个类似的警告信号是，你为 WithdrawalTransaction 类创建了一个特殊构造函数来获取其正常初始化数据的一个子集，以便写这样的代码：

C++示例：一个方法调用前后的设置和收尾代码——这样不好

```
withdrawal = new WithdrawalTransaction( customerId, balance,
   withdrawalAmount, withdrawalDate );
withdrawal.ProcessWithdrawal();
delete withdrawal;
```

但凡看到为了调用子程序而进行设置，或在调用子程序后进行收尾，就问自己子程序的接口是否呈现了正确的抽象。就本例来说，或许应修改 ProcessWithdrawal 的参数列表以支持下面这样的代码：

C++示例：不需要设置或收尾代码的子程序——这样才好

```
ProcessWithdrawal( customerId, balance, withdrawalAmount, withdrawalDate );
```

注意，这个例子反过来也存在类似的问题。如发现自己手上经常有一个 WithdrawalTransaction 对象，但需要把它的几个值传给如下所示的子程序，就应考虑重构 ProcessWithdrawal 接口，使它要求的是 WithdrawalTransaction 对象本身而非单独的字段：

C++示例：需要几个方法调用的代码

```
ProcessWithdrawal( withdrawal.GetCustomerId(), withdrawal.GetBalance(),
   withdrawal.GetWithdrawalAmount(), withdrawal.GetWithdrawalDate() );
```

这些方法中的任何一种都可能是正确的，任何一种都可能是错误的——具体取决于 ProcessWithdrawal()接口的抽象是期待 4 种不同的数据，还是期望一个 WithdrawalTransaction 对象。

程序包含的代码似乎有一天会被需要 在猜测某一天会需要什么功能方面，程序员是出了名的糟糕。“超前设计”（Designing ahead）会出现许多可预测的问题：

- 尚未完全确定对“超前设计”代码的需求，这意味着程序员很可能猜错。所谓的“超前设计”最终会被废弃。
- 即使程序员对未来需求的猜测八九不离十，但他们通常预见不到未来需求的所有复杂情况。这些错综复杂的问题破坏了程序员的基本设计假设，“超前设计”工作最终还是会被废弃。
- 未来使用“超前设计”代码的程序员并不知道那是“超前设计”的代码，或者对这些代码有超出正常的预期。他们以为这些代码在编码、测试和审查方面和其他代码处于一样的水平。所以，他们浪费大量时间来构建使用“超前设计”的代码，最终却发现这些代码实际上并不能工作。
- 额外的“超前设计”代码带来了额外的复杂性，需要额外的测试、额外的缺陷修正等等。最终反而拖延了项目的进度。

专家们一致认为，为未来需求做准备的最好方法不是写空中楼阁式的代码；而是使目前需要的代码尽可能清晰明了，这样未来的程序员就会知道它做什么和不做什么，并相应地做出修改（Fowler 1999, Beck 2000）。

检查清单：重构的理由

- 代码发生重复
- 子程序太长
- 循环太长或嵌套太深
- 类的内聚力很差

- 类的接口不能提供一致的抽象层级
- 参数表有太多参数
- 在类中进行的修改各自独立
- 必须平行修改多个类
- 必须平行修改继承层次结构
- 必须平行修改 case 语句
- 一起使用的相关数据项没有被组织成类
- 一个子程序使用了另一个类（而非它自己的类）更多的特性
- 无脑使用基本数据类型
- 类的作用不大
- 子程序链传递流浪数据
- 一个中间对象没有做任何事情
- 一个类与另一个类过于亲密
- 某个子程序的名字太差劲
- 公共数据成员
- 一个子类只使用了其父类的一小部分子程序
- 用注释解释难以理解的代码
- 全局变量的使用
- 子程序在调用前使用了设置（setup）代码，或在调用后使用了收尾（takedown）代码
- 程序包含的代码似乎有一天会被需要

不重构的理由

人们宽泛地用“重构”一词来指代修复缺陷、增加功能、修改设计——基本上成了对代码进行任何修改的同义词。这种对术语含义的普遍稀释是很不幸的。修改本身并不是一种美德，但有目的的修改，加上一点点严格的纪律，就可以成为关键的策略，从而通过维护稳步提高程序质量，并防止我们再熟悉不过的软件熵增的死亡螺旋。

24.3 特定的重构

本节展示了一个重构目录，其中许多是对《*Refactoring*》（重构：改善既有代码的设计）（Fowler 1999）一书的详尽描述的总结。但是，我不打算使这个目录详尽无遗。从某种意义上说，本书每一个同时显示了“坏代码（拙劣代码、糟糕代码）”和“好代码（良好的代码）”的例子都可供你参考一种重构方案。为节省篇幅，我将重点放在我个人觉得最有用的重构上。

数据级重构

以下重构方法旨在改善变量和其他类型的数据的使用。

用具名常量替换神秘数字（magic number） 如使用了像 3.14 这样的一个数值或字符串直接量，就用具名常量（如 PI）替换它。

用更清晰或更有信息量的名字重命名变量 如变量名不清不楚，把它改成一个更好的名字。当然，同样的建议也适用于常量、类和子程序的重命名。

使表达式内联（inline） 若将表达式的结果赋给一个中间变量，何不直接将变量替换为表达式本身？

用子程序替代表达式 为避免在代码中重复表达式，用子程序替代表达式。

引入中间变量 将表达式赋给中间变量，后者的名称概括了表达式的目的。

将一个多用途的变量转换为多个单用途的变量 如某个变量被用于一个以上的目的（对，说的就是 i、j、temp 和 x--这些家伙），就为每种用途创建单独的变量，为每个变量都指定一个更具体的名称。

局部的用途的就用局部变量，而不要用参数 如某个只供输入的子程序参数作为局部变量使用，就创建一个局部变量并改用它。

将数据基元（data primitive）转换为类 如数据基元需要额外的行为（包括更严格的类型检查）或额外的数据，就将数据转换为对象并添加你需要的行为。这一点适用于 Money 和 Temperature 等简单数值类型。也适用于 Color，Shape，Country 或 OutputType 等枚举类型。

将一组类型代码（type codes）转换为类或枚举 在一些较老的程序中，经常可以看到这样的关联：

```
const int SCREEN = 0;
const int PRINTER = 1;
const int FILE = 2;
```

不是定义独立的常量，而是创建一个类，这样就可获得更严格类型检查的好处，并在需要时为 OutputType 提供更丰富的语义。有的时候，创建枚举是创建类的一个好的替代方案。

将一组类型代码转换为带有子类的类 如果与不同类型相关的不同元素可能有不同的行为，考虑为类型创建一个基类，并为每个类型代码创建子类。对于 OutputType 基类，可考虑创建像 Screen、Printer 和 File 这样的子类

将数组改为对象 如果要使用一个数组，其中不同的元素是不同的类型，就创建一个对象，将之前的每个数组元素改为字段。

封装集合 如类返回集合，同时存在多个集合实例会造成同步困难。可考虑让类返回一个只读集合，并提供在集合中增删元素的子程序。

用数据类替代传统记录 创建一个类来包含记录的成员。创建类有利于集中错误检查、持久化和其他涉及记录的操作。

语句级重构

以下重构方法旨在改善单个语句的使用。

分解布尔表达式 通过引入命名良好的中间变量来简化布尔表达式，帮助澄清表达式的含义。

将复杂布尔表达式移入一个命名良好的布尔函数 如表达式足够复杂，这样重构可改善可读性。如表达式在多个地方使用，重构后就不需要进行并行修改，并减少了使用该表达式时出错的机率。

合并条件语句不同部分的重复片段 如果在 else 块的末尾重复了与 if 块末尾相同的几行代码，将这些行移至整个 if- then-else 块之后。

使用 break 或 return 替代循环控制变量 如循环内有一个用来控制循环的变量（例如 done），就改为使用 break 或 return 退出循环。

知道答案后立即返回，而不是在嵌套 if-then-else 语句中赋一个返回值 在知道返回值后立即退出子程序，代码通常更容易阅读，而且更不容易出错。相反，如果设置一个返回值，然后通过大量逻辑展开（unwind），则可能更难理解。

用多态替代条件语句(尤其是重复的 case 语句) 过去许多在结构化程序中包含在 case 语句中的逻辑，现在可以被转移到继承层次结构中，并通过多态的子程序调用来完成。

创建和使用空对象，而不是测试空值 有的时候，一个空对象可以关联泛型行为或数据，就好像将一个名字不详的居民称为“住户”。在这种情况下，考虑将处理空值的责任从客户代码转移到类中。换言之，让 Customer 类直接将未知居民定义为“住户”，而不是让 Customer 的客户代码反复测试客户的名字是否已知并在未知时用“住户”代替。

子程序级重构

以下重构方法旨在改善单独子程序一级的代码。

提取子程序/提取方法 从子程序中移除内联代码，将其提取为自己的子程序。

内联子程序的代码 如子程序的主体非常简单，而且一眼就能能看懂，就在使用该子程序的地方内联其代码。

将长的子程序转换为类 如子程序太长，有时可将其转换为类，并进一步将之前的子程序分解成多个子程序，从而改善可读性。

用简单算法代替复杂算法 用更简单的算法代替复杂的算法。

增加参数 如子程序需要它的调用者提供更多信息，就增加参数来提供这样的信息。

删除参数 如子程序不再使用一个参数，请删除它。

将查询操作与修改操作分开 查询操作一般不会改变对象的状态。如果像 GetTotals()这样的操作改变了对象的状态，请将查询功能和改变状态的功能分开，并提供两个单独的子程序。

通过参数化合并类似的子程序 两个类似的子程序可能只是内部使用的常量值有所不同。将其合并为一个，并将值作为参数传入。

分解行为依赖于传入参数的子程序 如子程序根据一个输入参数的值执行不同的代码，可考虑将其分解成可单独调用的独立子程序，避免传递那个特定的输入参数。

传递整个对象而不是特定的字段 如果从同一个对象中传递了几个值到一个子程序，考虑修改子程序的接口，改为获取整个对象。

传递特定的字段而不是整个对象 如果创建一个对象只是为了把它传给一个子程序，考虑修改子程序，使它获取特定的字段而不是整个对象。

封装向下转型(downcasting) 如子程序返回一个对象，通常应返回它所知道的最具体的对象类型。返回迭代器、集合、集合元素等的子程序尤其要注意这一点。

类实现重构

以下重构方法在类的一级加以改善。

将值对象修改为引用对象 如果发现自己需要创建和维护大型或复杂对象的大量拷贝，请修改这些对象的用法，使其只存在一个主拷贝（值对象），其余代码使用对该对象的引用（引用对象）。

将引用对象修改为值对象 如果发现对小的或简单的对象进行了大量引用，并因此要进行大量内务处理（housekeeping），请修改这些对象的用法，使所有对象都是值对象。

用数据初始化替代虚函数（virtual routines） 如一组子类只因其返回的常量值而变，就不要在派生类中覆盖（override）成员函数。相反，让派生类用恰当的常量值初始化类，再在基类中用泛型代码处理这些值。

改变成员函数或数据的位置 考虑对继承层次结构做出几个常规性的修改（通常是为了消除派生类中的重复）：

- 使子程序上移至它的超类。
- 使字段上移至它的超类。
- 使构造函数的主体上移至它的超类。

以下修改则支持派生类中的特化（specialization）：

- 使子程序下移至它的派生类：
- 使字段下移至它的派生类。
- 使构造函数的主体下移至它的派生类。

将特化代码提取到一个子类中 如果类包含的一些代码只被其实例的一个子集使用，就将这些特化的代码移到它自己的子类中。

将相似代码合并到超类中 如果两个子类有相似的代码，就合并这些代码，并移到超类中。

类接口重构

以下重构方法有助于获得更好的类接口。

将子程序移到另一个类中 在目标类中新建一个子程序，将子程序主体从源类移到目标类。然后就可以从旧的子程序中调用新的。

将一个类转换为两个 如某个类有两个或更多不同的职责范围，将该类拆分成多个类，每个都担负一个明确的职责。

淘汰类 如某个类没什么作用，将它的代码移到其他更有内聚性的类中，然后淘汰这个类。

隐藏委托 有时类 A 会调用类 B 和类 C，而实际上类 A 只应调用在 B，再由类 B 调用类 C，问问自己 A 和 B 交互的正确抽象是什么。如 B 应负责调用 C，就让 B 调用 C。

去掉中间人 如果类 A 调用类 B，类 B 调用类 C，有时让类 A 直接调用类 C 效果更好。是否应该委托给类 B 的问题取决于什么能最好地保持类 B 接口的完整性。

用委托代替继承 如某个类需使用另一个类，但又希望对其接口有更多的控制，就让超类成为原来子类的字段，然后公开一组子程序来提供一个内聚性的抽象。

用继承代替委托 如某个类公开了一个委托类（成员类）的所有公共子程序，就从委托类继承，而不要只是使用该类。

引入外来的子程序 如某个类需要一个额外的子程序，而你不能修改这个类来提供它，可在客户类中创建新建一个子程序来提供该功能。

引入扩展类 如某个类需要几个额外的子程序，而你不能修改这个类，可新建一个类来合并不可修改的那个类的功能与额外的功能。为此，要么创建原始类的子类并添加新的子程序，要么包装原始类并公开你需要的子程序。

封装公开的成员变量 如成员数据是公共的，将成员数据修改成私有，改为通过一个子程序来公开成员数据的值。

删除不可修改的字段的 Set()子程序 如字段在对象创建时赋值，之后便不可修改，就在对象的构造函数中初始化它，而不是提供一个误导性的 Set()子程序。

隐藏不打算在类外使用的子程序 如类的接口没有子程序会更有内聚性，就隐藏该子程序。

封装未使用的子程序 如发现自己经常只使用类接口的一部分，就为该类创建一个新接口，只公开那些必要的子程序。确保新接口提供一个内聚性的抽象。

合并实现非常相似的超类和子类 如子类没有提供太多的特化，就把它合并到超类中。

系统级重构

以下重构方法旨在改善系统一级的代码。

为你无法控制的数据创建一个明确的引用源 有的时候，你的一些数据由系统维护，无法从其他需要知道该数据的对象那里方便或一致地访问。一个常见的例子是在 GUI 控件吕维护

的数据。这个时候，可创建一个类来镜像 GUI 控件中的数据，再让 GUI 控件和其他代码都将该类作为数据的权威来源。

将单向类关联改为双向类关联 如果有两个类需要使用对方的功能，但只有一个可以知道另一个的情况，就修改这些类，让它们都知道对方的情况。

将双向类关联改为单向类关联 如果有两个类知道对方的功能，但只有一个类真正需要知道对方，就修改这些类，让其中一个知道另一个，反之则不知道。

提供工厂方法而不是简单构造函数 需要根据类型代码来创建对象时，或者要处理引用对象而不是值对象时，使用工厂方法（子程序）。

用异常代替错误代码或相反 根据你的错误处理策略，确保代码使用的是标准方法。

检查清单：重构总结

数据级重构

- 用具名常量替换神秘数字（magic number）。
- 用更清晰或更有信息量的名字重命名变量。
- 使表达式内联（inline） 。
- 用子程序替代表达式。
- 引入中间变量。
- 将一个多用途的变量转换为多个单用途的变量。
- 局部的用途的就用局部变量，而不要用参数。
- 将数据基元（data primitive）转换为类。
- 将一组类型代码（type codes）转换为类或枚举。
- 将一组类型代码转换为带有子类的类。
- 将数组改为对象。
- 封装集合。
- 用数据类替代传统记录。

语句级重构

- 分解布尔表达式。
- 将复杂布尔表达式移入一个命名良好的布尔函数。
- 合并条件语句不同部分的重复片段。
- 使用 break 或 return 替代循环控制变量。
- 知道答案后立即返回，而不是在嵌套 if-then-else 语句中赋一个返回值。

- 用多态替代条件语句(尤其是重复的 case 语句)。
- 创建和使用空对象，而不是测试空值。

子程序级重构

- 提取子程序。
- 内联子程序的代码。
- 将长的子程序转换为类。
- 用简单算法代替复杂算法。
- 增加参数。
- 删除参数。
- 将查询操作与修改操作分开。
- 通过参数化合并类似的子程序。
- 分解行为依赖于传入参数的子程序。
- 传递整个对象而不是特定的字段。
- 传递特定的字段而不是整个对象。
- 封装向下转型(downcasting)。

类实现重构

- 将值对象修改为引用对象。
- 将引用对象修改为值对象。
- 用数据初始化替代虚函数（virtual routines）。
- 改变成员函数或数据的位置。
- 将特化代码提取到一个子类中。
- 将相似代码合并到超类中。

类接口重构

- 将子程序移到另一个类中。
- 将一个类转换为两个。
- 淘汰类。
- 隐藏委托。
- 去掉中间人。
- 用委托代替继承。
- 用继承代替委托。

- 引入外来的子程序。
- 引入扩展类。
- 封装公开的成员变量。
- 删除不可修改的字段的 Set()子程序。
- 隐藏不打算在类外使用的子程序。
- 封装未使用的子程序。
- 合并实现非常相似的超类和子类。

系统级重构

- 为你无法控制的数据创建一个明确的引用源。
- 将单向类关联改为双向类关联。
- 将双向类关联改为单向类关联。
- 提供工厂方法而不是简单构造函数。
- 用异常代替错误代码或相反。

24.4 安全地重构

> “与打开水槽更换其中一个垫圈相比，打开一个正常工作的系统更像是打开人的大脑并更换其中的神经。若将软件维护称为“软件脑部外科手术”，维护起来会不更轻松一些？”
>
> ——Gerald Weinberg

重构是改善代码质量的一种强大技术。但和所有强大的工具一样，若使用不当，重构造成的伤害会大于它带来的好处。遵循几条简单的指导原则以避免走入重构的误区。

保存开始时的代码 重构之前，首先确保你能回到开始时的代码。在你的版本控制系统中保存一个版本，或将正确的文件复制到一个备份目录。

保持小幅重构 有的重构在规模上比其他重构要大，而且到底什么是“一次重构”可能有点模糊。每次重构都保持小的幅度，这样就能充分理解更改所造成的全部影响。《*Refactoring*》（重构：改善既有代码的设计）（Fowler 1999）一书所详尽描述的重构提供了许多具体如何做的好例子。

一次一个重构 有些重构比其他复杂。除了最简单的重构，其他重构都一次都只进行一个。完成一个重构后，就重新编译和重新测试，再进行下一个。

列出步骤清单 伪代码编程过程的一个自然延伸是列出从 A 点到 B 点的重构清单，列出清单有助于理解每个变更的前因后果。

做一个停车场 一次重构中途，有时会发现需要进行另一次重构。而在那次重构的中途，会发现有必要进行第三次重构。对于那些不需要立即进行的修改，可以做一个“停车场”，列出想在某个时候进行但现在暂时不需要的修改。

经常做检查点　重构期间，很容易发现代码突然开始走偏了。除了保存开始时的代码，在重构过程中的各个步骤都保存检查点。这样，如果代码进入了死胡同，可以还原之前能正常工作的程序。

利用编译器警告　一些小错误很容易被编译器漏掉。将你的编译器设置为最挑剔的警告级别，以便一开始就发现错误。

重新测试　对修改过的代码的审查应通过重新测试予以补充。当然，首先要有一套好的测试用例。回归测试和其他测试主题在第 22 章进行了更详细的描述。

添加测试用例　除了用旧的测试进行重测，还要添加新的单元测试来锤炼新代码。删除任何因重构而变得过时的测试用例。

关联参考　参见第 21 章详细了解代码评审。

审查修改　最开始的代码审查就已经很重要了，在之后的修改过程中更重要。Ed Yourdon 的报告称，程序员在第一次尝试修改时，通常有 50%以上的机会犯错（Yourdon 1986b）。有趣的是，如果程序员处理的代码量较大，而不是短短几行，做出正确修改的机率反而会提高，如图 24-1 所示。具体地说，随着修改行数从 1 行增至 5 行，做出错误修改的机率会增加。在此之后，机率反而下降。

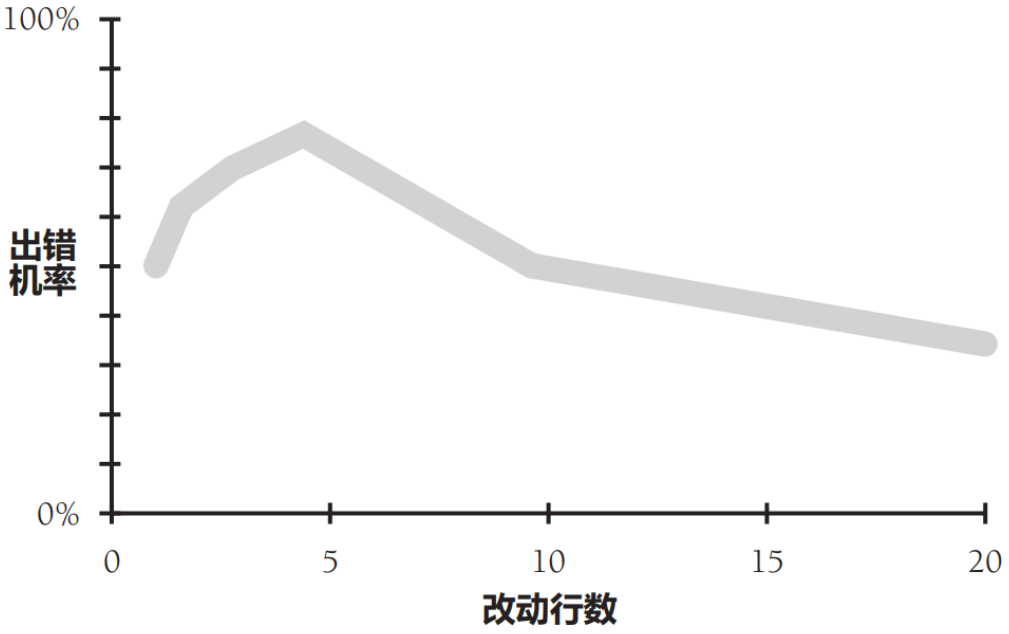
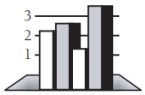


图 24-1　小幅修改比大幅修改更易出错(Weinberg 1983)

程序员对小的修改往往很随意。他们不进行桌面检查（desk-check），也不请人审查，有时甚至不运行代码来验证修复是否正确。



HARD DATA 道理很简单：将任何简单的修改当作复杂的修改来对待。某组织在对单行修改进行审查后发现，其错误率从审查前的 55%下降至审查后的 2%（Freedman and Weinberg 1982）。某电信公司在对修改的代码进行审查后，正确率从 86%上升至 99.6%（Perrott 2004）。

基于重构风险等级来调整方法 有的重构比其他的风险更大。像“用具名常量替换神秘数字”这样的重构相对没什么风险。而涉及类或子程序接口的变化、数据库模式的变化或布尔测试的变化等的重构往往风险更大。如果是较容易的重构，可考虑简化重构过程，一次进行多个重构，并简单地重新测试，无需走一遍正式审查流程。

若重构风险较大，则需谨慎行事。一次只进行一个重构。除了正常的编译器检查和单元测试，还要让其他人来审查重构，或通过结对编程来重构。

不宜重构的情况

重构是一种强大的技术，但并非万能，而且存在一些滥用的可能性。

> “不要只完成部分功能，并指望通过重构予以补全。”
>
> —John Manzo

不要将重构作为编码和修复的幌子 重构最糟糕的问题在于它可能被滥用。程序员有时会说自己在重构，真正做的却是调整代码，希望找到一种方法让它跑起来。重构是指对本来就能正常工作的、不影响程序行为的代码进行修改。对损坏的代码进行调整并不是在重构，而是在 hacking。

> “大规模重构是灾难的根源。”
>
> —Kent Beck

避免重构而不是重写 有的时候，代码不需要小的改动——它需要被扔掉，以便可以重新开始。如发现自己需要进行大幅重构，问问自己是否应从头设计和实现这部分代码。

24.5 重构策略

对任何特定程序有利的重构在数量基本上是无限的。和其他编程活动一样，重构受制于收益递减法则，80/20 规则同样适用。将时间花在 20%的重构上，这些重构能提供 80%的好处。在决定哪些重构是最重要的时候，请考虑以下指导原则：

添加子程序时重构 添加子程序时，检查相关的子程序是否组织良好。如果不是，就对它们进行重构。

添加类时重构 新类的添加往往会使现有代码的问题凸显出来。利用这个机会，重构与所添加的类密切相关的其他类。

修复缺陷时重构　利用通过修复缺陷所获得的理解来改进其他可能容易出现类似缺陷的代码。

关联参考　参见 22.4 节中的“哪些类中包含最多的错误”更多地了解容易出错的代码。

瞄准容易出错的模块　有的模块比其他模块更易出错、更脆弱。是否有一段你和你团队的其他人都害怕的代码？那可能就是一个容易出错的模块。虽然大多数人的自然倾向是避开这些具有挑战性的代码，但针对这些部分进行重构可能是更有效的策略之一（Jones 2000）。

瞄准高复杂度的模块　另一个办法是将重点放在复杂度最高的模块上(关于这些指标的细节，请参见第 19.6 节中的“如何度量复杂度”)。一项经典研究表明，当维护程序员将他们的改进工作集中在复杂度最高的模块上时，程序质量会有很大的提升（Henry and Kafura 1984）。

在维护环境中，改进你所接触的部分　从未改动过的代码无需重构。但只要亲自接触到了一段代码，就要确保你留下的比你最初发现的更好。

在干净的代码和丑陋的代码之间定义一个接口，再通过接口移动代码　“现实世界”往往比你想的更混乱。这种混乱可能来自复杂的业务规则、硬件接口或软件接口。老年系统的一个常见问题是必须一直运行写得不好的生产代码。

振兴老年生产系统的有效策略是指定一些代码在混乱的现实世界中保持原样，一些代码在理想的新世界中保持原样，再指定一些代码成为两者之间的接口。图 24-2 解释了这个思路。

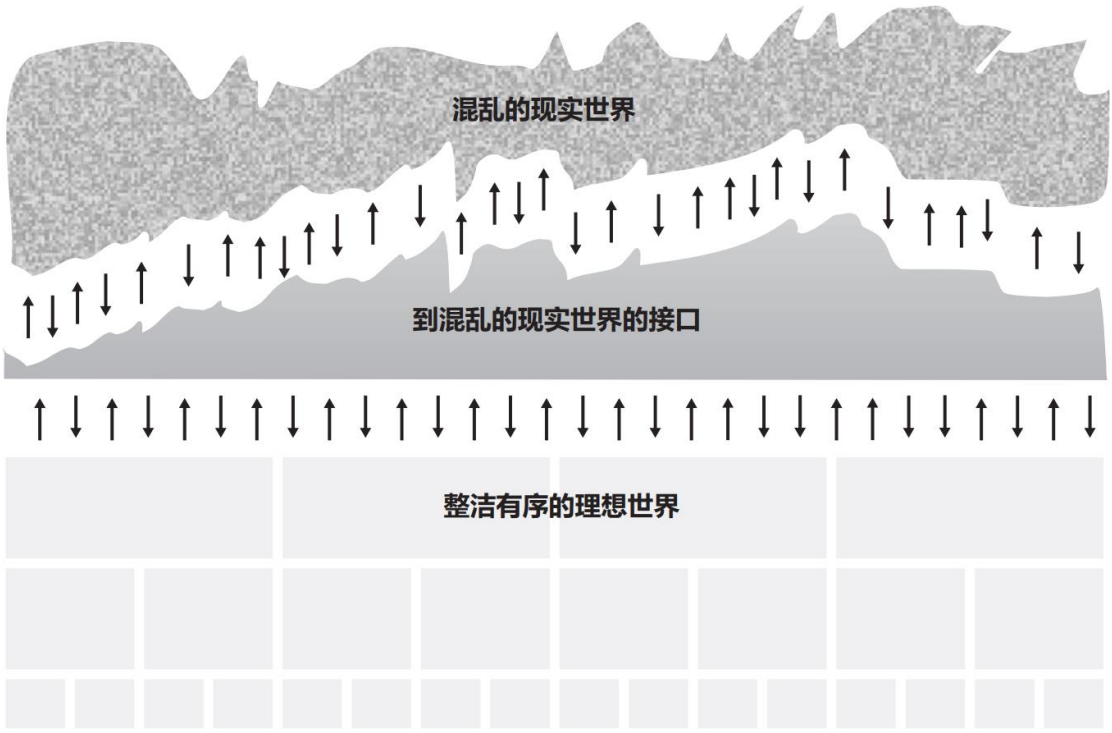


图 24-2　现实世界混乱，不意味着你的代码就必须混乱。将系统想象成理想代码、理想-现实接口以及混乱的现实世界的一种组合

拿到任何一个系统，都首先将代码从“现实世界接口”移动到一个更有条理的理想世界。拿到一个遗留系统时，可能整个系统都是那些写得很差的遗留代码。一个行之有效的策略是，

无论何时接触到一段混乱的代码，都必须使其符合当前的编码标准，指定明确的变量名称……等等——从而有效地将其转移到理想世界。随着时间的推移，这样可以迅速改善代码库，如图 24-3 所示。

