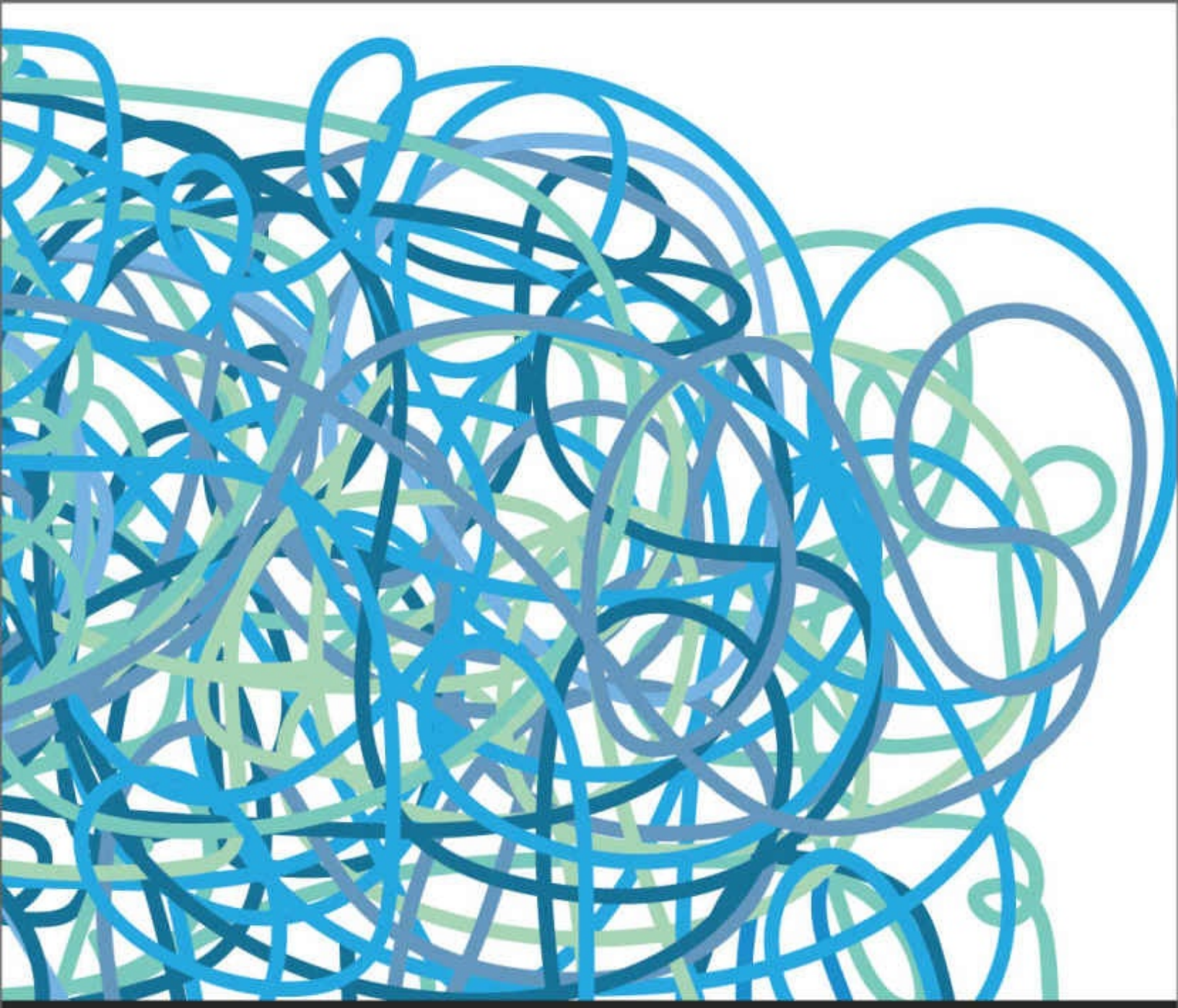

A PHILOSOPHY OF SOFTWARE DESIGN

JOHN OUSTERHOUT

软件设计哲学

约翰·奥斯特豪特
斯坦福大学

软件设计哲学

作者：约翰·奥斯特豪特

由 Yaknyam Press, Palo Alto, CA 出版。

封面设计：Pete Nguyen 和 Shirin Oreizy (www.hellonextstep.com).

印刷历史：
2018年4月： 第一版 (v1.0)
2018年11月： 第一版 (v1.01)

ISBN 978-1-7321022-0-0

数字图书（epub 和 mobi 格式）由 Booknook.biz 制作。

目录

前言

人们编写电子计算机程序已经超过 80 年了，但令人惊讶的是，关于如何设计这些程序或好的程序应该是什么样子，却很少有讨论。关于软件开发过程（如敏捷开发）以及开发工具（如调试器、版本控制系统和测试覆盖率工具）已经有很多讨论。对编程技术（如面向对象编程和函数式编程）以及设计模式和算法也进行了广泛的分析。所有这些讨论都很有价值，但软件设计的核心问题在很大程度上仍未触及。David Parnas 的经典论文“将系统分解为模块的标准”发表于 1971 年，但在随后的 45 年里，软件设计领域的最新技术并没有比这篇论文进步多少。

计算机科学中最根本的问题是问题分解：如何将一个复杂的问题分解成可以独立解决的各个部分。问题分解是程序员每天面临的核心设计任务，然而，除了这里描述的工作之外，我还没有在任何大学里找到一门以问题分解为中心主题的课程。我们教授for循环和面向对象编程，但不教授软件设计。

此外，程序员的质量和生产力差异很大，但我们很少尝试了解是什么让最好的程序员如此出色，或者在我们的课程中教授这些技能。我和几个我认为很棒的程序员交谈过，但他们中的大多数人很难清楚地表达出让他们获得优势的具体技术。许多人认为软件设计技能是一种无法传授的先天才能。

然而，有相当多的科学证据表明，许多领域的杰出表现与高质量的实践而非先天能力有关（例如，参见 Geoff Colvin 的《天赋被高估》）。

多年来，这些问题一直困扰着我，让我感到沮丧。我一直在想软件设计是否可以教授，并且我假设

设计技能是将优秀的程序员与普通的程序员区分开来的因素。我最终决定，回答这些问题的唯一方法是尝试开设一门关于软件设计的课程。结果就是斯坦福大学的 CS 190。在这门课上，我提出了一套软件设计原则。然后，学生们通过一系列项目来吸收和实践这些原则。这门课的教学方式类似于传统的英语写作课。在英语课上，学生们使用一个迭代过程，他们写一个草稿，得到反馈，然后重写以进行改进。在 CS 190 中，学生们从头开始开发一个重要的软件。然后，我们进行广泛的代码审查，以识别设计问题，学生们修改他们的项目以解决这些问题。

这使得学生们能够看到如何通过应用设计原则来改进他们的代码。

我已经教了三次软件设计课程，这本书是基于从课程中产生的设计原则。这些原则相当高层次，并且接近哲学层面（“将错误定义为不存在”），因此学生很难理解抽象的概念。

学生通过编写代码、犯错，然后了解他们的
错误以及随后的修复与原则之间的关系来学习得最好。

此时，你可能会想：是什么让我认为我知道所有关于软件设计的答案？老实说，我不知道。当我学习编程时，没有关于软件设计的课程，我也没有导师来教我设计原则。在我学习编程的时候，代码审查几乎不存在。我对软件设计的想法来自个人编写和阅读代码的经验。在我的职业生涯中，我写了大约

2 50,000 行各种语言的代码。我曾在团队中工作过
从头开始创建了三个操作系统，多个文件和存储系统，基础设施工具，如调试器、构建系统和 GUI 工具包，一种脚本语言，以及用于文本、绘图、演示文稿和集成电路的交互式编辑器。一路走来，我亲身经历了大型系统的问题，并尝试了各种设计技术。此外，我还阅读了大量其他人编写的代码，这让我接触到了各种方法，无论是好的还是坏的。

在所有这些经验中，我试图提取共同点，包括
要避免的错误和要使用的技术。这本书反映了我的
经验：这里描述的每个问题都是我亲身经历过的

并且每个建议的技术都是我在

自己的编码中成功使用过的。

我不期望这本书成为软件设计的最终定论；我确信
我遗漏了一些有价值的技术，而且从长远来看，我的一些建议可能会
变成坏主意。但是，我希望这本书能引发一场关于
软件设计的对话。将本书中的想法与你自己的
经验进行比较，然后自己决定这里描述的方法是否
真的能降低软件的复杂性。这本书是一篇观点文章，因此一些
读者会不同意我的一些建议。如果你确实不同意，请尝试
理解原因。我有兴趣听取对你有效的方法、
无效的方法以及你可能对软件设计有的任何其他想法。我希望
随之而来的对话能够提高我们对
软件设计的集体理解。我将在本书的未来版本中融入我所学到的知识。

与我沟通关于这本书的最佳方式是发送电子邮件至以下
地址：

software-design-book@googlegroups.com

我对听到关于这本书的具体反馈很感兴趣，例如错误或改进建议，以及与软件设计相关的总体想法和经验。我特别感兴趣的是可以在本书未来版本中使用的引人注目的例子。最好的例子阐明了一个重要的设计原则，并且足够简单，可以用一两个段落来解释。如果您想查看其他人在电子邮件地址上发表的内容并参与讨论，您可以加入 Google Group software-design-book。

如果由于某种原因，software-design-bookGoogle Group 在未来消失，请在网上搜索我的主页；它将包含关于如何沟通关于本书的更新说明。请不要将与本书相关的电子邮件发送到我的个人电子邮件地址。

我建议你对本书中的建议持保留态度。
总体目标是降低复杂性；这比你在这里读到的任何特定原则或想法都更重要。如果你尝试了本书中的一个想法，发现它实际上并没有降低复杂性，那么不要觉得有义务继续使用它（但是，请让我知道你的经验；我想获得关于什么有效和什么无效的反馈）。

许多人提出了批评或建议，提高了本书的质量。以下人员对本书的各个草案提出了有益的意见：Jeff Dean、Sanjay Ghemawat、John Hartman、Brian Kernighan、James Koppel、Amy Ousterhout、Kay Ousterhout、Rob Pike、Partha Ranganathan、Keith Schwartz 和 Alex Snaps。Christos Kozyrakis 建议使用“深”和“浅”这两个术语来表示类和接口，取代了之前的术语“厚”和“薄”，后者有些含糊不清。我感谢 CS 190 的学生；阅读他们的代码并与他们讨论的过程帮助我明确了关于设计的想法。

第一章

引言

（一切都与复杂性有关）

编写计算机软件是人类历史上最纯粹的创造性活动之一。程序员不受物理定律等实际限制的约束；我们可以创造令人兴奋的虚拟世界，其行为在现实世界中永远不会存在。编程不需要像芭蕾或篮球那样出色的身体技能或协调能力。所有编程都需要的是一个有创造力的头脑和组织你思想的能力。如果你能可视化一个系统，你可能可以在计算机程序中实现它。

这意味着编写软件的最大限制是我们理解我们正在创建的系统的能力。随着程序的发展并获得更多功能，它变得复杂，其组件之间存在微妙的依赖关系。随着时间的推移，复杂性会累积，程序员越来越难以在修改系统时将所有相关因素记在脑海中。这减慢了开发速度并导致错误，这进一步减慢了开发速度并增加了成本。复杂性在任何程序的生命周期中都不可避免地增加。程序越大，参与的人越多，管理复杂性就越困难。

好的开发工具可以帮助我们处理复杂性，并且在过去的几十年里已经创建了许多伟大的工具。但是，仅凭工具我们能做的事情是有限的。如果我们想让编写软件更容易，以便我们可以更便宜地构建更强大的系统，我们必须找到使软件更简单的方法。尽管我们尽了最大的努力，复杂性仍然会随着时间的推移而增加，但更简单的设计使我们能够在复杂性变得难以承受之前构建更大、更强大的系统。

对抗复杂性有两种通用方法，本书将讨论这两种方法。第一种方法是通过

使代码更简单和更明显来消除复杂性。例如，可以通过消除特殊情况或以一致的方式使用标识符来降低复杂性。

对抗复杂性的第二种方法是封装它，以便程序员可以处理一个系统，而无需一次接触到它的所有复杂性。这种方法被称为模块化设计。在模块化设计中，一个软件系统被划分为模块，例如面向对象语言中的类。这些模块被设计成彼此相对独立，因此程序员可以处理一个模块，而无需了解其他模块的细节。

由于软件具有很强的可塑性，因此软件设计是一个持续的过程，贯穿软件系统的整个生命周期；这使得软件设计不同于建筑物、船舶或桥梁等物理系统的设计。
然而，软件设计并非一直以这种方式看待。在编程的大部分历史中，设计都集中在项目的开始阶段，就像其他工程学科一样。这种方法的极端情况被称为瀑布模型，其中一个项目被划分为离散的阶段，例如需求定义、设计、编码、测试和维护。在瀑布模型中，每个阶段在下一个阶段开始之前完成；在许多情况下，不同的人负责每个阶段。整个系统在设计阶段一次性设计完成。设计在这个阶段结束时被冻结，后续阶段的作用是充实和实现该设计。

不幸的是，瀑布模型很少能很好地适用于软件。软件系统本质上比物理系统更复杂；不可能充分可视化大型软件系统的设计，以便在构建任何东西之前了解其所有含义。因此，最初的设计会存在许多问题。这些问题在实施顺利进行之前不会变得明显。然而，瀑布模型的结构并不适合在此时进行重大设计变更（例如，设计师可能已经转移到其他项目）。因此，开发人员试图在不改变整体设计的情况下修补这些问题。这导致了复杂性的爆炸。

由于这些问题，如今大多数软件开发项目都使用增量方法，例如敏捷开发，其中初始设计侧重于整体功能的一小部分子集。这个子集被设计、实现，然后进行评估。原始设计的问题被发现并纠正，然后设计、实现更多功能

并进行评估。每次迭代都会暴露现有设计的问题，这些问题会在设计下一组功能之前得到修复。通过以这种方式分散设计，可以在系统仍然很小时修复初始设计的问题；后来的功能受益于早期功能实施过程中获得的经验，因此它们的问题更少。

增量方法适用于软件，因为软件具有足够的可塑性，允许在实施过程中进行重大设计更改。与此相反，对于物理系统来说，重大设计更改更具挑战性：
例如，在建造过程中更改支撑
桥梁的塔的数量是不切实际的。

增量开发意味着软件设计永远不会完成。设计在系统的整个生命周期中持续进行：开发人员应始终考虑设计问题。增量开发也意味着持续的重新设计。系统或组件的初始设计几乎从来都不是最好的；经验不可避免地会显示出更好的做事方式。作为一名软件开发人员，您应该始终注意寻找改进您正在开发的系统设计的机会，并且您应该计划将一部分时间用于设计改进。

如果软件开发人员应该始终考虑设计问题，并且降低复杂性是软件设计中最重要的因素，那么软件开发人员应该始终考虑复杂性。本书讲述的是如何利用复杂性来指导软件在其整个生命周期中的设计。

本书有两个总体目标。首先是描述软件复杂性的本质：什么是“复杂性”，为什么它很重要，以及如何识别程序何时具有不必要的复杂性？本书的第二个，也是更具挑战性的目标是介绍您在软件开发过程中可以使用的技术，以最大限度地降低复杂性。不幸的是，没有一个简单的秘诀可以保证出色的软件设计。相反，我将介绍一系列接近哲学层面的高级概念，例如“类应该很深”或“将错误定义为不存在”。这些概念可能无法立即识别出最佳设计，但您可以使用它们来比较设计方案并指导您探索设计空间。

1.1 如何使用本书

这里描述的许多设计原则有些抽象，因此如果不看实际代码，可能很难理解。找到足够小的例子来包含在书中，但又足够大以说明真实系统的问题一直是一个挑战（如果您遇到好的例子，请发送给我）。因此，本书本身可能不足以让您学习如何应用这些原则。

使用本书的最佳方式是结合代码审查。当您阅读别人的代码时，请考虑它是否符合此处讨论的概念，以及这与代码的复杂性有何关系。在别人的代码中比在自己的代码中更容易看到设计问题。您可以使用此处描述的红色标志来识别问题并提出改进建议。审查代码还将使您接触到新的设计方法和编程技术。

提高设计技能的最佳方法之一是学会识别红色
标志：表明一段代码可能比实际需要的更复杂的迹象。
在本书的过程中，我将指出与每个主要设计问题相关的红色标志；最重要的那些总结在本书的背面。然后，您可以在编写代码时使用它们：当您看到红色标志时，请停下来寻找可以消除问题的替代设计。

当您第一次尝试这种方法时，您可能需要尝试几种设计方案，然后才能找到一种可以消除红色标志的方案。不要轻易放弃：在解决问题之前尝试的替代方案越多，您学到的东西就越多。随着时间的推移，您会发现您的代码中的红色标志越来越少，并且您的设计越来越简洁。您的经验还将向您展示其他红色标志，您可以使用它们来识别设计问题（我很乐意听到这些）。

在应用本书中的想法时，重要的是要适度和谨慎。每条规则都有其例外，每个原则都有其局限性。如果您将任何设计理念推向极端，您最终可能会陷入困境。

精美的设计反映了相互竞争的想法和方法之间的平衡。
有几个章节包含名为“走得太远”的部分，描述了如何识别你何时过度追求一件好事。

本书中的几乎所有示例都使用 Java 或 C++，并且大部分讨论都是关于在面向对象的语言中设计类。
但是，这些想法也适用于其他领域。几乎所有与方法相关的想法也可以应用于没有面向对象特性的语言（如 C）中的函数。

这些设计思想也适用于类以外的模块，例如子系统或网络服务。

有了这些背景知识，让我们更详细地讨论一下是什么导致了复杂性，以及如何使软件系统更简单。

第 2 章

复杂性的本质

本书是关于如何设计软件系统以最大限度地降低其复杂性的。第一步是了解敌人。到底什么是“复杂性”？你如何判断一个系统是否不必要地复杂？是什么导致系统变得复杂？本章将在较高层次上解决这些问题；后续章节将向你展示如何在较低层次上，根据特定的结构特征来识别复杂性。

识别复杂性的能力是一项至关重要的设计技能。它使你能够在投入大量精力之前识别问题，并且使你能够在备选方案中做出好的选择。判断一个设计是否简单比创建一个简单的设计更容易，但是一旦你能够识别出一个系统过于复杂，你就可以利用这种能力来指导你的设计理念朝着简单化发展。如果一个设计看起来很复杂，尝试一种不同的方法，看看是否更简单。随着时间的推移，你会注意到某些技术往往会导致更简单的设计，而另一些技术则与复杂性相关。这将使你能够更快地产生更简单的设计。

本章还阐述了一些基本假设，为本书的其余部分奠定了基础。后面的章节将本章的材料作为既定材料，并用它来证明各种改进和结论的合理性。

2.1 复杂性的定义

为了本书的目的，我以一种实用的方式定义“复杂性”。复杂性是指与软件系统的结构相关的任何使其难以理解和修改系统的东西。复杂性可以有多种形式。例如，可能难以理解一段代码是如何工作的；可能需要付出很大的努力才能实现一个小小的改进，或者可能不清楚必须修改系统的哪些部分才能进行改进；可能难以修复一个错误而不引入另一个错误。如果

一个软件系统难以理解和修改，那么它就是复杂的；如果它易于理解和修改，那么它就是简单的。

你也可以从成本和收益的角度来考虑复杂性。在一个复杂的系统中，即使是很小的改进也需要大量的工作才能实现。在一个简单的系统中，可以用更少的努力来实现更大的改进。

复杂性是开发人员在试图实现特定目标时在特定时间点所体验到的。它不一定与系统的整体大小或功能有关。人们经常用“复杂”这个词来描述具有复杂功能的大型系统，但如果这样一个系统很容易操作，那么，为了本书的目的，它并不复杂。当然，几乎所有大型和复杂的软件系统实际上都很难操作，所以它们也符合我对复杂性的定义，但这不一定是这样。一个小型且不复杂的系统也可能非常复杂。

复杂性是由最常见的活动决定的。如果一个系统有一些非常复杂的部分，但这些部分几乎不需要被触及，那么它们对系统的整体复杂性没有太大的影响。用一种粗略的数学方式来描述：

$$C = \sum_{p} c_p t_p$$

系统的整体复杂性（C）由每个部分p的复杂性（cp）决定，并根据开发人员花费在该部分上的时间比例（tp）进行加权。将复杂性隔离在一个永远不会被看到的地方几乎和完全消除复杂性一样好。

复杂性对于读者来说比对于作者来说更明显。如果你写了一段代码，你觉得它很简单，但其他人觉得它很复杂，那么它就是复杂的。当你发现自己处于这种情况时，值得探究其他开发人员，找出为什么代码对他们来说似乎很复杂；从你和他们的意见之间的脱节中，可能会学到一些有趣的教训。你作为开发人员的工作不仅仅是创建你可以轻松使用的代码，而是创建其他人也可以轻松使用的代码。

2.2 复杂性的症状

复杂性以三种常见方式体现，这些方式在下面的段落中描述。这些表现形式都使得执行开发任务更加困难。

开发任务。

变更放大：复杂性的第一个症状是，一个看似简单的变更需要在许多不同的地方修改代码。例如，考虑一个包含多个页面的网站，每个页面都显示一个带有背景颜色的横幅。在许多早期的网站中，颜色在每个页面上都明确指定，如图 2.1(a)所示。为了更改此类网站的背景，开发人员可能必须手动修改每个现有页面；对于一个拥有数千页面的大型网站来说，这几乎是不可能的。幸运的是，现代网站使用类似于图 2.1(b)中的方法，其中横幅颜色在一个中心位置指定一次，并且所有单独的页面都引用该共享值。使用这种方法，只需进行一次修改即可更改整个网站的横幅颜色。良好设计的目标之一是减少受每个设计决策影响的代码量，因此设计变更不需要太多的代码修改。

认知负荷：复杂性的第二个症状是认知负荷，它指的是开发人员完成一项任务需要了解多少信息。较高的认知负荷意味着开发人员必须花费更多的时间来学习所需的信息，并且由于他们遗漏了一些重要的东西，因此存在更大的错误风险。例如，假设 C 语言中的一个函数分配内存，返回指向该内存的指针，并假定调用者将释放该内存。这增加了使用该函数的开发人员的认知负荷；如果开发人员未能释放内存，则会出现内存泄漏。如果可以重构系统，使调用者无需担心释放内存（分配内存的同一模块也负责释放内存），则将减少认知负荷。认知负荷以多种方式产生，例如具有许多方法的 API、全局变量、不一致性和模块之间的依赖关系。

系统设计人员有时会假设复杂性可以通过代码行数来衡量。他们假设如果一个实现比另一个实现短，那么它一定更简单；如果只需几行代码即可进行更改，那么该更改一定很容易。但是，这种观点忽略了与认知负荷相关的成本。我见过一些框架，允许应用程序只用几行代码编写，但弄清楚这些行是什么却极其困难。有时，需要更多代码行的方法实际上更简单，因为它减少了认知负荷。