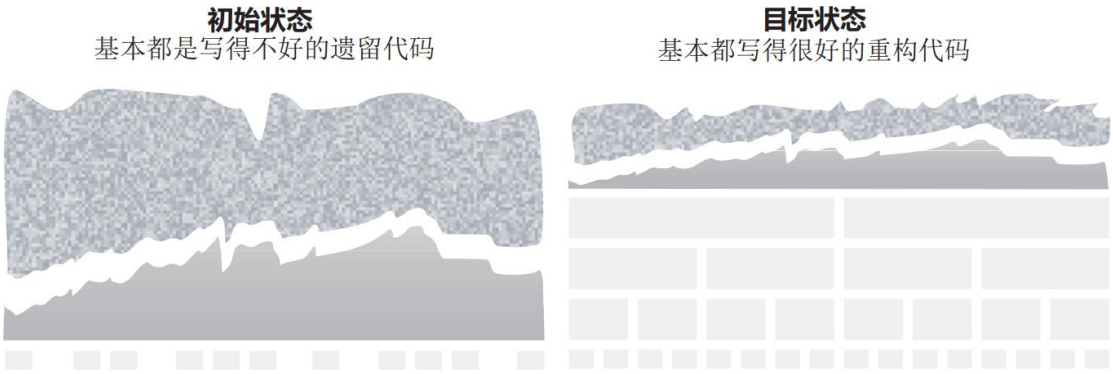


图 24-3　改进生产代码的一个策略是，一旦接触到写得不好的遗留代码就进行重构，将它转移到“混乱的现实世界的接口”的另一边

检查清单：安全地重构

- 每次改动都是一个语言改动策略的一部分吗？
- 重构前是否保存了最开始的代码？
- 是否保持每次重构的幅度都很小？
- 是否一次只进行一个重构？
- 是否列出了在重构过程中打算采取的步骤？
- 是否做了一个停车场，以便记住重构中途产生的想法？
- 每次重构后都重新测试了吗？
- 若改动很复杂，或者会影响关键任务，是否进行了代码审查？
- 是否考虑过特定重构的风险等级并相应地调整了你的方法？
- 这个修改是否改善而非降低了程序的内部质量？
- 是否避免了用重构作为编码和修复的幌子，或者作为不重写坏代码的借口？

更多资源

重构过程与修复缺陷的过程有很多共通之处，请参见 23.3 节更多地了解关于修复缺陷的信息。与重构相关的风险类似于与代码调优相关的风险，请参见 25.6 节更多地了解关于管理代码调优风险的信息。

《*Refactoring*》（重构：改善既有代码的设计）（Fowler 1999）一书是关于重构的权威指南。它详细讨论了本章概括的许多重构方法，还描述了本书没有涉及的其他重构方法。Fowler 通过大量示例代码说明了每一种重构的步骤。

要点回顾

- 无论是在最初的开发过程中还是在最初的发布之后，程序的变化都是生命中要接受的现实。
- 软件在变化过程中，要么改进，要么退化。软件进化的基本规则是：进化应该提高程序的内部质量。软件进化的基本规则是，内部质量应随着代码的进化而提高。
- 成功重构的一个关键是学会注意许多表明需要重构的警告信号（或称臭味）。
- 成功重构的另一个关键是掌握多种特定的重构方法。
- 最后一个成功的关键是要有一个安全重构策略。有的重构方法比其他方法更好。
- 为了在一开始就把事情做好，开发过程中的重构是你改进程序的最佳机会。在开发过程中要利用好这些机会！

第 25 章　代码调优策略

内容

- 25.1　性能概述
- 25.2　代码调优简介
- 25.3　臃肿和糖浆
- 25.4　度量
- 25.5　迭代
- 25.6　代码调优方法总结

相关章节

- 代码调优技术：第 26 章
- 软件架构：第 3.5 节

本章讨论性能调优（performance tuning）问题，这历来就是一个有争议的话题。在 20 世纪 60 年代，计算机资源受到严重限制，效率是首要考虑的问题。随着计算机从 70 年代开始变得越来越强大，程序员们开始意识到，他们对性能的关注对代码可读性和可维护性的危害有多大，代码调优就不再那么受关注了。80 年代，随着微型计算机的兴起，因为性能重新受到限制，效率问题再次被推上风口浪尖。但在随后的整个 90 年代，对效率问题的关注又逐渐减弱。在 2000 年代，在手机和 PDA 等移动设备上运行的嵌入式软件的内存限制以及解释型代码的执行时间再次使效率成为焦点。

可从两个层面解决性能问题：战略和战术。本章讨论战略性能问题：什么是性能，它有多重要，以及提高性能的一般方法。如果你已经很好地掌握了性能策略，并在寻找能提高性能的具体代码级技术，请继续阅读第 26 章“代码调优技术”。但在开始任何重要的性能调优之前，至少要略读一下本章，这样才不会在本应做其他更重要的事情的时候浪费时间去搞优化。

25.1　性能概述

代码调优是提升程序性能的一种方式。除了代码调优，通常还可以找到其他方法来提升性能，它们所需的时间更短，对代码的伤害更小。本节介绍了这些选项。

质量特性和性能

> 相较于因其他任何单一原因（包括单纯的愚蠢）而犯下的计算罪行相比，因效率之名（还不一定真的能实现“效率”）而犯下的计算罪行要多得多。
>
> ——W. A. Wulf

有些人透过玫瑰色的眼镜[8]看世界。像你我这样的程序员则倾向于透过代码色的眼镜看世界。我们认为，代码做得越好，客户和顾客就越喜欢我们的软件。

这个观点可能在现实的某个地方有一个邮寄地址，但不会有街道号码，也绝不会拥有任何房地产。用户对有形的程序特性（tangible program characteristics）比对代码质量更感兴趣。有的时候，用户对原始性能感兴趣，但只有当它影响到自己的工作时才会如此。用户往往对程序的吞吐量比对原始性能更感兴趣。按时交付软件、提供干净的用户界面以及避免当机时间往往更有意义。

用个例子说明一下。我每周至少用数码相机拍 50 张照片。为了把照片上传到电脑，相机附带的软件要求我一张一张地选择所有照片，在一次只显示 6 张照片的窗口中查看它们。上传 50 张图片是一个乏味的过程，需要点击几十次鼠标，而且要经历 6 个图片窗口。在忍受了几个月后，我买了一个内存卡读卡器，直接插入电脑。我的电脑认为它是一个磁盘驱动器。现在我可以用 Windows 资源管理器将图片复制到电脑上。以前需要点击几十次鼠标和大量的等待，现在只需要点击两次鼠标，按一下 Ctrl+A，然后拖放即可。我真的不在乎读卡器传输每个文件的时间是其他软件的一半还是两倍，因为我的吞吐量更大。无论读卡器的代码是快还是慢，它的性能都更好。


KEY POINT 性能与代码速度只有松散的关系。在代码速度上下了太大功夫，就没有精力在其他质量特性上下功夫。要警惕牺牲其他特性来使你的代码更快。你在速度方面的工作可能会损害整体性能，而不是提升它。

性能和代码调优

一旦选择效率优先，无论其重点是速度（speed）还是规模（size），在代码一级选择改善速度或规模之前，都应考虑几个选项。从以下每个角度考虑效率：

- 程序需求
- 程序设计
- 类和子程序设计
- 操作系统交互
- 代码编译
- 硬件
- 代码调优

[8] rose-colored glasses，一种积极、乐观看待生活的方式。——译注

程序需求

性能被说成是一种需求的时候远比它真的是一种需求的时候多。Barry Boehm 讲述了 TRW 的一个系统的故事，该系统最初需要亚秒级的响应时间。这个要求导致了一个高度复杂的设计和一亿美元的预估成本。进一步分析表明，用户在 90%的情况下会对 4 秒的响应时间感到满意。修改响应时间的要求后，整个系统的成本减少了约 7000 万美元。(Boehm 2000b)。

在投入时间解决一个性能问题之前，请确保正在解决的是一个需要解决的问题。

程序设计

关联参考　有关通过设计来提升程序性能的详情，请参阅本章最后的“更多资源”一节。

程序设计涉及针对单个程序进行设计时的主要思路，其中主要是将程序分解为类的方式。有的程序设计使实现一个高性能的系统变得十分困难。另一些则很轻松。

考虑现实世界中一个数据采集程序的例子，它的高层设计将吞吐量的测量作为一项关键产品属性。每一次测量都包括进行电气测量、校准数值、缩放数值以及将其从传感器数据单位（如毫伏）转换成工程数据单位（如摄氏度）的时间

在这个例子中，如果不首先解决高层设计中存在的风险，程序员就会发现自己需要对数学进行优化以便在软件中求值 13 阶多项式——即一个有 14 项的多项式，其中含有最高 13 次幂的自变量。相反，他们采用不同的硬件以及涉及几十个三阶多项式的一个高级设计来解决了问题。这种高层次的变更不可能通过代码调优来实现，而且任何数量的代码调优都不可能解决这个问题。这是一个必须在程序设计层面解决问题的例子。

关联参考　参见 20.2 节中的“设置目标”更多地了解程序员如何朝目标努力。

如果知道一个程序的规模和速度很重要，就从程序架构的设计入手，使之能合理地满足规模和速度目标。先设计一个面向性能的架构，再为各个子系统、功能和类设定资源目标。这能在以下几方面提供助益：

- 设定单独的资源目标，使得系统最终的性能可以预测。如果每个功能都能达到其资源目标，整个系统就能达到。可提早发现那些难以达到目标的子系统，并针对它们进行重新设计或代码调优。
- 仅仅是明确一下自己的目标，就能提高实现目标的可能性。当程序员知道自己的目标是什么时，他们就会朝目标努力；目标越明确，他们越容易达到目标。

- 可设定一些和效率没有直接关系，但从长远来看有利于提升效率的目标。效率往往最好在其他问题的背景下处理。例如，实现高度的可修改性可为实现效率目标提供一个比明确设定效率目标更好的基础。基于一个高度模块化、可修改的设计，可以很容易地将效率较低的部件换成效率较高的。

类和子程序设计

> 关联参考　有关数据类型和算法的详情，请参阅本章最后的“更多资源”一节。

类和子程序内部结构的设计是另一个为性能而设计的机会。在这一层面上发挥作用的一个性能关键是对数据类型和算法的选择，这通常会影响到程序的内存使用和执行速度。在这个层面上，你可以选择 quicksort 而不是 bubblesort，或者选择二分查找而不是线性查找。

操作系统交互

> 关联参考　参见第 26 章更多地了解如何处理缓慢或者臃肿的操作系统子程序。

如果程序要与外部文件、动态内存或输出设备一起工作，它就可能要和操作系统交互。如果性能不好，可能是由于操作系统的子程序很慢或很臃肿。你可能没有意识到程序正在与操作系统交互。有的时候，编译器会生成系统调用，或者你的库会发出你做梦都想不到的系统调用。这方面的问题稍后详述。

代码编译

好的编译器能将清晰的高级语言代码转化为优化的机器码。如果选对了编译器，可能不需要再考虑速度优化的问题。

第 26 章中报告的优化结果提供了许多编译器优化的例子，它们生成的代码比手工代码调优效率更高。

硬件

有的时候，提高程序性能的最便宜和最好的方法是购买新硬件。如果在全国范围内分发一个程序，供成千上万客户使用，购买新硬件自然不是一个现实的选择。但如果是为少数内部用户开发定制软件，硬件升级可能是最便宜的选择。它节省了初始性能工作的成本，节省了由性能工作引起的未来维护问题的成本，还能提高在该硬件上运行的其他所有程序的性能。

代码调优

代码调优是指对正确的代码进行修改，使其更高效地运行，这也是本章其余部分的主题。“调优”指的是小规模修改，它影响单个类，影响单个子程序，或者更常见的是影响几行代码。“调优”并不是指大规模设计改动或其他更高层次的性能改进手段。

从系统设计到代码调优的每一个层面都可以实现显著改进。Jon Bentley 引用了一个论点：在某些系统中，每个层面的改进都可以是成倍的（1982 年）。如果能在上述 6 个层面中的每一个上实现 10 倍改进，潜在的性能改进最终可以达到一百万倍。虽然这样的倍增改进要求程序在一个层面上的收益独立于其他层面上的收益——这非常罕见——但这种潜力是令人鼓舞的。

25.2 代码调优简介

代码调优的魅力何在？它不是提升性能最有效的方法——程序架构、类的设计和算法的选择往往能带来更显著的改进。它也不是提升性能最容易的方法——购买新硬件或使用优化更好的编译器更容易。它也不是提高性能最便宜的方法——最开始需要花费更多时间来手动调优代码，而且手动调优的代码以后更难维护。

代码调优之所以吸引人，有几个方面的原因。原因之一是，它似乎违背了自然规律。一个要花 20 微秒执行的程序，调整几行就能将执行速度降至 2 微秒，这带来了令人难以置信的成就感。

之所以吸引人，另一个原因是掌握编写高效代码的艺术是成为一名真正的程序员的成人礼。在网球运动中，你不会因为捡球的方式而得到任何赛点，但仍需学习正确的方法。不能随便俯身用手拿起它。如果做得好，你要用球拍击打它，直到它弹至腰部，然后接住。击打三次以上，甚至第一次没有弹起，就是严重的失败。虽然看起来无伤大雅，但捡球方式在网球文化中具有一定的重要性。同样地，除了你和其他程序员，通常没人关心你的代码有严谨。但在编程文化中，如果能写出微效率（对应前面的“微秒”）的代码，就表明你真的很“酷”。

代码调优的问题在于，高效的代码不一定是“更好”的代码。这是接着几个小节的主题。

帕累托法则

帕累托法则（Pareto Principle，也称为 80/20 法则、关键少数法则、八二法则）是指可用 20%的努力获得 80%的结果。该原则适用于编程以外的很多领域，但它绝对适用于程序优化。

KEY POINT Barry Boehm 的报告称，一个程序 20%的子程序消耗了 80%的执行时间（1987b）。在 Donald Knuth 的经典论文“An Empirical Study of Fortran Programs”（Fortran 程序实证研究）中，他发现一个程序不到 4%的部分经常占总运行时间的 50%以上（1971）。

Knuth 用一个行数分析器发现了这种令人惊讶的关系，其对优化的影响也很明显。应度量代码以找到热点，然后将资源用于优化用得最多的那百分之几。Knuth 分析了他的行数程序，发现它有一半的执行时间花在了两个循环上。他修改了几行代码，花不到一个小时的时间，分析器的速度就翻了一番。

Jon Bentley 描述了这样一个案例：一个 1000 行的程序有 80%的时间花在一个 5 行的平方根子程序上。通过将这个子程序的速度提高到三倍，他将程序的速度提高到一倍（1988）。人们也正是基于帕累托原则，才建议先用 Python 这样的解释型语言写大部分代码，再用 C 语言这样更快的编译语言重写热点。

Bentley 还报告了一个团队的案例，他们发现某操作系统一半的时间都花在了一个小循环上。他们用微代码重写了该循环，使其速度提高到 10 倍，但系统性能没有发生变化——他们重写的是系统的空闲（idle）循环！。

设计 ALGOL 语言（大多数现代语言的鼻祖和有史以来最具影响力的语言之一）的团队曾收到过以下忠告：“别因强求最好，而使好事难成”（The best is the enemy of the good，伏尔泰名言）。追求完美可能会妨碍完成。先完成再完善。需完善的部分通常很小。

一些无稽之谈

你所听到的关于代码调优的大部分内容都是错的，下面列出了常见的误解。

减少高级语言中的代码行数可提升所生成的机器码的速度或缩小其规模——错！ 许多程序员顽固地坚持这样的信念，即如果他们能将代码缩短为一、两行，那将是最高效的。考虑以下代码，它初始化 10 个元素的一个数组：

```
for i = 1 to 10
   a[ i ] = i
end for
```

和下面这 10 行做同样事情的代码相比，猜猜哪个更快？

```
a[ 1 ] = 1
a[ 2 ] = 2
a[ 3 ] = 3
a[ 4 ] = 4
a[ 5 ] = 5
a[ 6 ] = 6
a[ 7 ] = 7
a[ 8 ] = 8
a[ 9 ] = 9
a[ 10 ] = 10
```

如遵循“行数越少越快”的老教条，会认为第一段代码更快。但在 Microsoft Visual Basic 和 Java 中的测试表明，后者至少比前者快 60%。下面是统计数字。

语言	for 循环时间	直接赋值时间	节省时间	性能比
Visual Basic	8.47	3.16	63%	2.5:1
Java	12.6	3.23	74%	4:1

注意：(1) 包括这个表格在内的本章所有表格的时间都以秒为单位，而且仅对每个表格中不同行的比较有意义。实际时间会因编译器、所用的编译器选项以及运行每个测试时的环境而异。(2) 基准测试结果通常来自几千到几百万次的代码片段执行，目的是在结果中抹平样本间的波动。(3) 未指明编译器的具体品牌和版本。不同品牌和不同版本的性能表现差异很大。(4) 对不同语言的结果进行比较并非总是有意义，因为不同语言的编译器并不总是提供有可比性的代码生成选项。(5) 解释型语言（PHP 和 Python）的结果通常基于不到其他语言 1%的测试。(6) 由于“直接时间”（straight time，本例就是“直接赋值时间”）和“代码调优时间”（code-tuned time，本例就是“for 循环时间”）进行了四舍五入，所以一些“节省时间”百分比可能无法从这些表格的数据中得以再现。

这当然不是说增加高级语言代码的行数总是能提高速度或减少规模。但它确实意味着，虽然用最少的代码写出的东西似乎有点“优雅”，但在高级语言的代码行数和程序的最终规模/速度之间，并不存在一种必然的联系。

某些操作可能比其他操作更快或者更小——错！ 性能不存在“可能”。必须经常度量性能，以了解你的更改是带来了帮助还是伤害。每次更改语言、编译器、编译器的版本、库、库的

版本、处理器、内存容量、你穿的衬衫的颜色（好吧，这个不算）等等，游戏规则都会发生改变。在一台机器上用一套工具可取，在另一台机器上用另一套工具就很容易不可取。

这个现象为不通过代码调优来提升性能提供了几个理由。如果希望程序具有可移植性，那么在一个环境中提升性能的技术在其他环境中可能会降低性能。如果更换了编译器或进行了升级，新的编译器可能自动按你当初手工调优的方式优化代码，所以你之前的工作就白费了。更糟的是，之前的代码调优可能会使更强大的编译器优化失效，这些优化是为直接的代码（straightforward code）设计的。

对代码进行调优，相当于你承诺了以后每次更改编译器品牌、编译器版本、库版本等时，都要对每个优化措施进行重新分析。如果不重新分析，在一个编译器或库的版本下提升性能的优化措施，在 build 环境发生变化时，很可能会降低性能。

> “97%的情况都应忘却琐碎的效率提升：草率的优化乃万恶之源。”
>
> ——Donald Knuth

即刻优化——错！ 一种理论认为，如果在写每个子程序时都努力编写最快、最小的代码，最终整个程序就会又快又小。这种方法会造成一叶障目的情况，即程序员会因为太忙于微观优化而忽略重要的全局优化。以下是一边写代码一边优化的主要问题：

- 几乎不可能在程序完全跑起来之前确定性能瓶颈。程序员非常不擅长猜测哪 4%的代码占了 50%的执行时间，所以即刻优化的程序员平均会花费 96%的时间来优化不需要优化的代码，几乎没时间来优化真正重要的 4%。
- 即使在极少数情况下开发人员正确识别出了瓶颈，他们也会过犹不及地对待这个瓶颈，以至于顾此失彼。所以，最终结果还是性能下降。在系统完成后再进行优化，可识别出每个问题区域及其相对重要性，从而有效分配优化时间。
- 在初始开发过程中专注于优化，有损于其他程序目标的达成。开发人员沉浸于算法分析和搞一些莫名其妙的辩论，最终对用户来说其实并没有多大价值。正确性、信息隐藏和可读性等问题反而成了次要目标——即使相比这些问题，性能在后期更容易改进。事后的性能改进工作通常只影响不到 5% 的程序代码。你愿意回去对 5% 的代码进行性能改进的工作，还是一开始就搞定 100% 的可读性问题？

简而言之，过早优化的主要缺点在于视角有限。它的受害者包括最终的代码速度、比代码速度更重要的性能表现、程序质量以及最终的软件用户。若将实现最简单的程序所节省的开发时间用于优化一个能跑起来的程序，其结果总是比无差别优化所开发的程序运行得更快（Stevens 1981）。

偶尔，事后优化并不足以达成性能目标，这时必须对已完成的代码进行重大修改。但在这种情况下，小型的、局部的优化无论如何也不会提供所需的收益。在这种情况下，问题不是代码质量不够好，而是软件架构不够好。

如必须在程序完成之前进行优化，就在自己的过程中建立多方面的视角来降低风险。一个办法是设定功能的规模和速度目标，然后在实现过程中进行优化以满足这些目标。在设计规范中建立这样的目标，目的是在搞清楚当前这个“木”有多大的时候，先把自己的一只眼睛对准整个森林。

深入阅读　关于其他更多富于哲理的逸闻趣事，请参阅 Gerald Weinberg 的著作《*Psychology of Computer Programming*》（计算机编程心理学，1998）。

快的程序和正确的程序同等重要，错！　程序需要快而小才能正确，这几乎不可能成立。Gerald Weinberg 讲了这样的一个故事：一个程序员被空运到底特律，帮助调试一个有问题的程序。这个程序员和开发团队一起工作，几天后得出的结论是，情况已无望。

在回家的飞机上，他思考了一下情况，意识到问题出在哪里。在飞行结束时，他有了一个新代码的大纲。测试了几天代码，正准备返回底特律时，他收到一封电报，说这个项目已经被取消了，因为这个程序不可能写出来。他还是回到了底特律，并说服高管们这个项目可以完成。

然后，他不得不说服项目的原始程序员。他们听了他的介绍，在他说完后，旧系统的创建者问："那你的程序需要多长时间？"

"这不尽相同，但每个输入要十秒钟算完吧。"

"啊哈！但我们的程序每个输入一秒钟就能算完。"老手向后靠了靠，满意地认为他把这个新来的家伙搞得晕头转向。其他程序员似乎也同意，但新程序员并没有被吓倒。

"是的，但你的程序不起作用。如果我的程序也不需要起作用，你信不信可以让它瞬间完成？"

对于某些类别的项目，速度或规模是一个主要关注点。但此类项目是少数，比大多数人想象的少得多，而且一直在变少。对于这些项目，性能风险必须通过前期设计来解决。对于其他项目，早期优化对整个软件质量（包括性能）构成了重大威胁。

何时调优

Jackson 优化规则：规则 1：不要进行优化。规则 2 （仅针对专家）：还是不要进行优化，也就是说，在你还没有绝对清晰的未优化方案之前，请不要进行优化。

——M. A. Jackson

使用高质量的设计。使程序正确。使其模块化并易于修改，以便日后处理。当它完成并正确时，检查其性能。如程序很笨重，把它变得又快又小。若非必要，否则不优化。

我几年前参与了一个 C++项目，该项目生成图形输出来分析投资数据。在我的团队完成了第一个图表的工作后，测试报告显示，程序需要 45 分钟来绘制图表，这显然不可接受。我们召开了一次团队会议，讨论如何处理这个问题。其中一个开发人员变得很生气，并喊道："如果我们想有任何机会发布一个可接受的产品，现在就得开始用汇编重写整个代码库。"我表示反对——4%的代码可能占性能瓶颈的 50%或更多。最好是在项目结束时再解决这 4%的

问题。经过一番争吵之后，我们的经理指派我去做一些初始的性能工作（这其实正中我的下怀[9]）。

就像通常的情况一样，一天的工作发现了代码中的几个明显的瓶颈。少量的代码调整变化将绘图时间从 45 分钟减少到不到 30 秒。不到百分之一的代码占了 90%的运行时间。几个月后，当我们发布该软件时，又有几处代码调整的改动将绘图时间减少到 1 秒多一点。

和往常一样，通过一天的工作，我发现了代码中几个明显的瓶颈。少量的代码调优将绘图时间从 45 分钟缩短至不到 30 秒。不到百分之一的代码占了 90%的运行时间。几个月后，当我们发布该软件时，又有几处代码调优的改动将绘图时间缩短至 1 秒多一点。

编译器优化

现代编译器的优化可能比你预期的更强大。在前面的例子中，编译器对嵌套循环的优化做得和我以所谓更高效的风格重写的一样好。选择编译器时，要比较每种编译器对于你的程序的性能。每个编译器都有不同的优势和劣势，有的编译器比其他编译器更适合你的程序。

相较于一些难以理解的代码，编译器对于直接代码的优化更佳。如果你做一些“聪明”的事情，比如在循环索引上做手脚，编译器恐怕很难完成它的工作，程序就会受到影响。以第 31.5 节中的“每行只有一条语句”为例，直接的写法使编译器能成功优化，结果比类似的“难以理解”的代码快 11%。

一个好的优化编译器能使代码速度整体提升 40%甚至更多。下一章描述的许多技术只能产生 15%~30%的收益。为什么不直接写清晰的代码，让编译器来做这些工作呢？下面是一些测试结果，它们验证了优化器对插入-排序子程序的速度有多大的提高。

编程语言	编译器未优化执行时间	编译器优化后执行时间	节省时间	性能比
C++ 编译器 1	2.21	1.05	52%	2:1
C++ 编译器 2	2.78	1.15	59%	2.5:1
C++ 编译器 3	2.43	1.25	49%	2:1
C# 编译器	1.55	1.55	0%	1:1
Visual Basic	1.78	1.78	0%	1:1
Java VM1	2.77	2.77	0%	1:1
Java VM2	1.39	1.38	<1%	1:1
Java VM3	2.63	2.63	0%	1:1

子程序不同版本唯一的区别在于，首次编译时关闭编译器的优化选项，第二次则打开。显然，有的编译器优化得比其他编译器好。另外，在不开启优化的前提下，有的编译器一开始就比

[9] 原文是“Oh no! Please don't throw me into that briar patch!”，出自 Br'er（兄弟）兔的故事。兔子当时动弹不得，所以告诉狐狸自己不想被扔到 briar patch（荆棘丛）。狐狸恶狠狠地把他扔进了荆棘丛，他因此而逃脱。——译注

别的好。一些 Java 虚拟机（JVM）也明显比其他的好。必须检查自己的编译器、JVM 或同时检查两者，以衡量其效果。

25.3 臃肿和糖浆

代码调优时，需找到程序中像冬天的糖浆一样慢，体积却像哥斯拉一样大的部分。然后修改它们，使其像闪电一样快，同时又很轻盈，能躲在 RAM 中其他字节之间的缝隙里。始终都要对程序进行分析（profile），才能确定哪些部分是缓慢和臃肿的。但是，有的操作有长期的懒惰和臃肿历史，可从调查它们开始。

效率低下的常见根源

下面是效率低下的一些常见的根源：

输入/输出操作 效率低下的最主要原因之一是不必要的输入/输出（I/O）。如果可以选择在内存中处理文件，而不必通过磁盘、数据库或网络处理，那么除非空间紧张，就使用内存中的数据结构。

下面是对访问 100 个元素的内存数组中的随机元素和访问 100 个记录的磁盘文件中相同大小的随机元素的代码之间的性能比较。

编程语言	访问外部文件数据执行时间	访问内存数据执行时间	节省时间	性能比
C++	6.04	0.000	100%	n/a
C#	12.80	0.010	100%	1000:1

根据这些数据，访问内存的速度是访问外部文件的 1000 倍左右。事实上，就我使用的 C++ 编译器来说，内存访问因为太快，执行时间根本测不出来。

顺序访问的性能比较与此相似：

编程语言	访问外部文件数据执行时间	访问内存数据执行时间	节省时间	性能比
C++	3.29	0.021	99%	150:1
C#	2.60	0.030	99%	85:1
注意：顺序访问测试数据量是随机访问测试数据量的 13 倍，所以两种测试的结果不具备可比性				

如测试使用较慢的媒介进行外部访问（例如通过网络连接访问硬盘），差异会更大。在网络位置而非本地机器上进行类似的随机访问测试时，性能数据如下所示：

编程语言	访问本地文件执行时间	访问网络文件执行时间	节省时间
C++	6.04	6.64	-10%
C#	12.80	14.1	-10%

当然，结果会因网络速度、网络负载、本地机器到网络硬盘的距离、网络硬盘的速度与本地硬盘的速度的差异、当前月相（好吧，这个不算）以及其他因素而有很大的不同。

总的来说，内存访问效果显著，足以让你在设计程序对速度要求高的部分时，对是否需要 I/O 再三斟酌。

分页 导致操作系统交换内存页的操作要比只操作一个内存页慢得多。有的时候，一处简单的修改就会带来巨大的差异。在下个例子中，某程序员写了一个初始化循环，在使用 4K 页面的系统上造成了许多缺页错误：

Java示例：造成许多缺页错误的初始化循环

```
for ( column = 0; column < MAX_COLUMNS; column++ ) {
   for ( row = 0; row < MAX_ROWS; row++ ) {
      table[ row ][ column ] = BlankTableElement();
   }
}
```

这是一个格式很好的循环，有很好的变量名称，那么问题出在哪里？问题在于，table 的每个元素都长约 4000 字节。如果 table 有太多的行，程序每次访问不同的行时，操作系统将不得不交换内存页。这样的循环结构决定了每次数组访问都会在不同的行之间切换，这意味着每次数组访问都会导致分页到磁盘。

程序员以这种方式重新构造了该循环：

Java示例：造成较少缺页错误的初始化循环

```
for ( row = 0; row < MAX_ROWS; row++ ) {
   for ( column = 0; column < MAX_COLUMNS; column++ ) {
      table[ row ][ column ] = BlankTableElement();
   }
}
```

这段代码在每次切换行时仍会造成缺页错误，但它现在切换行的次数变成 MAX_ROWS 次，而不是 MAX_ROWS * MAX_COLUMNS 次。

具体的性能惩罚差异很大。在一台内存有限的机器上，我测到第二段代码比第一段快 1000 倍左右。在内存充足的机器上，我测得的差异减小到 2 倍，而且除非 MAX_ROWS 和 MAX_COLUMNS 的值非常大，否则根本显示不出区别。

系统调用 对系统子程序（系统例程）的调用往往很昂贵。它们经常涉及到上下文切换，即保存程序状态、恢复内核状态以及相反。系统例程包括对磁盘、键盘、屏幕、打印机或其他设备的输入/输出操作；内存管理例程；以及某些工具例程。如果性能是一个问题，调查你的系统调用有多昂贵。如发现它们代价太高，可考虑以下选项：

- 写自己的服务。有时只需一个系统例程所提供的一小部分功能，并且可基于较低级别的系统例程构建自己的程序。写自己的替代程序可提供更快、更小、更适合自己需要的东西。
- 避免去找系统。

- 与系统供应商合作，使调用更快。大多数供应商都有改进其产品的意愿，并且很乐意了解其系统中性能不佳的部分。(也许一开始看起来可能有点不高兴，但他们真的很感兴趣)。

25.2 节的“何时调优”描述的那个程序使用了从商业版 BaseTime 类派生的 AppTime 类（这些名字都进行了修改以避嫌）。AppTime 对象是这个程序最常用的对象，我们实例化了数以万计的 AppTime 对象。几个月后，我们发现 BaseTime 在其构造函数中将自己初始化为系统时间。但是，我们这个程序并不需要系统时间，所以无谓地生成了成千上万的系统级调用。简单地重写（override）了 BaseTime 的构造函数，将 time 字段初始化为 0 而不是系统时间，就带来了巨大的性能提升。提升的幅度和我们做的其他所有改动加起来一样大。

解释型语言　解释型语言往往存在严重的性能惩罚，因其必须在创建和执行机器码之前处理每条编程语言指令。在我为本章和第 26 章进行的性能基准测试中，我观察到了如表 25-1 所示的不同语言之间的大致性能关系。

表 25-1　编程语言的相对执行时间

编程语言	语言类型	相对于 C++ 语言的执行时间
C++	编译型	1:1
Visual Basic	编译型	1:1
C#	编译型	1:1
Java	字节码	1.5:1
PHP	解释型	>100:1
Python	解释型	>100:1

如你所见，C++、Visual Basic 和 C#都差不多。Java 接近，但比其他语言慢。PHP 和 Python 是解释型语言，它们的代码运行速度比 C++、Visual Basic、C#和 Java 的代码慢 100 倍甚至更多。必须谨慎看待这个表格中的数据。对于任何特定的代码，C++、Visual Basic、C#或 Java 的速度可能是其他语言的两倍或一半(第 26 章会用详细的例子说明)。

错误　性能问题的最后一个来源是代码中的错误。错误包括启用调试代码（例如将跟踪信息记录到文件中）、忘记释放内存、不正确地设计数据库表、轮询不存在的设备直至超时等等。

我做过一个版本 1.0 的应用程序，它的一个特定的操作比其他类似的操作慢得多。为了解释为什么这么慢，人们对这个项目提出了许多解释。我们还是发布了 1.0 版本，却没有完全理解为何这个特定的操作会如此慢。但在做 1.1 版本时，我发现这个操作所使用的数据库表没有索引！简单地对表进行索引，就能提高性能。仅仅为该表编制索引，就使某些操作的性能提高了 30 倍之多。为常用的表定义索引不是优化，而是一个好的编程实践。

常见操作的相对性能开销

虽然不能在不亲自测量的前提下就认定某些操作比其他操作更昂贵，但某些操作确实往往更昂贵。在自己的程序中寻找缓慢流动的糖浆时，使用表 25-2 来帮助自己对程序中的粘性部分进行一些初步的猜测。

表 25-2 常见操作的开销

		相对时间消耗	
操作	示例	C++ 语言	Java 语言
基准（整数赋值）	i=j	1	1
函数调用			
调用无参数函数	foo()	1	n/a
调用无参数私有成员函数	this.foo()	1	0.5
调用含 1 个参数的私有成员函数	this.foo(i)	1.5	0.5
调用含 2 个参数的私有成员函数	this.foo(i，j)	2	0.51
对象成员函数调用	bar.foo()	2	1
派生成员函数调用	derivedBar.foo()	2	2
多态成员函数调用	abstractBar.foo()	2.5	
对象引用			
1 层对象解引用	i=obj.num	1	1
2 层对象解引用	i=obj1.obj2.num	1	1
每个额外的解引用	i=obj1.obj2.obj3...	无法度量	无法度量
整数运算			
整数赋值（局部）	i=j	1	1
整数赋值（继承）	i=j	1	1
整数加	i=j+k	1	1
整数减	i=j-k	1	1
整数乘	i=j*k	1	1
整数除	i=j\k	5	1.5
浮点运算			
浮点赋值	x=y	1	1
浮点加	x=y+z	1	1
浮点减	x=y-z	1	1
浮点乘	x=y*z	1	1
浮点除	x=y/z	4	1
超越函数			
浮点方根	x=sqrt(y)	15	4
浮点正弦	x=sin(y)	25	20
浮点对数	x=log(y)	25	20
浮点指数	x=exp(y)	50	20

（续表）

		相对时间消耗	
操作	示例	C++ 语言	Java 语言
数组			
用常量下标访问整数数组	i=a[5]	1	1
用变量下标访问整数数组	i=a[j]	1	1
用常量下标访问二维整数数组	i=a[3,5]	1	1
用变量下标访问二维整数数组	i=a[j,k]	1	1
用常量下标访问浮点数组	x=z[5]	1	1
用整数变量下标访问浮点数组	x=z[j]	1	1
用常量下标访问二维浮点数组	x=z[3,5]	1	1
用整数变量下标访问二维浮点数组	x=z[j,k]	1	1
说明：表格中的度量值对以下影响因素高度敏感：本地计算机的环境配置、编译器的优化选项以及由特定编译器生成的代码。度量数据在 C++ 和 Java 语言之间并没有直接可比性			

自本书第一版以来，这些操作的相对性能已发生了很大变化。如果还是用 10 年前的性能观念来处理代码调优，可能需要更新自己的想法。

大多数常见操作的开销都差不多。成员函数调用、赋值、整型运算和浮点运算大致一致。超越函数（Transcendental Functions，即变量之间的关系不能用有限次加、减、乘、除、乘方、开方运算表示的函数，即“超出”代数函数范围的函数）非常昂贵。多态函数调用比其他类型函数调用的开销要大一些。

表 25-2 或你自己做的类似的表是解锁第 26 章描述的所有速度改进措施的钥匙。在每一种情况下，速度的提升都来自于用一个更便宜的操作替代一个昂贵的操作。第 26 章提供了如何做到这一点的例子。

25.4 度量

由于经常都是程序的一小部分耗费了不成比例的运行时间，所以要度量代码以找到热点。一旦找到热点并对其进行了优化，就再次度量代码以评估改进程度。性能的许多方面都是反直觉的。本章前面的例子中，10 行代码比一行代码明显更快、更小，这就是代码会让你大吃一惊的明证。

KEY POINT 经验对优化也没什么帮助。一个人的经验可能来自于旧的机器、语言或编译器。当这些东西中的任何一个发生变化时，所有经验都失效。除非亲自测一下，否则永远无法确定优化的效果。

我多年前写了一个程序，对一个矩阵中的元素进行求和。最初的代码是这样的：

C++示例：对矩阵元素求和的直接代码

```
sum = 0;
for ( row = 0; row < rowCount; row++ ) {
   for ( column = 0; column < columnCount; column++ ) {
      sum = sum + matrix[ row ][ column ];
   }
}
```

这段代码很简单，但矩阵求和程序的性能很关键，我知道所有的数组访问和循环测试都代价不菲。从计算机科学课可知，代码每次访问一个二维数组时，都会执行代价高昂的乘法和加法运算。对于一个 100 X 100 的矩阵，总共需执行 10000 次乘法和加法，还要加上循环本身的开销。我推断，通过改为使用指针记号法，我可以递增一个指针，用一万次相对便宜的递增操作取代一万次昂贵的乘法。我小心翼翼地用指针重写了代码，得到了以下结果：

深入阅读　Jon Bentley 报告了一个类似的经历，即转换为指针后反而使性能下降了约 10%。但在另一个环境中，同样的转换反而使性能提高了 50%以上。详情请参见“Software Exploratorium: Writing Efficient C Programs”一文（Bentley 1991）。

C++示例：试图对矩阵求和代码进行调优

```
sum = 0;
elementPointer = matrix;
lastElementPointer = matrix[ rowCount - 1 ][ columnCount - 1 ] + 1;
while ( elementPointer < lastElementPointer ) {
   sum = sum + *elementPointer++;
}
```

虽然这段代码不像第一段代码那样好读，尤其是对那些不是 C++专家的程序员来说，我还是对自己感到非常满意。我算了一下，对于一个 100 X 100 的矩阵，节省了 10000 次乘法和大量循环开销。我非常高兴，所以决定测一下速度到底有多大的提升，这是我当时不常做的事情。目的嘛，自然是夸一下自己喽（以定量的方式）。

“没有数据作为支撑，任何程序员都无法预测或分析性能瓶颈出现于何处。无论怎么猜，都会惊奇地发现事实与自己的想象背道而驰。”

——Joseph M. Newcomer

知道我发现了什么吗？没有任何改善。100 X 100 的矩阵没有。10 X 10 的矩阵也没有。任何大小的矩阵都没有。我非常失望，于是深入研究了编译器生成的汇编代码，看看为什么优化没起作用。令我吃惊的是，我并不是第一个需要遍历数组元素的程序员——编译器的优化器早就知道将数组访问转换为指针。这个教训让我深刻意识到，如果不亲自测一下性能，有时优化的唯一结果就是使代码变得更难读。如果不值得通过度量来确定效率变高了，就不值得为一次性能的赌博而牺牲可读性。

度量需精确

关联参考　参见第 30.3 节中的“代码调优”更多地了解代码分析或剖测器（profiling tools）。

性能度量需精确。用秒表或通过数“一只大象、两只大象、三只大象”来给程序计时并不精确[10]。在这个时候，分析工具（profiling tools）很有用，或者可以使用自己的系统时钟，也可以使用记录计算操作耗时的子程序。

无论使用别人的工具还是自己写代码来进行度量，都要确保度量的只是需调优的代码的执行时间。使用分配给你的程序的 CPU 时钟滴答数量，而不要使用一天当中的时间。否则，当系统从你的程序切换至另一个程序时，你的一个子程序将因为执行另一个程序所花的时间而受到惩罚。类似地，尽量将度量本身的开销和程序启动的开销算进去。这样，无论原始代码还是调优尝试，都不至于受到不公平的惩罚。

25.5 迭代

确定了性能瓶颈后，你可能会对代码调优所实现的性能提升幅度感到惊讶。很少能从单一的技术中得到 10 倍的改善，但可有效结合各种技术；所以要不断尝试，即使在找到一种有效的技术之后。

我曾写过数据加密标准（DES）的一个软件实现。实际上，我不只写了一次，而是写了大约 30 次。根据 DES 加密法，用 DES 加密的数据在没有密码的前提下无法解开。加密算法是如此复杂，以至于它似乎自己都被“加密”了。我为自己的 DES 实现制定了一个性能目标，即在一台原版 IBM PC 上，在 37 秒内加密一个 18K 的文件。我的第一个实现在 21 分 40 秒内完成，所以我有很长的路要走。

虽然大多数单独的优化都很小，累积起来就很可观了。从改进的百分比来判断，没有三项甚至四项优化能达到我的性能目标。但最后的组合是有效的。这个故事的寓意是，只要挖得足够深，终究可以取得一些令人惊讶的宝藏。

我在这个案例中所做的代码调优是我所做过的最积极的代码调优。同时，最终的代码也是我写过的最不可读、最不可维护的代码。初始算法就很复杂。高级语言转换后生成的代码几乎无法阅读。翻译成汇编程序后产生了一个 500 行的子程序，我都不敢看。一般来说，代码调优和代码质量之间的这种关系是成立的。以下表格显示了优化的历史：

关联参考　表格所列出的方法在第 26 章有详细描述。

优化	**基准时间**	**提升**
最初的实现——直接代码(未优化)	21:40	—
将位域(bit field)转换为数组	7:30	65%
展开最内层的 for 循环	6:00	20%

[10] 读出“one elephant（一只大象）”大约耗时 1 秒，这是日常生活中一种粗略的计时方法。——译注

移除最后的排列(permutation)	5:24	10%
合并两个变量	5:06	5%
用一个逻辑标识（logical identity）合并 DES 算法最初的两个步骤	4:30	12%
使两个变量共享相同的内存以减少内层循环的数据传递	3:36	20%
使两个变量共享相同的内存以减少外层循环的数据传递	3:09	13%
展开所有循环，使用字面量下标	1:36	49%
移除子程序调用，内联所有代码	0:45	53%
用汇编语言重写整个子程序	0:22	51%
最终结果	**0:22**	**98%**
注意：本表展示的稳步优化过程并不意味着但凡优化一下都会生效。我的一些尝试居然使运行时间翻了一番，这些我并没有列出。在我尝试的所有优化中，至少有三分之二没起作用。		

25.6 代码调优方法总结

考虑代码调优是否有助于提升程序性能时，请按以下步骤操作：

1. 使用良好设计的代码来开发软件，使其易于理解和修改。

2. 如果性能很差：

    a. 保存一个能正常工作的版本，以便能回到“最后已知良好状态”。

    b. 度量系统以找到热点。

    c. 确定性能低下是否来自于设计、数据类型或算法的不足，以及是否适合代码调优。如果不适合代码调优，返回步骤 1。

    d. 对步骤（c）中确定的瓶颈进行调优。

    e. 逐一度量每一项改进。

f. 如果一项改进没有提升代码性能，就恢复步骤（a）保存的代码。(通常，超过半数的性能调优只能产生微不足道的性能改进，甚至会降低性能）。

3. 从第 2 步开始重复。

更多资源

本节包含与常规性能改进有关的资源。要获取关于特定代码调优技术的更多资源，请参见第 26 章末尾的“更多资源”小节。

性能

Smith，Connie U.和 Lloyd G. Williams 的《*Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*》（软件性能工程）一书，Addison-Wesley, 2002。本书介绍了软件性能工程，这是一种在软件系统开发的各个阶段保证性能的方法。它大量使用了几种程序的例子和案例研究。包括对 Web 应用的具体建议，对可扩展性予以了特别关注。

Newcomer, Joseph M 的“Optimization: Your Worst Enemy”（优化：你最糟糕的敌人）一文，2000 年 5 月，网址是 www.flounder.com/optimization.htm。Newcomer 是经验丰富的系统程序员，他以图形方式详细描述了无效优化策略的各种陷阱。

算法和数据类型

Knuth, Donald 的《The *Art of Computer Programming, vol. 1, Fundamental Algorithms*》*, 3d ed*. Reading, MA: Addison-Wesley, 1997.

Knuth, Donald 的《*The Art of Computer Programming, vol. 2, Seminumerical Algorithms*》*, 3d ed*. Reading, MA: Addison-Wesley, 1997.

Knuth, Donald 的《*The Art of Computer Programming, vol. 3, Sorting and Searching*》*, 2d ed*. Reading, MA: Addison-Wesley, 1998.

这是算法系列的前三卷，这一系列最初打算出 7 卷。似乎这三卷已经获得了某种震撼效果。除了使用了通俗的语言，该书还使用数学符号或 MIX（针对虚构的 MIX 计算机而创建的汇编语言）来描述算法。该书涵盖了对海量主题的详尽描述。如果读者对某种算法有浓厚的兴趣，或许找不到比这更好的参考资料了。

Sedgewick, Robert 的《*Algorithms in Java*》，Parts 1-4，3d ed. Boston，MA : Addison-Wesley, 2002。本书包含 4 个部分，涵盖了对解决各种类型问题的最佳方法的研究，涉及的主题包括基本原理、排序、搜索、抽象数据类型实现和一些高级主题。Sedgewick 的《*Algorithms in Java*》，Part 5，3d ed.(2003)覆盖了图论算法。Sedgewick 的《Algorithms in C++》，Parts 1-4，3d ed.(1998)、《Algorithms in C++》，Part 5，3d ed.(2002)、《Algorithms in C》，Parts 1-4，3d ed.(1997)和《Algorithms in C》，Part 5，3d ed.(2001)均按类似方式组织。Sedgewick 的博士导师是 Knuth（高德纳）。

检查清单：代码调优策略

程序整体性能

- 考虑通过变更程序需求来提升性能了吗？
- 考虑通过修改程序设计来提升性能了吗？
- 考虑通过修改类的设计来提升性能了吗？
- 考虑通过避免程序与操作系统的交互来提升性能了吗？
- 考虑通过避免 I/O 来提升性能了吗？
- 考虑用编译型语言替代解释型语言来提升性能了吗？
- 考虑启用编译器优化选项来提升性能了吗？
- 考虑通过切换到不同的硬件设备来提升性能了吗？
- 代码调优是不是万不得已的最后选择？

代码调优方法

- 开始代码调优之前，程序是完全正确的吗?
- 在代码调优之前，度量过性能瓶颈了吗?
- 度量过每一次代码调优的效果了吗？
- 如果代码调优并没有带来预期的性能提升，撤消了所做的改变了吗?
- 是否尝试过针对每一个性能瓶颈进行多次修改以提升性能（换言之，迭代过吗）？
- 性能只是整体软件质量的一个方面，而且通常并不是最重要的。精心调优的代码只是整体性能的一个方面，而且通常并不是最重要的。相较于代码的效率，程序架构、详尽的设计以及数据结构和算法的选择通常对程序的执行速度和规模有更大影响。
- 定量度量是实现性能最大化的关键。它是找出能真正提高性能的地方的必要手段。另外，为了验证优化是提升了性能而不是降级，还需要再次进行定量度量。
- 大多数程序的大部分时间都花在一小部分代码上。除非亲自度量，否则不会知道是哪些代码。
- 通常需要多次迭代才能通过代码调优达到预期的性能提升。
- 最开始编码时，为性能工作做好准备的最佳方式就是编写易于理解和修改的清晰的代码。

第 26 章　代码调优技术

内容

- 26.1 逻辑
- 26.2 循环
- 26.3 数据变换
- 26.4 表达式
- 26.5 子程序
- 26.6 用低级语言重新编码
- 26.7 改得越多，越发没有大的改观

相关章节

- 代码调优策略：第 25 章
- 重构：第 24 章

在计算机编程的大部分历史中，代码调优一直是热门话题。所以，一旦决定需要提高性能，并想在代码层面上做到这一点（牢记第 25 章“代码调优策略”的警告），就会有一套丰富的技术供你选择。

本章重点是提升速度，包括一些使代码变小的技巧。性能通常同时指速度和规模，但规模的减小往往更多地来自对类和数据的重新设计，而不是来自代码调优。代码调优指的是小幅改变，而不是对设计的大幅改变。

本章很少有技术是普遍适用的，所以无法直接将示例代码复制到你的程序中。讨论的主要目的是演示可针对自己的情况改编的一些代码调优技术。

本章描述的代码调优修改表面上与第 24 章描述的重构相似，但重构是通过修改来改善程序的内部结构（Fowler 1999）。本章的修改可能更适合被称为“反重构”。这些修改远非“改善内部结构”，反而是通过使内部结构降级以换取性能上的提升。按照定义就是如此。如果这些修改没有使内部结构降级，我们就不说它们是优化。相反，我们就会默认使用它们，并认为它们是标准的编码实践。

> 关联参考　代码调优采用了启发式方法。有关启发式方法的更多信息，请参阅第 5.3 节。

有的书将代码调优技术作为“经验法则”（rules of thumb）来介绍，或引用研究结果来说明特定的调优会产生预期的效果。但正如你很快就会看到的那样，“经验法则”的概念对代码调优的适用性很差。唯一可靠的经验法则是在你的环境中度量每个调优的效果。所以，本章展示的是一个“可供尝试的东西”的目录，其中许多东西在你的环境中不会起作用，但其中一些确实能起到很好的作用。

26.1 逻辑

关联参考　要想进一步了解如何使用语句逻辑，请参阅本书第 14 章~第 19 章。

编程的大部分内容都涉及逻辑处理。本节介绍了如何处理逻辑表达式，使之对你有利。

知道答案后停止测试

例如以下语句：

```
if ( 5 < x ) and ( x < 10 ) then ...
```

一旦确定 x 不大于 5，就无需执行另一半测试。

关联参考　有关短路求值的更多信息，请参阅第 19.1 节中的“理解布尔表达式如何求值”。

有的语言提供了一种称为“短路求值”的表达式求值形式，这意味着编译器生成的代码一旦知道答案就会自动停止测试。短路求值是 C++的标操作符和 Java“条件”操作符的一部分。

如果语言没有提供对短路求值的原生支持，就不要使用 and 和 or，而是自己添加逻辑。为了引入短路求值，可以这样修改上述代码：

```
if ( 5 < x ) then
   if ( x < 10 ) then ...
```

知道答案之后不继续测试的原则在其他许多情况下也很好用。其中最常见的是搜索循环。如果要扫描一个输入数字的数组，在其中寻找一个负值，而且只需知道是否存在一个负值，那么一个方法是检查每个值，找到负值就设置一个 negativeInputFound 变量。下面是搜索循环的样子：

C++示例：知道答案后不停止

```
negativeInputFound = false;
for ( i = 0; i < count; i++ ) {
   if ( input[ i ] < 0 ) {
      negativeInputFound = true;
   }
}
```

更好的方法是一旦发现负值就停止扫描。以下任何一种方法都可以解决问题。

- 在 negativeInputFound = true 一行之后添加 break 语句。
- 如果语言不支持 break 语句，可以用 goto 语句来模拟，跳转到循环后的第一个语句。
- 将 for 循环改为 while 循环，除了检查循环计数器是否超过 count，还要检查 negativeInputFound。
- 将 for 循环改成 while 循环，在数组最后一个值项后面放一个哨兵值，然后直接在 while 测试中检查负值。循环终止后，检查发现的第一个值的位置是在数组中还是过了终点。本章后面会详细讨论哨兵值。

下面是在 C++和 Java 中使用 break 关键字后的性能数据：

编程语言	直接时间	代码调优后的时间	节省时间
C++	4.27	3.68	14%
Java	4.85	3.46	29%
注意：(1) 包括这个表格在内的本章所有表格的时间都以秒为单位，而且仅对每个表格中不同行的比较有意义。实际时间会因编译器、所用的编译器选项以及运行每个测试时的环境而异。(2) 基准测试结果通常来自几千到几百万次的代码片段执行，目的是在结果中抹平样本间的波动。(3) 未指明编译器的具体品牌和版本。不同品牌和不同版本的性能表现差异很大。(4) 对不同语言的结果进行比较并非总是有意义，因为不同语言的编译器并不总是提供有可比性的代码生成选项。(5) 解释型语言（PHP 和 Python）的结果通常基于不到其他语言 1%的测试。(6) 由于“直接时间”（straight time）和“代码调优后的时间”（code-tuned time）进行了四舍五入，所以一些“节省时间”百分比可能无法从这些表格的数据中得以再现。			

取决于具体有多少个值，以及你期望找到一个负值的频率，这个修改的影响有很大的不同。该测试假设平均有 100 个值，并假定 50%的时间会找到负值。

按频率调整测试顺序

测试顺序很重要，要使最快和最有可能为 true 的测试被首先执行。最常见的情况应该最先测试，如果出现了效率低下的情况，应该是花了太多时间处理不常见的情况。该原则适用于 case 语句和 if-then-else 语句链。

以下 Select-Case 语句响应字处理软件中的键盘输入：

Visual Basic示例：顺序排列不佳的逻辑测试

```
Select inputCharacter
   Case "+", "="
      ProcessMathSymbol( inputCharacter )
   Case "0" To "9"
      ProcessDigit( inputCharacter )
   Case ",", ".", ":", ";", "!", "?"
      ProcessPunctuation( inputCharacter )
   Case " "
      ProcessSpace( inputCharacter )
   Case "A" To "Z", "a" To "z"
      ProcessAlpha( inputCharacter )
   Case Else
      ProcessError( inputCharacter )
End Select
```

该 case 语句中的 case 是按照接近 ASCII 排序顺序排列的。但在 case 语句中，效果往往与你写了一系列复杂的 if-then-else 相同。所以，如果得到一个"a"作为输入字符，程序在确定它是字母之前，会先测试它是数学符号、标点符号、数字还是空格。如果知道输入字符可能的频率，可将最常见的情况放在前面。下面是重新排序的 case 语句：

Visual Basic示例：良好排序的逻辑测试

```
Select inputCharacter
   Case "A" To "Z", "a" To "z"
      ProcessAlpha( inputCharacter )
   Case " "
      ProcessSpace( inputCharacter )
   Case ",", ".", ":", ";", "!", "?"
      ProcessPunctuation( inputCharacter )
   Case "0" To "9"
      ProcessDigit( inputCharacter )
   Case "+", "="
      ProcessMathSymbol( inputCharacter )
   Case Else
      ProcessError( inputCharacter )
End Select
```