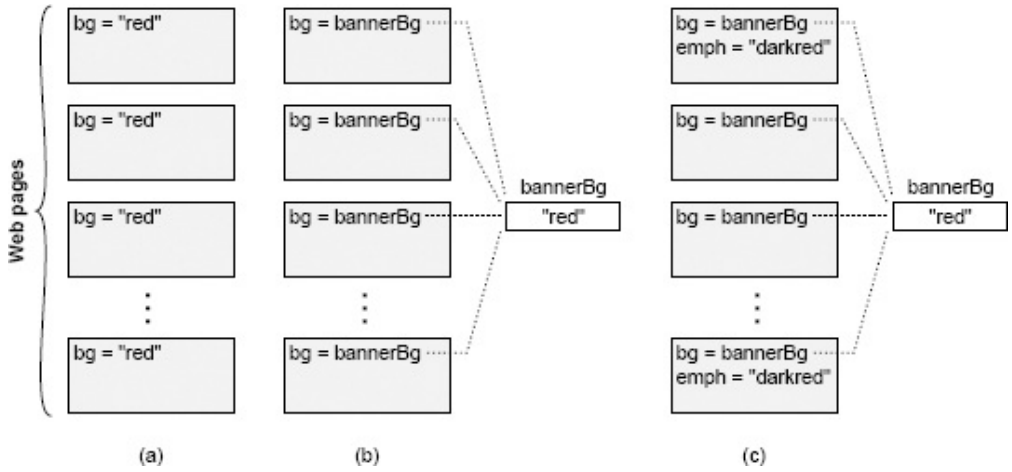

图 2.1：网站中的每个页面都显示一个彩色横幅。在 (a) 中，横幅的背景颜色在每个页面中明确指定。在 (b) 中，一个共享变量保存背景颜色，并且每个页面都引用该变量。在 (c) 中，某些页面显示额外的颜色以示强调，这是横幅背景颜色的较深阴影；如果背景颜色更改，则强调颜色也必须更改。

未知的未知：复杂性的第三个症状是，不清楚必须修改哪些代码才能完成任务，或者开发人员必须拥有哪些信息才能成功执行任务。图 2.1(c) 说明了这个问题。该网站使用一个中心变量来确定横幅背景颜色，因此更改起来似乎很容易。但是，一些网页使用背景颜色的较深阴影来表示强调，并且该较深颜色在各个页面中明确指定。如果背景颜色更改，则强调颜色必须更改以匹配。不幸的是，开发人员不太可能意识到这一点，因此他们可能会更改中心bannerBg变量而不更新强调颜色。即使开发人员意识到了这个问题，也不清楚哪些页面使用了强调颜色，因此开发人员可能必须搜索网站中的每个页面。

在复杂性的三种表现形式中，未知的未知是最糟糕的。
未知的未知意味着有些东西你需要知道，但是你无法找到它是什么，甚至无法确定是否存在问题。在你进行更改后出现错误之前，你不会发现它。变更放大令人讨厌，但只要清楚需要修改哪些代码，系统在完成更改后就可以正常工作。同样，较高的认知负荷会增加更改的成本，但如果清楚要读取哪些信息，则更改仍然可能是正确的。对于未知的未知，不清楚该做什么，或者提出的解决方案是否有效。唯一确定的方法是读取系统中每一行代码，

这对于任何规模的系统来说都是不可能的。即使这样可能也不够，因为更改可能取决于从未记录的微妙设计决策。

优秀设计的一个最重要目标是使系统显而易见。
这与高认知负荷和未知的未知情况相反。在一个显而易见的系统中，开发人员可以快速理解现有代码的工作方式以及进行更改所需的操作。一个显而易见的系统是指开发人员无需过多思考就能快速猜测该做什么，并且确信猜测是正确的系统。第 18 章讨论了使代码更显而易见的技术。

2.3复杂性的原因

既然您已经了解了复杂性的高级症状以及为什么复杂性
使软件开发变得困难，那么下一步就是了解导致
复杂性的原因，以便我们可以设计系统来避免这些问题。复杂性是
由两件事引起的：依赖性和模糊性。本节将从高层次上讨论这些
因素；后续章节将讨论它们与较低层次的
设计决策之间的关系。

就本书而言，当给定的代码片段无法独立理解和修改时，就存在依赖关系；代码以某种方式与其他代码相关，如果更改给定的代码，则必须考虑和/或修改其他代码。在图 2.1(a)的网站示例中，背景颜色在所有页面之间创建了依赖关系。所有页面都需要具有相同的背景，因此如果更改了一个页面的背景，则必须更改所有页面的背景。依赖关系的另一个例子发生在网络协议中。通常，协议的发送方和接收方都有单独的代码，但它们都必须符合协议；更改发送方的代码几乎总是需要在接收方进行相应的更改，反之亦然。方法的签名在该方法的实现和调用它的代码之间创建了依赖关系：如果向方法添加了新参数，则必须修改该方法的所有调用以指定该参数。

依赖关系是软件的基本组成部分，无法完全
消除。事实上，我们有意地引入依赖关系作为
软件设计过程的一部分。每次您编写一个新类时，您都会围绕该类的 API 创建依赖关系。然而，其中一个目标是

软件设计是减少依赖关系的
数量，并使剩余的依赖关系尽可能简单和明显。

考虑一下网站的例子。在旧的网站中，背景在每个页面上单独指定，所有网页都相互依赖。新的网站通过在中心位置指定背景颜色并提供一个 API 供各个页面在呈现时使用来检索该颜色，从而解决了这个问题。新的网站消除了页面之间的依赖关系，但它围绕用于检索背景颜色的 API 创建了一个新的依赖关系。幸运的是，新的依赖关系更加明显：很明显，每个单独的网页都依赖于 bannerBg 颜色，并且开发人员可以通过搜索其名称轻松找到使用该变量的所有位置。此外，编译器有助于管理 API 依赖关系：如果共享变量的名称发生更改，则在任何仍然使用旧名称的代码中都会发生编译错误。新的网站用一个更简单、更明显的依赖关系取代了一个不明显且难以管理的依赖关系。

复杂性的第二个原因是模糊性。当
重要信息不明显时，就会发生模糊性。一个简单的例子是变量名
过于通用，以至于没有携带太多有用的信息（例如，time）。或者，变量的文档可能没有指定其单位，因此找到
的唯一方法是扫描代码以查找使用该变量的位置。模糊性通常
与依赖关系相关联，在这种情况下，依赖关系的存在并不明显。
例如，如果向系统中添加了新的错误状态，则可能需要向保存每个状态的字符串消息的表中添加一个条目，但是对于查看状态声明的程序员来说，消息表的存在可能并不明显。不一致也是造成模糊性的一个主要因素：如果同一个变量名用于两个不同的目的，那么开发人员将无法清楚地知道特定变量用于哪个目的。

在许多情况下，模糊性是由于文档不足造成的；
第 13 章讨论了这个问题。然而，模糊性也是一个设计问题。如果一个系统具有清晰而明显的设计，那么它将需要更少的文档。对
大量文档的需求通常是一个危险信号，表明设计不太
正确。减少模糊性的最佳方法是简化系统设计。

总之，依赖性和模糊性导致了第 2.2 节中描述的复杂性的三种表现形式。依赖性导致变化放大和高认知负荷。模糊性会产生未知的未知，并且

会增加认知负荷。如果我们能找到能最大限度地减少依赖性和晦涩性的设计技术，那么我们就能降低软件的复杂性。

2.4 复杂性是递增的

复杂性不是由单一的灾难性错误引起的；它是由许多小块累积而成的。单一的依赖关系或模糊性本身不太可能显著影响软件系统的可维护性。复杂性是由于成百上千的小的依赖关系和模糊性随着时间的推移而积累起来的。最终，这些小问题太多了，以至于对系统的每一个可能的改变都会受到其中几个问题的影响。

复杂性的渐进性质使其难以控制。很容易说服自己，当前更改引入的少量复杂性没什么大不了的。但是，如果每个开发人员对每次更改都采用这种方法，复杂性会迅速累积。一旦复杂性积累起来，就很难消除，因为仅仅修复一个依赖项或模糊之处本身不会产生很大的影响。为了减缓复杂性的增长，您必须采取“零容忍”的理念，正如第 3 章中所讨论的那样。

2.5 结论

复杂性来自于依赖关系和模糊性的积累。随着复杂性的增加，会导致变化放大、高认知负荷和未知的未知。因此，需要更多的代码修改来实现每个新功能。此外，开发人员花费更多的时间来获取足够的信息以安全地进行更改，并且在最坏的情况下，他们甚至无法找到他们需要的所有信息。最重要的是，复杂性使得修改现有代码库变得困难和冒险。

第三章

能运行的代码还不够

（战略性与战术性编程）

良好软件设计最重要的要素之一是你在处理编程任务时所采取的心态。许多组织鼓励一种战术心态，专注于尽快使功能正常工作。然而，如果你想要一个好的设计，你必须采取一种更具战略性的方法，投入时间来产生清晰的设计并解决问题。本章讨论了为什么战略方法能够产生更好的设计，并且从长远来看实际上比战术方法更便宜。

3.1 战术编程

大多数程序员以一种我称之为“战术编程”的心态来对待软件开发。在战术方法中，你的主要重点是让某些东西能够工作，例如一个新功能或一个错误修复。乍一看，这似乎完全合理：有什么比编写能够工作的代码更重要的呢？
然而，战术编程几乎不可能产生一个好的系统设计。

战术编程的问题在于它目光短浅。如果你正在进行战术编程，你就会试图尽快完成一项任务。
也许你有一个很紧的截止日期。因此，为未来做计划不是优先事项。你不会花太多时间寻找最佳设计；你只是想尽快让某些东西能够工作。你告诉自己，如果增加一点复杂性或引入一两个小的临时解决方案能够更快地完成当前的任务，那也没关系。

系统就是这样变得复杂的。正如前一章所讨论的，复杂性是逐渐累积的。使系统变得复杂的不是某一件特定的事情，而是几十个或几百个小事情的积累。

如果你进行战术编程，每个编程任务都会贡献一些这样的复杂性。为了快速完成当前的任务，每一个复杂性可能看起来都是一个合理的妥协。然而，复杂性会迅速累积，特别是当每个人都在进行战术编程时。

不久之后，一些复杂性就会开始引起问题，你就会开始后悔当初走了那些捷径。但是，你会告诉自己，让下一个功能能够工作比回去重构现有代码更重要。重构可能从长远来看会有所帮助，但肯定会减慢当前任务的速度。因此，你会寻找快速的补丁来解决你遇到的任何问题。这只会产生更多的复杂性，然后需要更多的补丁。很快代码就会变得一团糟，但到了这个时候，情况已经变得非常糟糕，以至于需要几个月的工作才能清理干净。你的时间表根本无法容忍这种延迟，而且修复一两个问题似乎也不会产生太大的影响，所以你只能继续进行战术编程。

如果你在一个大型软件项目上工作了很长时间，我怀疑你已经看到了战术编程的运作方式，并且体验到了由此产生的问题。一旦你开始走上战术道路，就很难改变。

几乎每个软件开发组织都至少有一位将战术编程发挥到极致的开发人员：一个“战术龙卷风”。战术龙卷风是一位多产的程序员，他编写代码的速度远快于其他人，但工作方式完全是战术性的。当涉及到实现一个快速功能时，没有人比战术龙卷风做得更快。在一些组织中，管理层将战术龙卷风视为英雄。然而，战术龙卷风留下的是一片狼藉。那些将来必须使用他们的代码的工程师很少认为他们是英雄。通常，其他工程师必须清理战术龙卷风留下的烂摊子，这使得那些工程师（他们才是真正的英雄）看起来比战术龙卷风的进展更慢。

3.2 战略编程

成为一名优秀的软件设计师的第一步是意识到仅仅是能工作的代码是不够的。为了更快地完成当前任务而引入不必要的复杂性是不可接受的。最重要的是系统的长期结构。任何系统中的大部分代码都是通过扩展现有代码库编写的，因此作为一名

开发人员，你最重要的工作是促进未来的这些扩展。因此，你不应该把“能工作的代码”作为你的主要目标，当然你的代码必须能工作。
你的主要目标必须是产生一个伟大的设计，而这个设计恰好也能工作。这就是战略编程。

战略编程需要一种投资心态。你必须投入时间来改进系统的设计，而不是采取最快的途径来完成当前的项目。这些投资在短期内会让你慢一点，但从长远来看会加快你的速度，如图3.1所示。

一些投资将是主动的。例如，花一点额外的时间为每个新类找到一个简单的设计是值得的；与其实现脑海中出现的第一个想法，不如尝试几个替代设计并选择最简洁的一个。尝试想象一下系统将来可能需要更改的几种方式，并确保你的设计能够轻松实现。编写良好的文档是主动投资的另一个例子。

其他投资将是被动的。无论你预先投入多少，你的设计决策都不可避免地会出现错误。随着时间的推移，这些错误会变得显而易见。当你发现一个设计问题时，不要只是忽略它或修补它；花一点额外的时间来修复它。如果你进行战略性编程，你将不断地对系统设计进行小的改进。这与战术编程相反，在战术编程中，你不断地添加小的复杂性，从而在未来导致问题。

3.3 投资多少？

那么，正确的投资额是多少呢？大量的预先投资，例如试图设计整个系统，不会是有效的。这就是瀑布方法，我们知道它行不通。理想的设计往往会随着你对系统的经验而一点一点地出现。因此，最好的方法是持续不断地进行大量的小额投资。我建议将大约10–

2 0% 的总开发时间用于投资。这个数额很小
足以使其不会对您的日程安排产生重大影响，但又足够大，可以在一段时间内产生显著效益。因此，您的初始项目将需要 10–
2 比纯粹的战术方法长 0%。额外的时间将
带来更好的软件设计，并且您将在几个月内开始体验到好处。不久之后，您的开发速度将至少提高 10-20%，如果您采用战术编程。在这一点上，您的

投资变得免费：您过去投资的回报将节省
足够的时间来支付未来投资的成本。您将迅速收回
初始投资的成本。图 3.1说明了这种现象。

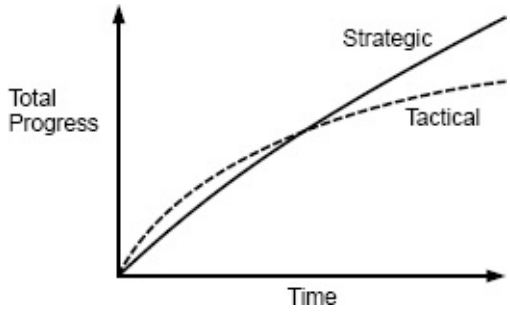

图 3.1：在开始时，与
战略方法相比，采用战术方法进行编程可以更快地取得进展。然而，在战术方法下，复杂性积累得更快，这降低了生产力。随着时间的推移，战略方法会带来更大的进步。注意：此图
仅作为定性说明；我不知道对精确
曲线形状的任何经验测量。

相反，如果您采用战术编程，您将以 10–
20% 的速度更快地完成您的第一个项目，但随着时间的推移，随着复杂性的积累，您的开发速度将会减慢。不久之后，您的编程速度将至少降低 10–20% 更慢。您将迅速归还您在开始时节省的所有时间，并且
在系统的剩余生命周期内，您的开发速度将比您
采用战略方法时更慢。如果您从未在严重退化的
代码库中工作过，请与曾经工作过的人交谈；他们会告诉您，糟糕的代码质量至少会使开发速度降低 20%。

3.4 创业公司和投资

在某些环境中，存在着强大的力量在对抗战略
方法。例如，早期创业公司感到巨大的压力，需要尽快发布他们的
早期版本。在这些公司中，即使是 10-20% 的
投资也似乎是负担不起的。因此，许多创业公司采取战术方法，
在设计上投入很少的精力，甚至在出现问题时更少地进行清理。
他们用这样的想法来合理化这一点：如果他们成功了，他们将有足够的钱来聘请额外的工程师来清理这些东西。

如果你所在的公司倾向于这个方向，你应该意识到一旦代码库变成了意大利面条，几乎不可能修复。在产品的整个生命周期中，你可能会支付高昂的开发成本。此外，

好的（或坏的）设计的回报来得很快，所以战术方法甚至不太可能加速你的第一个产品发布。

另一个需要考虑的因素是，公司成功的最重要的因素之一是其工程师的素质。降低开发成本的最佳方法是聘请优秀的工程师：他们的成本并不比平庸的工程师高多少，但生产力却高得多。然而，最好的工程师非常关心好的设计。如果你的代码库一团糟，消息会传出去，这将使你更难招聘。因此，你最终可能会得到平庸的工程师。这将增加你未来的成本，并可能导致系统结构更加退化。

Facebook 是一个鼓励战术编程的创业公司的例子。
多年来，该公司的座右铭是“快速行动，打破常规”。刚从大学毕业的新工程师被鼓励立即投入到公司的代码库中；工程师在入职的第一周就将提交推送到生产环境中是很正常的。从积极的方面来看，Facebook 树立了赋予员工权力的公司的声誉。工程师们拥有极大的自由度，几乎没有规则和限制来阻碍他们。

Facebook 作为一家公司取得了巨大的成功，但由于该公司的战术方法，其代码库受到了影响；大部分代码不稳定且难以理解，几乎没有注释或测试，而且使用起来很痛苦。随着时间的推移，该公司意识到其文化是不可持续的。
最终，Facebook 将其座右铭改为“以坚实的基础设施快速行动”，以鼓励其工程师更多地投资于好的设计。Facebook 是否能够成功清理多年战术编程积累的问题，仍有待观察。

公平地讲，我应该指出，Facebook 的代码可能并不比创业公司中的平均水平差多少。战术编程在创业公司中很常见；Facebook 只是一个特别明显的例子。

幸运的是，在硅谷，也有可能通过战略方法取得成功。谷歌和 VMware 与 Facebook 大约在同一时间发展起来，但这两家公司都采取了更具战略性的方法。这两家公司都非常重视高质量的代码和良好的设计，并且都构建了复杂的产品，这些产品通过可靠的软件系统解决了复杂的问题。这些公司强大的技术文化变得广为人知

在硅谷很有名。很少有其他公司能在招聘顶尖技术人才方面与他们竞争。

这些例子表明，一家公司可以通过任何一种方法取得成功。
然而，在一家关心软件设计并拥有干净代码库的公司工作会更有趣。

3.5 结论

好的设计不是免费的。它必须是你不断投资的东西，这样小问题才不会积累成大问题。幸运的是，好的设计最终会得到回报，而且比你想象的要快。

至关重要的是，要始终如一地应用战略方法，并将投资视为今天要做的事情，而不是明天。当你遇到困难时，很可能会把清理工作推迟到困难结束后。然而，这是一个滑坡；在当前的危机之后，几乎肯定会有另一个危机，以及之后的另一个危机。一旦你开始推迟设计改进，延迟很容易变成永久性的，你的文化很容易滑向战术方法。你等待解决设计问题的时间越长，问题就越大；解决方案变得更加令人生畏，这使得更容易进一步推迟它们。最有效的方法是每个工程师都不断地对好的设计进行小额投资。

第4章

模块应该足够深入

管理软件复杂性最重要的技术之一是设计系统，以便开发人员在任何给定时间只需要面对整体复杂性的一小部分。这种方法称为模块化设计，本章介绍其基本原理。

4.1 模块化设计

在模块化设计中，一个软件系统被分解成一组相对独立的模块。模块可以采用多种形式，例如类、子系统或服务。在理想的世界中，每个模块都将完全独立于其他模块：开发人员可以在任何模块中工作，而无需了解任何其他模块。在这个世界中，系统的复杂性将是其最差模块的复杂性。

不幸的是，这种理想是无法实现的。模块必须通过调用彼此的函数或方法来协同工作。因此，模块必须了解彼此的一些信息。模块之间会存在依赖关系：如果一个模块发生更改，其他模块可能需要更改以匹配。例如，方法的参数会在方法和任何调用该方法的代码之间创建依赖关系。如果所需的参数发生更改，则必须修改该方法的所有调用以符合新的签名。依赖关系可以采取许多其他形式，并且可能非常微妙。模块化设计的目标是最大限度地减少模块之间的依赖关系。

为了管理依赖关系，我们将每个模块分为两个部分：接口和实现。接口包含在不同模块工作的开发人员为了使用给定模块必须了解的所有内容。通常，接口描述了什么模块做什么，而不是如何它做。实现包含执行接口所做承诺的代码。在特定模块工作的开发人员必须

理解该模块的接口和实现，以及给定模块调用的任何其他模块的接口。开发人员不需要理解他或她正在工作的模块以外的模块的实现。

考虑一个实现平衡树的模块。该模块可能包含复杂的代码，用于确保树保持平衡。但是，这种复杂性对于模块的用户是不可见的。用户看到一个相对简单的接口，用于调用操作以插入、删除和获取树中的节点。要调用插入操作，调用者只需提供新节点的键和值；遍历树和拆分节点的机制在接口中不可见。

就本书而言，模块是任何具有接口和实现的代码单元。面向对象编程语言中的每个类都是一个模块。类中的方法或非面向对象语言中的函数也可以被认为是模块：每个模块都有一个接口和一个实现，并且模块化设计技术可以应用于它们。更高级别的子系统和服务也是模块；它们的接口可能采用不同的形式，例如内核调用或 HTTP 请求。本书中关于模块化设计的讨论主要集中在设计类上，但这些技术和概念也适用于其他类型的模块。

最好的模块是那些接口比实现简单得多的模块。这样的模块有两个优点。首先，一个简单的接口最大限度地减少了模块对系统其余部分施加的复杂性。
其次，如果以不更改其接口的方式修改模块，则没有其他模块会受到该修改的影响。如果模块的接口比其实现简单得多，那么可以更改模块的许多方面而不会影响其他模块。

4.2接口中有什么？

模块的接口包含两种信息：形式的和非形式的。接口的形式部分在代码中显式指定，并且编程语言可以检查其中一些部分的正确性。例如，方法的形式接口是其签名，其中包括其参数的名称和类型、其返回值的类型以及有关该方法引发的异常的信息。大多数编程语言确保方法的每次调用都提供正确数量和类型的

参数以匹配其签名。类的形式接口由其所有公共方法的签名以及任何公共变量的名称和类型组成。

每个接口还包括非正式元素。这些元素没有以编程语言可以理解或强制执行的方式指定。接口的非正式部分包括其高级行为，例如函数删除由其一个参数命名的文件这一事实。如果对类的使用存在约束（可能必须先调用一个方法才能调用另一个方法），那么这些约束也是类接口的一部分。一般来说，如果开发人员需要知道某个特定的信息才能使用一个模块，那么该信息就是该模块接口的一部分。接口的非正式方面只能使用注释来描述，并且编程语言无法确保描述是完整或准确的1。对于大多数接口来说，非正式方面比正式方面更大、更复杂。

一个明确指定的接口的好处之一是，它准确地表明了开发人员需要了解哪些内容才能使用相关的模块。这有助于消除第 2.2 节中描述的“未知未知”问题。

4.3 抽象

术语抽象与模块化设计的思想密切相关。抽象是实体的简化视图，它省略了不重要的细节。抽象很有用，因为它们使我们更容易思考和操作复杂的事物。

在模块化编程中，每个模块都以其接口的形式提供一个抽象。该接口呈现了模块功能的简化视图；从模块抽象的角度来看，实现的细节并不重要，因此它们从接口中省略。

在抽象的定义中，“不重要”这个词至关重要。从抽象中省略的不重要细节越多越好。但是，只有当细节不重要时，才能从抽象中省略它。抽象可能会在两个方面出错。首先，它可以包含实际上并不重要的细节；当发生这种情况时，它会使抽象比必要的更复杂，这会增加使用抽象的开发人员的认知负担。第二个错误是抽象省略了真正重要的细节。这会导致模糊：只查看抽象的开发人员将无法获得正确使用抽象所需的所有信息。一个

省略重要细节的抽象是虚假抽象：它可能看起来
很简单，但实际上并非如此。设计抽象的关键是理解
什么是重要的，并寻找能够最大限度地减少
重要信息量的设计。

举个例子，考虑一个文件系统。文件系统提供的抽象省略了许多细节，例如选择存储设备上哪些块用于给定文件中数据的机制。这些细节对于文件系统的用户来说并不重要（只要系统提供足够的性能）。

但是，文件系统实现的一些细节对于用户来说很重要。大多数文件系统将数据缓存在主内存中，并且可能会延迟将新数据写入存储设备，以提高性能。一些应用程序（如数据库）需要确切地知道何时将数据写入存储，以便它们可以确保在系统崩溃后数据将被保留。因此，将数据刷新到辅助存储的规则必须在文件系统的接口中可见。