代码经优化后，由于最常见的情况通常会更快地发现，所以会产生更少的测试，从而提升了性能。以下是此次优化的结果，使用的样本具有典型的字符组合：

编程语言	直接时间	调优后执行时间	节省时间
C#	0.220	0.260	-18%
Java	2.56	2.56	0%
Visual Basic	**0.280**	0.260	7%

说明：测试基准数据来自组合的输入，其中含有 78% 的字母、17% 的空格以及 5% 的标点符号

Microsoft Visual Basic 结果符合预期，但 Java 和 C# 结果不符合预期。显然，这是 switch-case 语句在 C# 和 Java 中的构造方式使然——每个值都必须单独枚举而不是放在一个范围中处理，C#和 Java 代码不会像 Visual Basic 代码那样从优化中受益。这个结果强调了不要盲目遵循任何优化建议的重要性——特定的编译器实现对结果有显著影响。

你或许以为 Visual Basic 编译器为执行与 case 语句相同的测试的一组 if-then-else 生成的代码是相似的。来看看结果：

编程语言	直接时间	调优后执行时间	节省时间
C#	0.630	0.330	48%
Java	0.922	0.460	50%
Visual Basic	**1.36**	**1.00**	**26%**

结果大不相同。针对相同数量的测试，Visual Basic 编译器在未优化的情况下花费的时间是 5 倍，在优化的情况下是 4 倍。这表明编译器为 case 方法和 if-then-else 方法生成的是不一样的代码。

使用 if-then-else 的改进比使用 case 语句的改进更一致，但这要分两面看。在 C#和 Visual Basic 中，两个版本的 case 语句方法都比两个版本的 if-then-else 方法快，而在 Java 中，两个版本都比较慢。这种结果上的差异表明了第三种可能的优化，详情见下一节。

相似逻辑结构之间的性能比较

上述测试可用 case 语句或 if-then-else 来执行。取决于不同的环境，任何一种方法都可能更好地工作。下面重新格式化了前两个表格的数据，展示了对 if-then-else 和 case 的性能进行比较的“代码调优”时间：

编程语言	case 语句	if-then-else 语句	节省时间	性能比
C#	0.260	0.330	−27%	1:1
Java	2.56	0.460	82%	6:1
Visual Basic	0.260	1.00	−258%	1:4

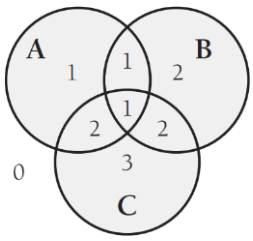
这些结果没有任何逻辑可言。在其中一种语言中，case 明显优于 if-then-else，而在另一种语言中，if-then-else 明显优于 case。在第三种语言中，差异则相对较小。你可能会认为，由于 C#和 Java 共享类似的 case 语法，所以结果会相似，但事实上它们的结果是相反的。

这个例子清楚说明了为代码调优设定任何形式的“经验法则”或“逻辑”的困难——只能亲自度量结果，没有别的替代物。

采用查询表替代复杂的表达式

> 关联参考　要详细了解如何使用查询表来替代复杂逻辑，请参见第 18 章“表驱动法”。

某些时候，查询表可能比遍历一个复杂的逻辑链更快。复杂逻辑链主要用于对某样东西进行分类，然后根据其所属的类别采取相应的行动。作为一个抽象的例子，假设要根据某样东西所属的三个组——A 组、B 组和 C 组——为它分配一个类别编号：



下面这个复杂逻辑链负责分配类别编号：

C++示例：复杂逻辑链

```
if ( ( a && !c ) || ( a && b && c ) ) {
   category = 1;
}
else if ( ( b && !a ) || ( a && c && !b ) ) {
   category = 2;
}
else if ( c && !a && !b ) {
   category = 3;
}
else {
   category = 0;
}
```

可用一个更容易修改、性能更高的查询表来替代这个测试：

表的定义较难理解。添加注释方便人们读懂表的定义。

C++示例：用查询表替代复杂逻辑

```
// define categoryTable
static int categoryTable[ 2 ][ 2 ][ 2 ] = {
   // !b!c  !bc  b!c  bc
       0,    3,   2,   2,   //   !a
       1,    2,   1,   1    //    a
};
...
category = categoryTable[ a ][ b ][ c ];
```

虽然表格定义比较难懂，但如果贴心地为它加上注释，就不会比复杂的逻辑链代码更难懂。以后若要修改定义，该表也比之前的逻辑更容易维护。下面是性能测试结果：

编程语言	直接时间	调优后执行时间	节省时间	性能比
C++	5.04	3.39	33%	1.5:1
Visual Basic	5.21	2.60	50%	2:1

使用惰性求值

我的一个前室友是重度拖延症患者。他为自己的懒惰辩解说，许多人们觉得急着要做的事情根本不需要做。他声称，只要等得足够久，那些不重要的事情就会被拖延到忘掉，所以不必浪费时间去做那些事情。

懒惰求值（lazy evaluation）正是基于我室友的原则。如程序使用了惰性求值，除非万不得已，否则会尽量避免做任何工作。惰性求值类似于即时（just-in-time）策略，即在马上就需要的时候才干活儿。

例如，假定程序包含 5000 个值的一个表格。程序启动时生成整个表格，并在执行期间使用它。但是，如果程序只使用表中的一小部分条目，那么更好的方案是在需要时才计算这些条目，而不是一次性全部计算好。一旦某个条目被计算出来，它仍可存储起来供将来引用——或者说“缓存”起来（cached）。

26.2 循环

关联参考　参见第 16 章更多地了解循环。

由于循环要执行许多次，所以程序中的热点通常都在循环中。利用本节的技术使循环本身变得更快。

循环判断外提

switching 是指每次执行循环时，都要在循环内做出一个判断。如果每次循环时的判断结果都一样，可将判断拿到循环外部（称为循环判断外提，或 unswitch）。通常，这需要将循环里外翻转，将循环放以条件里面，而不是将条件放到循环里面。下面是一个判断外提前的例子：

C++示例：一个switched(判断内置)循环

```
for ( i = 0; i < count; i++ ) {
   if ( sumType == SUMTYPE_NET ) {
      netSum = netSum + amount[ i ];
   }
   else {
      grossSum = grossSum + amount[ i ];
   }
}
```

在这段代码中，if (sumType == SUMTYPE_NET)这个测试每次迭代都会重复，即使每次的结果都一样。所以，可以这样重写代码以提高速度：

C++示例：一个unswitched(判断外提)循环

```
if ( sumType == SUMTYPE_NET ) {
   for ( i = 0; i < count; i++ ) {
      netSum = netSum + amount[ i ];
   }
}
else {
   for ( i = 0; i < count; i++ ) {
      grossSum = grossSum + amount[ i ];
   }
}
```

注意：这段代码违反了良好编程的若干规则。可读性和可维护性通常比执行速度或规模更重要，但本章的主题是性能，这意味着要与其他目标进行权衡。和上一章一样，本章会展示本书其他部分不推荐的编码实践的例子。

结果是节省了 20%左右的时间：

编程语言	直接时间	调优后执行时间	节省时间
C++	2.81	2.27	19%
Java	3.97	3.12	21%
Visual Basic	2.78	2.77	<1%
Python	8.14	5.87	28%

这个调优的一个明显的风险是两个循环必须平行维护。如果将 count 改为 clientCount，必须记住两个地方都要改，这不仅麻烦，其他人在维护时也会头痛。

这个例子还说明了代码调优的一项关键挑战：任何特定的代码调优效果都是不可预测的。代码调优在四种语言的三种中产生了明显的改进，但在 Visual Basic 中却没有。在这个特定版本的 Visual Basic 中执行这个特定的优化，会产生可维护性变差的代码，性能上却没有任何收益。总之，必须对每个特定的优化进行度量以确定其效果，没有例外。

合并

合并（jamming）或融合（fusion）是指对操作相同元素集合的两个循环进行合并。两个循环变成一个，自然减少了循环的开销。

下例适合进行循环合并：

Visual Basic示例：可以合并的两个单独的循环

```
For i = 0 to employeeCount - 1
   employeeName( i ) = ""
Next
...
For i = 0 to employeeCount - 1
   employeeEarnings( i ) = 0
Next
```

合并循环时，要在两个循环中找到可合并成一个的代码。通常，这意味着循环计数器必须相同。在本例中，两个循环都从 0 运行到 employeeCount - 1，所以可以合并：

Visual Basic示例：合并后的循环

```
For i = 0 to employeeCount - 1
   employeeName( i ) = ""
   employeeEarnings( i ) = 0
Next
```

下面对比了调优后的性能提升。

编程语言	直接时间	调优后执行时间	节省时间
C++	3.68	2.65	28%
PHP	3.97	2.42	32%
Visual Basic	3.75	3.56	4%

说明：本例中，测试基准数据 employeeCount 取值为 100

和之前一样，不同语言的结果差异很大。

循环合并有两个主要风险。首先，被合并的两个部分的索引可能会发生变化，造成它们不再兼容。其次，可能不容易合并循环。合并循环之前，一定确保它们相对于代码其余部分的顺序正确。

展开

循环展开（unrolling）的目的是减少循环的内务处理。在第 25 章中，一个循环被完全展开，10 行代码被证明比 3 行快。在这种情况下，循环从 3 行展开成 10 行，使全部 10 个数组访问均单独完成。

虽然完全展开循环是一个快速的解决方案，而且在处理少量元素时效果不错，但如果有大量元素或事先不知道会有多少元素，它就不实用了。下面是一个常规循环的例子：

通常应使用一个*for*循环来做这样的工作，但为了优化它，又必须转换成一个*while*循环。为方便讨论，这里直接展示一个*while*循环。

Java示例：该循环可以展开

```
i = 0;
while ( i < count ) {
   a[ i ] = i;
   i = i + 1;
}
```

为了部分展开循环，每次循环迭代都处理两个或更多情况，而不是一个。这种展开会损害可读性，但不会损害循环的通用性。下面是该循环展开一次的结果：

这几行处理因循环递增2而非1而可能造成的未被循环处理到的情况。

Java示例：循环被展开一次

```
i = 0;
while ( i < count - 1 ) {
   a[ i ] = i;
   a[ i + 1 ] = i + 1;
   i = i + 2;
}

if ( i == count - 1) {
   a[ count - 1 ] = count - 1;
}
```

该技术用两行替代了原来 a[i]=i 那一行，而且 i 递增 2 而非 1。需要在 while 循环之后增加代码，防止因 count 是奇数而造成循环终止后还剩一次迭代

当 5 行直接的代码扩展为 9 行“难以理解”（tricky）的代码后，代码变得较难阅读和维护。尽管速度提升了，但质量变差了。然而，任何设计准则都有一部分要求进行必要的权衡。所以，即使某个特定的技术通常代表着糟糕的编码实践，但在特定情况下，它也可能是最适合的。

以下循环展开后的性能数据：

编程语言	直接时间	调优后执行时间	节省时间
C++	1.75	1.15	34%
Java	**1.01**	**0.581**	**43%**
PHP	5.33	4.49	16%
Python	2.51	3.21	-27%

说明：本例的测试基准数据 count 取值为 100

16%到 43%的收益非常不错，虽然还是要注意不升反降的情况，如 Python 基准测试所显示的那样。循环展开的主要风险是在循环处理了最后一个情况后出现“相差 1”错误。

如进一步展开循环，进行两次或更多次展开呢？将循环展开两次，会获得更大收益吗？

Java示例：循环被展开两次

```
i = 0;
while ( i < count - 2 ) {
   a[ i ] = i;
   a[ i + 1 ] = i+1;
   a[ i + 2 ] = i+2;
   i = i + 3;
}
```

```
if ( i <= count - 1 ) {
    a[ count - 1 ] = count - 1;
}
if ( i == count - 2 ) {
    a[ count -2 ] = count - 2;
}
```

以下循环第二次展开后的性能数据：

编程语言	直接时间	循环展开两次后执行时间	节省时间
C++	1.75	1.01	42%
Java	**1.01**	**0.581**	**43%**
PHP	5.33	3.70	31%
Python	2.51	2.79	−12%

说明：本例的测试基准数据 count 取值为 100

结果表明，进一步展开循环也许能进一步节省时间，但并非肯定如此，如 Java 的度量结果所示。我们主要的关切在于，代码现在变得多么有多繁复。单纯看上面的代码，也许不觉得它有多复杂，但再看几页前那个仅 5 行的循环，就能体会是牺牲了多大的可读性才换来了这个性能。

最小化循环内部的工作

写高效循环的一个关键是循环内部的工作应最小化。如果能在循环外部求值一个语句或语句的一部分，在循环内部只使用结果，就应该将这些计算放到循环外部。这个良好的编程实践在很多情况下还有助于改善程序的可读性。

假定某个热循环（频繁运行的循环）内部有一个如下所示的复杂指针表达式：

C++示例：循环内部的复杂指针表达式

```
for ( i = 0; i < rateCount; i++ ) {
   netRate[ i ] = baseRate[ i ] * rates->discounts->factors->net;
}
```

这种情况下，将复杂指针表达式赋给一个良好命名的变量，不仅能改善代码可读性，还通常能提升性能：

C++示例：简化复杂指针表达式

```
quantityDiscount = rates->discounts->factors->net;
for ( i = 0; i < rateCount; i++ ) {
   netRate[ i ] = baseRate[ i ] * quantityDiscount;
}
```

引入的新变量 quantityDiscount 清楚表明 baseRate 数组中的每个元素都和数量折扣（quantity discount）系数相乘以计算出净费用（net rate）。而在循环中最初的表达式中，这样的含义并没有清楚表达出来。将复杂指针表达式放到循环外部的一个变量中，还使每次循环迭代都节省了 3 次指针解引用的时间。下面展示了性能提升数据：

编程语言	直接时间	调优后执行时间	节省时间
C++	3.69	2.97	19%
C#	2.27	1.97	13%
Java	4.13	2.35	43%

说明：本例的测试基准数据 rateCount 取值为 100

除了 Java 编译器，本例的代码调优对于其他编程语言的效果并没有什么太多值得说道的。这些数据也给出了启示：在最初的编码阶段，开发人员可以尽情使用可读性更好的方法，以后再关注代码速度。

哨兵值

带复合测试的循环通常可通过简化测试来节省时间。如果是一个搜索循环，简化测试的一个方法是使用哨兵值（sentinel value），即一个刚刚超过搜索范围末端，而且保证能终止搜索的值。

对于搜索循环，可通过哨兵值来改进复合测试的一个典型例子是既检查是否找到了所需的值，又检查是否用完了所有值，如下例所示：

这里是复合测试

C#示例：搜索循环中的复合测试

```
found = FALSE;
i = 0;
while ( ( !found ) && ( i < count ) ) {
   if ( item[ i ] == testValue ) {
      found = TRUE;
   }
   else {
      i++;
   }
}

if ( found ) {
   ...
```

在这段代码中，循环的每一次迭代都会测试!found 和 i＜count。前者确定何时找到了所需的元素，后者防止超出数组末端。在这个循环中，item[]的每个值都要被单独测试，所以该循环的每一次迭代实际有三个测试。

对于这种搜索循环，三个测试可合并成一个，并在搜索范围的末端放一个“哨兵”来终止循环，这样每次循环迭代只会产生一次测试。在这种情况下，可简单地将你要找的值赋给刚好超出搜索范围末端的那个元素（下图箭头所指的赋值语句。另外，记得在声明数组时为该元素留出空间。)然后检查每个元素，如果在找到卡在末尾的那个元素之前没有找到目标元素，就知道你要找的值并不在范围中。下面是具体的代码：

记住为数组末端的哨兵值留出空间。

C#示例：使用哨兵值加快循环速度

```
// set sentinel value, preserving the original value
initialValue = item[ count ];
item[ count ] = testValue;

i = 0;
while ( item[ i ] != testValue ) {
   i++;
}

// check if value was found
if ( i < count ) {
   ...
```

在 item 是整数数组的前提下，性能有了显著提升：

编程语言	直接时间	调优后执行时间	节省时间	性能比率
C#	0.771	0.590	23%	1.3:1
Java	1.63	0.912	44%	2:1
Visual Basic	1.34	0.470	65%	3:1
说明：搜索一个拥有 100 个整数元素的数组				

Visual Basic 的提升尤其显著，但其实所有结果都不错。然而，当数组类型发生变化时，结果也会发生变化。在 item 是单精度浮点数组时，结果如下：

编程语言	直接时间	调优后执行时间	节省时间
C#	1.351	1.021	24%
Java	1.923	1.282	33%
Visual Basic	1.752	1.011	42%
说明：搜索一个拥有 100 个 4 字节浮点数的数组			

和往常一样，结果呈现出很大差异。

哨兵技术几乎可应用于任何线性搜索的情况，包括链表和数组。唯一要注意的是，必须谨慎选择前哨值，而且必须注意将哨兵值放入数据结构的方式。

最忙的循环放在最内层

如果有嵌套循环，慎重考虑哪个循环在外，哪个在内。以下嵌套循环值得改进：

Java示例：可改进的嵌套循环

```
for ( column = 0; column < 100; column++ ) {
   for ( row = 0; row < 5; row++ ) {
      sum = sum + table[ row ][ column ];
   }
}
```

本例改进循环的关键在于，外层循环的执行频率比内层循环高得多。循环每次执行都必须初始化循环索引，而且每次迭代都要递增索引，并在迭代之后检查索引。外层循环的总执行次数为 100 次，内层循环为 100*5=500 次，共计 600 次。只需简单地交换内层和外层循环，即可将外层循环的总迭代次数变成 5 次，内层循环 5*100=500 次，共计 505 次。所以，通过交换内外循环，理论上能节省大约(600-505) / 600 = 16%。下面是测得的性能差异：

编程语言	直接时间	调优后执行时间	节省时间
C++	4.75	3.19	33%
Java	**5.39**	**3.56**	**34%**
PHP	4.16	3.65	12%
Python	3.48	3.33	4%

提升效果显著。这再次表明，必须在自己的特定环境中度量效果，然后才能确定优化有效。

降低强度

降低强度意味着用更便宜的操作（如加法）取代昂贵的操作（如乘法）。有的时候，循环内部会有一个表达式依赖于循环索引乘以一个系数。加法一般比乘法快，如果能在循环的每次迭代中通过加法而不是乘法来计算同样的数字，代码通常会运行得更快。下例使用的是乘法：

Visual Basic示例：对循环索引做乘法

```
For i = 0 to saleCount - 1
   commission( i ) = (i + 1) * revenue * baseCommission * discount
Next
```

这段代码很直接，但成本很高。可重写循环，通过累加来获得乘积，而不是每次都计算。这样就将操作的强度从乘法降至加法。

Visual Basic示例：做加法而非乘法

```
incrementalCommission = revenue * baseCommission * discount
cumulativeCommission = incrementalCommission
For i = 0 to saleCount - 1
   commission( i ) = cumulativeCommission
   cumulativeCommission = cumulativeCommission + incrementalCommission
Next
```

乘法很昂贵，这种修改就像拿到了厂商发的优惠券，让你在循环的成本上有所折扣。原来的代码每次都递增 i，然后乘以 revenue*baseCommission*discount——先乘以 1，再乘以 2，再

乘以 3……以此类推。优化后的代码将 incrementalCommission 设为 revenue*baseCommission*discount。然后，每次循环迭代都将 incrementalCommission 加到 cumulativeCommission 上。第一次迭代，它被加了一次；第二次加了两次；第三次加了三次……以此类推。其效果等同于 incrementalCommission 乘以 1，再乘以 2，再乘以 3……，但它更便宜。

关键在于，原来的乘法必须依赖于循环索引。而就本例来说，循环索引是表达式中唯一变化的部分，所以表达式可以更经济地重新编码。下面基于一些测试用例展示了重写后的性能提升。

编程语言	直接时间	调优后执行时间	节省时间
C++	4.33	3.80	12%
Visual Basic	3.54	1.80	49%

说明：测试基准数据 saleCount 取值为 20。所有参与计算的变量都是浮点类型

26.3 数据变换

改变数据类型可为缩减程序规模和提升执行速度提供有力帮助。数据结构的设计超出了本书的范围，但通过适度改变特定数据类型的实现，也有助于性能的提升。下面是一些数据类型调优方法。

使用整数而不是浮点数

交叉参考　要更详细地了解整数和浮点的使用，请参阅第 12 章。

整数加法和乘法比浮点快。例如，将循环索引从浮点变为整数可节省时间：

Visual Basic示例：该循环使用了耗时的浮点循环索引

```
Dim x As Single
For x = 0 to 99
   a( x ) = 0
Next
```

下面修改 Visual Basic 循环以显式使用整型：

Visual Basic示例：该循环使用了省时的整数循环索引

```
Dim i As Integer
For i = 0 to 99
   a( i ) = 0
Next
```

差别有多大？下面是上述 Visual Basic 代码和相似的 C++/PHP 代码的结果：

编程语言	直接时间	调优后执行时间	节省时间	性能比
C++	2.80	0.801	71%	3.5:1
PHP	5.01	4.65	96%	1:1
Visual Basic	6.84	0.280	7%	25:1

数组维度尽可能少

交叉参考　要更详细地了解数组，请参见 12.8 节。

多维数组开销很大。如果能用一维数组来组织数据而不是二维或三维数组，或许可以节省一些时间。以下面这段代码为例：

Java示例：标准的二维数组初始化

```
for ( row = 0; row < numRows; row++ ) {
   for ( column = 0; column < numColumns; column++ ) {
      matrix[ row ][ column ] = 0;
   }
}
```

如果这段代码需遍历 50 行、20 列的一个数组，那么使用我目前的 Java 编译器，运行时间将是调整为一维数组之后的两倍。修改后的代码如下所示：

Java示例：将数组修改为一维

```
for ( entry = 0; entry < numRows * numColumns; entry++ ) {
   matrix[ entry ] = 0;
}
```

下表汇总了测试结果，包括和其他几种编程语言的对比：

编程语言	直接时间	调优后执行时间	节省时间	性能比
C++	8.75	7.82	11%	1:1
C#	3.28	2.99	9%	1:1
Java	**7.78**	**4.14**	**47%**	**2:1**
PHP	6.24	4.10	34%	1.5:1
Python	3.31	2.23	32%	1.5:1
Visual Basic	9.43	3.22	66%	3:1

说明：Python 和 PHP的时间无法直接与其他编程语言比较，因其执行的迭代次数不及其他编程语言的 1%。

结果显示，该优化方法对 Visual Basic 和 Java 语言效果极佳，对 PHP 和 Python 语言效果良好，对 C++和 C#语言则表现平平。当然，C#编译器生成的代码在优化前的性能就已经出类拔萃了，所以不能对它要求太高。

测试结果的显著差异再次表明，盲目采纳各种代码调优建议是有风险的。除非在自己的特定应用场景下尝试过，否则永远无法打包票。

最小化数组引用

除了尽量减少对二维或三维数组的访问，还要尽量减少数组访问。重复使用数组中一个元素的循环就是很好的优化对象。下例进行了不必要的数组访问：

C++示例：不必要地在循环内部引用数组

```
for ( discountType = 0; discountType < typeCount; discountType++ ) {
   for ( discountLevel = 0; discountLevel < levelCount; discountLevel++ ) {
      rate[ discountLevel ] = rate[ discountLevel ] * discount[ discountType ];
   }
}
```

内层循环中的 discountLevel 发生变化时，对 discount[discountType]的引用不会变。所以可将其移出内层循环，这样每次执行外层循环就只产生一次数组访问，而不是每次执行内层循环都来一次。下例展示了修改后的代码：

C++示例：将数组引用放到循环外部

```
for ( discountType = 0; discountType < typeCount; discountType++ ) {
   thisDiscount = discount[ discountType ];
   for ( discountLevel = 0; discountLevel < levelCount; discountLevel++ ) {
      rate[ discountLevel ] = rate[ discountLevel ] * thisDiscount;
   }
}
```

性能测试结果如下所示：

编程语言	直接时间	调优后执行时间	节省时间
C++	**32.1**	**34.5**	**-7%**
C#	18.3	17.0	7%
Visual Basic	23.2	18.4	20%

说明：测试基准数据 typeCount 取值为 10，levelCount 取值为 100

和往常一样，不同编译器的结果呈现出很大差异。

使用辅助索引

使用辅助索引（supplementary index）是指添加相关数据，使得对某种数据类型的访问更高效。可将相关数据添加到主数据类型中或者存放到一个并行结构中。

字符串长度索引

在各种字符串存储策略都能找到使用辅助索引的例子。在 C 中，字符串以设为 0 的一个字节终止。在 Visual Basic 字符串格式中，每个字符串的起始位置都隐藏着一个长度字节，它标识了该字符串的长度。为确定 C 字符串的长度，程序需从字符串起始位置对各个字节计数，直到发现设为 0 的字节。为确定 Visual Basic 字符串的长度，程序只需查看那个长度字节。Visual Basic 长度字节就是通过增加索引来扩展数据类型，从而使某些操作（例如计算字符串长度）变得更快的例子。

可将这个长度索引的概念应用于任何长度可变的数据类型。跟踪数据结构的长度通常比每次需要时计算长度更高效。

独立、并行的索引结构

有的时候，与操作数据类型本身相比，操作数据类型的索引更高效。如果数据类型中的数据项过于庞大或很难移动（或许存放在磁盘上），那么对索引进行排序和搜索要比直接操作数据更快。如果每个数据项都很庞大，可考虑创建一个辅助结构，在其中保存键值和指向详细信息的指针。如果数据结构项和辅助结构项在大小上存在显著差异，那么即使数据不得不存放在外部，有时也可以将键存储在内存中。这样一来，所有的搜索和排序都可以在内存中完成，得知目标数据项的确切位置后，只需一次磁盘访问就可以了。

使用缓存

缓存（caching）是指将一些值按照某种方式存储起来，使最常用的值比不太常用的数值更容易被检索到。例如，假定某程序从磁盘上随机读取记录，那么其中某个子程序或许可用缓存来存储读取最频繁的记录。子程序收到访问某条记录的请求时，首先检查缓存中是否存在这条记录；如果有，就直接从内存中返回该记录，而无需访问磁盘。

除了缓存磁盘上的记录外，缓存技术还可应用于其他方面。在某个 Microsoft Windows 字体校正程序中，性能瓶颈出在每个字符显示时对其宽度的检索。缓存最近使用的字符宽度，其显示速度提高了一倍。

还可以缓存耗时计算的结果——特别是当计算的参数很简单的时候。例如，假定需要计算一个直角三角形的斜边长度（给定其他两边的长度）。该子程序的直接实现如下所示：

Java示例：该子程序能从缓存中获益

```
double Hypotenuse(
   double sideA,
   double sideB
   ) {
   return Math.sqrt( ( sideA * sideA ) + ( sideB * sideB ) );
}
```

如果知道相同的值会被反复请求，就可以像下面将值缓存起来：

Java示例：缓存以避免昂贵的计算

```
private double cachedHypotenuse = 0;
private double cachedSideA = 0;
private double cachedSideB = 0;

public double Hypotenuse(
   double sideA,
   double sideB
   ) {

   // check to see if the triangle is already in the cache
   if ( ( sideA == cachedSideA ) && ( sideB == cachedSideB ) ) {
      return cachedHypotenuse;
   }

   // compute new hypotenuse and cache it
   cachedHypotenuse = Math.sqrt( ( sideA * sideA ) + ( sideB * sideB ) );
   cachedSideA = sideA;
   cachedSideB = sideB;

   return cachedHypotenuse;
}
```

第二个版本比第一个更复杂，占用空间也更多，但速度足以作为补偿。许多缓存方案会缓存一个以上的元素，相应开销更大。两个版本在运行速度上的差异如下所示：

编程语言	直接时间	代码调优后执行时间	节省时间	性能比
C++	4.06	1.05	74%	4:1
Java	**2.54**	**1.40**	**45%**	**2:1**
Python	8.16	4.17	49%	2:1
Visual Basic	24.0	12.9	47%	2:1

说明：假设每次缓存了某个值后，该值会被命中两次。

缓存成功与否取决于访问缓存元素、创建未缓存元素以及在缓存中存储新元素时产生的相对成本。成功与否还取决于缓存信息的访问请求频率。有的时候，成功与否或许还要取决于硬件在缓存时是否给力。通常，生成新元素的成本越高，请求相同信息的次数越多，缓存就越有价值。访问缓存元素以及将新元素存储到缓存中的开销越低廉，缓存就越有价值。和其他优化技术一样，缓存增加了程序的复杂度，使得程序更容易出错。

26.4 表达式

交叉参考　要更多地了解表达式，请参阅第 19.1 节。

程序的许多工作都是在数学或逻辑表达式中完成的。复杂的表达式通常都很昂贵，所以本节探讨如何降低其成本。

利用代数恒等式

可以采用代数恒等式，以开销低的运算来替代开销大的运算。例如，怔两个表达式在逻辑上等价：

```
not a and not b
not（a or b）
```

如选用第二个表达式而不是第一个，就可省去一次 not 运算。

虽然避免一次 not 运算所节省时间也许微不足道，但它背后的基本原则是强大的。Jon Bentley 描述过一个用来测试 sqrt(x)＜sqrt(y)的程序（1982）。由于仅当 x 小于 y 时，sqrt(x)才会小于 sqrt(y)，所以可直接用 x＜y 来替代。由于 sqrt()子程序本身的代价就很高昂，所以可预见到大幅的优化。的确如此，测试结果如下表所示：

编程语言	直接时间	调优后执行时间	节省时间	性能比
C++	7.43	0.010	99.9%	750:1
Visual Basic	4.59	0.220	95%	20:1
Python	4.21	0.401	90%	10:1

降低强度

之前说过，降低强度是指用便宜的操作替代昂贵的。下面列举了一些可能的替代方法：

- 用加法代替乘法。
- 用乘法代替求幂。
- 用三角恒等式代替三角函数。
- 用 long 或 int 代替 longlong 整数（但要注意采用原生和非原生长度的整数所带来的性能差异）。
- 用定点数或整数代替浮点数。
- 用单精度数代替双精度数。
- 用移位操作代替整数乘 2 或除 2。

假设需要计算一个多项式。如果你对多项式感到生疏了，不妨把它们看成$Ax^2 + Bx + C$这样的东西。字母 A、B、C 都是系数，x 是变量。计算 *n* 阶多项式的代码一般像下面这样：

Visual Basic示例：多项式求值

```
value = coefficient( 0 )
For power = 1 To order
   value = value + coefficient( power ) * x^power
Next
```

考虑降低强度时，要用怀疑的眼光去看待求幂操作符（^）。一个方案是将求幂替换为每次循环做一次乘法运算，这类似于几个小节之前用加法代替乘法以降低计算强度。下面是降低了计算强度之后的多项式求值代码：

Visual Basic示例：降低多项式求值强度的方法

```
value = coefficient( 0 )
powerOfX = x
For power = 1 to order
   value = value + coefficient( power ) * powerOfX
   powerOfX = powerOfX * x
Next
```

如下表所示，这样的改动对二次或更高次多项式的效果相当显著（二次多项式就是最高次项为平方的多项式）：

编程语言	直接时间	调优后执行时间	节省时间	性能比
Python	3.24	2.60	20%	1:1
Visual Basic	**6.26**	**0.160**	**97%**	**40:1**

如果认真对待降低强度的问题，两次浮点乘法会让你无法释怀。降低强度的原则表明，可通过累加乘幂而不是每次都执行乘法运算来进一步降低循环中的运算强度。

Visual Basic示例：进一步降低多项式求值所需的运算强度

```
value = 0
For power = order to 1 Step -1
   value = ( value + coefficient( power ) ) * x
Next
value = value + coefficient( 0 )
```

这个方法消除了额外的 powerOfX 变量，并将每次循环迭代时的两次乘法运算替换为一次。性能测试结果如下所示：

编程语言	直接时间	第 1 次优化	第 2 次优化	与第 1 次优化相比节省的时间
Python	3.24	2.60	2.53	3%
Visual Basic	**6.26**	**0.16**	**0.31**	**-94%**

这是证明理论在实践中可能没有获得很好支持的一个好例子。降低强度的代码似乎应该更快，但实际并非如此。一种可能是，在 Visual Basic 中，将循环递减 1 而不是递增 1 会损害性能，但只有亲自度量这个假设才能确定。

在编译时初始化

如果在子程序调用中需用到一个具名常量或神奇数字，而且它是唯一实参，就可考虑事先计算好该数字，把它放到一个常量中，从而避免子程序调用。同样的原则也适用于乘法、除法、加法和其他运算。

有一次，我需要计算以 2 为底的整数对数，结果取整为最接近的整数。系统没有以 2 为底的对数子程序，所以我自己写了一个。最无脑的是使用以下公式：

```
log(x)base = log(x) / log(base)
```

> 交叉参考　要更详细地了解如何将变量和它们的值绑定，请参见第 10.6 节。

基于该恒等式，可以写出下面这样的一个子程序：

C++示例：基于系统子程序的以2为底的对数子程序

```
unsigned int Log2( unsigned int x ) {
   return (unsigned int) ( log( x ) / log( 2 ) );
}
```

这个子程序奇慢无比。由于 log(2)值不会改变，所以我用其计算值 0.69314718 取代了 log(2)，如下所示：

C++示例：基于系统子程序和常量的以2为底的对数子程序

```
const double LOG2 = 0.69314718;
...
unsigned int Log2( unsigned int x ) {
   return (unsigned int) ( log( x ) / LOG2 );
}
```

既然 log()计算往往是代价高昂的子程序（比类型转换或除法还要奢侈），所以可预期将 log()函数的调用削减一半后能节省一半的子程序运行时间。实际度量后的结果如下所示：

编程语言	直接时间	调优后执行时间	节省时间
C++	**9.66**	**5.97**	**38%**
Java	17.0	12.3	28%
PHP	2.45	1.50	39%

在本例中，对除法和类型转换的相对重要性以及对节省 50%时间的合理推测都非常接近于现实。但这不过是瞎猫正巧碰上死耗子罢了。本章一直在强调，不要对结果有任何预设。

小心系统子程序

系统子程序（系统例程）的开销很大，而且它提供的精度往往会被浪费掉。例如，典型的系统数学函数是按照将宇航员送上月球且其着陆点误差不超过正负两英尺的精度来设计的。如果不需要这么高的精度，就不必花那么多时间去计算它。

在上例中，Log2()子程序返回一个整数值，却通过一个浮点 log()子程序去计算它。对于整数结果，这显得过犹不及。所以，在第一次尝试之后，我写了一系列的整数测试，这对于整数 $\log_2$ 的计算来说已经足够精确了。下面是优化后的代码：

C++示例：基于整数的以2底为例的对数子程序

```
unsigned int Log2( unsigned int x ) {
   if ( x < 2 ) return 0 ;
   if ( x < 4 ) return 1 ;
   if ( x < 8 ) return 2 ;
   if ( x < 16 ) return 3 ;
   if ( x < 32 ) return 4 ;
   if ( x < 64 ) return 5 ;
   if ( x < 128 ) return 6 ;
   if ( x < 256 ) return 7 ;
   if ( x < 512 ) return 8 ;
   if ( x < 1024 ) return 9 ;
   ...
   if ( x < 2147483648 ) return 30;
   return 31 ;
}
```

该子程序只用到了整数运算，根本没有转换为浮点数，其效率一举击败了两个使用浮点数的版本。

编程语言	直接时间	调优后执行时间	节省时间	性能比
C++	9.66	0.662	93%	15:1
Java	17.0	0.882	95%	20:1
PHP	2.45	3.45	-41%	2:3

绝大多数所谓“超越”函数（Transcendental Functions，即变量之间的关系不能用有限次加、减、乘、除、乘方、开方运算表示的函数，即“超出”代数函数范围的函数）都是为了应对最糟糕的情况而设计的；换言之，即便传入的是整数实参，这些函数在内部还是会转换为双精度浮点数进行处理。如果在某段效率低下的代码中发现了此类函数，同时无需那么高的精度，就应立即引起重视。

另一个方法是借助于右移位操作等同于除以 2 这一事实。在结果非零的情况下，对该数字执行了多少次除以 2 的运算，其结果就等同于进行了多少次 $\log_2$ 运算。下面是基于这一事实而改写的代码：

C++示例：基于右移位操作符的以2为底的对数子程序

```
unsigned int Log2( unsigned int x ) {
   unsigned int i = 0;
   while ( ( x = ( x >> 1 ) ) != 0 ) {
      i++;
   }
   return i ;
}
```

非 C++程序员很难理解这段代码。while 条件中的复杂表达式属于典型的、除非有充分理由否则应当避免的编码实践。

该子程序比之前那个较长的版本多耗费了约 350%的执行时间，即花了 2.4 秒而非 0.66 秒才完成。但还是比第一种方法快，并且能非常容易地适配 32 位、64 位以及其他运行环境。

这个例子凸显了在一次成功优化后还应继续优化的价值。第一次优化带来了可观的 30%到 40%的时间节省，但根本无法同第二次和第三次的优化效果相提并论。

使用正确类型的常量

向变量赋值时，使用和变量同类型的具名常量和字面量。若常量及其相关变量的类型不一致，编译器就必须做一个类型转换将常量赋给变量。好的编译器会在编译时进行类型转换，这样就不会影响运行时性能。

不太先进的编译器或解释器则会生成运行时转换的代码，所以可能影响性能。下面针对两种情况测试浮点变量 x 和整数变量 i 在初始化上的性能差异。第一种情况的初始化如下所示：

```
x = 5
i = 3.14
```

假定 x 是浮点变量，i 是整数，所以这种情况需要类型转换。第二种情况如下所示：

```
x = 3.14
i = 5
```

它则不需要类型转换。下面是性能测试结果，不同编译器的差异同样很大：

编程语言	直接时间	调优后执行时间	节省时间	性能比
C++	1.11	0.000	100%	无法度量
C#	1.49	1.48	<1%	1:1
Java	1.66	1.11	33%	1.5:1
Visual Basic	0.721	0.000	100%	无法度量
PHP	0.872	0.847	3%	1:1

预计算结果

一个常见底层设计决策是从两种方案中选择一个：是动态计算结果，还是计算一次并保存结果，并在需要时查找。如结果需多次使用，通常采用计算一次并在后期查找的方式，这样系统开销更低。

该决策的价值体现在多个方面。在最简单的层面，可在循环外部而非内部计算某个表达式的一部分。本章前面已展示了这样的一个例子。在更为复杂的层面上，可在程序开始执行时计算好一张查询表，之后每次需要时都使用该表。另外，还可将计算结果存储到数据文件中，或直接嵌入程序。

关联参考　参见第 18 章更多地了解如何使用表中的数据替代复杂逻辑。

例如在一个太空大战电子游戏中，程序员最初是实时计算与太阳不同距离的重力系数。该计算很昂贵，影响了性能。但后来发现，程序只需为数不多的几个相对距离。所以，程序员可以预先计算出这些重力系数，并将其存储到一个 10 元素的数组中。和昂贵的实时计算相比，查询现有的数组要快得多。

下面是一个用于计算汽车贷款支付金额的子程序：

Java示例：可预计算的复杂计算

```
double ComputePayment(
   long loanAmount,
   int months,
   double interestRate
   ) {
   return loanAmount /
      (
      ( 1.0 - Math.pow( ( 1.0 + ( interestRate / 12.0 ) ), -months ) ) /
      ( interestRate / 12.0 )
      );
}
```

计算贷款支付金额的公式很复杂，而且相当昂贵。将这些信息提前放入表格，而不是每次都计算，可能会更便宜。

这个表会有多大？取值范围最大的变量是 loanAmount。变量 interestRate 的范围可能从 5% 到 20%（按 0.25%递增），只有 61 个不同的利率。months 范围从 12 到 72，也只有 61 个不同的还款期数。loanAmount（贷款金额）允许的范围是从$1000 到$100000，其中的条目太多，一般不会想用查询表来处理。

但是，大部分计算并不依赖于 loanAmount，所以可将计算中真正难看的部分（整个大表达式的分母）放到一个由 interestRate 和 months 索引的表中。每次只需重新计算 loanAmount 部分：

新建变量*interestIndex*来提供*loanDivisor*数组的下标。

Java示例：预计算一个复杂计算

```
double ComputePayment(
   long loanAmount,
   int months,
   double interestRate
   ) {
   int interestIndex =
      Math.round( ( interestRate - LOWEST_RATE ) * GRANULARITY * 100.00 );
   return loanAmount / loanDivisor[ interestIndex ][ months ];
}
```

在这段代码中，繁琐的计算被替换为数组索引的计算和一次数组访问。性能数据如下所示：

编程语言	直接时间	调优后执行时间	节省时间	性能比
Java	**2.97**	**0.251**	**92%**	**10:1**
Python	3.86	4.63	-20%	1:1

取决于具体情况，可在程序初始化阶段预先计算好 loanDivisor 数组，也可从磁盘文件读取。另外，也可先把它初始化为 0，在首次请求访问该数组时再计算每个元素，并将结果存好以备后续查询。这就是某种形式的缓存，我们之前讨论过。

即使不创建表，也能通过预计算表达式获得性能提升。和之前几个例子相似的代码可考虑采用一种不同的预计算方式。假设以下代码用于计算多种贷款金额的支付金额：

Java示例：可以预计算的第二个复杂计算

```
double ComputePayments(
   int months,
   double interestRate
   ) {
   for ( long loanAmount = MIN_LOAN_AMOUNT; loanAmount < MAX_LOAN_AMOUNT;
      loanAmount++ ) {
      payment = loanAmount / (
         ( 1.0 - Math.pow( 1.0+(interestRate/12.0), - months ) ) /
         ( interestRate/12.0 )
         );
      ...
   }
}
```

后续代码应处理 *payment*；本例略。

即使不预计算一个表，也可以在循环外部预计算表达式中的复杂部分，并在循环内部使用该结果。如下所示：

Java示例：预计算第二个复杂计算

```
double ComputePayments(
   int months,
   double interestRate
   ) {
   long loanAmount;
   double divisor = ( 1.0 - Math.pow( 1.0+(interestRate/12.0). - months ) ) /
      ( interestRate/12.0 );
   for ( long loanAmount = MIN_LOAN_AMOUNT; loanAmount <= MAX_LOAN_AMOUNT;
      loanAmount++ ) {
      payment = loanAmount / divisor;
      ...
   }
}
```

这里是预计算的部分。

这和之前建议的将数组引用和指针解引用放到循环外部的技术相似。在本例中，Java 的结果与第一次优化中使用预计算表的结果相当：

编程语言	直接时间	调优后执行时间	节省时间	性能比
Java	**7.43**	**0.24**	**97%**	**30:1**
Python	5.00	1.69	66%	3:1

Python 这次有了改进，在第一次优化尝试中则没有。许多时候，当一个优化没有产生预期的结果时，一个看似相似的优化会让你如愿以偿。

下面列举了通过预计算对程序进行优化的形式：

- 在程序执行之前计算出结果，并将结果写入在编译时赋值的常量。
- 在程序执行之前计算出结果，并将它们硬编码到运行时使用的变量中。
- 在程序执行之前计算出结果，并将结果放入在运行时加载的文件。
- 程序启动时一次性计算好结果，以后每次需要时就去引用它们。
- 尽可能在循环开始之前完成计算，最小循环内部的工作。
- 首次需要时计算并保存结果，以后再需要时就检索该结果。

消除公共子表达式

如发现某个表达式在代码中重复出现，就将其赋值给一个变量，然后在需要时引用该变量，而不是在多个位置重新计算该表达式。计算贷款支付金额的程序就包含一个应消除的公共子表达式。原始代码如下所示：

Java示例：一个公共子表达式

```
payment = loanAmount / (
      ( 1.0 - Math.pow( 1.0 + ( interestRate / 12.0 ), -months ) ) /
      ( interestRate / 12.0 )
   );
```

在这个示例中，可以把 interestRate/12.0 赋值给一个变量，然后在程序中引用两次，而不是将表达式计算两次。如果选取了良好的变量命名，该优化方法在提升性能的同时，也改善了代码的可读性。以下是修改后的代码：

Java示例：消除公共子表达式

```
monthlyInterest = interestRate / 12.0;
payment = loanAmount / (
      ( 1.0 - Math.pow( 1.0 + monthlyInterest, -months ) ) /
      monthlyInterest
   );
```

本例节省的时间似乎不是特别明显：

编程语言	直接时间	调优后执行时间	节省时间
Java	**2.94**	**2.83**	**4%**
Python	3.91	3.94	−1%

Math.pow()子程序似乎在严重消耗资源，抵消了消除子表达式所带来的优化效果。另一种可能是编译器可能已消除了子表达式。如子表达式战胜整个表达式的成本更多一些，或者编译器本身的优化不太有效，这种优化方法或许能产生更大的影响。

26.5 子程序

> 交叉参考　要更多地了解子程序的使用，请参阅第 7 章。

代码调优最强大的工具之一是好的子程序分解。小的、定义明确的子程序可节省空间，因其避免了将相同的代码分散于多处。它们使程序易于优化，因为可重构子程序中的代码，从而改进每个调用它的子程序。小的子程序在低级语言中相对容易重写。长而曲折的子程序本身就很难理解；在汇编语言这样的低级语言中，理解它们根本就不可能。

将子程序重写为内联

在计算机编程的早期岁月，一些机器对调用子程序施加了过高的性能惩罚。调用子程序意味着操作系统必须将程序换出，换入一个子程序目录，换入其中特定的子程序，执行该子程序，换出子程序，再换回发出调用的子程序。所有这些交换都会消耗资源，使程序变得缓慢。

而现代计算机的调用子程序的代价要小得多。下例是将一个字符串拷贝子程序内联后的结果：

编程语言	子程序执行时间	内联代码执行时间	节省时间
C++	0.471	0.431	8%
Java	13.1	14.4	-10%

某些情况下，可通过像 C++ inline 关键字这样的语言特性将子程序中的代码内联到程序中，从而节省几纳秒的时间。如所用的语言不直接支持 inline，但支持预处理宏，可用一个宏将代码放进去，并根据需要换入或换出。但是，现代机器（其实就是你现在使用的任何机器）对子程序的调用几乎没有性能惩罚。正如这个例子所显示的那样，代码内联和优化代码一样，都有可能使性能不升反降。

26.6 用低级语言重新编码

一句老话不得不得，那就是在遇到性能瓶颈时，应该用低级语言重新编码。如果用 C++编码，低级语言可能是汇编语言。如果用 Python 编码，低级语言可能是 C。用低级语言重新编码往往会改善速度和代码规模。下面列举了用低级语言进行优化的一种典型方式：

1. 100%用高级语言写应用程序。

2. 充分测试该应用程序，验证其正确性。

交叉参考　程序的少量代码耗费了绝大部分运行时间，有关这一现象的详细描述，请参阅第 25.2 节中的“帕累托法则”。

3. 如发现性能需要改进，就分析（profile）应用程序以确定热点。由于程序中约 5%的内容通常会占据约 50%的运行时间，所以通常都能将程序中的小部分内容确定为热点。

4. 用低级语言重新编码几个小的部分以提升整体性能。

你是否会走上前人走过无数遍的这条路，取决于你对低级语言的熟悉程度、问题对于低级语言的贴合程度以及你的绝望程度。在上一章提到的 DES 程序中，我第一次接触到了这个技术。当时，我已尝试了自己知道的全部优化措施，但程序的速度仍然只有目标的一半。用汇编程序重新编码部分程序是唯一剩下的选择。作为汇编程序的新手，我所能做的就是将高级语言直接翻译成汇编程序，但即使像我这样的“半吊子”，也获得了 50%的速度提升。

假定一个子程序负责将二进制数据转换为大写 ASCII 字符。下例用 Delphi 代码来实现：

Delphi示例：更适合用汇编语言写的代码

```
procedure HexExpand(
   var source: ByteArray;
   var target: WordArray;
   byteCount: word
);
var
   index: integer;
   targetIndex: integer;
begin
   targetIndex := 1;
   for index := 1 to byteCount do begin
      target[ targetIndex ] := ( (source[ index ] and $F0) shr 4 ) + $41;
      target[ targetIndex+1 ] := (source[ index ] and $0f) + $41;
      targetIndex := targetIndex + 2;
   end;
end;
```

虽然这段代码不好看出臃肿在什么地方，但它至少包含大量位操作，这并不是 Delphi 的强项。不过，位操作是汇编语言的强项，所以这段代码适合重新编码。下面是汇编语言代码：

示例：用汇编语言重新编码的子程序

```
procedure HexExpand(
   var source;
   var target;
   byteCount : Integer
);
    label
    EXPAND;

    asm
          MOV   ECX,byteCount      // load number of bytes to expand
          MOV   ESI,source         // source offset
          MOV   EDI,target         // target offset
          XOR   EAX,EAX            // zero out array offset

    EXPAND:
          MOV   EBX,EAX            // array offset
          MOV   DL,[ESI+EBX]       // get source byte
          MOV   DH,DL              // copy source byte

          AND   DH,$F              // get msbs
          ADD   DH,$41             // add 65 to make upper case

          SHR   DL,4               // move lsbs into position
          AND   DL,$F              // get lsbs
          ADD   DL,$41             // add 65 to make upper case
          SHL   BX,1               // double offset for target array offset
          MOV   [EDI+EBX],DX       // put target word

          INC   EAX                // increment array offset
          LOOP  EXPAND             // repeat until finished
    end;
```

用汇编语言重写的效果很明显，节省了 41%的运行时间。一个合乎逻辑的推断是，重新编码后，本来就更适合执行位操作的编程语言（例如 C++语言）在性能提升幅度上不如 Delphi 语言那么大。以下是结果：

编程语言	高级语言执行时间	汇编语言执行时间	节省时间
C++	4.25	3.02	29%
Delphi	**5.18**	**3.04**	**41%**

调优前的数据反映了两种语言在位操作上的不同实力。调优后的数据几乎一样，表明汇编代码将 Delphi 和 C++语言最初的性能差异最小化了。

该汇编语言子程序让我们看到，用汇编语言重写的代码未必就会生成庞大、难看的子程序。相反，如本例所示，这种子程序的大小适中。有的时候，汇编语言代码几乎与高级语言版本一样紧凑。

为了用汇编语言重新编码，一种相对简单有效的方式是使用一个能将汇编代码清单作为副产物来输出的编译器。提取需要调优的子程序的汇编代码，保存到单独的源文件中。然后，以汇编代码为基础对代码进行手动优化，每一步都检查正确性并度量性能改进。有的编译器会将高级语言的语句作为注释穿插在汇编代码中。如果你的编译器是这样做的，请在汇编代码中保留它们。

检查清单：代码调优技术

同时改进速度和规模

- 用查询表替代复杂逻辑。
- 合并循环。
- 用整数而不是浮点变量。
- 编译时初始化数据。
- 使用正确类型的常量。
- 预计算出结果。
- 消除公共子表达式。
- 将关键子程序转化为低级语言。

只改进速度

- 知道结果后停止测试。
- 按频率对 case 语句和 if-then-else 链中的测试进行排序。
- 比较相似逻辑结构的性能。
- 使用惰性求值。
- 将循环中的 if 判断外提。
- 展开循环。

- 最小化循环内部的工作。
- 在搜索循环中使用哨兵值。
- 最忙的循环放到嵌套循环的内层。
- 降低内层循环的运算强度。
- 多维数组改为一维。
- 最小化数组引用。
- 为数据类型增加辅助索引。
- 缓存频繁使用的值。
- 利用代数恒等式。
- 降低逻辑和数学表达式的运算强度。
- 小心系统子程序（系统例程）。
- 重写子程序以内联。

26.7 改得越多，越发没有大的改观

你可能预期在我写完第一版《*代码大全*》之后的 10 年里，系统的性能特性会有一些变化，某些方面确实如此。计算机速度大幅提升，内存也更充裕。在第一版中，本章大多数测试都运行了一万到五万次，以获得有意义的、可度量的结果。在这一版中，大多数测试不得不运行一百万到一亿次。如果一个测试不得不运行一亿次才能获得可度量的结果，就不得不问有谁会注意到这些优化在真实程序中的影响？计算机已变得如此强大，以至于对于许多常见的程序类型来说，本章所讨论的性能提升程度已变得无关紧要。

但在其他方面，性能问题一直都存在。编写桌面应用的人可能不需要这些信息，但为嵌入式系统、实时系统和其他有严格速度或空间限制的系统编写软件的人仍然可从中受益。

自 Donald Knuth 在 1971 年发表了他对 Fortran 程序的研究成果以来，我们一直坚持对每次代码调优尝试的影响进行度量。根据本章的度量，任何特定优化的效果实际上都不如 10 年前那样可以预测。每种代码调优的效果都受编程语言、编译器、编译器版本、代码库、库版本和编译器设置等的影响。

代码调优无一例外都涉及对以下两个方面的权衡：一方面是复杂性、可读性、简单性和可维护性，另一方面是对提高性能的渴望。由于需要重新分析（reprofiling）优化效果，它引入了沉重的维护开销。

我发现，坚持可度量的改进能很好地抵抗过早优化的诱惑，也能迫使自己有意识地写清晰、直接的代码。如果一个优化重要到足以让人拿出分析器来度量优化效果，那么只要它能起作用，就表明或许应该允许。但是，如果一个优化的重要性还没有达到可以动用分析器的程度，就不值得为它牺牲可读性、可维护性和其他代码特性。未经度量的代码调优对性能的影响充其量只是一个猜测，对可读性的负作用则是确定的。

更多资源

关于代码调优，我个人很喜欢的参考资料是《*Writing Efficient Programs*》（Bentley，Englewood Cliffs，NJ：Prentice Hall，1982）。该书已绝版，但如果能够找得到，绝对值得一读。人们公认这是关于代码调优的权威之作。Bentley 描述了用时间换空间和用空间换时间的技术。他提供了几个重新设计数据类型以减少空间和时间的例子。他采用的方法比本章的方法更“野”（anecdotal）一些，这些野路子还相当有趣。他对几个子程序执行了几个优化步骤，这样就能看到针对某个问题的第一次、第二次和第三次尝试效果。Bentley 仅用 135 页的篇幅便完成了这本书的主要内容，但其中精华众多，是每个从业程序员都应拥有的罕见的珍宝之一。

Bentley 在他的《*Programming Pearls*》第 2 版（Boston，MA：Addison-Wesley，2000）的附录 4 中对前面那本书的代码调优规则进行了总结。

另外，还有一系列非常全面的讨论和特定技术相关的优化方法的书籍。下面列出了其中几本：

- Booth, Rick. *Inner Loops: A Sourcebook for Fast 32-bit Software Development*. Boston, MA: Addison-Wesley, 1997.
- Gerber, Richard. *Software Optimization Cookbook: High-Performance Recipes for the Intel Architecture*. Intel Press, 2002.
- Hasan, Jeffrey and Kenneth Tu. *Performance Tuning and Optimizing ASP.NET Applications*. Berkeley, CA: Apress, 2003.
- Killelea, Patrick. *Web Performance Tuning, 2d ed*. Sebastopol, CA: O’Reilly & Associates, 2002.
- Larman, Craig and Rhett Guthrie. *Java 2 Performance and Idiom Guide*. Englewood Cliffs, NJ: Prentice Hall, 2000.
- Shirazi, Jack. *Java Performance Tuning*. Sebastopol, CA: O’Reilly & Associates, 2000.
- Wilson, Steve and Jeff Kesselman. *Java Platform Performance: Strategies and Tactics*. Boston, MA: Addison-Wesley, 2000.