我们依靠抽象来管理复杂性，不仅在编程中，而且在我们的日常生活中也普遍存在。微波炉包含复杂的电子设备，用于将交流电转换为微波辐射，并将该辐射分布在整个烹饪腔中。幸运的是，用户看到的是一个更简单的抽象，包括几个按钮来控制微波的时间和强度。汽车提供了一个简单的抽象，使我们能够在不了解电机、电池电源管理、防抱死制动系统、巡航控制等机制的情况下驾驶它们。

4.4 深度模块

最好的模块是那些提供强大功能但接口简单的模块。我用“深”这个词来描述这样的模块。为了形象地理解深度的概念，想象一下每个模块都用一个矩形表示，如图
4.1所示。每个矩形的面积与功能成正比
由模块实现。矩形的顶部边缘代表模块的接口；该边缘的长度表示接口的复杂性。最好的模块是深的：它们在简单的接口背后隐藏了大量的功能。一个深的模块是一个很好的抽象，因为只有一小部分内部复杂性对用户可见。

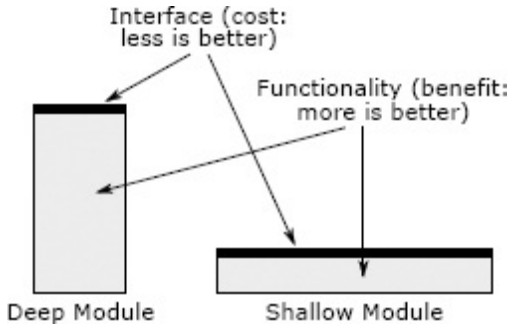

图 4.1：深模块和浅模块。最好的模块是深模块：它们允许通过简单的接口访问大量功能。浅模块是指接口相对复杂，但功能不多的模块：它没有隐藏太多的复杂性。

模块深度是一种思考成本与收益的方式。模块提供的收益是其功能。模块的成本（就系统复杂性而言）是其接口。模块的接口代表了模块强加给系统其余部分的复杂性：接口越小越简单，它引入的复杂性就越小。最好的模块是那些具有最大收益和最小成本的模块。接口是好的，但更多或更大的接口不一定更好！

Unix 操作系统及其后代（如 Linux）提供的文件 I/O 机制是深度接口的一个很好的例子。只有五个基本的 I/O 系统调用，具有简单的签名：

```
int open(const char* path, int flags, mode_t permissions);
ssize_t read(int fd, void* buffer, size_t count);
ssize_t write(int fd, const void* buffer, size_t count);
off_t lseek(int fd, off_t offset, int referencePosition);
int close(int fd);
```

open系统调用接受一个分层文件名，例如/a/b/c，并返回一个整数文件描述符，该描述符用于引用打开的文件。open的其他参数提供可选信息，例如文件是为读取还是写入而打开，如果没有现有文件是否应创建新文件，以及如果创建新文件，文件的访问权限。read和write系统调用在应用程序内存中的缓冲区区域和文件之间传输信息；close结束对文件的访问。大多数文件都是按顺序访问的，因此这是默认设置；但是，可以通过调用lseek系统调用来更改当前访问位置，从而实现随机访问。

Unix I/O 接口的现代实现需要数十万行代码，这些代码解决了以下复杂问题：

- 为了实现高效访问，文件在磁盘上是如何表示的？
- 目录是如何存储的，以及如何处理分层路径名来查找它们所引用的文件？
- 权限是如何强制执行的，以防止一个用户修改或删除另一个用户的文件？
- 文件访问是如何实现的？例如，中断处理程序和后台代码之间如何划分功能，以及这两个元素如何安全地通信？
- 当并发访问多个文件时，使用什么调度策略？
- 如何将最近访问的文件数据缓存在内存中，以减少磁盘访问的次数？
- 如何将各种不同的辅助存储设备（如磁盘和闪存驱动器）集成到单个文件系统中？

所有这些问题，以及更多的问题，都由 Unix 文件系统实现来处理；对于调用系统调用的程序员来说，它们是不可见的。
Unix I/O 接口的实现多年来发生了根本性的变化，但五个基本的内核调用没有改变。

另一个深层模块的例子是 Go 或 Java 等语言中的垃圾收集器。这个模块根本没有接口；它在幕后默默地工作，以回收未使用的内存。向系统中添加垃圾收集实际上会缩小其整体接口，因为它消除了释放对象的接口。垃圾收集器的实现非常复杂，但这种复杂性对使用该语言的程序员来说是隐藏的。

像 Unix I/O 和垃圾收集器这样的深层模块提供了强大的抽象，因为它们易于使用，但它们隐藏了重要的实现复杂性。

4.5 浅模块

另一方面，浅模块是指其接口相对于其提供的功能而言比较复杂的模块。例如，实现链表的类就是一个浅模块。操作链表不需要太多代码（插入或删除一个元素只需要几行代码），因此链表抽象并没有隐藏很多细节。链表的复杂性

接口几乎与其实现的复杂性一样大。浅类有时是不可避免的，但它们在管理复杂性方面并没有提供太多帮助。

这是一个来自软件设计课程项目中的浅方法的一个极端例子：

```
private void addNullValueForAttribute(String attribute) {
    data.put(attribute, null);
}
```

From the standpoint of managing complexity, this method makes things worse, 没有更好。该方法没有提供任何抽象，因为它的所有功能都可以通过其接口看到。例如，调用者可能需要知道该属性将存储在 data 变量中。考虑接口并不比考虑完整的实现更简单。如果该方法被正确地记录，那么文档将比该方法的代码更长。

调用该方法甚至比调用者直接操作 data 变量需要更多的击键次数。该方法增加了复杂性（以开发人员需要学习的新接口的形式），但没有提供任何补偿性的好处。

危险信号：浅模块

浅模块是指其接口相对于其提供的功能而言比较复杂。浅模块在对抗复杂性的战斗中并没有提供太多帮助，因为它们提供的好处（不必了解它们内部是如何工作的）被学习和使用它们的接口的成本所抵消。小型模块往往是浅模块。

4.6 类炎

不幸的是，深度类的价值在今天并没有得到广泛的认可。编程中的传统观点是，类应该是小的，而不是深的。
学生们经常被教导，类设计中最重要的是将较大的类分解成较小的类。关于方法，也经常给出同样的建议：“任何超过N行的方法都应该分成多个

方法”（N可以低至10）。这种方法会导致大量的浅类和方法，从而增加整体系统的复杂性。

“类应该小”这种做法的极端情况是一种我称之为类炎的综合症，它源于一种错误的观点，即“类是好的，所以更多的类更好”。在患有类炎的系统中，鼓励开发人员尽量减少每个新类中的功能：如果你想要更多的功能，就引入更多的类。类炎可能会导致类在个体上很简单，但它会增加整体系统的复杂性。小类不能贡献太多的功能，所以必须有很多小类，每个类都有自己的接口。这些接口累积起来，会在系统层面造成巨大的复杂性。由于每个类都需要样板代码，小类也会导致冗长的编程风格。

4.7 示例：Java 和 Unix I/O

如今，类炎最明显的例子之一是 Java 类库。Java 语言并不需要大量的小类，但类炎文化似乎已经在 Java 编程社区扎根。例如，为了打开一个文件以便从中读取序列化的对象，你必须创建三个不同的对象：

```
FileInputStream fileStream =
        new FileInputStream(fileName);
BufferedInputStream bufferedStream =
        new BufferedInputStream(fileStream);
ObjectInputStream objectStream =
        new ObjectInputStream(bufferedStream);
```

一个 FileInputStream对象仅提供基本的 I/O：它不能执行缓冲 I/O，也不能读取或写入序列化的对象。The BufferedInputStream 对象向 FileInputStream 添加缓冲，而 ObjectInputStream 添加读取和写入序列化对象的功能。上面的代码中的前两个对象，fileStream 和 bufferedStream，在文件打开后就再也不会使用；所有未来的操作都使用 objectStream。

特别令人恼火（且容易出错）的是，必须通过创建一个单独的 BufferedInputStream 对象来显式请求缓冲；如果开发人员忘记创建此对象，则不会有缓冲，并且 I/O 将会很慢。

也许 Java 开发人员会争辩说，并非每个人都想使用缓冲

用于文件 I/O，因此不应将其构建到基本机制中。他们可能会争辩说，最好将缓冲分开，以便人们可以选择是否使用它。提供选择是好的，但是接口的设计应该使常见情况尽可能简单（参见第 6 页的公式）。几乎每个文件 I/O 用户都希望使用缓冲，因此应该默认提供。对于那些少数不希望使用缓冲的情况，库可以提供一种禁用它的机制。任何禁用缓冲的机制都应该在接口中清晰地分离（例如，通过为 FileInputStream 提供不同的构造函数，或者通过一种禁用或替换缓冲机制的方法），以便大多数开发人员甚至不需要意识到它的存在。

相比之下，Unix 系统调用的设计者使常见情况变得简单。例如，他们认识到顺序 I/O 是最常见的，因此他们将其作为默认行为。随机访问仍然相对容易实现，使用 lseek 系统调用，但是仅进行顺序访问的开发人员无需了解该机制。如果一个接口有很多特性，但是大多数开发人员只需要了解其中的几个，那么该接口的有效复杂性就是常用特性的复杂性。

4.8 结论

通过将模块的接口与其实现分离，我们可以向系统的其余部分隐藏实现的复杂性。模块的用户只需要理解其接口提供的抽象。设计类和其他模块时，最重要的问题是使它们具有深度，以便它们为常见用例提供简单的接口，但仍然提供重要的功能。这最大限度地提高了隐藏的复杂性。

1存在一些语言，主要是在研究社区中，其中方法或函数的整体行为可以使用规范语言正式描述。可以自动检查规范，以确保它与实现匹配。一个有趣的问题是，这种正式规范是否可以取代接口的非正式部分。我目前的观点是，用英语描述的接口可能比用正式规范语言编写的接口对开发人员来说更直观和易于理解。

第五章

信息隐藏（和泄漏）

第四章认为模块应该具有深度。本章以及接下来的几章将讨论创建深度模块的技术。

5.1 信息隐藏

实现深度模块最重要的技术是信息隐藏。这项技术最早由大卫·帕纳斯[1]描述。其基本思想是每个模块都应该封装一些知识，这些知识代表设计决策。这些知识嵌入在模块的实现中，但不会出现在其接口中，因此其他模块无法看到。

模块中隐藏的信息通常包含关于如何实现某种机制的细节。以下是一些可能隐藏在模块中的信息的示例：

- 如何在 B 树中存储信息，以及如何高效地访问它。
- 如何识别与文件中每个逻辑块对应的物理磁盘块。

- 如何实现 TCP 网络协议。
- 如何在多核处理器上调度线程。
- 如何解析 JSON 文档。

隐藏的信息包括与该机制相关的数据结构和算法。它还可以包括较低级别的细节，例如页面大小，并且它可以包括更高级别的概念，这些概念更抽象，例如假设大多数文件都很小。

信息隐藏通过两种方式降低复杂性。首先，它简化了模块的接口。该接口反映了模块功能的更简单、更抽象的视图，并隐藏了细节；这降低了使用该模块的开发人员的认知负荷。例如，使用 B 树类的开发人员无需担心树中节点的理想扇出，或者如何保持

树的平衡。其次，信息隐藏使系统更容易演进。如果一条信息被隐藏，那么在该信息之外的模块中，就不会存在对该信息的依赖关系，因此与该信息相关的设计变更只会影响这一个模块。例如，如果 TCP 协议发生变化（例如，引入一种新的拥塞控制机制），则协议的实现必须进行修改，但使用 TCP 发送和接收数据的高级代码不需要进行任何更改。

在设计新模块时，您应该仔细考虑哪些信息可以隐藏在该模块中。如果您可以隐藏更多信息，您还应该能够简化模块的接口，这使得模块更深入。

注意：通过将类中的变量和方法声明为private来隐藏它们与信息隐藏不是一回事。私有元素可以帮助实现信息隐藏，因为它们使得无法从类外部直接访问这些项。但是，关于私有项的信息仍然可以通过公共方法（如 getter 和 setter 方法）公开。当这种情况发生时，变量的性质和用法与变量是公共变量时一样暴露。

信息隐藏的最佳形式是信息完全隐藏在模块中，因此对于模块的用户来说，它是无关紧要且不可见的。但是，部分信息隐藏也具有价值。例如，如果某个特定的特性或信息仅被类的少数用户需要，并且它通过单独的方法访问，因此在最常见的用例中不可见，那么该信息在很大程度上是被隐藏的。与对类的每个用户都可见的信息相比，此类信息将产生更少的依赖关系。

5.2 信息泄露

信息隐藏的反面是信息泄露。当一个设计决策反映在多个模块中时，就会发生信息泄露。这会在模块之间创建依赖关系：对该设计决策的任何更改都需要更改所有涉及的模块。如果一条信息反映在模块的接口中，那么根据定义，它已经被泄露；因此，更简单的接口往往与更好的信息隐藏相关。但是，即使信息没有出现在模块的接口中，也可能被泄露。假设两个类都知道一种特定的文件格式（可能一个类读取该格式的文件，另一个类写入该格式的文件）。即使这两个类都没有在其接口中公开

该信息，它们都依赖于该文件格式：如果该格式
发生更改，则需要修改这两个类。像这样的后门泄露比通过接口泄露更有害，因为它不明显。

信息泄露是软件设计中最重要的危险信号之一。作为软件设计师，您可以学习的最佳技能之一是对信息泄露的高度敏感性。如果您遇到类之间的信息泄露，请问自己“我如何重新组织这些类，以便这种特定知识仅影响单个类？”如果受影响的类相对较小并且与泄露的信息紧密相关，那么将它们合并到单个类中可能是有意义的。另一种可能的方法是将信息从所有受影响的类中提取出来，并创建一个新的类，该类仅封装该信息。但是，只有在您可以找到一个简单的接口来抽象出细节时，此方法才有效；如果新类通过其接口公开了大部分知识，那么它不会提供太多价值（您只是用通过接口泄露代替了后门泄露）。

红旗：信息泄露

当相同知识在多个地方使用时，就会发生信息泄露，例如两个不同的类都理解特定类型文件的格式。

5.3 时间分解

信息泄露的一个常见原因是一种我称之为时间分解的设计风格。在时间分解中，系统的结构对应于操作发生的时序。考虑一个以特定格式读取文件、修改文件内容然后再次写出文件的应用程序。通过时间分解，此应用程序可能会被分解为三个类：一个用于读取文件，另一个用于执行修改，第三个用于写出新版本。文件读取和文件写入步骤都具有关于文件格式的知识，这会导致信息泄露。解决方案是将读取和写入文件的核心机制组合成一个类。这个类将在应用程序的读取和写入阶段被使用。

很容易陷入时间分解的陷阱，因为在您编写代码时，操作必须发生的顺序通常在您的脑海中。但是，大多数设计决策在应用程序的生命周期中会在几个不同的时间显现出来；因此，时间分解通常会导致信息泄露。

顺序通常确实很重要，因此它将反映在应用程序的某个地方。但是，它不应反映在模块结构中，除非该结构与信息隐藏一致（可能不同的阶段使用完全不同的信息）。在设计模块时，请关注执行每个任务所需的知识，而不是任务发生的顺序。

危险信号：时间分解

在时间分解中，执行顺序反映在代码结构中：
在不同时间发生的操作位于不同的方法或类中。如果在执行的不同点使用相同的知识，它将被编码在多个地方，从而导致信息泄露。

5.4 示例：HTTP服务器

为了说明信息隐藏中的问题，让我们考虑一下学生在软件设计课程中实现HTTP协议时所做的设计决策。
了解他们做得好的地方和他们存在问题的地方都很有用。

HTTP 是 Web 浏览器用于与 Web 服务器通信的机制。当用户点击 Web 浏览器中的链接或提交表单时，浏览器使用 HTTP 通过网络向 Web 服务器发送请求。一旦服务器处理了请求，它就会向浏览器发回响应；响应通常包含要显示的新网页。HTTP 协议指定了请求和响应的格式，两者都以文本形式表示。图 5.1显示了一个描述表单提交的 HTTP 请求示例。

课程中的学生被要求实现一个或多个类，以便 Web 服务器能够轻松接收传入的 HTTP 请求并发送响应。

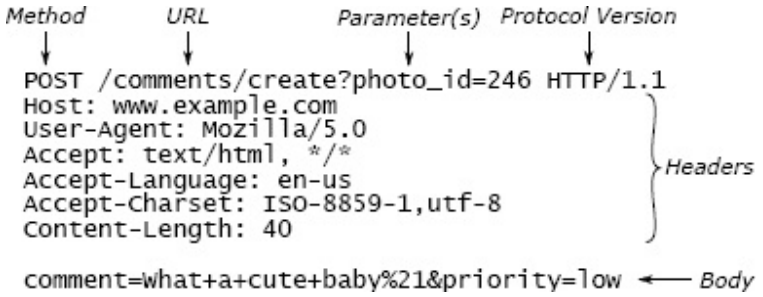

图 5.1：HTTP 协议中的 POST 请求由通过 TCP 套接字发送的文本组成。每个请求都包含一个初始行，一个由空行终止的标头集合，以及一个可选的主体。初始行包含请求类型（POST用于提交表单数据），一个指示操作的 URL（/comments/create）和可选参数（photo_id的值为 246），以及发送者使用的 HTTP 协议版本。每个标头行都包含一个名称，例如Content-Length，后跟其值。对于此请求，主体包含其他参数（comment和priority）。

5.5 示例：类太多

学生们最常犯的错误是将他们的代码分成大量的浅层类，这导致了类之间的信息泄露。
一个团队使用了两个不同的类来接收 HTTP 请求；第一个类从网络连接中读取请求到一个字符串，第二个类解析该字符串。这是一个时间分解的例子（“首先我们读取请求，然后我们解析它”）。信息泄露发生的原因是，如果不解析消息的大部分内容，就无法读取 HTTP 请求；例如，

Content-Length标头指定了请求主体的长度，因此必须解析标头才能计算总请求长度。因此，两个类都需要理解 HTTP 请求的大部分结构，并且解析代码在两个类中都是重复的。这种方法也为调用者带来了额外的复杂性，他们必须以特定的顺序调用不同类中的两个方法才能接收请求。

由于这些类共享了如此多的信息，因此最好将它们合并到一个类中，该类同时处理请求读取和解析。
这提供了更好的信息隐藏，因为它将请求格式的所有知识隔离在一个类中，并且它还为调用者提供了一个更简单的接口（只需调用一个方法）。

这个例子说明了软件设计中的一个普遍主题：信息隐藏通常可以通过稍微扩大一个类来实现。这样做的一个原因是将所有与特定功能相关的代码（例如解析 HTTP 请求）集中在一起，以便生成的类包含所有内容

与该功能相关。增加类大小的第二个原因是提高接口的级别；例如，与其为计算的三个步骤分别设置单独的方法，不如设置一个执行整个计算的单一方法。这可以产生更简单的接口。这两个好处都适用于前一段的例子：合并这些类将与解析 HTTP 请求相关的所有代码集中在一起，并将两个外部可见的方法替换为一个。合并后的类比原始类更深入。

当然，将更大的类的概念推得太远也是有可能的（例如，整个应用程序使用一个类）。第 9 章将讨论在什么情况下将代码分离成多个较小的类是有意义的。

5.6 示例：HTTP 参数处理

在服务器收到 HTTP 请求后，服务器需要访问请求中的一些信息。图 5.1 中的请求处理代码可能需要知道 `photo_id` 参数的值。参数可以在请求的第一行（图 5.1 中的 `photo_id`）中指定，或者有时在正文中（图 5.1 中的 `comment` 和 `priority`）。每个参数都有一个名称和一个值。参数的值使用一种特殊的编码，称为

URL 编码；例如，在图 5.1 中 `comment` 的值中，“+”用于表示空格字符，而“%21”用于代替“！”。为了处理请求，服务器将需要一些参数的值，并且希望它们以未编码的形式出现。

大多数学生项目在参数处理方面做出了两个不错的选择。首先，他们认识到服务器应用程序并不关心参数是在标头行还是请求正文中指定的，因此他们向调用者隐藏了这种区别，并将来自两个位置的参数合并在一起。其次，他们隐藏了 URL 编码的知识：HTTP 解析器在将参数值返回给 Web 服务器之前对其进行解码，因此图 5.1 中 `comment` 参数的值将作为“What a cute baby!”返回，而不是“What+a+cute+baby%21”）。在这两种情况下，信息隐藏都为使用 HTTP 模块的代码带来了更简单的 API。

然而，大多数学生使用了一个过于浅显的接口来返回参数，这导致了信息隐藏的机会丧失。

大多数项目使用 `HTTPRequest` 类型的对象来保存已解析的 HTTP

请求，并且 `HTTPRequest` 类有一个像下面这样的单一方法来返回参数：

```
public Map<String, String> getParams() {
    return this.params;
}
```

Rather than returning a single parameter, the method returns a reference to the Map在内部用于存储所有参数。此方法是浅层的，它
公开了HTTPRequest类用于存储的内部表示
参数。对该表示的任何更改都将导致对
接口的更改，这将需要对所有调用者进行修改。当实现
被修改时，更改通常涉及关键数据的表示形式的更改
结构（例如，为了提高性能）。因此，重要的是要避免
尽可能多地公开内部数据结构。这种方法也使得
调用者需要做更多的工作：调用者必须首先调用getParams，然后它必须调用
另一个方法来从Map中检索特定参数。最后，调用者
必须意识到他们不应该修改getParams返回的Map，因为
这将影响HTTPRequest的内部状态。

以下是检索参数值的更好接口：

```
public String getParameter(String name) { ... }
public int getIntParameter(String name) { ... }
```

getParameter以字符串形式返回参数值。它提供了比上面的getParams更深入的
接口；更重要的是，它隐藏了参数的内部
表示。getIntParameter将参数的值从
HTTP请求中的字符串形式转换为整数（例如，Figure 5.1中的photo_id
参数）。这节省了调用者单独请求字符串到整数的
转换，并对调用者隐藏了该机制。
如果需要，可以定义其他数据类型的其他方法，例如getDoubleParameter。（所有这些方法都会在所需参数不存在时，或者无法转换为请求的类型时抛出异常；异常声明已在上面的代码中省略）。

5.7 示例：HTTP 响应中的默认值

HTTP 项目还必须提供对生成 HTTP 响应的支持。
学生们在这个领域最常犯的错误是默认值不足。
每个 HTTP 响应都必须指定一个 HTTP 协议版本；一个团队要求

调用者在创建响应对象时显式指定此版本。
然而，响应版本必须与请求对象中的版本相对应，并且在发送响应时必须已经将请求作为参数传递（它指示将响应发送到哪里）。因此，HTTP类自动提供响应版本更有意义。调用者不太可能知道要指定什么版本，如果调用者确实指定了一个值，则可能会导致HTTP库和调用者之间的信息泄漏。HTTP响应还包括一个Date标头，用于指定发送响应的时间；HTTP库也应该为此提供一个合理的默认值。

默认值阐明了接口设计应尽可能简化常见情况的原则。它们也是部分信息隐藏的一个例子：在正常情况下，调用者不需要知道默认项的存在。在调用者需要覆盖默认值的极少数情况下，它必须知道该值，并且可以调用一个特殊方法来修改它。

只要有可能，类就应该在没有被明确要求的情况下“做正确的事情”。默认值就是这样的一个例子。第26页上的Java I/O示例以一种消极的方式说明了这一点文件I/O中的缓冲是如此普遍地受欢迎，以至于没有人应该明确地要求它，甚至不知道它的存在；I/O类应该做正确的事情并自动提供它。最好的功能是那些你甚至不知道它们就存在的功能。