要点回顾

- 优化结果在不同编程语言、编译器和环境下存在显著差异。不对每次特定的优化进行度量，就无法确定优化对程序到底是提升还是损害。
- 第一次优化通常并不是最好的。即使找到了好的，也要继续寻找更好的。
- 代码调优有点像核能，是一个争议性的、情绪化的主题。有的人认为这对可靠性和可维护性非常不利，所以根本不会去做。另一些人则认为，只要有适当的保障措施，它还是有益的。无论如何，本章的技术使用需谨慎。

第 VI 部分　系统考虑

第 27 章　程序规模对构建的影响

内容

- 27.1 沟通和规模
- 27.2 项目规模的范围
- 27.3 项目规模对错误的影响
- 27.4 项目规模对生产率的影响
- 27.5 项目规模对开发活动的影响

相关章节

- 构建的前期准备：第 3 章
- 确定要开发的软件类型：第 3.2 节
- 管理构建：第 28 章

在软件开发中，扩大规模不是简单地将小项目的每一部分都做大。假设用 20 个人月写了 25000 行的一个 Gigatron 软件，现场测试发现了 500 个错误。假设 Gigatron 1.0 很成功，Gigatron 2.0 也很成功，于是着手开发 Gigatron Deluxe，这是一个大幅增强的版本，预计将有 25 万行代码。

虽然规模是初版 Gigatron 的 10 倍，但开发 Gigatron Deluxe 的工作量并不是 10 倍，而是 30 倍。另外，30 倍的工作量并不意味着 30 倍的构建量。它可能意味着 25 倍的构建和 40 倍的架构和系统测试。错误也不是 10 倍，而是 15 倍或更多。

如习惯于做小项目，你的第一个中大型项目可能失控，成为一头无法控制的野兽，而不是你所设想的令人愉快的成功。本章将告诉你会出现什么样的野兽，以及在哪里找到工具来驯服它们。相反，如习惯于做大型项目，可能会倾向于在小型项目上使用过于正式的方法。本章描述了如何节省成本，使小项目不至于因为不堪重负而被拖垮。

27.1 沟通和规模

如果项目只有你一个人，那么唯一的沟通路径就是你和客户之间的沟通，除非你还忿忿不平地算上穿过胼胝体的路径，也就是连接你左脑和右脑的路径。随着项目中人数的增加，沟通路径的数量也会增加。这个增加不是根据人数做加法，而是做乘法。换言之，与人数的平方成正比，如图 27-1 所示。

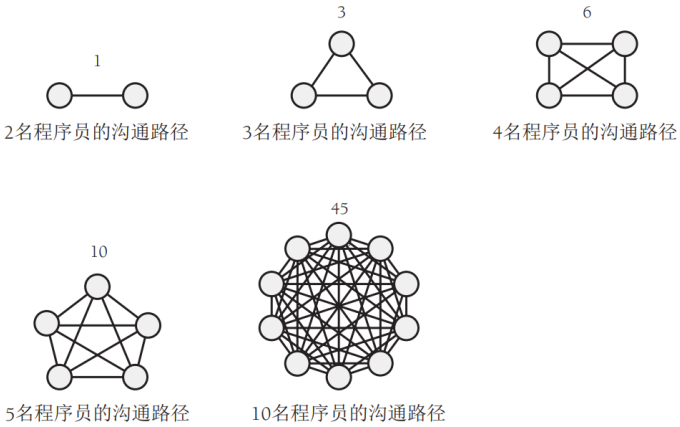


图 27-1　沟通路径的数量与团队人数的平方成正比

KEY POINT 如图所示，两个人的项目仅一条沟通路径。5 人项目有 10 条。10 人项目有 45 条（假设每个人都和其他每个人交谈）。10%的项目有 50 名或更多程序员，其潜在的路径至少有 1200 条。路径越多，花在沟通上的时间就越多，也就为沟通错误（各式各样的误解）创造了更多机会。规模较大的项目需要对沟通进行精简的组织技术，或以合理方式限制沟通。

精简沟通所采取的典型方法是在文件中把它正式化。不是让 50 个人以各种可以想象的组合互相交谈，而是让 50 个人阅读和撰写文件。这些文件有的是文本，有的是图。有的打印在纸上，有的以电子形式存储。

27.2 项目规模的范围

你认为自己从事的项目的规模很典型吗？由于项目规模非常宽泛，所以不能将任何单一的规模视为典型。思考项目规模的一种方式是思考项目团队的规模。下面粗略估计了所有项目中不同规模团队所占的百分比。

团队规模（人）	占所有项目的大致比例
1~3	25%
4~10	30%
11~25	20%
26~50	15%
50+	10%

来源：改编自“A Survey of Software Engineering Practice: Tools, Methods, and Results”(Beck and Perkins1983)，*Agile Software Development Ecosystems* (Highsmith 2002)和*Balancing Agility and Discipline* (Boehmand Turner 2003)

项目规模数据有一个方面可能不是很引人注目，那就是各种规模的项目占比和为其工作的程序员占比的区别。由于每个大项目都会比小项目使用更多的程序员，所以也会占用更大比例的程序员。下面粗略估计了各种规模和项目中的程序员占比：

团队规模（人）	占程序员总数的大致比例
1~3	5%
4~10	10%
11~25	15%
26~50	20%
50+	50%

来源：数据来自“A Survey of Software Engineering Practice: Tools, Methods, and Results”（Beck and Perkins1983），*Agile Software Development Ecosystems*（Highsmith 2002）和 *Balancing Agility and Discipline*（Boehmand Turner 2003）

27.3 项目规模对错误的影响

关联参考　要了解关于错误的更多细节，请参阅第 22.4 节。

无论错误的数量还是类型都受项目规模的影响。你可能没想到错误类型也会受影响。但是，随着项目规模的扩大，会有越来越多的错误归咎于需求和设计，如图 27-2 所示。

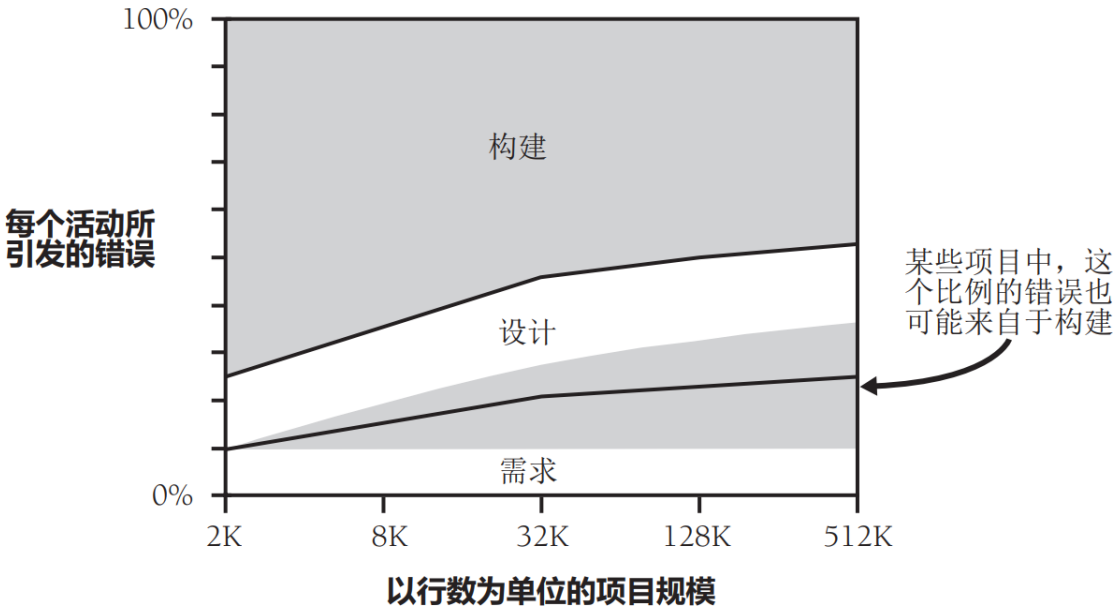
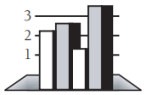


图 27-2　随着项目规模的扩大，更多的错误会来自于需求和设计。有的时候，错误仍然主要来自于构建（Boehm 1981，Grady 1987，Jones 1998）



HARD DATA 在小项目中，构建错误（construction errors）占所有被发现的错误的 75%左右。方法论对代码质量的影响较小，对程序质量影响最大的往往是写程序的人的技术水平（Jones 1998）。

在较大的项目中，构建错误会逐渐减少至总错误的 50%左右；需求和架构错误占比则越来越大。推测这是由于大项目需要更多的需求开发和架构设计，所以这些活动所引发的错误也会增加。但在一些非常大的项目中，构建错误的占比依然很高；有时即使是 50 万行的代码，也有高达 75%的错误要归咎于构建（Grady 1987）。

缺陷的种类随规模的变化而变，缺陷数量也会随之变化。你会很自然地以为比另一个项目大一倍的项目会产生两倍的错误。但是，缺陷的密度（每 1000 行代码的缺陷数量）也会增大。所以，两倍大的产品可能会有两倍以上的错误。表 27-1 总结了不同规模的项目可以预期的缺陷密度范围。

表 27-1 项目规模和典型的错误密度

项目规模（以代码行数为单位）	典型的错误密度
少于 2000 行	每千行 0 到 25 个错误
2000 到 16 000 行	每千行 0 到 40 个错误
16 000 到 64 000 行	每千行 0.5 到 50 个错误
64 000 到 512 000 行	每千行 2 到 70 个错误
512 000 行或者更多	每千行 4 到 100 个错误

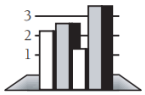
来源：“Program Quality and Programmer Productivity”（Jones 1977），*Estimating Software Costs*（Jones 1998）

关联参考 表中数据代表的是平均表现。一些机构报告的错误率比本表最低的数值更低。详情参见第 22.4 节中的“期望发现多少错误”。

该表的数据来自特定的项目，这些数字可能与你所从事的项目的数字没有什么相似之处。但是，作为行业的缩影，这些数据还是很有启发性的。它表明，错误的数量随着项目规模的扩大而急剧增加，非常大的项目每千行代码的错误数量达到了小项目的四倍。大型项目需要比小型项目更努力才能达到相同的错误率。

27.4 项目规模对生产率的影响

就项目规模来说，生产率(productivity，或称“生产力”)与软件质量有许多共通之处。在小项目中（2000 行代码或更小），对生产率影响最大的是程序员个人的技术水平（Jones 1998）。随着项目规模的扩大，团队规模和组织对生产率的影响越来越大。



HARD DATA 项目需要达到多大的规模，团队的规模才会开始影响生产率？根据 Boehm、Gray 和 Seewaldt 在“Prototyping Versus Specifying: a Multiproject Experiment”中的报告，小团队在完成他们的项目时，生产率比大团队高出 39%。团队规模具体是指什么？小项目两人，大项目三人（1984）。表 27-2 总结了项目规模和生产率之间的一般关系。

表 27-2　项目规模和生产率

项目规模（以代码行数为单位）	每人年的代码行数（括号里是 Cocomo II 均值）
1K	2500~25000(4000)
10K	2000~25000(3200)
100K	1000~20000(2600)
1000K	700~10000(2000)
10000K	300~5000(1600)

来源：数据来自 *Measures for Excellence* (Putnam and Meyers 1992)，*Industrial Strength Software* (Putnamand Meyers 1997)，*Software Cost Estimation with Cocomo II* (Boehm et al. 2000) 和“Software DevelopmentWorldwide: The State of the Practice” (Cusumano et al. 2003)

生产率在很大程度上取决于软件类型、人员素质、编程语言、方法论（methodology）、产品复杂度、编程环境、工具支持、“代码行数”的统计方式、如何将非编程人员的支持工作计入“每人年的代码行数”以及许多其他因素，因此表 27-2 的具体数字会有很大差异。

但同时要意识到，这些数字所显示的总体趋势是很重要的。小项目的生产率可能是大项目生产率的 2~3 倍，而且从最小到最大的项目，生产率可能相差 5~10 倍。

27.5 项目规模对开发活动的影响

如从事的是单人项目，对项目成败影响最大的就是你自己。如果是一个 25 人的项目，可以想象你仍然是最大的影响者，但更有可能的是，没有一个人会因为这一殊荣而戴上奖章；你所在的组织对项目的成败会有更大的影响。

活动占比和规模

随着项目规模和对正式沟通的需求的增大，项目所需的活动种类也会发生显著变化。图 27-3 展示了不同规模的项目的开发活动占比。

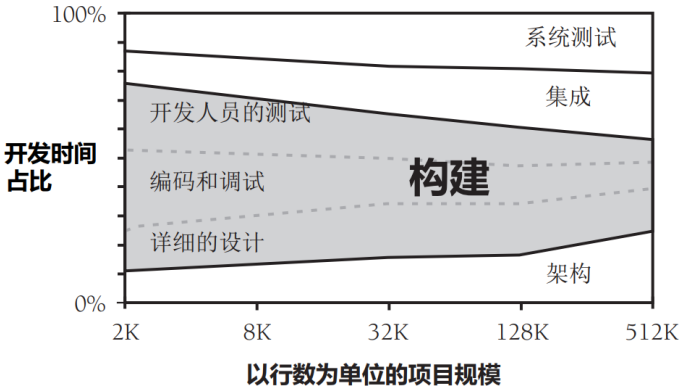


图 27-3　构建活动在小型项目中占主导地位。较大的项目需要更多的架构、集成工作和系统测试才能成功。图中未显示需求工作（requirements work），因为在需求上的付出不像其他活动那样直接与项目规模相关(Albrecht 1979; Glass 1982; Boehm, Gray, and Seewaldt 1984; Boddie 1987; Card 1987; McGarry, Waligora, and McDermott 1989; Brooks 1995; Jones 1998; Jones 2000; Boehm et al. 2000)

在小型项目中，构建（construction）是迄今为止最突出的活动，占总开发时间的 65%。在中等规模的项目中，构建仍然是最主要的活动，但它在总工作量中的比重下降至 50%左右。在非常大的项目中，架构、集成和系统测试占用了更多时间，构建则变得不是那么占主导。简单地说，随着项目规模的扩大，构建在总工作量中的比重越来越小。这张图看起来好像你可以把它向右延伸，让构建完全消失。所以，为了保住我的工作，我把它截断在 512K。

构建之所以越来越不占主导，是因为随着项目规模的扩张，构建活动（包括详细设计、编码、调试和单元测试）虽然规模也会相应扩张，但许多其他活动的规模扩张得更快。图 27-4 对此进行了演示。

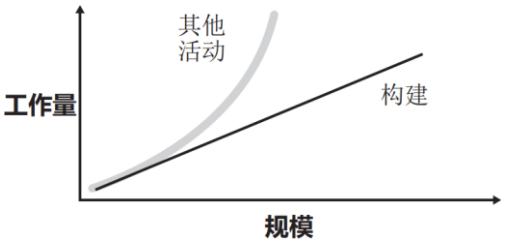
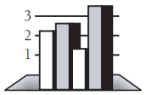


图 27-4 软件构建的工作量与项目规模呈近似线性的关系。其他类型的工作量随项目规模的扩张非线性地增大

规模相近的项目将采取类似的活动，但活动类型随项目规模而异。如本章最开头所述，当 Gigatron Deluxe 版本的规模达到 Gigatron 初始版本的 10 倍时，其构建活动的工作量是原来的 25 倍，项目规划的工作量是原来的 25~50 倍，系统集成的工作量是原来的 30 倍，架构设计和系统测试的工作量则为原来的 40 倍。



HARD DATA 活动的比例之所以发生变化，是因为它们对不同规模的项目的重要性不一样。

Barry Boehm 和 Richard Turner 发现，对于代码不超过 10000 行的项目，把大约 5%的项目总成本花在架构和需求上，会使整个项目的成本最低。但对于代码不超过 100000 行的项目，需要在架构和需求上花费 15%到 20%的工作量才能产生最佳结果（Boehm and Turner 2004）。

随着项目规模的扩大，以下活动的工作量增长高于线性比率：

- 沟通
- 规划
- 管理
- 需求开发
- 系统功能设计
- 接口设计和规格
- 架构
- 集成
- 缺陷消除
- 系统测试
- 文档制作

无论项目规模如何，总有一些技术是价值的：训练有素的编码实践、其他开发人员进行的设计审查和代码审查、好的工具支持以及高级语言的使用。这些技术对小项目很有价值，对大项目的价值更是无法衡量。

程序、产品、系统和系统产品

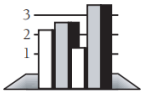
深入阅读　关于这个观点的另一种解释，请参阅《*The Mythical Man-Month*》（人月神话）（Brooks1995）第 1 章。

代码行数和团队规模并不是影响项目规模的唯一因素。一个更微妙的影响是最终软件的质量和复杂性。初版 Gigatron（即 Gigatron Jr.）可能只花了一个月的时间来编写和调试。它是由一个人编写、测试和撰写文档的单一程序。既然 2500 行的 Gigatron Jr.只花了一个月的时间，为何完善的、25000 行的 Gigatron 要花 20 个月？

最简单的软件是单一的“程序”，由开发它的人自用，或者非正式地由其他几个人使用。

更复杂的程序是软件“产品”，一个旨在供原开发者之外的其他人使用的程序。软件产品的使用环境和产品创建时的环境不同。它在发布之前经过了全面的测试，要有文档，并能由其他人维护。软件产品的开发成本约为软件程序的三倍。

另一个层次的复杂性发生在需要开发一组协同工作的程序的时候。这样的一组程序称为软件“系统”。开发系统比开发简单的程序更复杂，因为这涉及到开发各个部分之间的接口的复杂性，而且需要小心翼翼地集成各个部分。总的来说，系统的成本也是简单程序的三倍左右。



HARD DATA 而“系统产品”在开发完毕后，它既有单一“产品”的完善度，也有“系统”的多个部分。系统产品的成本约为简单程序的 9 倍（Brooks 1995, Shull et al. 2002）。

没有意识到程序、产品、系统和系统产品之间在完善度和复杂度上的差异，是导致估计错误的一个常见原因。程序员如果用他们构建“程序”的经验来估计构建“系统产品”的时间表，可能会低估近 10 倍的时间。在考虑下面的例子时，请同时参考图 27-3。若用你写 2K 行代码的经验来估计开发一个 2K 程序所需的时间，你的估计只占实际开发一个程序的全部活动所需总时间的 65%。写 2K 行代码所花的时间并不等同于创建包含 2K 行代码的一个完整程序的时间。如果不考虑非构建活动所需的时间，开发会比你估计的多花 50%的时间。

随着规模的扩大，“构建”在项目的总工作量中变得越来越小。如果只依据构建经验来估计，估计的误差就会增加。如果用自己 2K 行代码的构建经验来估计开发一个 32K 行程序所需的时间，你的估计将只占总时间的 50%；开发过程的耗时比你估计的多出 100%。

这里的估计错误完全归咎于你不了解规模对于开发大型程序的影响。除此之外，如果没有考虑到一个“产品”而不是一个单纯的“程序”所需的额外完善度，这个误差很容易增加三倍或更多。

方法论和规模

方法论（methodologies）有各种规模（size）的项目中都有使用。在小项目中，方法论往往是随意和出于本能的。而在大项目中，它们往往是严格和精心规划的。

有的方法论很松散（loose），以至于程序员甚至没有意识到他们正在使用它们。一些程序员认为方法论过于死板（rigid），表示自己绝不会去碰。虽然程序员可能确实没有有意识地选择一个方法论，但任何编程方法都构成了方法论，无论这种方法论是多么无意识或原始。虽然没啥创意，但仅仅是早上起床和上班就是一种基本的方法论。坚持不去碰方法论的程序员实际只是在明确地避免选择一种——事实上，没人能完全避开它们。

KEY POINT 正式方法并不总是那么有趣，而且如果被错误地应用，它们产生的开销会非常大，以至于得不偿失。但是，大项目的复杂性要求我们更有意识地关注方法论。建造摩天大楼和搭狗窝需要不同的方法。不同规模的软件项目亦是如此。在大项目中，无意识的选择不足以完成任务。成功的项目规划者会明确地为大项目选择策略。

在社交场合，活动越正式，你的衣服就越不舒服（高跟鞋、领带等等）。在软件开发中，项目越正式，文书越多，这样才能证明你已经做好了功课。Capers Jones 指出，一个 1000 行代码的项目，平均约 7%的精力要花在文书工作上。而一个 10 万行代码的项目，平均约 26%的精力要花在文书工作上（Jones 1998）。

这些文书工作并不是为了单纯写文档的乐趣而产生的。它的产生是如图 27-1 所示的现象的直接结果：要协调的人越多，协调他们需要的正式文档越多。

任何这样的文档都不是为了创建而创建。例如，编写配置管理计划的目的并不是锻炼你的写作能力。编写计划的目的在于迫使自己仔细考虑配置管理并向其他人解释你的计划。文档只在计划和构建软件系统时所做的实际工作的一种有形的副产品。如果觉得自己正在照本宣科地写一些常规文档，那肯定是出了问题。

KEY POINT 就方法论而言，“更多”并不意味着更好。Barry Boehm 和 Richard Turner 对敏捷和计划驱动的方法论进行了对比，他们警告说，相较于从一个包罗万象的方法开始，并针对小项目缩减规模，更好的做法是从小方法开始，并针对大型项目进行扩充（Boehm 和 Turner 2004）。一些软件专家会谈论所谓的“轻量级”（lightweight）和“重量级”（heavyweight）方法论，但在实践中，关键在于考虑项目的具体规模和类型，然后找到“适量级”（right-weight）的方法论。

更多资源

通过以下资源进一步探索本章的主题。

Boehm, Barry and Richard Turner. *Balancing Agility and Discipline: A Guide for the Perplexed.* Boston, MA: Addison-Wesley, 2004。两位作者描述了项目规模对使用敏捷方法和计划驱动方法会有怎样的影响，还探讨了其他一些与敏捷和计划驱动有关的话题。

Cockburn, Alistair. *Agile Software Development*. Boston, MA: Addison-Wesley, 2002。该书第 4 章描述了与选择适当的项目方法论有关的问题，包括项目规模。第 6 章介绍了 Cockburn 的 Crystal 方法论，亦即用于开发不同规模、不同紧要程度的项目而采取的一系列方法。

Boehm, Barry W. *Software Engineering Economics*. Englewood Cliffs,NJ: Prentice Hall, 1981。作者在本书中广泛探讨了项目规模和软件开发过程中的其他变量对成本、生产率和质量的影响。本书探讨了项目规模对构建和其他活动的影响。其中第 11 章非常精彩地解释了成本因软件因规模扩大而增加的现象。有关项目规模的其他信息则散布于本书的其他章节。作者在其 2000 年出版的 *Software Cost Estimation with Cocomo II* 一书中为 Cocomo 评估模型给出了更多最新的参考资料，但前一本书就该模型的背景讨论更为深入，而且这些信息仍然适用。

Jones, Capers. *Estimating Software Costs*. New York, NY: McGraw-Hill, 1998。本书用大量表格和图表深入解析软件开发生产率的根源。如果特别关注项目规模所带来的影响，不妨参考 Jones 在 1986 年出版的 *Programming Productivity* 一书，第 3 章的“The Impact of Program Size”一节对此做了精彩论述。

Brooks, Frederick P., Jr. *The Mythical Man-Month: Essays on Software Engineering*, *Anniversary Edition* (2d ed.). Reading, MA: Addison-Wesley,1995。作者是 IBM OS/360 项目的开发经理，这是一个花了 5000 人年的大项目。他讲述了与小型团队和大型团队相关的管理问题，在这个引人入胜的论文集当中，还对首席程序员团队进行了特别生动的描述。

DeGrace, Peter, and Leslie Stahl. *Wicked Problems, Righteous Solutions: A Catalogue of Modern Software Engineering Paradigms*. Englewood Cliffs, NJ: Yourdon Press, 1990。书如其名，其中

收编了众多软件开发方法。就像本章一直在强调的那样，采用的方法应随项目规模而变，DeGrace 和 Stahl 更清晰地说明了这一点。第 5 章的“Attenuating and Truncating”一节讲述了如何依据项目的规模和正式程度来定制软件开发过程。书中包含对 NASA 和美国国防部的模型讲解，还列举了大量启发性的示例。

Jones, T. Capers.“Program Quality and Programmer Productivity.”*IBM Technical Report TR 02.764* (January 1977): 42–78。也见于 Jones 的 *Tutorial: Programming Productivity: Issues for the Eighties, 2d ed*. LosAngeles, CA: IEEE Computer Society Press, 1986。作者首次深入分析了在大型项目与小型项目中，导致工作量分配支出模式不同的原因，深入探讨了大型项目和小型项目的诸多不同之处，包括需求分析和质保度量。内容有点过时，但还是很有趣。

要点回顾

随着项目规模的扩大，沟通交流需要有保障。大多数方法论的要点在于减少沟通问题，而一个方法论的存亡也取决于它是否能有效促进沟通。

其他所有条件都等同的情况下，大项目的生产率将低于小项目。

其他所有条件都等同的情况下，大项目的每千行代码错误率高于小项目。

在小项目中一些看起来理所当然的活动，在大型项目中则必须认真规划。随着项目规模的扩大，构建活动越来越不占主导。

与缩减“重量级”方法论相比，对“轻量级”方法论进行扩充往往效果更佳。在所有方法论当中，最有效的当属“适量级”方法论。

第 28 章　管理构建

内容

- 28.1 鼓励良好的编码实践
- 28.2 配置管理
- 28.3 评估构建进度表
- 28.4 度量
- 28.5 以人为本，善待每一位程序员
- 28.6 向上管理

相关章节

- 构建的前期准备：第 3 章
- 确定要开发的软件类型：第 3.2 节
- 程序规模：第 27 章
- 软件质量：第 20 章

过去几十年里，软件开发的管理一直是一项艰巨的挑战。如图 28-1 所示，软件项目管理的一般主题超出了本书的范围，但本章讨论了直接适用于构建的一些具体的管理主题。如果你是开发人员，本章将帮助你了解管理者需要考虑的问题。如果你是管理者，本章将帮助你了解管理在开发人员看来是怎样的，以及如何有效地管理构建。由于本章涵盖了广泛的主题，所以还用几个小节描述了可以去哪里获得更多信息。

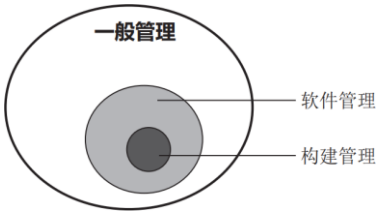


图 28-1　本章讨论了和构建相关的管理主题

如果对软件管理感兴趣，请务必阅读第 3.2 节“确定要开发什么类型的软件”，以理解传统顺序开发方法和现代迭代方法之间的区别。还要阅读第 20 章“软件技术概述”和第 27 章“项目规模对构建的影响”。质量目标和项目规模都会极大地影响一个特定的软件项目的管理方式。

28.1 鼓励良好的编码实践

由于代码是构建的主要产出，所以对构建进行管理的一个关键问题是“如何鼓励良好的编码实践？”一般来说，从管理岗位强制要求一套严格的技术标准并不是一个好主意。程序员往往认为管理者处于技术进化的低级阶段，介于单细胞生物和冰河时期灭绝的长毛象之间，如果要有编程标准，程序员必须买账才行。

如果项目中要有人定义标准，请让一个受人尊敬的架构师来定义，而不是由管理者。软件项目在“专业技术等级结构”和“权力等级结构”上的运作方式是差不多的。若架构师被认为是项目的精神领袖，则项目团队通常会遵循其制定的标准。

如选择这种方式，请确保架构师真的受尊重。有的时候，项目架构师只是因为呆的时间够久而资深而已，早就和生产编码问题脱节了。如果这种“架构师”定义的标准与正在做的工作脱节，程序员会有怨气。

设定标准时的注意事项

标准在某些组织中比在其他组织更有用。一些开发者欢迎标准，因其可减少项目中随性而为的差异。如团队抵制严格的标准，可考虑一些替代方案：灵活的指导原则、一系列建议而不是指导原则或者一组体现最佳实践的例子。