危险信号：过度暴露

如果常用功能的API迫使用户学习很少使用的其他功能，这会增加不需要这些很少使用的功能的用户的认知负担。

5.8 类内的信息隐藏

本章中的示例侧重于与类的外部可见API相关的信息隐藏，但信息隐藏也可以应用于系统中的其他级别，例如在类中。尝试设计类中的私有方法，以便每个方法封装一些信息或

功能并将其隐藏在类的其余部分。此外，尽量减少每个实例变量的使用位置数量。一些变量可能需要在整个类中广泛访问，但其他变量可能只需要在几个地方使用；如果可以减少变量的使用位置数量，则可以消除类中的依赖关系并降低其复杂性。

5.9 走得太远

只有当被隐藏的信息在模块外部不需要时，信息隐藏才有意义。如果模块外部需要该信息，那么你必须不能隐藏它。假设一个模块的性能受到某些配置参数的影响，并且该模块的不同用途将需要不同的参数设置。在这种情况下，重要的是这些参数在模块的接口中公开，以便可以适当地调整它们。作为一名软件设计师，你的目标应该是尽量减少模块外部所需的信息量；例如，如果一个模块可以自动调整其配置，那比公开配置参数更好。但是，重要的是要认识到哪些信息是模块外部需要的，并确保它被公开。

5.10 结论

信息隐藏和深度模块密切相关。如果一个模块隐藏了大量信息，那么它往往会增加模块提供的功能量，同时减少其接口。这使得模块更深入。

相反，如果一个模块没有隐藏太多信息，那么要么它没有太多功能，要么它有一个复杂的接口；无论哪种方式，该模块都是浅层的。

当将一个系统分解成模块时，尽量不要受到操作在运行时发生的顺序的影响；这将引导你走向时间分解的道路，这将导致信息泄露和浅层模块。相反，考虑一下执行应用程序任务所需的各种知识，并将每个模块设计为封装一个或几个这些知识。这将产生一个干净而简单的设计，具有深度模块。

[1]David Parnas，“将系统分解为模块时使用的标准”，*Communications of the ACM*，1972年12月。

第6章

通用模块更深入

在设计新模块时，你将面临的最常见的决定之一是以通用还是专用方式实现它。有些人可能会认为你应该采取通用的方法，在这种方法中，你实现一种机制，可以用来解决广泛的问题，而不仅仅是今天重要的问题。在这种情况下，新的机制可能会在未来找到意想不到的用途，从而节省时间。通用方法似乎与第3章中讨论的投资心态一致，你在前期花费更多的时间来节省以后的时间。

另一方面，我们知道很难预测软件系统未来的需求，因此通用的解决方案可能包括实际上永远不需要的设施。此外，如果你实现的东西过于通用，它可能无法很好地解决你今天遇到的特定问题。因此，有些人可能会认为最好关注今天的需求，只构建你知道你需要的东西，并针对你今天计划使用它的方式进行专门化。如果你采用专用方法并在以后发现其他用途，你可以随时重构它以使其通用。专用方法似乎与软件开发的增量方法一致。

6.1 使类在某种程度上具有通用性

以我的经验来看，最佳方案是以某种通用方式实现新模块。 “某种通用”这个短语意味着模块的功能应该反映您当前的需求，但其接口不应如此。 相反，接口应该足够通用，以支持多种用途。 该接口应该易于满足当今的需求，而又不会专门与这些需求相关联。 “某种”这个词很重要：不要得意忘形

并构建一些通用的东西，以至于难以用于您当前的需求。

通用方法最重要的（也许令人惊讶的）好处是，它比专用方法产生更简单和更深入的接口。如果将来您将该类重用于其他目的，通用方法还可以节省您的时间。但是，即使该模块仅用于其原始目的，由于其简单性，通用方法仍然更好。

6.2 示例：为编辑器存储文本

让我们考虑一个来自软件设计课程的例子，学生们被要求构建简单的GUI文本编辑器。编辑器必须显示一个文件，并允许用户通过点击和输入来编辑文件。编辑器必须支持在不同窗口中同时显示同一文件的多个视图；它们还必须支持对文件修改的多级撤销和重做。

每个学生项目都包含一个管理文件底层文本的类。文本类通常提供将文件加载到内存、读取和修改文件文本以及将修改后的文本写回文件的方法。

许多学生团队为文本类实现了专用API。他们知道该类将用于交互式编辑器，因此他们考虑了编辑器必须提供的功能，并针对这些特定功能定制了文本类的API。例如，如果编辑器的用户按下退格键，编辑器会立即删除光标左侧的字符；如果用户按下删除键，编辑器会立即删除光标右侧的字符。了解到这一点，一些团队在文本类中创建了一个方法来支持每个这些特定功能：

```
void backspace(Cursor cursor);
void delete(Cursor cursor);
```

这些方法中的每一个都将光标位置作为其参数；一种特殊类型的光标表示这个位置。编辑器还必须支持可以复制或删除的选择。学生们通过定义一个Selection类并在删除期间将该类的对象传递给文本类来处理这个问题：

```
void deleteSelection(Selection selection);
```

学生们可能认为，如果文本类的方法与用户可见的功能相对应，那么实现用户界面会更容易

用户。然而，在现实中，这种专业化并没有为用户界面代码带来多少好处，而且为从事用户界面或文本类工作的开发人员带来了很高的认知负担。文本类最终包含了大量浅层方法，每种方法只适用于一种用户界面操作。许多方法，例如 delete，只在一个地方被调用。因此，从事用户界面工作的开发人员必须了解文本类的大量方法。

这种方法在用户界面和文本类之间造成了信息泄漏。与用户界面相关的抽象概念，如选择或退格键，都反映在文本类中；这增加了从事文本类工作的开发人员的认知负担。每个新的用户界面操作都需要在文本类中定义一个新的方法，因此从事用户界面工作的开发人员很可能最终也会从事文本类的工作。类设计的目标之一是允许每个类独立开发，但这种专门化的方法将用户界面和文本类联系在一起。

6.3 一个更通用的API

一个更好的方法是使文本类更通用。它的API应该只根据基本的文本特征来定义，而不反映将使用它实现的更高级别的操作。例如，只需要两种方法来修改文本：

```
void insert(Position position, String newText);
void delete(Position start, Position end);
```

第一个方法在文本中的任意位置插入任意字符串，第二个方法删除位置大于或等于 start 但小于 end 的所有字符。此API还使用更通用的类型

Position 而不是 Cursor，它反映了特定的用户界面。文本类还应提供用于操作文本中位置的通用工具，例如以下：

```
Position changePosition(Position position, int numChars);
```

此方法返回一个新位置，该位置与给定位置相距给定的字符数。如果 numChars 参数为正数，则新位置比 position 在文件中更靠后；如果 numChars 为负数，则新位置在 position 之前。当

必要时，该方法会自动跳到下一行或上一行。使用这些方法，可以使用以下代码实现删除键（假设 cursor 变量保存当前光标位置）：

```
text.delete(光标, text.changePosition(光标, 1));
```

类似地，退格键可以如下实现：

```
text.delete(text.changePosition(cursor, -1), cursor);
```

使用通用的文本API，实现诸如删除和退格等用户界面功能的代码比使用专用文本API的原始方法要长一些。然而，新代码比旧代码更清晰。在用户界面模块工作的开发人员可能关心退格键删除了哪些字符。使用新代码，这一点很明显。使用旧代码，开发人员必须转到文本类并阅读退格方法的文档和/或代码，以验证其行为。此外，通用方法比专用方法总体上代码更少，因为它用少量通用方法替换了文本类中的大量专用方法。

使用通用接口实现的文本类可能用于交互式编辑器之外的其他目的。举个例子，假设您正在构建一个应用程序，该应用程序通过将所有出现的特定字符串替换为另一个字符串来修改指定的文件。专用文本类的方法，如退格和删除，对于此应用程序几乎没有价值。但是，通用文本类已经具有新应用程序所需的大部分功能。所有缺少的是一种搜索给定字符串的下一个出现位置的方法，例如：

```
Position findNext(Position start, String string);
```

当然，交互式文本编辑器很可能具有搜索和替换的机制，在这种情况下，文本类已经包含此方法。

6.4 通用性带来更好的信息隐藏

通用方法在文本和用户界面类之间提供了更清晰的分离，从而实现了更好的信息隐藏。文本类不需要知道用户界面的具体细节，例如如何处理退格键；这些细节现在封装在用户界面类中。

可以在不创建文本类中的新支持功能的情况下添加新的用户界面功能。通用接口还减少了认知负荷：在用户界面上工作的开发人员只需要学习一些简单的方法，这些方法可以重复用于各种目的。

文本类的原始版本中的退格方法是一个错误的抽象。它声称隐藏了有关删除哪些字符的信息，但是用户界面模块确实需要知道这一点；用户界面开发人员很可能会阅读退格方法的代码，以便确认其精确行为。将该方法放在文本类中只会使用户界面开发人员更难获得他们需要的信息。软件设计最重要的要素之一是确定谁需要知道什么，以及何时需要知道。

当细节很重要时，最好将它们明确化，并尽可能地显而易见，例如后退键操作的修订实现。
将这些信息隐藏在接口后面只会造成模糊。

6.5 自我提问

识别一个清晰的通用类设计比创建一个更容易。这里有一些你可以问自己的问题，这将帮助你找到接口的通用性和专用性之间的正确平衡。

覆盖我当前所有需求的最简单接口是什么？如果你在不降低其整体功能的情况下减少 API 中的方法数量，那么你可能正在创建更通用的方法。专用文本 API 至少有三种删除文本的方法：退格键、删除和删除选择。更通用的 API 只有一种删除文本的方法，它可以满足所有三种用途。只有在每个单独方法的 API 保持简单的情况下，减少方法数量才有意义；如果你必须引入大量额外的参数才能减少方法数量，那么你可能并没有真正简化事情。

此方法将在多少种情况下使用？如果一个方法是为一种特定用途而设计的，例如退格键方法，这是一个危险信号，表明它可能过于专用。看看你是否可以用一个通用的方法替换几个专用的方法。

这个 API 对于我当前的需求是否易于使用？这个问题可以帮助你确定你在使 API 变得简单和通用方面是否走得太远。如果你必须编写大量的额外代码才能将一个类用于你当前的目的，这是一个危险信号，表明该接口没有提供正确的功能。例如，文本类的一种方法是围绕单字符操作进行设计：插入插入一个字符，删除删除一个字符。这个 API 既简单又通用。

然而，对于文本编辑器来说，它并不是特别容易使用：更高级别的代码将包含大量的循环来插入或删除字符范围。单字符方法对于大型操作来说也是低效的。因此，文本类最好内置对字符范围操作的支持。

6.6 结论

通用接口比专用接口具有许多优势。
它们往往更简单，方法更少，深度也更浅。它们还在类之间提供更清晰的分离，而专用接口往往会在类之间泄漏信息。使您的模块在某种程度上具有通用性是降低整体系统复杂性的最佳方法之一。

第七章

不同的层，不同的抽象

软件系统由多层组成，其中较高层使用较低层提供的设施。在一个设计良好的系统中，每一层都提供与上下层不同的抽象；如果你跟踪一个单一操作，当它通过调用方法在各层之间上下移动时，抽象会随着每次方法调用而改变。例如：

- 在文件系统中，最上层实现文件抽象。一个文件由一个可变长度的字节数组组成，可以通过读取和写入可变长度的字节范围来更新。文件系统中下一层较低的层在内存中实现固定大小的磁盘块的缓存；调用者可以假定经常使用的块将保留在内存中，以便可以快速访问它们。最底层由设备驱动程序组成，它们在辅助存储设备和内存之间移动块。

- 在诸如TCP之类的网络传输协议中，最上层提供的抽象是从一台机器可靠地传送到另一台机器的字节流。此级别构建在较低级别之上，该级别在机器之间以尽力而为的方式传输有界大小的数据包：大多数数据包将成功传递，但是某些数据包可能会丢失或乱序传递。

如果一个系统包含具有相似抽象的相邻层，这是一个危险信号，表明类分解存在问题。本章讨论发生这种情况的情况、由此产生的问题以及如何重构以消除这些问题。

7.1 通过方法

当相邻层具有相似的抽象时，问题通常以传递方法的形式表现出来。传递方法是指除了调用另一个方法（其签名与调用方法相似或相同）之外，几乎不做任何事情的方法。例如，一个实现GUI的学生项目

文本编辑器包含一个几乎完全由传递方法组成的类。

以下是从该类中提取的内容：

```
public class TextDocument ... {
     private TextArea textArea;
     private TextDocumentListener listener;
     ...
     public Character getLastTypedCharacter() {
          return textArea.getLastTypedCharacter();
     }
     public int getCursorOffset() {
          return textArea.getCursorOffset();
     }
     public void insertString(String textToInsert,
            int offset) {
          textArea.insertString(textToInsert, offset);
     }
     public void willInsertString(String stringToInsert, int offset) {
          if (listener != null) {
             listener.willInsertString(this, stringToInsert, offset);
          }
     }
     ...
}
```

13 of the 15 public methods in that class were pass-through methods.

危险信号：传递方法

直通方法是指除了将其参数传递给另一个方法（通常具有与直通方法相同的 API）之外，什么也不做的方法。这通常表明类之间没有明确的责任划分。

直通方法使类变得更浅：它们增加了类的接口复杂性，从而增加了复杂性，但它们并没有增加系统的总体功能。在上面的四种方法中，只有最后一种方法具有任何功能，即使在那里，它也是微不足道的：该方法检查一个变量的有效性。直通方法还在类之间创建依赖关系：如果 TextArea 中 insertString 方法的签名发生更改，那么 TextDocument 中的 insertString 方法也必须更改以匹配。

直通方法表明类之间的责任划分存在混淆。在上面的示例中， TextDocument 类提供了一个 insertString 方法，但是插入文本的功能完全在 TextArea 中实现。这通常是一个坏主意：功能的接口应该与实现该功能的类相同。当你看到从一个类到另一个类的直通方法时，请考虑这两个类，并问自己“这些类分别负责哪些特性和抽象？”你可能会注意到这些类之间的责任存在重叠。

解决方案是重构这些类，以便每个类都具有独特且连贯的责任集。图 7.1 说明了几种方法。图 7.1(b) 中显示的一种方法是将较低级别的类直接暴露给较高级别类的调用者，从而消除了较高级别类对该功能的所有责任。另一种方法是在类之间重新分配功能，如 Figure 中所示

7.1(c)。最后，如果这些类无法解开，最好的解决方案可能是将它们合并，如 图 7.1(d) 所示。

在上面的示例中，有三个类具有相互交织的责任：TextDocument、TextArea 和 TextDocumentListener。学生通过在类之间移动方法并将这三个类合并为两个类来消除直通方法，这两个类的职责更加明确。

7.2 何时可以接受接口重复？

具有相同签名的方法并不总是坏事。重要的是每个新方法都应该贡献重要的功能。直通方法不好，因为它们不贡献任何新功能。

一个方法调用另一个具有相同签名的方法的一个有用示例是 dispatcher。调度器是一种使用其参数的方法

选择调用其他几种方法之一；然后它将其大部分或全部参数传递给所选方法。调度器的签名通常与其调用的方法的签名相同。即便如此，调度器也提供了有用的功能：它选择其他几种方法中的哪一种应该执行每个任务。

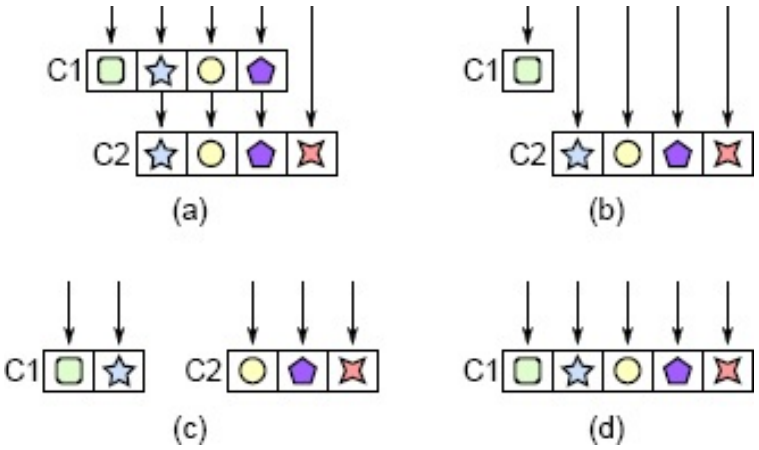

图 7.1：直通方法。在 (a) 中，类 C1 包含三个直通方法，它们除了调用 C2 中具有相同签名的方法之外什么也不做（每个符号代表一个特定的方法签名）。
通过让 C1 的调用者直接调用 C2（如 (b) 所示），通过在 C1 和 C2 之间重新分配功能以避免类之间的调用（如 (c) 所示），或者通过组合类（如 (d) 所示），可以消除直通方法。

例如，当 Web 服务器收到来自 Web 浏览器的传入 HTTP 请求时，它会调用一个调度器，该调度器检查传入请求中的 URL，并选择一个特定的方法来处理该请求。某些 URL 可能通过返回磁盘上文件的内容来处理；其他 URL 可能通过调用 PHP 或 JavaScript 等语言中的过程来处理。调度过程可能非常复杂，并且通常由一组与传入 URL 匹配的规则驱动。

只要每个方法都提供有用且不同的功能，多个方法具有相同的签名就可以了。调度器调用的方法具有此属性。另一个例子是具有多个实现的接口，例如操作系统中的磁盘驱动程序。每个驱动程序都为不同类型的磁盘提供支持，但它们都具有相同的接口。

当多个方法提供同一接口的不同实现时，它可以减少认知负荷。一旦你使用过其中一种方法，就可以更容易地使用其他方法，因为你不需要学习新的接口。

这样的方法通常位于同一层，并且它们不会相互调用。

7.3 装饰器

装饰器设计模式（也称为“包装器”）是一种鼓励跨层 API 复制的模式。装饰器对象接受一个现有对象并扩展其功能；它提供与底层对象相似或相同的 API，并且其方法调用底层对象的方法。在第 4 章的 Java I/O 示例中，BufferedInputStream 类是一个装饰器：给定一个 InputStream 对象，它提供相同的 API 但引入了缓冲。例如，当调用其 read 方法来读取单个字符时，它会调用底层 InputStream 上的 read 来读取更大的块，并保存额外的字符以满足未来的 read 调用。另一个例子出现在窗口系统中：Window 类实现了一种简单的不可滚动窗口形式，而 ScrollableWindow 类通过添加水平和垂直滚动条来装饰 Window 类。

装饰器的动机是将类的专用扩展与更通用的核心分开。但是，装饰器类往往比较浅薄：
它们为少量的新功能引入了大量的样板代码。装饰器类通常包含许多传递方法。很容易过度使用装饰器模式，为每个小的新功能创建一个新的类。这导致了浅层类的爆炸式增长，例如Java I/O示例。

在创建装饰器类之前，请考虑以下替代方案：

- 您能否直接将新功能添加到底层类，而不是创建装饰器类？如果新功能是相对通用的，或者它在逻辑上与底层类相关，或者底层类的大多数用途也会使用新功能，那么这样做是有意义的。

  例如，几乎每个创建 Java InputStream 的人也会创建一个 BufferedInputStream，并且缓冲是 I/O 的一个自然组成部分，因此这些类应该被合并。
- 如果新功能是专门针对特定用例的，那么将其与用例合并，而不是创建一个单独的类，是否有意义？

- 您能否将新功能与现有的装饰器合并，而不是创建一个新的装饰器？这将导致一个更深的装饰器类，而不是多个浅层的装饰器类。
- 最后，问问自己，新功能是否真的需要包装

现有功能：您能否将其实现为一个独立于基类的独立类？在窗口示例中，滚动条可能可以与主窗口分开实现，而无需包装其所有现有功能。

有时装饰器是有意义的，但通常有更好的替代方案。

7.4 接口与实现

“不同层，不同抽象”规则的另一个应用是，类的接口通常应该与其实现不同：内部使用的表示应该与接口中出现的抽象不同。如果两者具有相似的抽象，那么该类可能不是很深。例如，在第 6 章中讨论的文本编辑器项目中，大多数团队都根据文本行来实现文本模块，每行文本单独存储。一些团队还围绕行设计了文本类的 API，其中包含诸如 getLine 和 putLine 等方法。

然而，这使得文本类变得浅显且难以使用。在更高级别的用户界面代码中，通常需要在行中间插入文本（例如，当用户正在输入时）或删除跨越多行的文本范围。对于文本类的面向行 API，调用者被迫拆分和连接行以实现用户界面操作。这段代码非常复杂，并且在用户界面的实现中被重复和分散。

当文本类提供面向字符的接口时，它们更容易使用，例如一个 insert 方法，该方法在文本中的任意位置插入任意字符串的文本（可能包括换行符），以及一个 delete 方法，该方法删除文本中两个任意位置之间的文本。
在内部，文本仍然以行的形式表示。面向字符的接口封装了文本类内部的行拆分和连接的复杂性，这使得文本类更深入，并简化了使用该类的更高级别代码。通过这种方法，文本 API 与面向行的存储机制截然不同；这种差异代表了该类提供的有价值的功能。

7.5 通过变量

跨层 API 重复的另一种形式是 通过变量，它是一个通过长方法链传递的变量。图 7.2(a)

显示了来自数据中心服务的示例。命令行参数描述了用于安全通信的证书。此信息仅由低级方法 m3 需要，该方法调用库方法以打开套接字，但它通过 main 和 m3 之间的路径上的所有方法传递。

cert 变量出现在每个中间方法的签名中。

通过变量增加了复杂性，因为它们迫使所有中间方法都知道它们的存在，即使这些方法不需要这些变量。此外，如果出现一个新变量（例如，一个系统最初构建时不支持证书，但您稍后决定添加该支持），您可能需要修改大量接口和方法，以通过所有相关路径传递该变量。

消除通过变量可能具有挑战性。一种方法是查看在最顶层和最底层方法之间是否已经存在共享对象。在 图 7.2 的数据中心服务示例中，可能存在一个包含有关网络通信的其他信息的对象，main 和 m3 都可以访问该对象。如果是这样，main 可以将证书信息存储在该对象中，因此无需通过到 m3 的路径上的所有中间方法（参见 图 7.2(b)）。但是，如果存在这样的对象，那么它本身可能是一个通过变量（m3 如何访问它？）。

另一种方法是将信息存储在全局变量中，如 图 7.2(c)这避免了将信息从一个方法传递到另一个方法的需要，但是全局变量几乎总是会产生其他问题。例如，全局变量使得在同一进程中创建同一系统的两个独立实例成为不可能，因为对全局变量的访问会冲突。在生产环境中，您可能不太需要多个实例，但它们在测试中通常很有用。

我最常用的解决方案是引入一个上下文对象，如图所示 7.2(d)。上下文存储应用程序的所有全局状态（任何原本会是传递变量或全局变量的东西）。大多数应用程序在其全局状态中都有多个变量，代表诸如配置选项、共享子系统和性能计数器之类的内容。每个系统实例都有一个上下文对象。上下文允许多个系统实例在单个进程中共存，每个实例都有自己的上下文。

不幸的是，上下文可能在很多地方都需要，因此它可能会变成一个传递变量。为了减少必须知道它的方法的数量，可以在大多数

系统的主要对象中保存对上下文的引用。在图 7.2(d)的例子中，包含m3的类在其对象中将对上下文的引用存储为实例变量。当创建一个新对象时，创建方法从其对象中检索上下文引用，并将其传递给新对象的构造函数。通过这种方法，上下文在任何地方都可用，但它只作为构造函数中的显式参数出现。