促进良好编码的技术

本节描述了为实现良好的编码实践而采用的几种技术，这些编码实践不像死板的编码标准那样严厉。

关联参考　结对编程的详情请参见第 21.2 节。

为项目的每个部分都分派两个人　如果每行代码都由两个人共同完成，就可保证至少有两个人认为这段代码是可以正常工作的，而且是清晰可读的。两人组队的机制包括结对编程（pair programming）、师徒制（mentor-trainee pairs）以及伙伴系统评审（buddy-system reviews）等。

关联参考　代码评审的详情请参见第 21.3 节和第 21.4 节。

逐行评审代码　代码评审（code review）通常涉及程序员本人和至少两名评审者。这意味着至少会有 3 个人来逐行阅读全部代码。同行评审（peer review）的另一种说法是“同行压力”（peer pressure）。除了能够为原程序员离开项目时提供一层安全保障外，评审还有助于提升代码质量，因为程序员知道会有其他人看自己的代码。即使你的组织并未明确制订编码标准，评审也会以一种微妙的方式促成小组的编码标准：小组成员会在评审过程中做出一些决定，而随着时间的推移，小组会衍生出自己的编码标准。

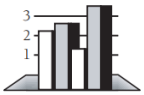
要求代码签核　在其他领域里，技术图纸需要由工程师主管来批准和签字。签字表明，在该工程师的认知范围内，确认这些图纸在技术上可行并且没有错误。一些组织也用相同的方法来管理代码。在认定代码完成之前，高级技术人员必须在代码清单上签字。

选取优秀的代码示例作参考　但凡优秀的管理，其中一个重要方面就是清晰表达自己的目标。传达目标的一种方式是向程序员传递一些优秀的代码或者公开张贴出来。这样可通过清

晰的样例来说明质量目标。类似地，编码标准手册也可以主要由一份“最佳代码清单”构成。将一些代码清单标识为“最佳”，就树立了一个其他人可以遵循的榜样。这种手册比一大堆文字的标准手册更容易更新，而且很容易将编码风格中的细微之处表达清楚，这是文字描述难以做到的。

关联参考　编程在很大程度上就是将你的工作与别人进行沟通。详情请参见第 33.5 节和第 34.3 节。

强调代码是公有财产　程序员有时认为自己写的代码是“自己的代码”，就像私有财产一样。虽然这是他们的工作成果，但代码属于项目的一部分，应提供给项目组中其他任何成员根据需要自由获取。即使其他时间不会公开，但至少在代码评审和维护期间应该让别人看到。



HARD DATA 据报道称，一个最成功的项目花 11 个工作年开发了 83000 行代码。在投入运行后的前 13 个月里，只找出了一个导致系统故障的错误。要知道，该项目是在 20 世纪 60 年代后期完成的（当时没有联机编译或交互式调试工具），所以这个成就更加引人注目。该项目的生产率——在 20 世纪 60 年代末期，每工作年写 7500 行代码——即使按今天的标准来看也仍然令人印象深刻。该项目的首席程序员在报告中陈述，项目成功的关键因素之一就是将所有计算机的运行记录（无论出错与否）都视为公产而非私产（Baker and Mills 1973）。这一理念延伸到好多现代的场景中，包括开源软件（Raymond 2000）和极限编程（XP）所倡导的集体所有权（Beck 2000）以及其他应用场景。

奖励好的代码　使用你的组织的奖励制度来激励良好的编码实践。开发激励系统时记住以下几点。

- 奖励是程序员想要的东西。（许多程序员认为“干得好”形式的奖励是令人厌恶的，特别是当它们来自非技术岗的经理时）。
- 获得奖励的代码应该是特别好的。如果给一个其他人都知道做得不好的程序员颁奖，你看起来就像荷马·辛普森（Homer Simpson）试图运行一个核反应堆[11]。这个程序员是否有合作精神或总是按时上班并不重要。如果你的奖励和技术脱节，你就会失去可信度。如果在技术上不熟练，无法对代码做出良好判断，那就别做！根本就不要颁奖，或者让你的团队选择获奖者。

一个简单的标准　如果你负责管理一个编程项目，而且你有编程背景，一个简单而有效的激励良好工作的技巧是宣布：“我必须能阅读和理解为该项目写的任何代码”。管理者不是最炙手可热的技术专家，这可能反而是一种优势，因为它能阻止某些“聪明”（clever）或“难以理解”（tricky）的代码。

[11] 荷马·辛普森是美国电视动画《辛普森一家》中的一名虚构角色，辛普森一家五口中的父亲。虽然他贪食、懒惰、常惹事故且非常愚蠢，但却偶尔能展现出自身的才智与真实价值，譬如对自己家人的热爱及保护。——译注

本书的角色

本书大多数篇幅都在讨论良好的编程实践。但是，本书的目的并不是为死板而严格的标准做辩护，更不是打算作为一套这样的标准来使用。相反，本书旨在提供一个讨论的基础，帮助你理解什么是好的编程实践，并帮助你在自己的环境中识别那些真正有益的实践。

28.2 配置管理

软件项目是动态变化的。代码在变，设计在变，需求也在变。更重要的是，需求的变化会导致设计的更多变化，而设计的变化会导致代码和测试用例的更多变化。

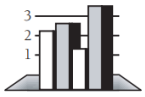
什么是配置管理？

配置管理（configuration management）是指标识出项目工件（project artifacts）并以系统化的方式处理变更，使项目能随时间的推移保持其完整性。它的另一个名字是是“变更控制”（change control）。它包含多项技术：评估提议的变更请求、跟踪变更、保留系统在不同时间点的多个历史版本。

如果对需求的变更不加以控制，就会为系统中最终会被移除的部分写代码。另外，还会写出一些和系统新增部分不兼容的代码。可能要等到集成的时候，才发现大量不兼容。这会导致相互指责的局面，因为没人真正知道到底发生了什么。

如果对代码的变更不加以控制，可能出现别人正在修改一个子程序，而你也跑去修改的情况。这会造成很难将自己的更改和别人的更改合并。不加控制的代码变更会使代码表面上看起来似乎做了更多的测试，但实则不然。测试过的很可能是旧的、未更改的版本；而修改过的版本可能还没有测试。如果没有良好的变更控制，可能会去修改一个子程序，发现新错误，但再也无法恢复到最初可以正常工作的旧版本。

这些问题产生的时间并不确定。如果不系统化地对变更加以控制，那么就像是在迷雾中随意游走，而不是直接朝着清晰的目标迈进。缺乏良好的变更控制，与其说是在开发代码，还不如说是在浪费时间。配置管理可帮助你有效地利用时间。



HARD DATA 虽然配置管理有显而易见的必要性，但几十年来，相当多的程序员并不愿意做配置管理。二十世纪八十年代的一份调查表明，超过三分之一的程序员甚至还不熟悉这一概念（Beck and Perkins 1983），而且也几乎没有迹象表明这一状况有所改变。SEI 软件工程研究所的一份研究表明，在采用非正式软件开发实践的组织当中，有适当配置管理的不到 20%（SEI 2003）。

配置管理并不是由程序员发明的，但由于程序项目相当不稳定，所以配置管理对程序员极其实用。应用于软件项目的配置管理通常也称为“软件配置管理”（Software Configuration Management，SCM）。SCM 关注程序的需求、源代码、文档和测试数据。

关联参考　参见 27 章更详细地了解项目规模对构建的影响。

SCM 的系统性问题是过度控制。阻止交通事故的最可靠的方法是阻止所有人开车，而阻止软件开发问题的一个可靠方法是阻止所有的软件开发。虽然这也是一种控制变更的方法，却无疑是开发软件中极其糟糕的方法。必须认真做 SCM 计划，使其成为一种资产而非负担。

在小型单人项目中，除了计划一下非正式的周期性备份，不用 SCM 也行。尽管如此，配置管理仍然非常有用（而且事实上，我在写这本书时就用到了配置管理）。在一个大型的 50 人项目里，或许就要采用最全面的 SCM 方案，包括相当正式的备份规程、控制需求变更和设计变更以及对文档、源代码、内容资源、测试用例和其他项目工件进行全面控制。如项目规模不大也不小，就需要在两种极端情况之间确定正规程度。以下几个小节描述了实现 SCM 时的几个选项。

需求和设计变更

关联参考　有的开发方法能更好地支持变更。详情参见第 3.2 节。

在开发过程中，一定会萌发很多关于如何改进系统的想法。如果每有一个想法就实现相应的变更，很快就会发现自己走上了软件开发的不归路：虽然系统在发生变化，离终点却渐行渐远。以下是一些用于控制设计变更的指导原则。

遵循系统化的变更控制规程　如第 3.4 节所述，在面临很多变更请求时，系统化的变更控制规程简直就是天赐之物。通过建立一套系统化的规程，就能明确变更应放在对整个项目最为有利的场景下考虑。

整体处理变更请求　人们的倾向于一有想法就去实现那些容易处理的变更。但这样处理变更的问题在于，好的变更可能会被错过。如果在项目进行到 25%时想起一项简单的变更，当时一切都按计划顺利进行，开发者就会去实现该变更。如果在项目进行到 50%并且进度已经滞后时又想到另外一项简单的变更，或许就不会去实现它。等到项目进行到最后，时间差不多用完，这时即使后一项变更要比前一项好上 10 倍，也都没有机会去做了。一些最好的变更就这样错过了，原因仅仅是想到它时才发现为时已晚。

解决该问题的一个办法是先记录所有想法和建议，无论它们实现起来有多容易。做好记录，直到有时间才处理。到那时，作为一个整体来处理，从中选出最有利的来实现。

评估每项变更的成本　每当客户、老板或自己试图修改系统时，不妨评估一下需要花多少时间，包括对修改的代码进行评审以及重新测试整个系统的时间。在评估时，还要把该变更所导致的连锁反应（需求、设计、编码、测试以及用户文档的变更）的耗时都考虑进去。让所有利益相关方都知道软件是错综复杂交织在一起的，而且评估变更的耗时非常有必要，哪怕要做的变更表面上微不足道。

无论首次提出变更请求时自己觉得有多么乐观，都不要信口开河，随便给出评估。这种评估通常带有 2 倍甚至更高的误差。

关联参考　参见第 3.4 节中的“在构建期间处理需求变更”从另一个角度看待变更处理。参见第 24 章了解如何安全地处理代码变更。

提防大量的变更请求　虽然某种程度的变更不可避免，但大量的变更请求是非常重要的警告信号，表明需求、架构设计或顶层设计做得不够好，无法有效地支撑有效的构建。对需求或

架构进行返工表面上看起来代价昂贵，但相较于多次构建软件，或者将实际并不需要的功能的代码丢掉相比，仍然值得考虑。

成立变更控制委员会或适合自己项目的类似组织 变更控制委员会（change-control board）的职责是在收到变更请求后进行去芜存菁。任何提议变更的人都要将变更请求提交给变更控制委员会。这里的“变更请求”（change request）指的是任何可能改变软件的请求：对于一个新功能的想法、对现有功能的更改以及一份“错误报告”（可能报告了真正的错误，也可能没有）等等。委员会成员定期开会评审提交的变更请求。每一项变量都可能批准、驳回或推迟。变更控制委员会被认为是设定需求变更的优先级以及控制需求变更的最佳实践；然而，在商业环境中仍然应用得不够广泛（Jones 1998，Jones 2000）。

警惕官僚主义，但不要因为惧怕官僚主义而排斥有效的变更控制 缺乏规范的变更控制是当今软件行业面临的重大管理难题之一。在被认为进度落后的项目中，有相当大的比例本可按时完成——只要考虑到未做跟踪却同意变更所带来的影响。糟糕的变更控制会导致变更堆积如山，从而破坏项目状态的可见性、长期可预测性、项目计划、风险管理（这一点很重要）以及一般意义上的项目管理。

变更控制往往会滋生官僚主义，所以，重要的是想办法简化变更控制过程。如果不愿意采用传统的变更请求，那么不妨设置一个简单的“Change Board”（变更委员会）电子邮件别名，然后让大家将变更请求发送到该地址。或者，让人们在变更委员会的会议上进行互动，提出各自的变更建议。一种特别有效的方法就是将所有的变更请求都作为“缺陷”记录到缺陷跟踪软件中。有的纯化论者会将这些变更归为“需求缺陷”，或者你也可以将它们归为变更而不是缺陷。

既可以正式实施变更控制委员会制度，也可以设立一个产品规划小组（Product Planning Group）或者作战指挥委员会（War Council），由他们履行变更控制委员会的传统职责。再或者，也可以指派某个人来担任变更最高统帅。无论怎么称呼，都要去做！

KEY POINT 我偶尔会看到一些项目因为变更控制的粗暴实施而饱受折磨。但十倍以上的项目是因为根本就不存在有意义的变更控制才饱受折磨。变更控制的精华在于确定什么最重要，所以不要因为对官僚主义的恐惧而阻止你享受它的许多好处。

软件代码变更

另一个配置管理问题是对源代码的控制。如果变更了代码，然后出现了一个似乎与你所做的变更无关的新错误，你可能会将新版本的代码与旧版本的代码进行比较，以寻找错误根源。如果找不到答案，你可能想看更早的版本。如果有版本控制工具来跟踪多个版本的源代码，这种历史考究就很容易了。

KEY POINT **版本控制软件** 好的版本控制软件很好用，你几乎注意不到它在发挥作用。它对团队项目特别有帮助。版本控制的一种方式是锁定源文件，一个文件每次只能有一个人修改。通常，在需要处理某个文件的源代码时，会将该文件从版本控制中签出（check out）。如当前已由别人签出，会通知你当前不能签出。签出后，可像在没有版本控制的情况下一样处理

它，直到准备好签入（check in）。另一种方式是允许多人同时处理同一个文件，并在代码签入时处理合并修改的问题。无论哪种情况，在签入文件时，版本控制都会问你为什么要改变它，而你需要输入理由。

通过这种适度的投入，可以获得几个大的好处：

- 不会在别人修改文件时候跟他/她发生冲突（或者至少在发生冲突时会知道）。
- 可以很容易地将项目中所有文件的副本更新为最新版本，通常一个指令即可。
- 可回溯到任何文件的任何版本，这些文件曾被签入版本控制系统。
- 可获得任何文件的任何版本的改动清单。
- 不必担心个人备份，因为有一个版本控制副本的安全网。

版本控制对于团队项目是不可缺少的。当版本控制、缺陷跟踪和变更管理被合并到一起时，它变得更加强大。Microsoft 的应用程序部门认为，其专有的版本控制工具是一项“主要竞争优势”（Moore 1992）。

工具版本

对于某些类型的项目，可能需要重建当初在创建软件每个特定版本时的环境，包括编译器、链接器、代码库等等。在这种情况下，应将所有这些工具也纳入版本控制。

机器配置

许多公司（包括我自己的）都因创建标准化的开发机器配置而受益。具体的做法是创建一个标准开发者工作站的磁盘镜像，其中包括所有常见的开发者工具、办公应用等等。然后，将该镜像加载到每个开发人员的机器上。创建标准化的配置有助于避免因为配置的少许差异、工具版本的差异而造成一系列问题。相较于必须单独安装每个软件，标准化的磁盘镜像还能大幅简化新机器的安装与配置。

备份计划

备份计划不是什么新概念；就是定期备份自己的工作。如果手写一本书，你不会将写好的书页堆在门廊上。如果这样做了，它们可能会被雨淋或被风吹走，邻居家的狗也可能会借它们当睡前读物。你会把它们放在安全的地方。软件不那么有形，所以更容易忘记自己在一台机器上拥有巨大价值的东西。

电脑数据可能发生许多事情：磁盘可能失效；你或其他人可能意外删除关键文件；一个愤怒的员工可能破坏你的机器；或者你可能因为盗窃、水灾或火灾而失去一台机器。采取措施来保护自己的工作。备份计划应包括定期进行备份和定期将备份转移到异地存储，它应包括项目的所有重要资料——文件、图形和笔记——以及源代码。

制定备份计划时，一个经常被忽视的方面是要对备份程序进行测试。试着在某个时间点做一次还原，确定备份包含你所需要的一切，而且能正常还原。

项目完成后，建立一个项目档案。保存所有东西的副本：源代码、编译器、工具、需求、设计、文档——一切重新创建产品所需的东西。把它们放在安全的地方。

检查清单：配置管理

概要

- 软件配置管理计划是设计用来帮助程序员并将开销最小化吗？
- 软件配置管理（SCM）避免了对项目的过度控制吗？
- 变更请求是进行了分组，当作一个整体处理的吗？无论采用非正式的方法（例如创建一份待定变更的清单）还是更系统化的方法（例如设立变更控制委员会）。
- 系统化地评估了提交的每一项变更请求对于成本、进度和质量的影响了吗？
- 将重大的变更视为需求分析不够完善的一个警示了吗？

工具

- 采用版本控制软件来帮助配置管理了吗？
- 采用版本控制软件减少团队工作中的协调问题了吗?

备份

- 定期备份所有项目资料了吗?
- 定期将项目备份数据转移到异地存储了吗?
- 所有资料（包括源代码、文档、图片和重要的笔记）都备份了吗?
- 测试过备份与还原过程吗?

配置管理的更多资源

由于本书的主题是软件构建，所以本节是以构建的视角看待变更控制。但是，变更会从各个层面上对项目产生影响。所以，一个全面的变更控制策略也需要如此。

Hass, Anne Mette Jonassen. *Configuration Management Principles and Practices*. Boston, MA: Addison-Wesley, 2003。这本书描述了软件配置管理的全景，还描述了如何将软件配置管理融入软件开发过程的实践细节。重点在于配置项的管理和控制。

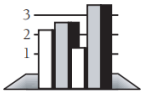
Berczuk, Stephen P. and Brad Appleton. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Boston, MA: Addison-Wesley, 2003。和 Hass 的书一样，这本书概述了 SCM，而且相当实用。作为对 Hass 那本书的补充，书中提出了一些实用的指导原则，帮助开发团队隔离和协调他们的工作。

SPMN. *Little Book of Configuration Management*. Arlington, VA: Software Program Managers Network, 1998。这是一本介绍配置管理活动的小册子，对成功的关键因素进行了定义。可从 SPMN 网站 www.spmn.com/products_guidebooks.html 免费下载。

Bays, Michael. *Software Release Methodology*. Englewood Cliffs, NJ: Prentice Hall, 1999。本书也讨论了软件管理管理，侧重于如何发布软件以投入生产环境。

Bersoff, Edward H., and Alan M. Davis. “Impacts of Life Cycle Models on Software Configuration Management”，Communications of the ACM 34, no. 8 (August 1991): 104–118。这篇文章描述了软件开发的一些较新的方法（特别是原型法）对 SCM 的影响。本文尤其适合采用敏捷开发实践的场景。

28.3 评估构建进度表



HARD DATA 软件项目管理是人类在 21 世纪所面临的一项重大挑战。评估项目的规模和完成项目所需要的工作量是软件项目管理中最具挑战性的工作内容之一。大型软件项目平均延期一年，且超出预算 100%（Standish Group 1994，Jones 1997，Johnson 1999）。在个人层面上，对预估和实际的进度表的调查显示，开发人员的估计值比实际值要外观 20%~30%（van Genuchten 1991）。这既与对项目规模和工作量糟糕的评估有关，也和开发能力不足有关。本节讨论评估软件项目时涉及的一些问题，并指出去哪里获得更多信息。

评估方法

> 深入阅读　要更多地了解进度评估技术，请参见《*Rapid Development*》的第 8 章 (McConnell 1996)以及《*Software Cost Estimation with Cocomo II*》(Boehm et al. 2000)。

可采取以下几种方法评估项目规模及其涉及的工作量。

- 使用评估软件。
- 采用算法模型，例如 Barry Boehm 提出的 Cocomo II 评估模型（Boehm et al. 2000）。
- 从外面聘请评估专家来评估项目。
- 为评估举办一次走查（walk-through）会议。
- 评估项目的每一部分，然后汇总。
- 先让人员评估各自的任务，然后汇总。
- 参考以往项目的经验。
- 保留之前的评估值，检查其准确度。据此调整新的评估值。

可从本节末尾的“软件评估的更多资源”获取有关这些方法的更多信息。下面是一个比较好的项目评估方法：

> 深入阅读　这种方法改编自《*Software Engineering Economics*》(Boehm 1981)。

建立目标　为什么需要评估？评估什么？只评估构建活动，还是评估所有开发活动？只评估项目所需的工作量，还是将休假、节假日、培训和其他非项目活动都算在内？评估需要多准确才能达到你的目标？评估需要达到什么样的确定度（degree of certainty）？乐观评估和悲观评估会产生截然不同的结果吗？

为评估留出时间，并且做出计划 草率的评估是不准确的。如果评估的是大型项目，需要将评估工作当作一个迷你项目来做，并且要花时间为评估制订迷你计划，这样才能把评估做好。

关联参考 有关软件需求的更多信息，请参阅第 3.4 节。

阐明软件需求 就像建筑师无法估算一座“相当大”的房子需要花费多少钱一样，你也无法靠谱地评估一个“相当大”的软件项目。在一个“东西”还没有被定义的时候，任何人期望你估计出建造这个“东西”所需的工作量，都是不合理的。评估之前，先定义好需求，或者计划一个初步的探索阶段。

在底层细节层面进行评估 根据已确定的目标，基于对项目各项活动的详细检查进行评估。通常，检查得越细致，评估结果越准确。根据大数定律（Law of Large Numbers），如果整体评估存在 10%的误差，那么只做一次评估，结果可能高出 10%，也可能低出 10%。但是，若将其拆分成 50 个小块再评估，某些块的结果会高出 10%，某些块会低出 10%，而这些误差趋向于相互抵消。

关联参考 在软件开发中，很难找出一个迭代没有价值的领域。迭代对评估也很有帮助。有关迭代技术的总结，请参见第 34.8 节。

采用多种评估方法并比较结果 本节开头列出的评估方法涉及几种技术。它们的评估结果并不完全一样，所以要多尝试多个，并研究不同方法所产生的不同结果。小孩们很早就知道，如果分别向父母要第 3 碗冰淇淋，比单独向父亲或母亲索要的成功率高一些。有的时候，家长也很聪明，会给出一致的回答；有时则不会。看看能从不同的评估方法获得哪些不同的结果。

不存在所有情况都适用的方法，而且它们之间的差异还具有一定的启发性。例如，写本书第一版时，我最初粗略估算只有 250~300 页。但在我最终做了深入估算时，结果是 873 页。“这恐怕不对。”我想。于是，我用一种完全不同的技术来估算，结果是 828 页。考虑到这些估算结果彼此相差约 5%，所以我得出结论，这本书将更接近 850 页而不是 250 页。这样一来，我就能相应地调整写作计划。

定期重新评估 软件项目的因素在最初的评估后会发生变化，所以做好计划定期更新。如图 28-2 所示，随着项目趋近于完成，估算的准确性应该提高。不时将实际结果与评估的结果进行比较，并据此改进对项目剩余部分的评估。

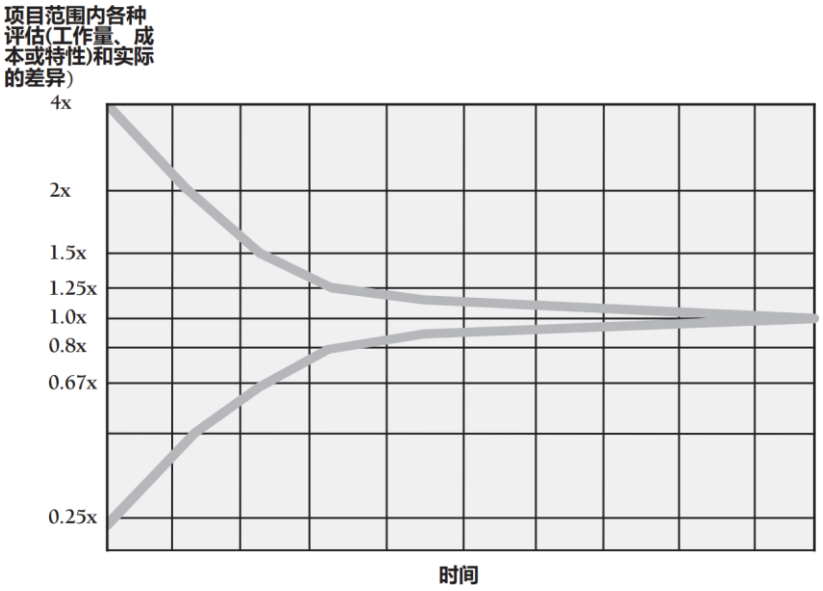


图 28-2　在项目早期创建的评估注定不准确。随着项目的进展，评估结果变得越来越准确。在整个项目中定期重新评估，并利用从每个活动中学到的知识来改进对下一个活动的评估

评估构建工作量

> 关联参考　要详细了解不同规模项目的编码工作量，请参见第 27.5 节中的“活动占比和规模”。

构建会对项目进度造成多大程度的影响，部分取决于构建活动（详细设计、编码和调试以及单元测试等）在项目中所占的比例。如上一章的图 27-3 所示，构建所占比例随项目规模而变。除非公司保留了自己的项目历史数据，否则图中展示的各种活动时间占比就是进行新项目评估时的一个很好的出发点。

关于项目中构建活动占多大比例这一问题，最好的回答就是该比例因项目和组织而变。保存组织的项目历史数据，并根据它们评估未来的项目要花费多少时间。

对进度的影响

> 关联参考　程序规模对生产率和质量的影响并非总是那么显而易见。第 27 章解释了规模对构建的影响。

对软件项目进度影响最大的就是程序的规模。但其他很多因素也会对开发进度造成影响。一些针对商业应用程序做的研究将部分因素的影响效果做了量化，如表 28-1 所示。

表 28-1 影响软件项目工作量的因素

影响因素	潜在有益影响	潜在有害影响
集中开发 vs. 分布式开发	−14%	22%
数据库规模	−10%	28%
文档符合项目需求	−19%	23%
解释需求时的灵活性	−9%	10%
积极应对风险	−12%	14%
使用语言和工具的经验	−16%	20%
人员连续性（流动性）	−19%	29%
平台稳定性	−13%	30%
过程成熟度	−13%	15%
产品复杂度	−27%	74%
程序员的能力	−24	34%
需要的可靠性	−18%	26%
需求分析师的能力	−29%	42%
对重用的要求	−5%	24%
最先进的应用程序	−11%	12%
存储限制（将消耗多少可用存储空间）	0%	46%
团队凝聚力	−10%	11%
团队在该应用领域的经验	−19%	22%
团队在该技术平台上的经验	−15%	19%
时间限制（来自应用程序自身）	0%	63%
对软件工具的使用	−22%	17%

来源：《*Software Cost Estimation with Cocomo II*》(Boehm et al. 2000)

以下是一些影响软件开发进度但不易被量化的因素，它们选自 Barry Boehm 的《*Software Cost Estimation with Cocomo II*》(2000)和 Capers Jones 的《*Estimating Software Costs*》(1998)。

- 需求开发人员（requirements developer）的经验和能力
- 程序员的经验和能力
- 团队的动力
- 管理质量
- 重用代码量
- 人员流动性
- 需求稳定性
- 客户关系质量
- 用户对需求的参与度
- 客户对此类应用程序的经验
- 程序员对需求开发的参与度
- 计算机、程序和数据的分级安全环境
- 文档量
- 项目目标（进度、质量、可用性以及其他可能的目标）

每一项因素都可能极其重要，因此需要与表 28-1 列出的因素（这里的部分因素也包括在其中）一同考虑。

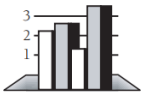
评估与控制

> “重要的问题是：要预测，还是要控制？”——Tom Gilb

在为按时完成软件项目而制定的计划中，评估是非常重要的一部分。一旦确定交付日期和产品规范，剩下的主要问题就是如何控制好人员和技术资源的开销，以便能够按时交付产品。从这个角度来说，最初评估时准确度的重要性，远远比不上后期为了如期交付而成功控制好资源的重要性。

进度落后怎么办？

本章前面提到，项目平均会超出原定时间的 100%。进度落后时，增加时间通常并不可行。如果可行，就那么做。否则可以尝试以下一种或多种解决方案：



HARD DATA **希望能迎头赶上** 项目落后于进度时，人们通常的反应都是乐观的。典型的理由是：“需求所花的时间比我们预期的要长一些，但现在需求已经固定，所以我们一定会在后面省下时间。我们将在编码和测试阶段把时间补回来。”这种情况几乎不会发生。一个对 300 个软件项目展开的调查显示，越接近项目后期，延误和超支的现象越严重（van Genuchten 1991）。项目并不能在后期把时间补回来，而是越来越落后。

扩充团队 根据 Brooks 定律，向一个已经延期的软件项目增加人手只会使其进一步延期（Brooks 1995）。这无异于火上浇油。Brooks 的解释很有说服力：新手需要先花时间熟悉项目，然后才能有高生产率。培训他们要占用已受训人员的时间。而且，如果只是增加人员数量，会导致项目沟通的复杂度和数量随之增加。Brooks 指出，一个女人可以在 9 个月内怀胎生子这一事实，并不意味着 9 个女人可以在 1 个月内生个孩子出来。

毫无疑问，Brooks 定律应该引起更广泛的关注。人们往往更愿意往一个项目中增派人手，希望他们可以使项目按时完成。管理者需要理解的是，开发软件不同于搬砖，不是说干活的人越多，完成的工作越多。

然而，往延期的项目增加人手会使其延期更久这一论断过于简单，它掩盖了这样的事实：在某些场景下，向延期的项目增加人手是可能提速的。正如 Brooks 在分析这一定律时所说的，若项目中的任务不可拆分而且不能单独完成，那么增加人手也无济于事。但如果项目中的任务可以拆分，就可以将其细分，然后分配给不同的人来做，甚至可以分配给项目后期才加入的成员。其他研究人员已明确了一些应用场景，在这些场景下可以向延期的项目增加人手，而不会导致项目进一步延期（Abdel-Hamid 1989，McConnell 1999）。

> 深入阅读 参见《*Rapid Development*》(McConnell 1996) 的第 14 章了解只构建最需要的功能的观点。

缩减项目范围 缩减项目范围这个强大的方法常常会被人们忽视。如去掉一项特性，也就消除了相应的设计、编码、调试、测试和文档工作，同时也移除了该特性和其他特性之间的接口。

最初计划产品时，需要将产品的功能划分成“必须有”、“有了更好”和“可选”三类。如进度落后，就调整“可选”和“有了更好”的优先级，丢弃那些最不重要的。

如果做不到完整移除某项特征，可提供该相同功能的简化版本。或许还可以按时交付一个尚未进行性能调优的版本。还可以提供一个版本，其中最不重要的功能可以实现得相当粗略。还可以放宽对速度的要求，因为提供一个速度缓慢的版本更容易。还可以放宽对空间的要求，因为提供一个占用更多内存的版本更容易。

重新评估实现最不重要的特性的开发时间。在两小时、两天或两周之内能提供什么功能？花两周打造的版本比花两天打造的版本好在哪里？或者花两天打造的版本比花两小时打造的版本好在哪里？

软件评估的更多资源

下面列出了有关软件评估的更多资源：

Boehm, Barry, et al. *Software Cost Estimation with Cocomo II*. Boston, MA: Addison-Wesley, 2000。这本书描述了 Cocomo II 评估模型的来龙去脉，这无疑是当今最流行的模型。

Boehm, Barry W. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, 1981。这本老书对软件项目评估做出了详细的描述，内容比 Boehm 上面那本新书更通用。

Humphrey, Watts S. *A Discipline for Software Engineering*. Reading, MA: Addison-Wesley, 1995。这本书的第 5 章讲述了 Humphrey 的探查法，该方法可用于开发者个人工作评估。

Conte, S. D., H. E. Dunsmore, and V. Y. Shen. *Software Engineering Metrics and Models*. Menlo Park, CA: Benjamin/Cummings, 1986。这本书的第 6 章包含一份对各种评估技术的综述，包括评估的历史、统计模型、基于理论的模型以及复合模型。这本书还针对一个项目数据库演示了各种评估方法的应用，并将评估结果与项目的实际用时进行了对比。

Gilb, Tom. *Principles of Software Engineering Management*. Wokingham, England: Addison-Wesley, 1988。这本书第 16 章的标题“评估软件特性的 10 个原则”有点调侃的意思。作者反对进行项目评估，赞成进行项目控制。他指出，人们并不真的想准确预测，但确实想控制最终的结果。作者给出了 10 个原则，用来控制项目以使其满足最后期限、成本目标或者其他项目目标。

28.4 度量

软件项目可通过许多方式来度量。以下是对过程进行度量的两个非常充分的理由：

- 

任何项目特性都能找到一种合适的度量方法，而且肯定比完全不度量好。度量可能不是完全精确的，可能很难进行，而且可能需要随着时间的推移而改进，但度量会让你掌控软件开发过程，不度量就不可能掌控（Gilb 2004）。

- 用于科学实验的数据必须量化。你能想象一个科学家因为一组小白鼠比另一组小白鼠“似乎更容易生病”而建议禁止一种新的食品吗？这太荒谬了。你会要求一个量化的理由，例如“吃了新食品的小白鼠比没吃的小白鼠每月多病 3.7 天”。为了评估软件开发方法，必须对这些方法进行度量。像“这种新方法似乎生产率更高”这样的陈述是不够的。

> “测得出，才完得成。”（What gets measured, gets done）—Tom Peters

留意度量的副作用 度量具有激励作用。人们在意任何被度量的东西，假设它是用来评价他们的。仔细选择你所度量的东西。人们倾向于关注那些被度量的工作，忽略那些没有的。

反对度量相当于说最好不要知道项目中真正发生了什么 度量项目的某个方面时，会知道一些你从前不知道的事情。可以看到这个方面是变大、变小还是保持不变。度量为你提供了至少项目的这一方面的窗口。在对度量进行完善之前，这个窗口可能是小和模糊的，但总比没有好。如果仅仅因为某些度量结果不确定而反对所有度量，就相当于因为有些窗户看出去是阴天就反对所有窗户。

软件开发过程的几乎任何方面都可以度量。表 28-2 列出了其他从业人员认为有用的一些度量。

表 28-2 有用的软件开发度量

类型	描述
规模	编写的代码总行数 注释总行数 类或子程序总数 数据声明总数 空行总数
缺陷跟踪	每个缺陷的严重程度 每个缺陷的位置（类或子程序） 每个缺陷的根源（需求、设计、构建、测试） 修正每个缺陷的方式 每个缺陷的责任人 修正每一个缺陷所影响的代码行数 修正每一个缺陷所花费的工作时间 找到每一个缺陷的平均用时

	修正每一个缺陷的平均用时 修正每一个缺陷的尝试次数 修正缺陷而引发的新错误数
生产率	项目花费的工间 编写每个类或子程序花费的工间 每个类或子程序修改的次数 项目花费的金钱 编写每行代码花费的金钱 修正每个缺陷花费的金钱
整体质量	缺陷总数 每个类或子程序的缺陷数 每千行代码的平均缺陷数 平均故障间隔时间 编译器检测到的错误数
可维护性	每个类的 public 子程序数量 传递给每个子程序的参数数量 每个类的 private 子程序以及/或者 private 变量的数量 每个子程序使用的局部变量的数量 每个类或者子程序调用子程序的数量 每个子程序中决策点的数量 每个子程序中控制流程的复杂度 每个类或者子程序中的代码行数 每个类或者子程序中的注释行数 每个类或者子程序中数据声明的数量 每个类或者子程序中空白行的数量 每个类或者子程序中 goto 语句的数量 每个类或者子程序中输入语句或者输出语句的数量

大多数度量数据都可用当前已有的软件工具来收集。我们通过全书在多个地方的讨论解释了每一种度量为什么有用。目前，大多数度量都不是特别适合对程序、类和例程做出精细的区

分（Shepperd and Ince 1989）。它们主要用于识别“出乎寻常”的子程序；子程序中的异常度量数据是一个警告信号，表明应重新检查该子程序，找出质量出乎寻常偏低的原因。

不要一开始就试图收集所有可能的度量数据，否则会淹没于过度复杂的数据中，以至于无法弄清楚其中的含义。从一组简单的度量开始，例如缺陷数、工作月数、总费用和代码总行数。在所有项目中对度量进行标准化。然后，随着对自己要测量的东西的理解程度的提高，对它们进行完善和补充（Pietrasanta 1990）。

确保收集数据是出于一个目的。设定目标，确定需要提出哪些问题以达到目标，然后通过度量来回答这些问题（Basili 和 Weiss 1984）。确保只要求获得能获得的信息，并牢记数据收集的重要性总是排在最后期限之后（Basili et al. 2002）。

软件度量的更多资源

下面是更多的资源：

Oman, Paul and Shari Lawrence Pfleeger, eds. *Applying Software Metrics*. Los Alamitos, CA: IEEE Computer Society Press, 1996。这本书收录了超过 25 篇有关软件度量的重要论文。

Jones, Capers. *Applied Software Measurement: Assuring Productivity and Quality*, 2d ed. New York, NY: McGraw-Hill, 1997。作者是软件度量领域的领袖，书中汇集了这一领域的知识，讲述了度量方法的权威理论和实践， 描述了传统度量手段的缺陷。 书中给出了一个完整的用于收集“功能点指标”（function-point metrics）的程序。作者曾收集分析了大量有关质量和生产率的数据，并将分析结果提炼到本书当中，其中有一章非常精彩地描述了美国软件开发的平均水平。

Grady, Robert B. *Practical Software Metrics for Project Management and Process Improvement*. Englewood Cliffs, NJ: Prentice Hall PTR, 1992。作者讲述了惠普公司在创建软件度量程序时的经验和教训，并且告诉读者如何在自己的组织中创建软件度量程序。

Conte, S. D., H. E. Dunsmore, and V. Y. Shen. *Software Engineering Metrics and Models*. Menlo Park, CA: Benjamin/Cummings, 1986。该书编录了截止 1986 年的软件度量方面的知识，包括常用的度量方法、实验方法以及实验结果的评判标准。

Basili, Victor R., et al. 2002. “Lessons learned from 25 years of process improvement: The Rise and Fall of the NASA Software Engineering Laboratory”，*Proceedings of the 24th International Conference on Software Engineering*. Orlando, FL, 2002。收录了顶尖软件开发组织之一 NASA 软件工程实验室的经验和教训，集中反映了度量的话题。

NASA Software Engineering Laboratory. *Software Measurement Guidebook*, June 1995, NASA-GB-001-94。这本约 100 页的手册或许是介绍如何建立并运行度量程序的最佳用信息来源。该手册可从 NASA 网站下载。

Gilb, Tom. *Competitive Engineering*. Boston, MA: Addison-Wesley, 2004。书中讲述了一种以度量为中心的方法，可采用该方法来定义需求、评估设计、度量质量以及一般意义上的项目管理。可从作者网站下载。

28.5 以人为本，善待每一位程序员

编程活动的抽象性要求自然的办公环境和同事之间的丰富接触。高技术公司提供公园式的公司园区、有机的组织结构、舒适的办公室以及其他“高接触”（high-touch）环境特征，以平衡这种有时会显得枯燥的智力密集型工作。最成功的技术公司结合了高科技和高接触的元素（Naisbitt 1982）。本节从多个方面解释了程序员为什么不仅仅是他们的“第二硅芯片人格”的有机反射。

程序员的时间是怎么花的？

程序员会花时间编程，但也会花时间在开会、培训、阅读邮件以及思考上。1964 年贝尔实验室的一项研究发现程序员是这样花时间的，如表 28-3 所示。

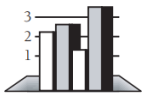
表 28-3　程序员怎么分配时间的一个观点

活动	源代码	业务	个人	会议	培训	邮件 / 闲杂文档	技术手册	操作规程, 杂项	程序测试	总计
交谈或者倾听	4%	17%	7%	3%				1%		32%
与管理人员交谈		1%								1%
电话		2%	1%							3%
阅读	14%					2%	2%			18%
写作 / 记录	13%					1%				14%
离开或者外出		4%	1%	4%	6%					15%
步行	2%	2%	1%			1%				6%
杂项	2%	3%	3%			1%		1%	1%	11%
总计	**35%**	**29%**	**13%**	**7%**	**6%**	**5%**	**2%**	**2%**	**1%**	**100%**

来源：*Research Studies of Programmers and Programming*（Bairdain 1964，载于 Boehm 1981）

这些数据基于一项针对 70 位程序员的时间和活动所进行的研究。数据很老旧，而且不同程序员花在各项活动上的时间比例也不尽相同，但其结果仍然发人深省。一位程序员大约有 30% 的时间花在对项目并没有直接助益的非技术活动之上：步行和个人事务等。在这份调查中，程序员有 6%的时间花在步行上；这相当于一周约 2.5 个小时、一年约 125 个小时。看起来似乎不算什么，但一旦意识到程序员每年花在走路上的时间和花在培训上的时间相等，并且 3 倍于他们阅读技术手册的时间，6 倍于他们和管理人员交谈的时间之后，你就会感到惊讶了。到现在为止，我本人并没有觉得这个模式有了太多变化。

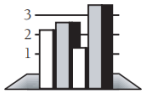
性能差异与质量差异



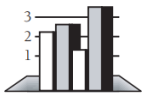
HARD DATA 不同程序员在天赋和努力程度方面差异很大，这一点与其他所有领域一样。有项针对不同职业（写作、橄榄球、发明、警务工作和飞行驾驶）做的调查研究表明，排名在前20%的人员占全部产出的 50%（Augustine 1979）。这一研究结果基于对生产率数据的分析，如触地得分、专利数量和侦破案件的数量等。有的人并没有做出切实的贡献，因而未能包括在调查当中（例如没有得分的橄榄球运动员、没有获得专利的发明家和没有结案的侦探等）。所以，这份数据可能还低估了生产率的实际差异。

具体到编程领域，很多研究表明，程序员在编写程序的质量、编写程序的规模以及生产率等方面，都存在着数量级的差异。

个体差异



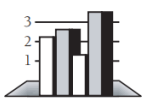
HARD DATA 程序员在编程生产率上呈现出巨大个体差异，其最早一项研究是由 Sackman、Erikson 和 Grant 在 20 世纪 60 年代末期做出的（Sackman，Erikson，and Grant 1968）。他们对平均工作经验为 7 年的专业程序员进行了调查，发现最优秀和最糟糕的程序员初始编码时所花的时间为 20:1，调试所花的时间为 25:1，程序规模比例为 5:1，程序执行速度比例为 10:1。他们并未发现程序员的经验与其代码质量/生产率之间有什么关联。



HARD DATA 虽然像 25:1 这样特殊的比例并不是特别有意义，但像“程序员之间存在着数量级的差异”这样更一般的陈述却是非常有意义的，并已由其他许多针对专业程序员的研究所证实（Curtis 1981, Mills 1983, DeMarco and Lister 1985, Curtis et al. 1986, Card1987, Boehm and Papaccio 1988, Valett and McGarry 1989, Boehm et al. 2000）。

团队差异

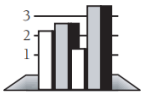
不同编程团队在软件质量和生产率上也存在相当大的差异。优秀的程序员倾向于抱团，糟糕的程序员也不例外，一项针对 18 个组织中 166 名专业程序员所做的研究证实了这个观点（Demarco and Lister1999）。



HARD DATA 一项针对 7 个相同项目所做的研究表明，花费的工作量的变化范围大到 3.4:1，程序规模的比例为 3:1（Boehm, Gray, and Seewaldt1984）。虽然存在这样的生产率差异，上

述研究中程序员的差别却并不大。他们都是经验相当丰富的专业程序员，并且都是计算机科学专业的研究生毕业。由此可以合理推断得出一个结论：如果被研究的团队并不是特别相似，那么存在的差异更大。

更早一份对多个编程团队做的研究发现，由不同的团队来完成相同的项目，其程序规模之比为 5:1，花费时间之比为 2.6:1（Weinberg and Schulman 1974）。



HARD DATA 创建 Cocomo II 评估模型时，在对超过 20 年的数据进行研究之后，Barry Boehm 和其他研究人员得出结论，由程序员能力排名位于第 15 个百分位的人员组成的团队，其开发应用程序所需的工作月数，大约是由程序员能力排名位于第 90 个百分位的人员组成的团队的 3.5 倍（Boehm et al. 2000）。Boehm 和其他研究人员发现，80%的贡献来自 20%的贡献者（Boehm 1987b）。

这其中的含义对招聘和录用来说不言而喻。如果为了聘到排名在前 10%而不是后 10%的程序员而不得不支付更多薪酬，请不要犹豫！这会因为聘用了高品质和高生产率的程序员而迅速获得回报，而且这么做还有一个好处，那就是组织中其他程序员的素质和生产率也会得以维持，因为优秀的程序员倾向于抱团。

信仰问题

软件项目经理并非总是注意到一些编程问题和信仰有关。如果你是管理者，并试图要求统一某些编程实践，可能会招致程序员的愤怒。以下是关于信仰的一些问题：

- 编程语言
- 缩进风格
- 大括号位置
- IDE 的选择
- 注释风格
- 效率与可读性的权衡
- 方法论的选择：例如，Scrum，极限编程，还是演进式交付
- 编程工具
- 命名规范
- goto 的使用
- 全局变量的使用
- 度量，尤其是生产率的度量，例如每天写多少行代码

以上问题的共同特征就是，每一项都是程序员个人风格的反映。如管理者认为有必要控制程序员的某些信仰，不妨考虑以下几点：

要清楚自己进入了敏感领域 做决定前先试探一下程序员对每个敏感主题的看法。

针对这些领域要采用“建议”或“指导原则” 避免制订死板的“规则”或“标准”。

尽量不要通过强制性的规定来处理问题 为了处理缩进风格或者大括号的位置，可以要求在源代码宣告完成之前先通过一个格式化工具来处理。让格式化工具来处理格式。为了处理注释风格，可以要求对所有代码进行评审，修改不清晰的代码，直到变得清晰为止。

让程序员自己设立标准 正如本书其他地方所提到的，某个标准已经存在的事实通常比一个特定标准中的细节更重要。不要为程序员设立标准，但坚持让他们在对你来说很重要的领域设立标准。

有哪些关于信仰的话题值得分出个胜负呢？在所有方面的所有小细节上都要求一致，可能不会产生足够的好处来抵消士气低落的影响。但是，如果发现有人不分青红皂白地使用 goto 或全局变量、不可读的风格或其他影响整个项目的做法，就准备忍受一些摩擦以提高代码质量。如果你的程序员是认真的，这很少会成为问题。最大的争斗往往是在编码风格的细微差别上，只有项目没什么损失，你完全可以置身事外。

物理环境

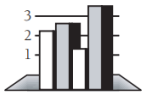
以下是一个实验：来到乡下，找个农场，见到一位农场主，问他为每个工人的装备花了多少钱。农场主来到粮仓，看了看里面的几台拖拉机、一些小货车、一台联合收割机和一台豌豆脱粒机，然后回答说，每个工人的花费超过 100000 美元。

然后来到城市，找一家软件公司，见到一位程序开发经理，问他为每个工作人员的设备投了多少钱。经理扫视一下办公室，一张桌子、一把椅子、几本书和一台电脑，然后告诉你，每个工作人员的花费不到 25000 美元。

物理环境对生产率有着巨大的影响。DeMarco 和 Lister 向来自 35 个组织的 166 名程序员询问了工作环境的质量。大多数员工都对工作环境感到不满。在此后举行的一次编程竞赛中，排名在前 25%的程序员都拥有更宽敞、更安静、更私密的办公室，而且更少受到其他人员和电话的干扰。表现最好和最差的参赛者在办公环境上的差异如下表所示。

环境因素	排名在前 25%	排名在后 25%
专用办公空间	78 平方英尺	46 平方英尺
可接受的安静的工作空间	57%	29%
可接受的私人工作空间	62%	19%
电话静音的能力	52%	10%
电话呼叫转移的能力	76%	19%
经常不必要的打扰	38%	76%
工作空间得到程序员赞赏	57%	29%

来源：*Peopleware*（DeMarco and Lister 1999）



HARD DATA 这些数据表明，生产率与工作场所的质量有着很强的相关性。排名前 25%的程序员的生产率是排名后 25%的程序员的 2.6 倍。DeMarco 和 Lister 原以为更优秀的程序员是由于获得晋升才自然拥有更好的办公室。但是，进一步的调查却显示事实并非如此。来自同一组织的程序员，不论绩效表现如何，他们的办公设施均相差无几。

大型的软件密集型组织均有类似的经验。Xerox、TRW、IBM 和贝尔实验室都表示，他们以人均 1 万到 3 万美元的资本投资实现了生产率的大幅提高，这些投资通过生产率的提高得到了更多回报（Boehm 1987a）。建立“有利于提高生产率的办公室”后，自评报告中估计生产率提高了 39%~47%（Boehm et al. 1984）。

总之，如果个人工作环境排名在后 25%，就可以通过将工作环境改善为排名前 25%的方式，实现大约 100%的生产率提升。如果个人工作环境处于平均水平，仍然可以通过将工作环境改善为排名至前 25%的方式，从而实现 40%或更多的生产率提升。

善待每一位程序员：更多资源

下面是针对这一主题的更多资源：

Weinberg, Gerald M. *The Psychology of Computer Programming*, 2ded. New York, NY: Van Nostrand Reinhold, 1998。这是第一本明确提出善待程序员的书籍，而且到目前为止仍然是论述编程活动作为人的活动这一主题的最佳书籍。书中饱含对程序员人性的敏锐观察，并且阐释了其蕴含的意义。

DeMarco, Tom and Timothy Lister. *Peopleware: Productive Projects and Teams*, 2d ed. New York, NY: Dorset House, 1999。书如其名，这本书所关注的同样是编程活动中人的因素。其中包括很多奇闻逸事，内容涉及人员管理、办公环境、聘用和培养正确的人员、团队成长以及享受工作。作者依靠一些奇闻轶事来支撑自己的一些不同寻常的观点，不过有些地方其逻辑显得比较牵强。但书中以人为中心的思想却最重要，作者毫不犹豫地阐述并传达了这一理念。

McCue, Gerald M. “IBM's Santa Teresa Laboratory – Architectural Design for Program Development”，IBM Systems Journal 17, no. 1 (1978):4 – 25。讲述了 IBM 创建 Santa Teresa 办公大楼的过程。IBM 研究了程序员的需求，以此创建出了建筑指导方案，并为程序员精心设计了办公场所。程序员们全程参与了整个过程。其结果是，在每年的意见调查中，员工对 Santa Teresa 其物理设施的满意度是全公司最高的。

McConnell, Steve. *Professional Software Development. Boston*, MA: Addison-Wesley, 2004。这本书的第 7 章总结了一些针对程序员的人口统计学研究成果，内容包括人格类型、教育背景和职业前景。

Carnegie, Dale. *How to Win Friends and Influence People*, Revised Edition. New York, NY: Pocket Books, 1981。当作者于 1936 年写下本书第 1 版的书名时，他无法想象该书的内容在今天会有什么样的含义。听起来像是应该与马基雅维利的书放在一起。然而，该书的思想与马基雅维利的控制手段针锋相对，作者有一个核心观点，真诚地关切他人相当重要。作者对如何处理日常人际关系具有敏锐的洞察力，他讲述了怎样通过更好地关切了解他人来与之共事。书

中提供了大量令人难忘的奇闻逸事，有时一页上甚至有两三个之多。任何需要和他人合作的人都应在适当的时候读一下这本书，所有管理者更应该马上读。

28.6 向上管理

> “在等级制度中，每位员工都倾向于升职到自己无法胜任的职位上。”——彼得原理

在软件开发领域，非技术出身的管理者随处可见，有技术经验但落后于这个时代 10 年的管理者也比比皆是。技术出色又能紧跟技术发展趋势的管理者简直是凤毛麟角。如果开发者正在效力于这样的管理者，就千方百计保住这份工作吧，这样的机遇千载难逢。

如果你的管理者属于更典型的那种，你将要面临一项异常艰巨的向上管理的任务。“向上管理”（Managing your manager）意味着，需要告诉管理者要这样做而不要那样做。其中的窍门在于，你的举止行为要让管理者以为他还是老大。下面是向上管理的一些方法：

- 向其植入自己的想法与创意，等着管理者来一场头脑风暴（其实就是你的意思），跟你讨论你本来就想做的事情。
- 将正确的做事方式教给管理者。这是一项需要持之以恒的工作，因为管理者经常会被提升、调动或解聘。
- 关注管理者的兴趣，按他们真正想要的方式做事，不要让他们注意到不必要的实现细节（想象成对自己的工作的一种“封装”）
- 拒绝按管理者说的做，坚持用正确的方法做事。
- 另外找份工作。

最好的长远解决方案就教会管理者做事的方式。这通常相当不易，但可以通过 Dale Carnegie 的《*How to Win Friends and Influence People*》（人性的弱点）一书来做好准备。

管理构建的更多资源

下面这些书涵盖了软件项目管理中普遍受到关注的问题：

Gilb, Tom. *Principles of Software Engineering Management*. Wokingham, England: Addison-Wesley, 1988。作者以图表方式展现了自己 30 年的经历，并且在大多数时候，他的见解都很超前，无论其他人是否认识到这一点。这本书最先讨论演进式开发实践、风险管理以及正式审查的使用等相关专题。作者熟知前沿方法，而且事实上，这本在 1988 年出版的书已经包含敏捷开发阵营中大多数良好的实践。作者非常注重实效，而本书至今也仍然是最佳的软件管理书籍之一。

McConnell, Steve. *Rapid Development*. Redmond, WA: Microsoft Press, 1996。本书着眼于进度压力超大的项目，涵盖项目领导力和项目管理的相关问题。以作者本人的经验来看，这正是绝大多数项目的真实写照。

Brooks, Frederick P., Jr. *The Mythical Man-Month: Essays on Software Engineering*（人月神话）, Anniversary Edition(2d ed). Reading, MA: Addison-Wesley,1995。汇集有关编程项目管理的隐喻及民间传说，趣味横生，并且对于认知自己的项目也颇有启示作用。该书基于作者开发 OS/360 操作系统这一挑战性任务而写成，虽然我对此还持有一些保留意见。书中很多诸如

“我们这样做但失败了”和“我们本该这样做就成功了”这样的建议。作者在本书中对那些不成功的技术做出了令人信服的评论，但他宣称其他技术之所以行得通却是因为运气好。阅读本书时，一定要持有批判的眼光，把作者的观察和猜测区分开来。这个提醒并不会缩减本书的基本价值。相比其他任何计算机书籍，该书仍然是被引用次数最多的图书，而且即便本书初版于 1975 年发行，内容至今仍未过时。在阅读本书的过程中，每隔几页不附和一句“对极了!”是相当难受的。

相关标准

IEEE Std 1058-1998, Standard for Software Project Management Plans.

IEEE Std 12207-1997,Information Technology—Software Life CycleProcesses.

IEEE Std 1045-1992,Standard for Software Productivity Metrics.IEEE Std 1062-1998,Recommended Practice for Software Acquisition.

IEEE Std 1540-2001,Standard for Software Life Cycle Processes—RiskManagement.

IEEE Std 828-1998,Standard for Software Configuration ManagementPlans

IEEE Std 1490-1998,Guide—Adoption of PMI Standard—A Guide tothe Project Management Body of Knowledge.

要点回顾

- 好的编码实践可通过强制的标准或更宽松的方法来实现。
- 如应用得当，配置管理（尤其是变更控制）能使程序员的工作变得更容易。
- 好的软件评估是一项重大的挑战。成功的关键在于采用多种方法，随着项目的开展调整评估，并利用之前确定的数据来创建新的评估。
- 构建管理要取得成功，度量是关键。项目的任何方面都可以找到一些方法进行度量，这比根本不测量要好。为了实现准确的进度表、质量控制和改善开发过程，准确的度量是关键。
- 程序员和管理者都是人，以人为本并善待他们，他们的工作效果最好。