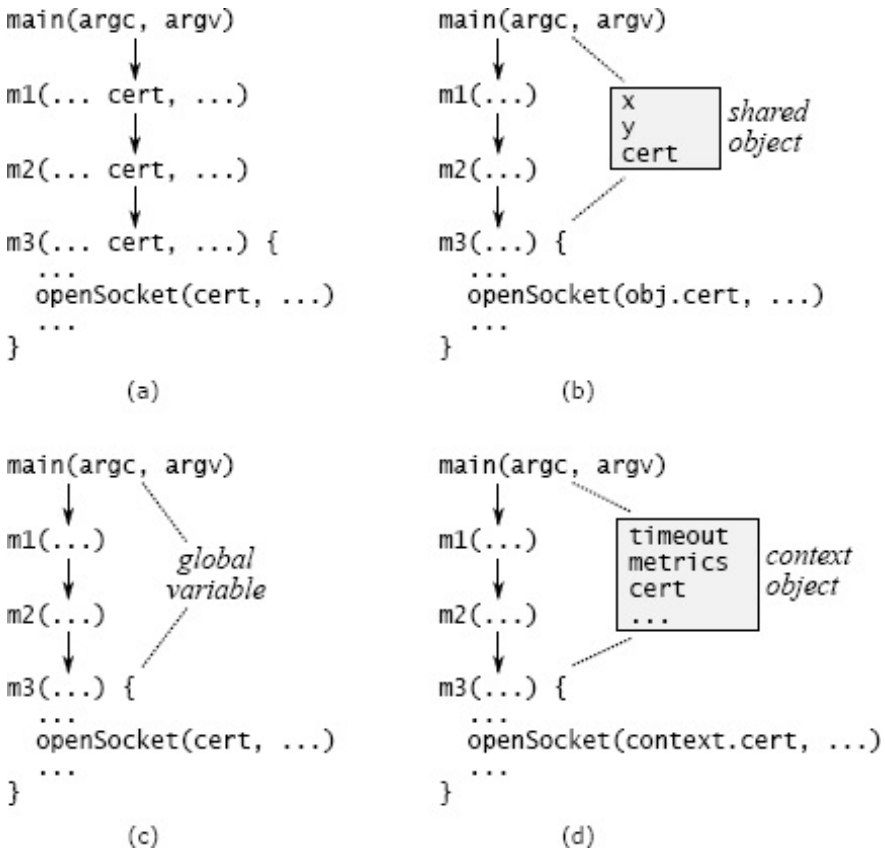

图 7.2：处理传递变量的可能技术。在 (a) 中，cert 通过方法 m1 和 m2 传递，即使它们不使用它。在 (b) 中，main 和 m3 共享对一个对象的访问权限，因此该变量可以存储在那里，而不是通过 m1 和 m2 传递。在 (c) 中，cert 作为全局变量存储。在 (d) 中，cert 与其他系统范围的信息（如超时值和性能计数器）一起存储在上下文对象中；对上下文的引用存储在所有需要访问它的对象的方法中。

上下文对象统一了所有系统全局信息的处理，并消除了对传递变量的需求。如果需要添加一个新的变量，可以将其添加到上下文对象中；除了上下文的构造函数和析构函数之外，没有现有的代码会受到影响。上下文使得识别和管理系统的全局状态变得容易，因为它都存储在一个

地方。上下文也便于测试：测试代码可以通过修改上下文中的字段来更改应用程序的全局配置。如果系统使用传递变量，则实现这些更改将更加困难。

上下文远非理想的解决方案。存储在上下文中的变量具有全局变量的大部分缺点；例如，可能不清楚为什么存在某个特定的变量，或者它在哪里被使用。如果没有约束，上下文可能会变成一个巨大的数据包，从而在整个系统中创建不明显的依赖关系。上下文也可能产生线程安全问题；避免问题的最好方法是使上下文中的变量是不可变的。不幸的是，我没有找到比上下文更好的解决方案。

7.6 结论

添加到系统的每个设计基础设施，例如接口、参数、函数、类或定义，都会增加复杂性，因为开发人员必须了解这个元素。为了使一个元素能够提供相对于复杂性的净收益，它必须消除在没有该设计元素的情况下会存在的某些复杂性。否则，最好在没有该特定元素的情况下实现系统。例如，一个类可以通过封装功能来降低复杂性，这样类的用户就不需要知道它。

“不同的层，不同的抽象”规则只是这个想法的应用：如果不同的层具有相同的抽象，例如直通方法或装饰器，那么它们很可能没有提供足够的收益来补偿它们所代表的额外基础设施。类似地，直通参数要求多个方法中的每一个都意识到它们的存在（这增加了复杂性），而没有贡献额外的功能。

第八章

将复杂性向下转移

本章介绍了另一种关于如何创建更深层次类别的思考方式。假设您正在开发一个新的模块，并且发现了一块不可避免的复杂性。哪种方式更好：您应该让模块的用户处理这种复杂性，还是应该在模块内部处理这种复杂性？如果这种复杂性与模块提供的功能相关，那么通常第二个答案是正确的。大多数模块的用户比开发者多，因此开发者受苦比用户受苦更好。作为模块开发者，您应该努力让模块的用户尽可能轻松，即使这意味着您需要付出额外的工作。表达这个想法的另一种方式是，一个模块拥有简单的接口比拥有简单的实现更重要。

作为一名开发者，很容易以相反的方式行事：解决简单的问题，并将难题推给其他人。如果出现您不确定如何处理的情况，最简单的方法是抛出一个异常，让调用者来处理它。如果您不确定要实施什么策略，您可以定义一些配置参数来控制该策略，并将其留给系统管理员来确定最佳值。

像这样的方法在短期内会让您的生活更轻松，但它们会放大复杂性，从而使许多人必须处理一个问题，而不是仅仅一个人。例如，如果一个类抛出一个异常，那么该类的每个调用者都必须处理它。如果一个类导出配置参数，那么每个安装中的每个系统管理员都必须学习如何设置它们。

8.1 示例：编辑器文本类

考虑一下为GUI文本编辑器管理文件文本的类，该类在第6章和第7章中讨论过。该类提供了将文件从磁盘读取到内存中的方法，查询和修改文件的内存副本，并写入

修改后的版本返回到磁盘。当学生们必须实现这个类时，他们中的许多人选择了一个面向行的接口，其中包含读取、插入和删除整行文本的方法。这导致了该类的一个简单实现，但它为更高级别的软件带来了复杂性。在用户界面级别，操作很少涉及整行。例如，击键会导致单个字符插入到现有行中；复制或删除选择可以修改几个不同行的部分。使用面向行的文本接口，更高级别的软件必须拆分和连接行才能实现用户界面。

一个面向字符的接口，例如第6.3节中描述的接口，会将复杂性向下转移。用户界面软件现在可以插入和删除任意范围的文本，而无需拆分和合并行，因此它变得更简单。文本类的实现可能会变得更复杂：

如果它在内部将文本表示为行的集合，那么它将需要拆分和合并行来实现面向字符的操作。这种方法更好，因为它将拆分和合并的复杂性封装在文本类中，从而降低了系统的整体复杂性。

8.2 示例：配置参数

配置参数是将复杂性向上移动而不是向下的一个例子。一个类可以不从内部确定特定的行为，而是可以导出一些控制其行为的参数，例如缓存的大小或放弃请求前重试的次数。然后，该类的用户必须为这些参数指定适当的值。配置参数在当今的系统中变得非常流行；有些系统有数百个配置参数。

倡导者认为配置参数是好的，因为它们允许用户根据其特定需求和工作负载来调整系统。在某些情况下，底层基础设施代码很难知道要应用的最佳策略，而用户更熟悉他们的领域。例如，用户可能知道某些请求比其他请求更具时间紧迫性，因此用户可以为这些请求指定更高的优先级。在这种情况下，配置参数可以在更广泛的领域内带来更好的性能。

然而，配置参数也提供了一个简单的借口来避免处理重要问题并将它们传递给其他人。在许多情况下，

用户或管理员很难或不可能确定参数的正确值。在其他情况下，通过在系统实现中进行一些额外的工作，可以自动确定正确的值。

考虑一个必须处理丢包的网络协议。如果它发送一个请求但在一定时间内没有收到响应，它会重新发送该请求。

确定重试间隔的一种方法是引入一个配置参数。

但是，传输协议可以通过测量成功请求的响应时间，然后将其倍数用于重试间隔来自行计算一个合理的值。这种方法将复杂性向下转移，并使用户不必弄清楚正确的重试间隔。它还具有动态计算重试间隔的额外优势，因此如果操作条件发生变化，它将自动调整。相比之下，配置参数很容易过时。

因此，您应该尽可能避免配置参数。在导出配置参数之前，问问自己：“用户（或更高级别的模块）是否能够确定比我们在此处确定的更好的值？”当您创建配置参数时，看看是否可以自动计算合理的默认值，这样用户只需要在特殊情况下提供值。理想情况下，每个模块都应该完全解决一个问题；配置参数会导致不完整的解决方案，从而增加系统复杂性。

8.3 过犹不及

在降低复杂性时要谨慎，这种想法很容易被过度使用。一个极端的方法是将整个应用程序的所有功能都拉到一个类中，这显然没有意义。

如果 (a) 被拉低的复杂性与该类现有的功能密切相关，(b) 拉低复杂性将导致应用程序中其他地方的许多简化，以及 (c) 拉低复杂性简化了该类的接口，那么拉低复杂性最有意义。请记住，目标是最大限度地降低整体系统复杂性。

第 6 章描述了一些学生如何在文本类中定义反映用户界面的方法，例如实现退格键功能的方法。这似乎很好，因为它降低了复杂性。然而，向文本类添加用户界面知识并不能很好地简化更高级别的代码，而且用户界面知识

与文本类的核心功能无关。在这种情况下，降低复杂性只会导致信息泄露。

8.4 结论

在开发模块时，要寻找机会让自己承担一点额外的痛苦，以减少用户的痛苦。

第 9 章

聚在一起更好还是分开更好？

软件设计中最基本的问题之一是：给定两个功能，它们应该在同一个地方一起实现，还是应该将它们的实现分开？这个问题适用于系统中的所有级别，例如函数、方法、类和服务。例如，缓冲应该包含在提供面向流的文件 I/O 的类中，还是应该包含在一个单独的类中？HTTP 请求的解析应该完全在一个方法中实现，还是应该将其划分为多个方法（甚至多个类）？本章讨论了在做出这些决定时需要考虑的因素。其中一些因素已经在前面的章节中讨论过，但为了完整起见，这里将重新讨论。

在决定是组合还是分离时，目标是降低整个系统的复杂性并提高其模块化。似乎实现此目标的最佳方法是将系统划分为大量的小组件：组件越小，每个单独的组件可能就越简单。然而，细分的行为会产生细分之前不存在的额外复杂性：

- 有些复杂性仅仅来自于组件的数量：组件越多，就越难跟踪所有组件，也越难在大型集合中找到所需的组件。细分通常会导致更多的接口，而每个新接口都会增加复杂性。
- 细分可能导致需要额外的代码来管理组件。例如，在细分之前使用单个对象的代码现在可能需要管理多个对象。
- 细分会产生分离：细分的组件将比细分前更远。例如，在细分之前在单个类中的方法可能在细分之后位于不同的类中，并且可能位于不同的文件中。分离使得开发人员更难以同时看到组件，甚至难以意识到

它们的存在。如果组件是真正独立的，那么分离是好的：它允许开发人员一次专注于一个组件，而不会被其他组件分散注意力。另一方面，如果组件之间存在依赖关系，那么分离是不好的：

开发人员最终会在组件之间来回切换。

更糟糕的是，他们可能没有意识到这些依赖关系，这可能导致错误。

- 细分可能导致重复：细分之前存在于单个实例中的代码可能需要存在于每个细分组件中。

如果代码片段之间关系密切，那么将它们放在一起是最有益的。如果这些代码片段不相关，那么最好将它们分开。以下是一些表明两段代码相关的迹象：

- 它们共享信息；例如，两段代码可能都依赖于特定类型文档的语法。
- 它们一起使用：任何使用其中一段代码的人也可能使用另一段代码。只有当这种关系是双向的时，这种关系才具有说服力。作为一个反例，磁盘块缓存几乎总是涉及哈希表，但是哈希表可以在许多不涉及块缓存的情况下使用；因此，这些模块应该是分开的。

- 它们在概念上是重叠的，因为存在一个简单的高层类别，它包含了这两段代码。例如，搜索子字符串和大小写转换都属于字符串操作的范畴；流控制和可靠交付都属于网络通信的范畴。

- 如果不看另一段代码，就很难理解其中一段代码。

本章的其余部分使用更具体的规则和示例来展示何时将代码片段放在一起是有意义的，以及何时将代码片段分开是有意义的。

9.1 如果信息是共享的，则放在一起

第 5.4 节在实现HTTP 服务器的项目的上下文中介绍了这一原则。在其最初的实现中，该项目在不同的类中使用了两种不同的方法来读取和解析 HTTP 请求。第一种方法从网络套接字读取传入请求的文本，并将其放入字符串中

对象。第二种方法解析字符串以提取请求的各种组件。通过这种分解，这两种方法最终都对 HTTP 请求的格式有了相当多的了解：第一种方法只是试图读取请求，而不是解析它，但如果不做大部分解析工作，它就无法识别请求的结尾（例如，它必须解析标头行才能识别包含总体请求长度的标头）。

由于这种共享信息，最好在同一位置读取和解析请求；当两个类合并为一个类时，代码变得更短更简单。

9.2 如果它将简化接口，则放在一起

当两个或多个模块组合成一个模块时，可以为新模块定义一个比原始接口更简单或更易于使用的接口。当原始模块各自实现问题的一部分解决方案时，通常会发生这种情况。在前面章节的 HTTP 服务器示例中，原始方法需要一个接口来从第一个方法返回 HTTP 请求字符串，并将其传递给第二个方法。当这些方法组合在一起时，这些接口就被消除了。

此外，当两个或多个类的功能组合在一起时，可以自动执行某些功能，以便大多数用户不必知道它们。Java I/O 库说明了这种机会。如果

如果将FileInputStream和BufferedInputStream类合并，并默认提供缓冲，那么绝大多数用户甚至不需要意识到缓冲的存在。一个合并的FileInputStream类可能会提供禁用或替换默认缓冲机制的方法，但大多数用户不需要了解它们。

9.3 整合以消除重复

如果您发现相同的代码模式一遍又一遍地重复出现，请看看是否可以重新组织代码以消除重复。一种方法是将重复的代码分解成一个单独的方法，并用对该方法的调用来替换重复的代码片段。如果重复的代码片段很长，并且替换方法的签名很简单，那么这种方法最有效。如果代码片段只有一两行长，那么用方法调用替换它可能没有太大的好处。如果代码片段以复杂的方式与其环境交互（例如通过访问大量的局部变量），那么替换方法可能

需要一个复杂的签名（例如许多按引用传递的参数），这将降低它的价值。

消除重复的另一种方法是重构代码，使有问题的代码片段只需要在一个地方执行。假设您正在编写一个需要在几个不同点返回错误的方法，并且在返回之前需要在每个点执行相同的清理操作（有关示例，请参见图 9.1）。如果编程语言支持

goto，您可以将清理代码移动到方法的末尾，然后在需要错误返回的每个点goto该代码片段，如图所示
9.2。Goto语句通常被认为是一个坏主意，如果滥用它们会导致难以理解的代码，但在这种情况下它们很有用，因为它们用于从嵌套代码中转义。

9.4 分离通用代码和专用代码

如果一个模块包含一种可以用于几种不同目的的机制，那么它应该只提供那一种通用机制。它不应该包含为特定用途专门化该机制的代码，也不应该包含其他通用机制。与通用机制相关的专用代码通常应该放在不同的模块中（通常是与特定用途相关的模块）。第 6 章中的 GUI 编辑器讨论说明了这一原则：最好的设计是文本类提供通用文本操作，而特定于用户界面的操作（例如删除选择）在用户界面模块中实现。这种方法消除了信息泄漏和额外的接口，这些接口存在于早期设计中，其中专门的用户界面操作是在文本类中实现的。

危险信号：重复

如果同一段代码（或几乎相同的代码）一遍又一遍地出现，这是一个危险信号，表明您还没有找到正确的抽象。

```
switch (common->opcode) {
    case DATA: {
        DataHeader* header = received->getStart<DataHeader>();
        if (header == NULL) {
            LOG(WARNING, "%s packet from %s too short (%u bytes)",
                    opcodeSymbol(common->opcode),
                    received->sender->toString(),
                    received->len);
            return;
        }
        ...
    case GRANT: {
        GrantHeader* header = received->getStart<GrantHeader>();
        if (header == NULL) {
            LOG(WARNING, "%s packet from %s too short (%u bytes)",
                    opcodeSymbol(common->opcode),
                    received->sender->toString(),
                    received->len);
            return;
        }
        ...
    case RESEND: {
        ResendHeader* header = received->getStart<ResendHeader>();
        if (header == NULL) {
            LOG(WARNING, "%s packet from %s too short (%u bytes)",
                    opcodeSymbol(common->opcode),
                    received->sender->toString(),
                    received->len);
            return;
        }
        ...
}
```

图 9.1：此代码处理不同类型的传入网络数据包；对于每种类型，如果数据包对于该类型来说太短，则会记录一条消息。在此版本的代码中，LOG语句对于几种不同的数据包类型是重复的。

```
switch (common->opcode) {
    case DATA: {
        DataHeader* header = received->getStart<DataHeader>();
        if (header == NULL)
            goto packetTooShort;
        ...
    case GRANT: {
        GrantHeader* header = received->getStart<GrantHeader>();
        if (header == NULL)
            goto packetTooShort;
        ...
    case RESEND: {
        ResendHeader* header = received->getStart<ResendHeader>();
        if (header == NULL)
            goto packetTooShort;
        ...
}
...
packetTooShort:
LOG(WARNING, "%s packet from %s too short (%u bytes)",
        opcodeSymbol(common->opcode),
        received->sender->toString(),
        received->len);
return;
```

图 9.2：对图 9.1中的代码进行重组，以便LOG语句只有一个副本。

一般来说，系统的底层往往更通用，而上层则更专用。例如，应用程序的最顶层完全由特定于该应用程序的功能组成。将专用代码与通用代码分离的方法是将专用代码向上拉到更高的层中，使较低的层保持通用。当您遇到一个类，其中包含同一抽象的通用和专用功能时，请查看该类是否可以分为两个类，一个包含通用功能，另一个在其之上分层以提供专用功能。

9.5 示例：插入光标和选择

接下来的章节将通过三个例子来说明上面讨论的原则。在其中两个例子中，最好的方法是分离相关的代码片段；在第三个例子中，最好将它们连接在一起。

第一个例子包括来自第 6 章的 GUI 编辑器项目中的插入光标和选择。编辑器显示一条闪烁的垂直线，指示用户键入的文本将出现在文档中的位置。它还显示一个称为选择的突出显示的字符范围，该范围用于复制或删除文本。插入光标始终可见，但有时可能没有选择任何文本。如果存在选择，则插入光标始终位于其一端。

选择和插入光标在某些方面是相关的。例如，光标始终位于选择的一端，并且光标和选择往往一起操作：单击并拖动鼠标会同时设置它们，并且文本插入首先删除所选文本（如果有），然后在光标位置插入新文本。因此，似乎可以使用单个对象来管理选择和光标，并且一个项目团队采用了这种方法。该对象在文件中存储了两个位置，以及指示哪个端点是光标以及是否存在选择的布尔值。

然而，组合对象很笨拙。它没有为更高级别的代码提供任何好处，因为更高级别的代码仍然需要意识到选择和光标是不同的实体，并且它分别操作它们（在文本插入期间，它首先在组合对象上调用一个方法来删除

所选文本；然后它调用另一个方法来检索光标位置以便插入新文本）。组合对象实际上比单独的对象更难实现。它避免将光标位置存储为单独的实体，而是必须存储一个布尔值，指示选择的哪个端点是光标。为了检索光标位置，组合对象必须首先测试布尔值，然后选择选择的适当端点。

危险信号：特殊-通用混合

当一个通用机制也包含专门用于该机制特定用途的代码时，就会出现这个危险信号。这使得该机制更加复杂，并在该机制和特定用例之间产生信息泄漏：未来对用例的修改可能需要对底层机制进行更改。

在这种情况下，选择和光标之间的关联不够紧密，无法将它们组合在一起。当代码被修改为分离选择和光标时，用法和实现都变得更简单。与必须从中提取选择和光标信息的组合对象相比，单独的对象提供了更简单的接口。光标的实现也变得更简单，因为光标位置是直接表示的，而不是通过选择和一个布尔值间接表示的。事实上，在修改后的版本中，选择或光标都没有使用特殊的类。相反，一个新的

Position 类被引入来表示文件中的位置（行号和行内字符）。选择用两个 Positions 表示，光标用一个表示。Positions 也在项目中找到了其他用途。这个例子也展示了更低级别但更通用的接口的好处，这在 第 6 章中讨论过。

9.6 示例：用于日志记录的单独类

第二个例子涉及一个学生项目中的错误日志记录。一个类包含几个像下面这样的代码序列：

```
try {
```

```
    rpcConn = connectionPool.getConnection(dest);
} catch (IOException e) {
    NetworkErrorLogger.logRpcOpenError(req, dest, e);
    return null;
}
```

Rather than logging the error at the point where it was detected, a separate 调用了特殊错误日志记录类中的方法。错误日志记录类在同一个源文件的末尾定义：

```
private static class NetworkErrorLogger {
   /**
    * 输出与尝试打开连接以发送 RPC 时发生的错误相关的信息。

    *
    * @param req（请求）
    * 本应通过连接发送的 RPC 请求

    * @param dest（目标）
    * RPC的目标
    * @param e
    * 捕获的错误
    */
   public static void logRpcOpenError(RpcRequest req, AddrPortTuple
                目标地址, 异常 e) {
      logger.log(Level.WARNING, "无法发送消息：" + req + "。" +"无法找到或打开到 " +
                dest + " 的连接：" +e);

    }
...
}
```

T哈恩 恩特诺克多尔奥隆格多克尔 阿默特 恩恩德恩恩恩 尔阿姆特哈 奥德恩多克哈 恩

logRpcSendError and logRpcReceiveError，它们各自记录了不同类型的错误。

这种分离增加了复杂性，却没有带来任何好处。这些日志记录方法都很浅显：大多数都只包含一行代码，但却需要大量的文档。每个方法只在一个地方被调用。

日志记录方法高度依赖于它们的调用：阅读调用的人很可能会翻到日志记录方法，以确保记录了正确的信息；同样，阅读日志记录方法的人可能会翻到调用位置，以了解该方法的目的。

在这个例子中，最好消除日志记录方法，并将日志语句放在检测到错误的位置。这将使代码更易于阅读，并消除日志记录方法所需的接口。

9.7 示例：编辑器撤销机制

在第 6.2节的 GUI 编辑器项目中，其中一个要求是支持多级撤销/重做，不仅针对文本本身的更改，还包括选择、插入光标和视图的更改。例如，如果用户选择了一些文本，删除了它，滚动到文件中的不同位置，然后调用撤销，编辑器必须将其状态恢复到删除之前的状态。这包括恢复已删除的文本，再次选择它，并使所选文本在窗口中可见。

一些学生项目将整个撤销机制作为文本类的一部分来实现。文本类维护一个所有可撤销更改的列表。每当文本发生更改时，它会自动将条目添加到此列表中。对于选择、插入光标和视图的更改，用户界面代码调用文本类中的其他方法，然后将这些更改的条目添加到撤销列表中。当用户请求撤销或重做时，用户界面代码调用文本类中的一个方法，然后该方法处理撤销列表中的条目。对于与文本相关的条目，它更新文本类的内部结构；对于与选择等其他事物相关的条目，文本类回调用户界面代码以执行撤销或重做。

这种方法导致文本类中出现了一组笨拙的特性。撤销/重做的核心包括一个用于管理已执行操作列表并在撤销和重做操作期间逐步执行这些操作的通用机制。核心位于文本类中，以及实现针对文本和选择等特定事物的撤销和重做的专用处理程序。选择和光标的专用撤销处理程序与文本类中的任何其他内容无关；它们导致文本类和用户界面之间的信息泄漏，以及额外的

每个模块中的方法来来回传递撤销信息。如果将来向系统中添加一种新的可撤销实体，则需要更改文本类，包括特定于该实体的新方法。此外，通用撤销核心与类中的通用文本设施几乎没有关系。

这些问题可以通过提取撤销/重做机制的通用核心并将其放置在一个单独的类中来解决：

```
public class History {
    public interface Action {
        public void redo();
        public void undo();
    }

    History() {...}

    void addAction(Action action) {...}
    void addFence() {...}

    void undo() {...}
    void redo() {...}
}
```

在这个设计中，History 类管理着一个实现了
接口History.Actio的对象集合。每个History.Actio描述了一个单独的操作，例如文本插入或光标位置的改变，并且它提供了
可以撤销或重做操作的方法。History类对存储在动作中的信息或它们如何实现它们的undo 和
redo 方法一无所知。History 维护着一个历史列表，描述了所有
在应用程序的生命周期内执行的动作，并且它提供了undo 和redo
方法，这些方法在列表中向后和向前移动，以响应用户-
请求的撤销和重做，调用undo 和redo 方法在
History.Actions中。

History.Actions 是专用对象：每个对象都理解
特定类型的可撤销操作。它们在History
类之外实现，在理解特定类型的可撤销操作的模块中。文本
类可能实现UndoableInsert 和UndoableDelete 对象来描述

文本插入和删除。每当它插入文本时，文本类都会创建一个新的 `UndoableInsert` 对象，描述插入操作，并调用`History.addAction` 将其添加到历史记录列表中。编辑器的用户界面代码可能会创建 `UndoableSelection` 和`UndoableCursor` 对象，这些对象描述了对选择和插入光标的更改。

`History` 类还允许将操作分组，以便例如，来自用户的单个撤销请求可以恢复已删除的文本，重新选择已删除的文本，并重新定位插入光标。有很多方法可以分组操作；`History` 类使用*fences*，它们是放置在历史列表中的标记，用于分隔相关操作的组。每次调用`History.redo` 都会向后遍历历史记录列表，撤消操作，直到到达下一个fence。围栏的放置由更高级别的代码通过调用来确定`History.addFence`.

这种方法将撤销功能划分为三个类别，每个类别都在不同的地方实现：

- 一种用于管理和分组操作以及调用撤销/重做操作的通用机制（由 History 类实现）。
- 特定操作的细节（由各种类实现，每个类理解少量操作类型）。
- 用于分组操作的策略（由高级用户界面代码实现，以提供正确的整体应用程序行为）。

这些类别中的每一个都可以在不了解其他类别的情况下实现。History 类不知道正在撤消的操作类型；它可以用于各种应用程序。每个操作类只理解一种操作，并且 History 类和操作类都不需要知道分组操作的策略。

关键的设计决策是将撤销机制的通用部分与专用部分分开，并将通用部分放在一个类中。一旦完成，其余的设计自然而然地就出来了。

注意：将通用代码与专用代码分离的建议是指与特定机制相关的代码。例如，专用撤销代码（例如撤销文本插入的代码）应与通用撤销代码（例如管理历史记录列表的代码）分离。

但是，将一个机制的专用代码与另一个机制的通用代码组合在一起通常是有意义的

机制与另一个机制的通用代码结合起来通常是有意义的。文本类就是一个例子：它实现了一种用于管理文本的通用机制，但它包括与撤销相关的专用代码。撤销代码是专用的，因为它只处理文本修改的撤销操作。将此代码与 History 类中的通用撤销基础设施结合起来没有意义，但将其放在文本类中是有意义的，因为它与其他文本功能密切相关。

9.8 分割和连接方法

何时细分的问题不仅适用于类，也适用于方法：
是否有时候将现有方法划分为多个较小的方法更好？或者，是否应该将两个较小的方法合并为一个较大的方法？长方法往往比短方法更难理解，因此许多人认为，仅凭长度就可以充分证明分解方法的合理性。课堂上的学生经常被给予严格的标准，例如“拆分任何超过 20 行的方法！”

然而，仅凭长度很少是拆分方法的好理由。一般来说，开发人员往往过度拆分方法。拆分方法会引入额外的接口，从而增加复杂性。它还会分离原始方法的各个部分，如果这些部分实际上是相关的，这会使代码更难阅读。除非它使整个系统更简单，否则你不应该拆分方法；我将在下面讨论这种情况是如何发生的。

长方法并不总是坏事。例如，假设一个方法包含五个 20 行的代码块，这些代码块按顺序执行。如果这些块相对独立，那么可以一次读取和理解一个块；将每个块移动到单独的方法中没有太多好处。如果这些块具有复杂的交互，那么将它们放在一起就更加重要，这样读者可以一次看到所有代码；如果每个块都在一个单独的方法中，读者将不得不在这些分散的方法之间来回翻阅，以便理解它们是如何协同工作的。包含数百行代码的方法，如果它们具有简单的签名并且易于阅读，那么也是可以的。这些方法是深入的（功能多，接口简单），这很好。

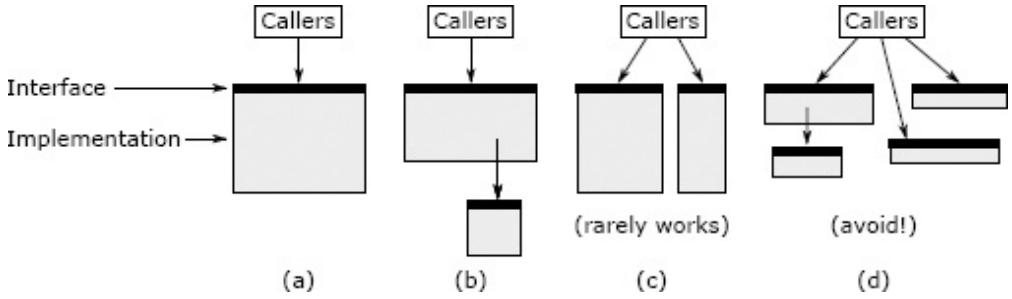

图 9.3：一种方法 (a) 可以通过提取子任务 (b) 或将其功能划分为两个单独的方法 (c) 来拆分。如果拆分导致浅显的方法，如 (d) 所示，则不应拆分方法。

在设计方法时，最重要的目标是提供清晰而简单的抽象。每个方法都应该做一件事，并且完整地完成它。
该方法应该具有清晰而简单的接口，以便用户不需要在脑海中记住太多信息才能正确使用它。该方法应该是深入的：它的接口应该比它的实现简单得多。如果一个方法具有所有这些属性，那么它是否长并不重要。

只有当拆分方法能够产生更清晰的抽象时，才有意义。有两种方法可以做到这一点，如图 9.3所示。最好的方法是将子任务分解为单独的方法，如图 所示
9 .3(b)。细分会产生一个包含子任务的子方法和一个
包含原始方法其余部分的父方法；父方法调用子方法。新父方法的接口与原始方法相同。如果存在一个可以从原始方法的其余部分干净地分离出来的子任务，那么这种形式的细分是有意义的，这意味着 (a) 阅读子方法的人不需要了解任何关于父方法的信息，并且 (b) 阅读父方法的人不需要理解子方法的实现。通常，这意味着子方法是相对通用的：除了父方法之外，它还可以被其他方法使用。如果你进行了这种形式的拆分，然后发现自己在父方法和子方法之间来回翻阅以了解它们是如何协同工作的，这是一个危险信号（“连接方法”），表明拆分可能是一个坏主意。

分解方法的第二种方式是将其拆分为两个独立的方法，每个方法对于原始方法的调用者都是可见的，如图 9.3(c)所示。如果原始方法有一个过于复杂的接口，因为它试图做多个不密切相关的事情，那么这才有意义。如果是这种情况，它可能是

可以将方法的功能划分为两个或更多个较小的方法，每个方法只具有原始方法的部分功能。如果进行这样的拆分，则每个生成方法的接口应比原始方法的接口更简单。理想情况下，大多数调用者只需要调用两个新方法中的一个；如果调用者必须调用这两个新方法，那么会增加复杂性，这使得拆分不太可能是一个好主意。新方法将更加专注于它们所做的事情。如果新方法比原始方法更通用（即，您可以想象在其他情况下单独使用它们），这是一个好兆头。

如图 9.3(c) 所示的拆分通常没有意义，因为它们会导致调用者必须处理多个方法而不是一个方法。当您以这种方式拆分时，您可能会最终得到几个浅层方法，如图 9.3(d) 所示。如果调用者必须调用每个单独的方法，并在它们之间来回传递状态，那么拆分不是一个好主意。如果您正在考虑像图 9.3(c) 这样的拆分，您应该根据它是否简化了调用者的事情来判断它。

在某些情况下，通过将方法连接在一起来简化系统。例如，连接方法可以用一个更深的方法替换两个浅层方法；它可以消除代码的重复；它可以消除原始方法之间或中间数据结构的依赖关系；它可以实现更好的封装，从而使先前存在于多个位置的知识现在被隔离在单个位置；或者它可以产生更简单的接口，如第 9.2 节所述。

危险信号：连接方法

应该能够独立理解每个方法。如果您不能理解一个方法的实现，而不理解另一个方法的实现，这是一个危险信号。这个危险信号也可能出现在其他上下文中：如果两段代码在物理上是分开的，但每一段代码只能通过查看另一段代码来理解，这是一个危险信号。

9.9 结论

拆分或连接模块的决定应基于复杂性。选择

拆分或连接模块的决定应基于复杂性。选择能够实现最佳信息隐藏、最少依赖关系和最深接口的结构。

第10章

定义不存在的错误

异常处理是软件系统中复杂性的最糟糕来源之一。处理特殊情况的代码本质上比处理正常情况的代码更难编写，并且开发人员经常在没有考虑如何处理异常的情况下定义异常。本章讨论了为什么异常会不成比例地增加复杂性，然后展示了如何简化异常处理。本章的主要总体教训是减少必须处理异常的地方的数量；在许多情况下，可以修改操作的语义，以便正常行为处理所有情况，并且没有异常情况需要报告（因此本章的标题）。

10.1 为什么异常会增加复杂性

我使用术语“异常”来指代任何改变程序中正常控制流程的非常见情况。许多编程语言都包含一种正式的异常机制，允许底层代码抛出异常，并由封闭代码捕获。然而，即使不使用正式的异常报告机制，也可能发生异常，例如当方法返回一个特殊值，表明它没有完成其正常行为时。所有这些形式的异常都会增加复杂性。

一段特定的代码可能会以几种不同的方式遇到异常：

- 调用者可能提供错误的参数或配置信息。
- 被调用的方法可能无法完成请求的操作。例如，I/O 操作可能失败，或者所需的资源可能不可用。

- 在分布式系统中，网络数据包可能会丢失或延迟，服务器可能无法及时响应，或者对等方可能会以意想不到的方式进行通信。

- 代码可能会检测到错误、内部不一致或无法处理的情况。

大型系统必须处理许多异常情况，尤其是在它们是分布式的或需要具有容错能力的情况下。异常处理可能占系统中所有代码的很大一部分。

异常处理代码本质上比正常情况的代码更难编写。异常会中断代码的正常流程；它通常意味着某些事情没有按预期工作。当发生异常时，程序员可以通过两种方式处理它，每种方式都可能很复杂。第一种方法是继续前进，尽管发生了异常，也要完成正在进行的工作。例如，如果网络数据包丢失，可以重新发送；如果数据损坏，也许可以从冗余副本中恢复。第二种方法是中止正在进行的操作，并将异常向上报告。

然而，中止可能会很复杂，因为异常可能发生在系统状态不一致的点（数据结构可能已被部分初始化）；异常处理代码必须恢复一致性，例如通过撤销在异常发生之前所做的任何更改。

此外，异常处理代码会创造更多异常的机会。考虑一下重新发送丢失的网络数据包的情况。也许数据包实际上并没有丢失，只是延迟了。在这种情况下，重新发送数据包将导致重复的数据包到达对等方；这引入了一个新的异常情况，对等方必须处理。或者，考虑一下从冗余副本中恢复丢失数据的情况：如果冗余副本也丢失了怎么办？恢复期间发生的次级异常通常比主要异常更微妙和复杂。如果通过中止正在进行的操作来处理异常，则必须将其作为另一个异常报告给调用者。为了防止无休止的异常级联，开发人员最终必须找到一种处理异常的方法，而不会引入更多异常。

对异常的语言支持往往是冗长而笨拙的，这使得异常处理代码难以阅读。例如，考虑以下代码，它使用 Java 对对象序列化和反序列化的支持从文件中读取一系列推文：

```
try (
    FileInputStream fileStream =
            new FileInputStream(fileName);
    BufferedInputStream bufferedStream =
```

```
            new BufferedInputStream(fileStream);
    ObjectInputStream objectStream =
            new ObjectInputStream(bufferedStream);
) {
    for (int i = 0; i < tweetsPerFile; i++) {
        tweets.add((Tweet) objectStream.readObject());
    }
}
catch (FileNotFoundException e) {
    ...
}
catch (ClassNotFoundException e) {
    ...
}
catch (EOFException e) {
    // 不是问题：并非所有推文文件都包含完整的
    // 推文集。
}
catch (IOException e) {
    ...
}
catch (ClassCastException e) {
    ...
}
```


try-catch 样板代码比正常情况下的代码行数还要多，甚至没有考虑到实际处理异常的代码。很难将异常处理代码与正常情况下的代码联系起来：例如，不清楚每个异常是在哪里生成的。另一种方法是将代码分解成许多不同的try块；在极端情况下，对于每一行可能生成异常的代码，都可能有一个 try 。这将清楚地表明异常发生的位置，但是

try 块本身会破坏代码的流程，使其更难阅读；
此外，一些异常处理代码最终可能会在多个

try 块中重复出现。

很难确保异常处理代码真正有效。有些异常，例如 I/O 错误，很难在测试环境中生成，因此很难测试处理这些异常的代码。异常在运行的系统中并不经常发生，因此异常处理代码很少执行。Bug 可能会长时间未被发现，当最终需要异常处理代码时，很有可能它无法工作（我最喜欢的一句话是：

“未执行的代码无法工作”）。最近的一项研究发现，分布式数据密集型系统中超过 90% 的灾难性故障是由不正确的错误处理1引起的。当异常处理代码失败时，很难调试问题，因为它发生的频率太低了。

10.2 异常过多

程序员通过定义不必要的异常来加剧与异常处理相关的问题。大多数程序员被教导说，检测和报告错误非常重要；他们通常将此理解为“检测到的错误越多越好”。这导致了一种过度防御的风格，即任何看起来有点可疑的东西都会被异常拒绝，从而导致不必要的异常激增，从而增加了系统的复杂性。

在设计 Tcl 脚本语言时，我自己也犯了这个错误。Tcl包含一个 unset 命令，可用于删除变量。我定义了unset ，如果变量不存在，它会抛出一个错误。当时我认为，如果有人试图删除一个不存在的变量，那一定是bug，所以Tcl应该报告它。然而，unset 最常见的用途之一是清理先前操作创建的临时状态。通常很难准确地预测创建了什么状态，特别是如果操作中途中止。因此，最简单的方法是删除所有可能已创建的变量。 unset 的定义使这变得很尴尬：

开发人员最终会将对 unset 的调用包含在 catch 语句中，以捕获和忽略 unset 抛出的错误。回想起来，unset
命令的定义是我在 Tcl 设计中犯下的最大错误之一。

使用异常来避免处理困难的情况是很诱人的：
与其想出一个干净的处理方法，不如抛出一个异常，把问题推给调用者。有些人可能会认为这种方法赋予了调用者权力，因为它允许每个调用者以不同的方式处理异常。但是，如果你在弄清楚特定情况下该怎么做时遇到麻烦，那么

调用者很可能也不知道该怎么做。在这种情况下生成异常只会将问题传递给其他人，并增加系统的复杂性。

类抛出的异常是其接口的一部分；具有大量异常的类具有复杂的接口，并且比异常较少的类更浅。 异常是接口中一个特别复杂的元素。

它可以在被捕获之前通过多个堆栈级别传播，因此它不仅会影响
方法调用者，还会影响更高级别的调用者（及其
接口）。

抛出异常很容易；处理它们很难。因此，异常的复杂性来自异常处理代码。减少异常处理造成的复杂性损害的最佳方法是 减少必须处理异常的地方的数量。 本章的其余部分将讨论减少异常处理程序数量的四种技术。

10.3 定义不存在的错误

消除异常处理复杂性的最佳方法是定义你的 API，以便没有要处理的异常： 定义不存在的错误。 这可能看起来很亵渎神灵，但它在实践中非常有效。考虑上面讨论的 Tcl unset命令。当 unset 被要求删除未知变量时，它不应该抛出错误，而应该直接返回而不做任何事情。我应该稍微改变 unset 的定义：与其删除变量，unset 应该确保变量不再存在。使用第一个定义，如果变量不存在，unset 无法完成其工作，因此生成异常是有意义的。使用第二个定义，使用不存在的变量名调用unset 是非常自然的。在这种情况下，它的工作已经完成，因此它可以简单地返回。不再有需要报告的错误情况。

10.4 示例：Windows 中的文件删除

文件删除提供了另一个关于如何通过定义来消除错误的例子。如果文件在一个进程中打开，Windows 操作系统不允许删除该文件。这对于开发者和用户来说是一个持续的挫折来源。为了删除正在使用的文件，用户必须在系统中搜索

找到打开该文件的进程，然后终止该进程。有时用户会放弃并重启他们的系统，只是为了删除一个文件。

Unix 操作系统对文件删除的定义更为优雅。在 Unix 中，如果一个文件在被删除时处于打开状态，Unix 不会立即删除该文件。相反，它会将该文件标记为删除，然后删除操作成功返回。文件名已从其目录中删除，因此没有其他进程可以打开旧文件，并且可以创建具有相同名称的新文件，但现有的文件数据仍然存在。已经打开该文件的进程可以继续正常读取和写入它。一旦所有访问进程都关闭了该文件，其数据就会被释放。

Unix 方法通过定义消除了两种不同类型的错误。首先，如果文件当前正在使用，删除操作不再返回错误；删除成功，文件最终将被删除。其次，删除正在使用的文件不会为使用该文件的进程创建异常。解决这个问题的一种可能方法是立即删除该文件，并将所有打开的文件标记为禁用；其他进程尝试读取或写入已删除的文件将失败。但是，这种方法会为这些进程创建新的错误来处理。相反，Unix 允许它们继续正常访问该文件；延迟文件删除消除了错误的产生。

Unix 允许进程继续读取和写入一个注定要失败的文件，这可能看起来很奇怪，但我从未遇到过这种情况导致重大问题。与 Windows 的定义相比，Unix 对文件删除的定义对于开发者和用户来说都更容易使用。

10.5 示例：Java substring 方法

作为最后一个例子，考虑 Java String 类及其 substring 方法。给定字符串中的两个索引， substring 返回从第一个索引给出的字符开始，到第二个索引之前的字符结束的子字符串。但是，如果任一索引超出字符串的范围，则 substring 抛出 IndexOutOfBoundsException。此异常是不必要的，并且使此方法的使用复杂化。我经常发现自己处于这样一种情况，即一个或两个索引可能超出字符串的范围，并且我想提取字符串中与指定范围重叠的所有字符。

不幸的是，这需要我检查每个索引并将它们向上舍入到

零或者到字符串的末尾；一个单行方法调用现在变成了5-10行代码。

如果 Java substring 方法能够自动执行此调整，使其实现以下 API，则该方法将更易于使用："返回索引大于或等于 beginIndex 且小于 endIndex 的字符串的字符（如果有）。"这是一个简单而自然的 API，它定义了 IndexOutOfBoundsException 异常不再存在。即使一个或两个索引为负数，或者 beginIndex 大于 endIndex，该方法的行为现在也得到了很好的定义。这种方法简化了该方法的 API，同时增加了其功能，因此它使该方法更深入。许多其他语言都采用了无错误方法；例如，Python 为超出范围的列表切片返回一个空结果。

当我主张将错误定义为不存在时，人们有时会反驳说，抛出错误会捕获错误；如果错误被定义为不存在，那不会导致软件出现更多错误吗？也许这就是 Java 开发人员决定 substring 应该抛出异常的原因。充满错误的方法可能会捕获一些错误，但它也会增加复杂性，从而导致其他错误。在充满错误的方法中，开发人员必须编写额外的代码来避免或忽略错误，这增加了出现错误的可能性；或者，他们可能忘记编写额外的代码，在这种情况下，可能会在运行时抛出意外错误。相反，将错误定义为不存在简化了 API，并减少了必须编写的代码量。

总的来说，减少错误的最佳方法是使软件更简单。

10.6 屏蔽异常

减少必须处理异常的地方数量的第二种技术是 异常屏蔽。通过这种方法，异常情况在系统的较低级别被检测和处理，因此较高级别的软件不需要知道这种情况。异常屏蔽在分布式系统中尤其常见。例如，在诸如 TCP 之类的网络传输协议中，数据包可能由于各种原因（例如损坏和拥塞）而被丢弃。TCP 通过在其实现中重新发送丢失的数据包来屏蔽数据包丢失，因此所有数据最终都会通过，并且客户端不知道丢弃的数据包。

NFS 网络文件系统中发生了一个更具争议的屏蔽示例。如果 NFS 文件服务器崩溃或由于任何原因无法响应，客户端会一遍又一遍地向服务器重新发出请求，直到问题最终得到解决。客户端上的底层文件系统代码不会向调用应用程序报告任何异常。正在进行的操作（以及因此应用程序）只是挂起，直到操作可以成功完成。如果挂起持续的时间超过很短的时间，NFS 客户端会在用户的控制台上打印诸如“NFS 服务器 xyzzy 未响应，仍在尝试”之类的消息。

NFS 用户经常抱怨他们的应用程序在等待 NFS 服务器恢复正常运行时挂起的事实。许多人建议 NFS 应该使用异常中止操作，而不是挂起。但是，报告异常会使事情变得更糟，而不是更好。

如果应用程序失去对其文件的访问权限，则应用程序无法执行太多操作。一种可能性是应用程序重试文件操作，但这仍然会挂起应用程序，并且在 NFS 层中的一个位置执行重试比在每个应用程序中的每个文件系统调用中执行重试更容易（编译器不必担心这一点！）。另一种选择是应用程序中止并将错误返回给它们的调用者。调用者也不太可能知道该怎么做，因此它们也会中止，从而导致用户工作环境崩溃。在文件服务器关闭时，用户仍然无法完成任何工作，并且一旦文件服务器恢复正常，他们将不得不重新启动所有应用程序。

因此，最好的选择是让 NFS 屏蔽错误并挂起应用程序。通过这种方法，应用程序不需要任何代码来处理服务器问题，并且一旦服务器恢复正常，它们就可以无缝地恢复。如果用户厌倦了等待，他们总是可以手动中止应用程序。

异常屏蔽并非在所有情况下都有效，但它在适用的情况下是一种强大的工具。它会产生更深层次的类，因为它减少了类的接口（用户需要注意的异常更少），并以屏蔽异常的代码的形式添加了功能。异常屏蔽是向下传递复杂性的一个例子。

10.7 异常聚合

第三种降低与异常相关的复杂性的技术是异常聚合。异常聚合背后的思想是用一段代码处理许多异常

用一段代码处理许多异常；与其为许多单独的异常编写不同的处理程序，不如用一个处理程序在一个地方处理所有异常。

考虑如何在 Web 服务器中处理缺失的参数。Web 服务器实现了一组 URL。当服务器收到一个传入的 URL 时，它会分派到一个特定于 URL 的服务方法来处理该 URL 并生成一个响应。该 URL 包含用于生成响应的各种参数。每个服务方法都会调用一个较低级别的方法（我们称之为

getParameter）来从 URL 中提取它需要的参数。如果 URL 不包含所需的参数，getParameter 会抛出一个异常。

当软件设计课程的学生实现这样一个服务器时，他们中的许多人将对getParameter 的每个不同调用都包装在一个单独的异常处理程序中，以捕获NoSuchParameter 异常，如图 10.1所示。这导致了大量的处理程序，所有这些处理程序本质上都做了相同的事情（生成一个错误响应）。

图 10.1：顶部的代码分派到 Web 服务器中的多个方法之一，每个方法处理一个特定的 URL。这些方法（底部）中的每一个都使用来自传入 HTTP 请求的参数。在此图中，对getParameter的每个调用都有一个单独的异常处理程序；这会导致重复的代码。

一个更好的方法是聚合异常。与其在各个服务方法中捕获异常，不如让它们传播到 Web 服务器的顶层分派方法，如图 10.2所示。一个处理程序在

此方法中可以捕获所有异常，并为缺失的参数生成适当的错误响应。

在Web示例中，聚合方法可以更进一步。
除了缺少参数之外，在处理网页时还可能发生许多其他错误；例如，参数可能没有正确的语法（服务方法期望一个整数，但值是“xyz”），或者用户可能没有获得请求操作的权限。在每种情况下，错误都应导致错误响应；错误仅在于响应中包含的错误消息不同（“参数'quantity'未在URL中出现”或“'quantity'参数的错误值'xyz'；必须是正整数”）。因此，所有导致错误响应的条件都可以通过单个顶级异常处理程序来处理。错误消息可以在抛出异常时生成，并作为异常记录中的变量包含；例如，getParameter将生成“参数'quantity'未在URL中出现”消息。顶级处理程序从异常中提取消息，并将其合并到错误响应中。

图10.2：此代码在功能上等同于图10.1，但异常处理已聚合：调度程序中的单个异常处理程序捕获来自所有URL特定方法的所有NoSuchParameter异常。

前一段中描述的聚合从封装和信息隐藏的角度来看具有良好的属性。顶级异常处理程序封装了有关如何生成错误响应的知识，但它对特定错误一无所知；它只是使用异常中提供的错误消息。getParameter方法封装了有关如何从URL中提取参数的知识，并且它也知道如何

以人类可读的形式描述提取错误。这两条信息密切相关，因此将它们放在同一个地方是有意义的。但是，getParameter对HTTP错误响应的语法一无所知。当向Web服务器添加新功能时，可能会创建像

getParameter这样的新方法，它们有自己的错误。如果新方法以与getParameter相同的方式抛出异常（通过生成从同一超类继承的异常，并在每个异常中包含错误消息），它们可以插入到现有系统中，而无需进行其他更改：顶级处理程序将自动为它们生成错误响应。

此示例说明了一种通常有用的异常处理设计模式。如果系统处理一系列请求，那么定义一个异常来中止当前请求、清理系统状态并继续处理下一个请求是很有用的。该异常在系统请求处理循环顶部的单个位置被捕获。可以在请求处理的任何时候抛出此异常以中止请求；可以为不同的条件定义不同的异常子类。这种类型的异常应与对整个系统来说是致命的异常明确区分开来。

如果异常在被处理之前向上传播多个堆栈级别，则异常聚合效果最佳；这允许在同一位置处理来自更多方法的更多异常。这与异常屏蔽相反：

如果异常在低级方法中处理，则屏蔽通常效果最佳。对于屏蔽，低级方法通常是许多其他方法使用的库方法，因此允许异常传播会增加处理它的位置数量。屏蔽和聚合的相似之处在于，这两种方法都将异常处理程序放置在可以捕获最多异常的位置，从而消除了许多原本需要创建的处理程序。

异常聚合的另一个例子发生在用于崩溃恢复的RAMCloud存储系统中。RAMCloud系统由一组存储服务器组成，这些服务器保留每个对象的多个副本，因此系统可以从各种故障中恢复。例如，如果服务器崩溃并丢失所有数据，RAMCloud会使用存储在其他服务器上的副本重建丢失的数据。错误也可能发生在较小的范围内；例如，服务器可能会发现单个对象已损坏。

RAMCloud没有针对每种不同类型错误的单独恢复机制。相反，RAMCloud将许多较小的错误“提升”为较大的错误。原则上，RAMCloud可以通过恢复该

从备份副本中获取一个对象。然而，它并没有这样做。相反，如果它发现一个损坏的对象，它会使包含该对象的服务器崩溃。
RAMCloud 使用这种方法是因为崩溃恢复非常复杂，并且这种方法最大限度地减少了必须创建的不同恢复机制的数量。为崩溃的服务器创建恢复机制是不可避免的，因此 RAMCloud 对其他类型的恢复也使用相同的机制。这减少了必须编写的代码量，并且还意味着服务器崩溃恢复会更频繁地被调用。因此，恢复中的错误更有可能被发现和修复。

将损坏的对象升级为服务器崩溃的一个缺点是，它大大增加了恢复的成本。这在 RAMCloud 中不是问题，因为对象损坏非常罕见。但是，对于频繁发生的错误，错误升级可能没有意义。举个例子，如果服务器的任何一个网络数据包丢失，就让服务器崩溃是不切实际的。

考虑异常聚合的一种方式是，它用一个可以处理多种情况的通用机制，取代了几个专门用途的机制，每个机制都针对特定情况量身定制。这提供了通用机制优势的另一个例证。

10.8 直接崩溃？

减少与异常处理相关的复杂性的第四种技术是使应用程序崩溃。在大多数应用程序中，会有一些不值得尝试处理的错误。通常，这些错误很难或不可能处理，而且不会经常发生。响应这些错误最简单的方法是打印诊断信息，然后中止应用程序。

一个例子是存储分配期间发生的“内存不足”错误。
考虑 C 语言中的 malloc 函数，如果它无法分配所需的内存块，则返回 NULL 。这是一个不幸的行为，因为它假设 malloc 的每个调用者都会检查返回值，并在没有内存时采取适当的措施。应用程序包含大量对 malloc的调用，因此在每次调用后检查结果会增加显著的复杂性。如果程序员忘记了检查（这很可能发生），那么如果内存耗尽，应用程序将取消对空指针的引用，导致崩溃，从而掩盖了真正的问题。

此外，当应用程序发现内存耗尽时，它能做的事情并不多。原则上，应用程序可以寻找不需要的

内存来释放，但如果应用程序有不需要的内存，它可能已经释放了它，这本可以避免首先出现内存不足的错误。今天的系统拥有如此多的内存，以至于内存几乎永远不会耗尽；如果确实耗尽了，通常表明应用程序中存在错误。因此，尝试处理内存不足错误很少有意义；这会为过少的收益带来过多的复杂性。

一个更好的方法是定义一个新的方法 ckalloc，它调用 malloc，检查结果，如果内存耗尽，则中止应用程序并显示错误消息。应用程序永远不会直接调用 malloc ；它总是调用ckalloc。

在诸如 C++ 和 Java 等较新的语言中， new 运算符在内存耗尽时会抛出异常。捕获此异常意义不大，因为异常处理程序很可能也会尝试分配内存，这也会失败。动态分配的内存是任何现代应用程序的基本要素，如果内存耗尽，应用程序继续运行是没有意义的；最好在检测到错误时立即崩溃。

在许多其他错误示例中，崩溃应用程序是有意义的。对于大多数程序，如果在读取或写入打开的文件时发生 I/O 错误（例如磁盘硬错误），或者无法打开网络套接字，则应用程序无法执行太多操作来恢复，因此使用清晰的错误消息中止是一种明智的方法。这些错误很少发生，因此不太可能影响应用程序的整体可用性。如果应用程序遇到内部错误（例如不一致的数据结构），则使用错误消息中止也是合适的。这种情况可能表明程序中存在错误。

在特定错误时崩溃是否可以接受取决于应用程序。对于复制的存储系统，在 I/O 错误时中止是不合适的。相反，系统必须使用复制的数据来恢复丢失的任何信息。恢复机制将大大增加程序的复杂性，但恢复丢失的数据是系统为其用户提供的价值的重要组成部分。

10.9 设计不存在的特殊情况

出于与定义不存在的错误相同的原因，定义不存在的其他特殊情况也是有意义的。特殊情况可能导致代码中充斥着 if 语句，这使得代码难以理解并导致错误。因此，应尽可能消除特殊情况。最好的方法是以一种自动处理特殊情况而无需任何额外代码的方式设计正常情况。

在 Chapter 6 中描述的文本编辑器项目中，学生必须实现一种机制来选择文本并复制或删除所选内容。大多数学生在其选择实现中引入了一个状态变量，以指示选择是否存在。他们可能选择了这种方法，因为有时屏幕上没有可见的选择，因此在实现中表示这个概念似乎很自然。但是，这种方法导致了无数次检查以检测“无选择”条件并对其进行特殊处理。

通过消除“无选择”特殊情况，可以简化选择处理代码，以便选择始终存在。当屏幕上没有可见的选择时，它可以在内部用一个空选择来表示，其起始和结束位置相同。使用这种方法，可以编写选择管理代码，而无需任何“无选择”检查。复制选择时，如果选择为空，则将在新位置插入 0 个字节（如果实现正确，则无需检查 0 个字节作为特殊情况）。同样，应该可以设计用于删除选择的代码，以便在没有任何特殊情况检查的情况下处理空情况。考虑一下单行上的所有选择。要删除选择，请提取选择之前的行部分，并将其与选择之后的行部分连接起来以形成新行。如果选择为空，则此方法将重新生成原始行。

此示例还说明了 Chapter 7 中的“不同层，不同抽象”思想。“无选择”的概念在用户如何考虑应用程序的界面方面是有意义的，但这并不意味着它必须在应用程序内部显式表示。拥有一个始终存在但有时为空且因此不可见的选择，会导致更简单的实现。

10.10 过犹不及

定义掉异常或在模块内部屏蔽它们只有在模块外部不需要异常信息时才有意义。对于

本章中的示例，例如 Tcl unset 命令和 Java substring方法；在极少数情况下，调用者关心异常检测到的特殊情况，还有其他方法可以获取此信息。

然而，将这种想法发挥过度也是有可能的。在一个用于网络通信的模块中，一个学生团队屏蔽了所有网络异常：如果发生网络错误，该模块会捕获它，丢弃它，然后像没有问题一样继续运行。这意味着使用该模块的应用程序无法知道消息是否丢失或对等服务器是否发生故障；如果没有这些信息，就不可能构建健壮的应用程序。在这种情况下，模块必须公开异常，即使它们增加了模块接口的复杂性。

对于异常，就像软件设计中的许多其他领域一样，您必须确定什么是重要的，什么是不重要的。不重要的东西应该被隐藏，而且隐藏得越多越好。但是当某些东西很重要时，它必须被暴露出来。

10.11 结论

任何形式的特殊情况都会使代码更难理解，并增加出现错误的几率。本章重点介绍了异常，异常是特殊情况代码的最重要来源之一，并讨论了如何减少必须处理异常的地方的数量。最好的方法是重新定义语义以消除错误条件。对于无法消除的异常，您应该寻找在较低级别屏蔽它们的机会，以便限制它们的影响，或者将几个特殊情况处理程序聚合到一个更通用的处理程序中。总之，这些技术可以对整体系统复杂性产生重大影响。

[1]丁源等人，“简单测试可以预防大多数严重故障：分布式数据密集型系统中的生产故障分析”，2014 年 USENIX 操作系统设计与实现会议。

第 11 章

设计两次

设计软件很难，因此您第一次考虑如何
构建模块或系统不太可能产生最佳设计。如果您考虑每个主要设计
决策的多个选项，您最终会得到
更好的结果：设计两次。

假设您正在设计一个类，该类将管理 GUI 文本编辑器的文件文本。第一步是定义该类将呈现给编辑器其余部分的接口；而不是选择首先想到的想法，考虑几种可能性。一种选择是面向行的接口，具有插入、修改和删除整行文本的操作。另一种选择是基于单个字符插入和删除的接口。第三种选择是面向字符串的接口，该接口对可能跨越行边界的任意字符范围进行操作。您不需要确定每个替代方案的每个功能；在这一点上，草拟出一些最重要的方法就足够了。

尝试选择彼此截然不同的方法；您会
学到更多。即使您确定只有一种合理的
方法，也要考虑第二个设计，无论您认为它有多糟糕
。思考该设计的弱点并将其与其他设计的特征进行对比将
具有启发意义。

在您粗略地设计出备选方案后，列出每个方案的优缺点。对于一个界面来说，最重要的考虑因素是更高级别软件的易用性。在上面的例子中，面向行的界面和面向字符的界面都需要在使用文本类的软件中进行额外的工作。面向行的界面将需要更高级别的软件在诸如剪切和粘贴选择等部分行和多行操作期间分割和连接行。面向字符的界面将需要循环来实现修改多个字符的操作。同样值得考虑其他因素：

- 一个备选项是否比另一个备选项具有更简单的接口？在文本示例中，所有文本接口都相对简单。
- 一个接口是否比另一个接口更通用？
- 一个接口是否比另一个接口实现得更有效率？在文本示例中，面向字符的方法可能比其他方法慢得多，因为它需要为每个字符单独调用文本模块。

一旦你比较了不同的设计方案，你就能更好地确定最佳设计方案。最好的选择可能是其中一个备选项，或者你可能会发现你可以将多个备选项的特性组合成一个比任何原始选择都更好的新设计。

有时，所有备选项都不太吸引人；当发生这种情况时，看看你是否能提出其他的方案。利用你在原始备选项中发现的问题来驱动新的设计。如果你正在设计文本类，并且只考虑了面向行和面向字符的方法，你可能会注意到每个备选项都很笨拙，因为它需要更高级别的软件来执行额外的文本操作。这是一个危险信号：如果有一个文本类，它应该处理所有的文本操作。为了消除额外的文本操作，文本接口需要更紧密地匹配更高级别软件中发生的操作。这些操作并不总是对应于单个字符或单行。这种推理应该引导你使用面向范围的文本API，从而消除早期设计中的问题。

设计两次原则可以在系统的多个层面上应用。对于一个模块，你可以首先使用这种方法来选择接口，如上所述。
然后，你可以在设计实现时再次应用它：对于文本类，你可以考虑诸如行链表、固定大小的字符块或“间隙缓冲区”之类的实现。对于实现来说，目标将与接口不同：对于实现来说，最重要的是简单性和性能。在系统的更高层次上探索多种设计也很有用，例如在为用户界面选择功能时，或者在将系统分解为主要模块时。在每种情况下，如果你能比较几个备选项，就更容易确定最佳方法。

设计两次并不需要花费大量的额外时间。对于像类这样较小的模块，你可能只需要一两个小时来考虑备选项。与你将花费数天或数周来实现该类相比，这只是一小段时间。最初的设计实验可能会产生一个明显更好的设计，这将超过花费在设计两次上的时间。对于较大的模块，你将在最初的设计探索中花费更多的时间，但实现也将花费更长的时间，并且更好的设计的好处也将更高。

我注意到，对于真正聪明的人来说，设计两次原则有时很难接受。当他们成长时，聪明的人发现他们对任何问题的第一个快速想法足以获得好成绩；没有必要考虑第二或第三种可能性。这使得他们很容易养成不良的工作习惯。然而，随着这些人年龄的增长，他们会被提升到问题越来越困难的环境中。最终，每个人都会达到这样一个地步：你的第一个想法不再足够好；如果你想获得真正好的结果，你必须考虑第二种可能性，或者可能是第三种，无论你多么聪明。大型软件系统的设计就属于这一类：没有人足够优秀，第一次尝试就能成功。

不幸的是，我经常看到聪明的人坚持实现他们脑海中出现的第一个想法，这导致他们无法发挥真正的潜力（这也使得他们很难合作）。也许他们下意识地认为“聪明的人第一次就能做对”，所以如果他们尝试多种设计，那就意味着他们毕竟不够聪明。事实并非如此。

不是因为你不聪明，而是因为问题真的很难！此外，这是一件好事：解决一个需要仔细思考的难题，比解决一个根本不需要思考的简单问题有趣得多。

设计两次的方法不仅能改进你的设计，还能提高你的设计技能。构思和比较多种方法的过程会让你了解哪些因素能使设计更好或更差。

随着时间的推移，这将使你更容易排除糟糕的设计，并专注于真正优秀的设计。

第12章

为什么要写注释？四个借口

代码内文档在软件设计中起着至关重要的作用。注释对于帮助开发人员理解系统并高效工作至关重要，但注释的作用远不止于此。文档在抽象中也起着重要作用；没有注释，你就无法隐藏复杂性。最后，如果正确地编写注释，实际上会改进系统的设计。 相反，如果一个好的软件设计文档记录不佳，它的大部分价值就会丧失。

不幸的是，这种观点并没有得到普遍认同。很大一部分生产代码基本上不包含任何注释。许多开发人员认为注释是浪费时间；另一些人看到了注释的价值，但不知何故总是无法抽出时间来编写它们。幸运的是，许多开发团队认识到文档的价值，而且似乎这些团队的普及率正在逐渐提高。然而，即使在鼓励文档编写的团队中，注释通常也被视为繁琐的工作，许多开发人员不理解如何编写注释，因此最终的文档通常很平庸。不充分的文档给软件开发带来了巨大而不必要的阻力。

在本章中，我将讨论开发人员用来避免编写注释的借口，以及注释真正重要的原因。 第13章 将描述如何编写好的注释，以及之后的几章将讨论相关问题，例如选择变量名以及如何使用文档来改进系统的设计。我希望这些章节能让你相信三件事：好的注释可以大大提高软件的整体质量；编写好的注释并不难；而且（这可能很难相信）编写注释实际上可能很有趣。

当开发人员不编写注释时，他们通常会用以下一个或多个借口来为自己的行为辩解：

- “好的代码本身就是文档。”

- “我没有时间写注释。”“注释会过时
- 并产生误导。”“我见过的注释都毫无价值；何必
- 费心呢？”在下面的章节中，我将依次讨论这些借口。

12.1 好的代码是自文档化的

有些人认为，如果代码写得好，那么它就非常明显，不需要任何注释。这是一种美妙的神话，就像冰淇淋对你的健康有益的谣言一样：我们真的很想相信它！不幸的是，这根本不是真的。当然，在编写代码时，你可以做一些事情来减少对注释的需求，例如选择好的变量名（参见第14章）。

尽管如此，仍然有大量的设计信息无法在代码中表示。例如，一个类接口中只有一小部分，比如其方法的签名，可以在代码中正式指定。接口的非正式方面，例如对每个方法作用的高级描述或其结果的含义，只能在注释中描述。还有许多其他无法在代码中描述的例子，例如特定设计决策的理由，或在什么条件下调用特定方法才有意义。

一些开发人员认为，如果其他人想知道一个方法的作用，他们应该直接阅读该方法的代码：这将比任何注释都更准确。读者有可能通过阅读方法的代码来推断出该方法的抽象接口，但这将是耗时且痛苦的。此外，如果你编写代码时期望用户阅读方法实现，你将尝试使每个方法尽可能短，以便于阅读。如果该方法做了任何重要的事，你将把它分解成几个较小的方法。这将导致大量浅层方法。

此外，这实际上并没有使代码更容易阅读：为了理解顶层方法的行为，读者可能需要理解嵌套方法的行为。对于大型系统，用户阅读代码来学习行为是不切实际的。

此外，注释是抽象的基础。回想一下第4章，抽象的目标是隐藏复杂性：抽象是实体的简化视图，它保留了基本信息，但省略了可以安全忽略的细节。如果用户必须阅读方法的代码才能使用它，那么就没有抽象：该方法的所有复杂性都暴露出来了。

如果没有注释，方法的唯一抽象就是它的声明，它指定了它的名称以及它的参数和结果的名称和类型。声明缺少太多基本信息，无法单独提供有用的抽象。例如，一个提取子字符串的方法可能有两个参数，start和end，表示要提取的字符范围。仅从声明中，无法判断提取的子字符串是否包含end指示的字符，或者如果start > end会发生什么。

注释允许我们捕获调用者需要的额外信息，从而完成简化的视图，同时隐藏实现细节。同样重要的是，注释是用人类语言（如英语）编写的；这使得它们不如代码精确，但它提供了更强的表达能力，因此我们可以创建简单、直观的描述。如果你想使用抽象来隐藏复杂性，注释是必不可少的。

12.2 我没有时间写注释

将注释的优先级排在其他开发任务之下是很诱人的。如果在添加新功能和记录现有功能之间做出选择，选择新功能似乎是合乎逻辑的。然而，软件项目几乎总是面临时间压力，并且总会有一些事情看起来比编写注释更重要。因此，如果你允许文档被降低优先级，你最终将没有任何文档。

对这种借口的辩驳是第15页讨论的投资心态。如果你想要一个清晰的软件结构，让你能够长期高效地工作，那么你必须预先花一些额外的时间来创建这个结构。好的注释对软件的可维护性有很大的影响，所以在注释上花费的精力会很快得到回报。

此外，编写注释并不需要花费大量时间。问问你自己，假设你不包含任何注释，你花了多少开发时间来输入代码（而不是设计、编译、测试等）；我怀疑答案是否超过10%。现在假设你花在输入注释上的时间和输入代码一样多；这应该是一个安全的上限。有了这些假设，编写好的注释不会使你的开发时间增加超过10%。拥有良好文档的好处将很快抵消这一成本。

此外，许多最重要的注释是那些与抽象相关的注释，例如类和方法的顶级文档。

第15章将论证这些注释应该作为设计过程的一部分来编写，并且编写文档的行为本身就是一种重要的设计工具，可以改进整体设计。这些注释会立即得到回报。

12.3 注释会过时并产生误导

注释有时确实会过时，但这在实践中不一定是一个主要问题。保持文档的更新不需要付出巨大的努力。
只有当代码发生重大更改时，才需要对文档进行大量更改，而代码更改将比文档更改花费更多的时间。第16章讨论了如何组织文档，以便在代码修改后尽可能容易地保持更新（关键思想是避免重复的文档，并使文档靠近相应的代码）。代码审查提供了一种很好的机制来检测和修复过时的注释。

12.4 我所见过的所有注释都是毫无价值的

在四个借口中，这可能是最站得住脚的一个。每个软件开发人员都见过没有提供任何有用信息的注释，而且大多数现有文档充其量也只是平庸之作。幸运的是，这个问题是可以解决的；一旦你知道如何编写，编写可靠的文档并不难。接下来的章节将阐述一个框架，说明如何编写好的文档并长期维护它。

12.5 编写良好的注释的好处

既然我已经讨论了（并且，希望已经驳斥了）反对编写注释的论点，那么让我们考虑一下你将从好的注释中获得的好处。注释背后的总体思想是捕捉设计者头脑中的信息，但这些信息无法在代码中表示出来。这些信息范围从低级细节（例如，驱动一段特别棘手的代码的硬件怪癖）到高级概念（例如，类的基本原理）。当其他开发人员稍后进行修改时，注释将使他们能够更快、更准确地工作。如果没有文档，未来的开发人员将不得不重新推导或猜测开发人员的原始知识；这将花费额外的时间，并且

如果新开发人员误解了原始设计师的意图，则存在出现错误的风险。即使原始设计师正在进行更改，注释也很有价值：如果您上次在一个代码片段中工作已经超过几周，您将忘记原始设计的许多细节。

第 2 章描述了复杂性在软件系统中表现出来的三种方式：

变更放大：一个看似简单的变更需要在许多地方进行代码修改。
认知负荷：为了进行变更，开发人员必须积累大量信息。

未知的未知：不清楚需要修改哪些代码，或者为了进行这些修改必须考虑哪些信息。

良好的文档有助于解决后两个问题。文档可以通过向开发人员提供他们进行更改所需的信息，并使开发人员可以轻松忽略不相关的信息来减少认知负荷。如果没有足够的文档，开发人员可能需要阅读大量代码来重建设计师的想法。文档还可以通过阐明系统的结构来减少未知的未知，从而清楚地了解哪些信息和代码与任何给定的更改相关。

章节 2 指出复杂性的主要原因是依赖性和模糊性。良好的文档可以阐明依赖性，并填补空白以消除模糊性。

接下来的几章将向您展示如何编写良好的文档。它们还将讨论如何将文档编写集成到设计过程中，以便改进软件的设计。

第13章

注释应该描述代码中不明显的内容

编写注释的原因是编程语言中的语句无法捕捉到开发人员编写代码时脑海中的所有重要信息。注释记录这些信息，以便后来的开发人员可以轻松理解和修改代码。注释的指导原则是：注释应该描述代码中不明显的内容。

有很多事情从代码中并不明显。有时是一些低级别的细节并不明显。例如，当一对索引描述一个范围时，由索引给出的元素是在范围内还是在范围外并不明显。有时不清楚为什么需要代码，或者为什么以特定的方式实现代码。有时开发人员会遵循一些规则，例如“始终在a之前调用b”。您可能可以通过查看所有代码来猜测规则，但这很痛苦且容易出错；注释可以使规则明确而清晰。

注释最重要的原因之一是抽象，其中包含许多从代码中不明显的信息。抽象的想法是提供一种思考事物的简单方法，但代码非常详细，以至于很难仅通过阅读代码来理解抽象。

注释可以提供一个更简单、更高级的视图（“在调用此方法后，网络流量将限制为maxBandwidth字节/秒”）。即使可以通过阅读代码来推断出这些信息，我们也不希望强迫模块的用户这样做：阅读代码既耗时又迫使他们考虑许多使用模块不需要的信息。开发人员应该能够理解模块提供的抽象，而无需阅读除其外部可见声明之外的任何代码。 做到这一点的唯一方法是用注释补充声明。

本章讨论了需要在注释中描述哪些信息以及如何编写好的注释。正如您将看到的，好的注释通常以与代码不同的详细程度解释事物，在某些情况下更详细，而在另一些情况下则不太详细（更抽象）。

13.1 选择约定

编写注释的第一步是确定注释的约定，例如您将注释什么以及您将使用的注释格式。如果您正在使用一种语言进行编程，该语言存在文档编译工具，例如Java的Javadoc，C++的Doxygen或Go的godoc，请遵循这些工具的约定。这些约定都不是完美的，但是这些工具提供了足够的好处来弥补这一点。如果您在没有现有约定可遵循的环境中进行编程，请尝试采用来自其他类似语言或项目的约定；这将使其他开发人员更容易理解和遵守您的约定。

约定有两个目的。首先，它们确保一致性，这使注释更易于阅读和理解。其次，它们有助于确保您实际编写注释。如果您不清楚要注释什么以及如何注释，很容易最终根本不写注释。

大多数评论属于以下类别之一：

接口：紧接在诸如类、数据结构、函数或方法等模块声明之前的注释块。该注释描述了模块的接口。对于一个类，该注释描述了该类提供的总体抽象。对于一个方法或函数，该注释描述了它的总体行为、它的参数和返回值（如果有的话）、它产生的任何副作用或异常，以及调用者在调用该方法之前必须满足的任何其他要求。

数据结构成员：数据结构中字段声明旁边的注释，例如类的实例变量或静态变量。

实现注释：方法或函数代码内部的注释，用于描述代码的内部工作原理。

跨模块注释：描述跨模块边界的依赖关系的注释。

最重要的注释是前两类中的注释。每个类
都应该有一个接口注释，每个类变量都应该有一个注释，
并且每个方法都应该有一个接口注释。偶尔，

对于变量或方法的声明非常明显，以至于在注释中添加任何有用的内容都是多余的（getter和setter有时属于这一类），但这种情况很少见；与其花费精力担心是否需要注释，不如注释所有内容更容易。实现注释通常是不必要的（见下面的第13.6节）。跨模块注释是最罕见的，而且编写起来很有问题，但当需要它们时，它们非常重要；第13.7节更详细地讨论了它们。

13.2 不要重复代码

不幸的是，许多注释并没有特别的帮助。最常见的原因是注释重复了代码：注释中的所有信息都可以很容易地从注释旁边的代码中推断出来。这是一个最近发表的研究论文中的代码示例：

```
ptr_copy = 获取副本(obj)                          # 获取指针副本
如果 is_unlocked(ptr_copy):                       # 对象是否空闲？
  返回 obj                                        # 返回当前对象
如果 is_copy(ptr_copy):                           # 已经是副本？
  返回 obj                                        # 返回 obj
thread_id = get_thread_id(ptr_copy)
如果 thread_id == ctx.thread_id:                  # 当前上下文锁定
  返回 ptr_copy                                   # 返回副本
```

除了“Lockedby”注释之外，所有这些注释中没有任何有用的信息，该注释暗示了关于线程的一些可能不明显的内容。请注意，这些注释与代码的详细程度大致相同：每行代码都有一个注释，用于描述该行。这样的注释很少有用。

以下是更多重复代码的注释示例：

```
// 添加一个水平滚动条
hScrollBar = new JScrollBar(JScrollBar.HORIZONTAL);
add(hScrollBar, BorderLayout.SOUTH);

// 添加一个垂直滚动条
vScrollBar = new JScrollBar(JScrollBar.VERTICAL);
add(vScrollBar, BorderLayout.EAST);

// 初始化与插入符位置相关的值
caretX = 0;
```

```
caretY     = 0;
caretMemX  = null;
```

这些注释都没有提供任何价值。对于前两个注释，代码已经足够清晰，不需要注释；在第三种情况下，注释可能有用，但当前的注释没有提供足够的细节来提供帮助。

在编写注释后，问自己以下问题：从未见过代码的人是否可以通过查看注释旁边的代码来编写注释？如果答案是肯定的，就像上面的例子一样，那么注释不会使代码更容易理解。像这样的注释是为什么有些人认为注释毫无价值。

另一个常见的错误是在注释中使用与被记录实体名称中相同的词语：

```
/*
 * 从REQ获取规范化的资源名称。
 */
private static String[] getNormalizedResourceNames(
       HTTPRequest req) ...

/*
 * 将PARAMETER向下转型为TYPE。
 */
private static Object downCastParameter(String parameter, String type)
         ...

/*
 * 文本中每行的水平填充。
 */
private static final int textHorizontalPadding = 4;
```

这些注释只是从方法或变量名中提取单词，可能
从参数名和类型中添加一些单词，并将它们组成一个句子。
例如，第二个注释中唯一不在代码中的是
单词“to”！ 再次，这些注释可以通过查看
声明来编写，而无需理解方法或变量；因此，
它们没有价值。

危险信号：注释重复代码

如果注释中的信息已经从注释旁边的代码中显而易见，那么该注释就没有帮助。这方面的一个例子是当注释使用构成其描述对象的名称的相同单词时。

与此同时，注释中缺少重要的信息：例如，“规范化资源名称”是什么，以及getNormalizedResourceNames返回的数组的元素是什么？“向下转型”是什么意思？填充的单位是什么，填充是在每行的一侧还是两侧？在注释中描述这些内容将很有帮助。

编写好的注释的第一步是在注释中使用与被描述实体的名称不同的单词。为注释选择提供有关实体含义的附加信息的单词，而不仅仅是重复其名称。例如，以下是textHorizontalPadding的更好的注释：

```
/*
 * 在每行文本的左侧和右侧留出的空白量，以像素为单位。

 */
private static final int textHorizontalPadding = 4;
```

此注释提供了从声明本身并不明显的额外信息，例如单位（像素）以及填充适用于每行的两侧。注释没有使用术语“填充”，而是解释了什么是填充，以防读者不熟悉该术语。

13.3 低级别注释增加精确度

既然您已经知道不该做什么，那么让我们讨论一下您应该在注释中放入哪些信息。注释通过在不同的详细程度级别提供信息来增强代码。一些注释提供的信息比代码更低级、更详细；这些注释通过阐明代码的确切含义来增加精确度。另一些注释提供的信息比代码更高级、更抽象；这些注释提供直觉，例如

代码背后的推理，或者一种更简单、更抽象的思考代码的方式。与代码处于同一级别的注释很可能会重复代码。
本节更详细地讨论了低级别方法，下一节讨论了高级别方法。

在注释变量声明（如类实例变量、方法参数和返回值）时，精确度最有用。变量声明中的名称和类型通常不是很精确。注释可以填补缺失的细节，例如：

- 此变量的单位是什么？
- 边界条件是包含性的还是排他性的？
- 如果允许空值，这意味着什么？
- 如果变量引用了最终必须释放或关闭的资源，谁负责释放或关闭它？

- 对于变量（*invariants*），是否总有一些属性是成立的，
  例如“此列表始终包含至少一个条目”？

其中一些信息可以通过检查所有使用该变量的代码来推断出来。然而，这既耗时又容易出错；声明的注释应该足够清晰和完整，以避免这种不必要的情况。当我说声明的注释应该描述代码中不明显的内容时，“代码”指的是注释旁边的代码（声明），而不是“应用程序中的所有代码”。

变量注释最常见的问题是注释过于模糊。以下是两个不够精确的注释示例：

```
// resp Buffer 中的当前偏移量
uint32_t offset;

// 包含文档中的所有行宽和
// 出现次数。
private TreeMap<Integer, Integer> lineWidths;
```

在第一个示例中，“当前”的含义不明确。在第二个示例中，不清楚 TreeMap 中的键是行宽，值是出现次数。此外，宽度是以像素还是字符为单位测量的？下面修改后的注释提供了更多细节：

```
// 此缓冲区中尚未返回给客户端的第一个对象的位置。
```

```
uint32_t offset;

// 保存有关行长度的统计信息，格式为 <长度，计数>
// 其中长度是行中的字符数（包括
// 换行符），计数是具有以下字符数的行数
// 正好是那么多字符。如果没有特定长度的行，则该长度没有条目。

私有 TreeMap<Integer, Integer> numLinesWithLength;
```

第二种声明使用了更长的名称，传达了更多的信息。它还将“width”更改为“length”，因为这个术语更可能让人认为单位是字符而不是像素。请注意，第二条注释不仅记录了每个条目的详细信息，还记录了如果缺少条目意味着什么。

在记录变量时，考虑名词，而不是动词。换句话说，关注变量代表什么，而不是如何操作它。考虑以下注释：

```
/* 追随者变量：指示变量，允许接收者和
    the
 * 用于通信心跳是否已发生的 PeriodicTasks 线程

 * 在追随者的选举超时窗口内收到。
 * 当收到有效的心跳信号时，切换为 TRUE。
 * 当选举超时窗口重置时，切换为 FALSE。*/
private boolean receivedValidHeartbeat;
```

本文档描述了类中多个代码片段如何修改变量。如果注释描述变量的含义，而不是反映代码结构，那么注释将更短且更有用：

```
/* True 表示自上次选举计时器重置后，已收到心跳信号。* 用于接收器和周期性任务线程
 之间的通信。*/

private boolean receivedValidHeartbeat;
```

根据这份文档，很容易推断出，当收到心跳信号时，该变量必须设置为 true，而当选举计时器重置时，该变量必须设置为 false。

13.4 更高级别的注释增强直觉

注释增强代码的第二种方式是通过提供直觉。
这些注释的编写层次高于代码。它们省略了细节，并帮助读者理解代码的整体意图和结构。
这种方法通常用于方法内部的注释和接口注释。例如，考虑以下代码：

```
// 如果存在一个使用与 assignPos 指向的 PKHash 相同的会话的 LOAD
ING readRpc，并且该 readRPC 中的最后一个 PKHash 小于当前分配的 P
KHash
// PKHash，然后我们将分配 PKHash 的操作放入该 readRPC 中。
int readActiveRpcId = RPC_ID_NOT_ASSIGNED;
for (int i = 0; i < NUM_READ_RPC; i++) {
    if (session == readRpc[i].session
          && readRpc[i].status == LOADING
          && readRpc[i].maxPos < assignPos
          && readRpc[i].numHashes < MAX_PKHASHES_PERRPC) {
      readActiveRpcId = i;
      中断；
    }
}
```

这条注释过于底层和详细。一方面，它部分重复了代码：“如果存在 LOADING readRPC”只是重复了测试 readRpc[i].status == LOADING。另一方面，注释没有解释这段代码的总体目的，或者它如何融入包含它的方法中。因此，注释无法帮助读者理解代码。

这是一个更好的注释：

```
// 尝试将当前密钥哈希附加到现有的
// 尚未发送到所需服务器的 RPC。
```

此注释不包含任何细节；相反，它在更高的层次上描述了代码的总体功能。有了这些高级信息，读者几乎可以解释代码中发生的一切：循环必须迭代所有现有的远程过程调用 (RPC)；session test 可能用于查看特定 RPC 是否用于正确的服务器；LOADING test 表明 RPC 可以有多种状态，并且在某些状态下添加更多哈希是不安全的； MAX - PKHASHES_PERRPC test 表明在单个 RPC 中可以发送的哈希数量存在限制。注释中唯一没有解释的是 maxPos test。此外，新注释为

读者判断代码提供了一个基础：它是否完成了将密钥哈希添加到现有 RPC 所需的一切？原始注释没有描述代码的总体意图，因此读者很难判断代码的行为是否正确。

与较低级别的注释相比，编写较高级别的注释更困难，因为您必须以不同的方式思考代码。问问自己：这段代码试图做什么？您可以用来说明代码中所有内容的最简单的事情是什么？这段代码最重要的是什么？

工程师往往非常注重细节。我们喜欢细节并且擅长管理大量细节；这对于成为一名优秀的工程师至关重要。但是，优秀的软件设计师也可以从细节中抽身出来，并在更高的层次上思考系统。这意味着决定系统的哪些方面最重要，并且能够忽略底层细节，仅根据其最基本的特征来思考系统。这是抽象的本质（找到一种简单的方法来思考复杂的实体），这也是您在编写更高级别的注释时必须做的事情。一个好的更高级别的注释表达了一个或几个简单的想法，这些想法提供了一个概念框架，例如“附加到现有的 RPC”。有了这个框架，就很容易看出特定的代码语句如何与总体目标相关。

这是另一个代码示例，它有一个很好的更高级别的注释：

```
if (numProcessedPKHashes < readRpc[i].numHashes) {// 某些
    密钥哈希无法在此请求中查找// （要么是因为它们未存储在服务器
    上，服务器崩溃，或者// 响应消息中没有足够的空间）。


    // 标记未处理的哈希值，以便它们将被
    // 重新分配给新的 RPC。
    for (size_t p = removePos; p < insertPos; p++) {
        if  (activeRpcId[p] == i) {
            if  (numProcessedPKHashes > 0) {
                numProcessedPKHashes--;
            } else {
                if  (p < assignPos)
                   assignPos = p;
                activeRpcId[p] = RPC_ID_NOT_ASSIGNED;
```

```
            }
        }
    }
}
```

This comment does two things. 第二句话提供了代码功能的抽象描述。第一句话则不同：它（以高级术语）解释了代码执行的原因。诸如“我们如何到达这里”之类的注释对于帮助人们理解代码非常有用。例如，在记录一个方法时，描述该方法最有可能被调用的条件（尤其是在该方法仅在不寻常的情况下被调用时）会非常有帮助。

13.5 接口文档

注释最重要的作用之一是定义抽象。回想一下第 4 章，抽象是实体的简化视图，它保留了基本信息，但省略了可以安全忽略的细节。代码不适合描述抽象；它太底层了，并且包含不应在抽象中可见的实现细节。描述抽象的唯一方法是使用注释。如果你想要呈现良好抽象的代码，你必须使用注释来记录这些抽象。

记录抽象概念的第一步是将接口注释与实现注释分开。接口注释提供某人为了使用类或方法需要知道的信息；它们定义了抽象。实现注释描述了类或方法如何在内部工作以实现抽象。将这两种注释分开非常重要，这样接口的用户就不会接触到实现细节。此外，这两种形式最好有所不同。如果接口注释还必须描述实现，那么类或方法就是肤浅的。这意味着编写注释的行为可以提供关于设计质量的线索；第15章将回到这个想法。

类的接口注释提供了对该类提供的抽象的高级描述，例如以下内容：

```
/**
 * 这个类实现了一个简单的服务器端 HTTP 接口：通过使用这个类，应用程序可以接收
 HTTP
```

```
 * 请求，处理它们，并返回响应。这个类的每个实例对应于一个用于接收请求的特定套
 接字。当前的实现是单线程的，并且一次处理一个请求。


 */
public class Http {...}
```

这段注释描述了类的整体功能，没有任何实现细节，甚至没有特定方法的细节。它还描述了类的每个实例代表什么。最后，注释描述了类的局限性（它不支持来自多个线程的并发访问），这对于考虑是否使用它的开发人员来说可能很重要。

方法的接口注释既包括用于抽象的更高级别信息，也包括用于精确的更低级别细节：

- 注释通常以一两句话开始，描述调用者感知到的方法的行为；这是更高层次的抽象。
- 注释必须描述每个参数和返回值（如果有）。这些注释必须非常精确，并且必须描述对参数值的任何约束以及参数之间的依赖关系。
- 如果该方法有任何副作用，则必须在接口注释中记录这些副作用。副作用是指该方法的任何影响系统未来行为的后果，但不是结果的一部分。例如，如果该方法向内部数据结构添加一个值，该值可以通过未来的方法调用来检索，这就是一个副作用；写入文件系统也是一个副作用。

- 一个方法的接口注释必须描述可以从该方法发出的任何异常。

- 如果在调用方法之前必须满足任何前提条件，则必须描述这些前提条件（可能必须首先调用一些其他方法；对于二分查找方法，被搜索的列表必须已排序）。尽量减少前提条件是一个好主意，但任何保留下来的前提条件都必须记录在案。

这是一个从 Buffer 对象复制数据的方法的接口注释：

```
/**
 * 将一系列字节从缓冲区复制到外部位置。
```

```
 *
 * \param offset
 *        要复制的第一个字节在缓冲区中的索引。
 * \param length
 *        要复制的字节数。
 * \param 目标
 *        复制字节的位置：必须有足够的空间容纳至少
 *        length 个字节。
 *
 * \return
 *        返回值是实际复制的字节数，*          如果请求的字节范围超过*
 缓冲区的末尾，则该值可能小于 length。如果请求的范围与*          实际缓冲区之间
 没有重叠，则返回 0。


 */
uint32_t
Buffer::copy(uint32_t offset, uint32_t length, void* dest)
...
```

此注释的语法（例如，\return）遵循 Doxygen 的约定，该程序从 C/C++ 代码中提取注释并将其编译成 Web页面。注释的目标是提供开发人员需要的所有信息，以便调用该方法，包括如何处理特殊情况（请注意此方法如何遵循 第 10 章 的建议，并定义了与范围规范相关的任何错误）。开发人员无需阅读该方法的主体即可调用它，并且接口注释不提供有关该方法如何实现的信息，例如它如何扫描其内部数据结构以查找所需的数据。

为了更详细的例子，让我们考虑一个名为 `IndexLookup` 的类，它是分布式存储系统的一部分。存储系统保存
表的集合，每个表包含许多对象。此外，每个表
可以有一个或多个索引；每个索引提供对表中对象的有效访问，
基于对象的特定字段。例如，一个索引可能
用于根据它们的 `name` 字段查找对象，而另一个索引可能

用于根据它们的 age 字段查找对象。通过这些索引，应用程序可以快速提取具有特定 name 的所有对象，或者具有给定范围内的 age 的所有对象。

IndexLookup 类为执行索引查找提供了一个方便的接口。以下是如何在应用程序中使用它的示例：

```
query = new IndexLookup(table, index, key1, key2);
while (true) {
    object = query.getNext();
    if (object == NULL) {
        break;
    }
    ... 处理对象 ...
}
```

The application first constructs an object of type IndexLookup，提供参数来选择一个表、一个索引以及索引中的一个范围（例如，如果索引基于年龄字段，key1和key2可能被指定为 21 和 65，以选择年龄在这些值之间的所有对象）。然后应用程序重复调用 getNext 方法。每次调用都会返回一个落在所需范围内的对象；一旦所有匹配的对象都被返回，getNext 返回 NULL。由于存储系统是分布式的，因此这个类的实现有些复杂。表中的对象可能分布在多个服务器上，并且每个索引也可能分布在不同的服务器集合中；IndexLookup 类中的代码必须首先与所有相关的索引服务器通信，以收集有关范围内对象的信息，然后它必须与实际存储对象的服务器通信，以便检索它们的值。

现在，让我们考虑一下需要在该类的接口注释中包含哪些信息。对于下面给出的每一条信息，问问自己，开发人员是否需要知道这些信息才能使用该类（我对这些问题的答案在本章末尾）：

1. IndexLookup 类发送到保存索引和对象的服务器的消息格式。

2. 用于确定特定对象是否落在所需范围内的比较函数（比较是使用整数、浮点数还是字符串完成的？）。

3. 用于在服务器上存储索引的数据结构。
4. IndexLookup 是否并发地向不同的服务器发出多个请求。

5. 处理服务器崩溃的机制。

这是 IndexLookup类的接口注释的原始版本；摘录还包括该类定义中的几行，这些行在注释中被引用：

```
/*
* 这个类实现了索引范围查找的客户端框架。* 它管理一个 LookupIndexKeys RPC 和
多个 IndexedRead RPC。客户端只需在其头文件中包含 "IndexLookup.h" 即可使
用 IndexLookup 类。以下配置中可以设置几个参数：



* - 并发 indexedRead RPC 的数量
* - 一个 indexedRead RPC 一次可以容纳的最大 PKHashes 数量
* - 活动 PKHashes 的大小
*
* To use IndexLookup, the client creates an object of this class by
* 提供所有必要的信息。构造 IndexLookup 后，客户端可以调用 getNext() 函
数来移动到下一个可用对象。如果 getNext() 返回 NULL，则表示我们已到达最后一个
对象。客户端可以使用 getKey、getKeyLength、getValue 和 getValueLength 来获取
当前对象的对象数据。

*/
class IndexLookup {
    ...
 private:
    /// 最大并发 indexedRead RPC 数
    static const uint8_t NUM_READ_RPC = 10;
    /// 一个 indexedRead RPC 中可以发送的最大 PKHashes 数

    static const uint32_t MAX_PKHASHES_PERRPC = 256;
    /// activeHashes 一次可以
    /// 容纳的最大 PKHashes 数。
```

```
static const size_t MAX_NUM_PK = (1 << LG_BUFFER_SIZE);
```

在继续阅读之前，看看你是否能找出这条评论的问题所在。以下是我发现的问题：

- 第一段的大部分内容都与实现有关，而不是接口。举个例子，用户不需要知道用于与服务器通信的特定远程过程调用的名称。第一段后半部分提到的配置参数都是私有变量，只与类的维护者相关，与用户无关。所有这些实现信息都应该从注释中省略。

- 这条评论还包括一些显而易见的事情。例如，没有必要告诉用户包含IndexLookup.h：任何编写C++代码的人都能猜到这是必要的。此外，“通过提供所有必要的信息”这句话什么也没说，所以可以省略。

对于这个类来说，一个更短的注释就足够了（而且更好）：

```
/*
 * 此类供客户端应用程序使用，以使用索引进行范围查询
 *。每个实例代表一个范围查询。
 *
 * To start a range query, a client creates an instance of this
 * 类。然后，客户端可以调用 getNext() 来检索所需范围内的对象。* 对于 getNext() 返回的每个对象，调用者可以调用 getKey()、getKeyLength()、getValue() 和

 * getValueLength() 来获取有关该对象的信息。
 */
```

此评论的最后一段并非绝对必要，因为它主要重复了各个方法评论中的信息。然而，在类文档中提供一些示例，说明其方法如何协同工作可能会有所帮助，特别是对于具有非显而易见的使用模式的深层类。请注意，新的评论没有提及来自 getNext 的 NULL 返回值。此评论并非旨在记录每个方法的每个细节；它只是提供高级别的信息，以帮助读者理解这些方法如何协同工作以及何时可能调用每个方法。有关详细信息，读者可以参考各个方法的接口注释。这

评论也没有提到服务器崩溃；这是因为服务器崩溃对于此类用户是不可见的（系统会自动从中恢复）。

危险信号：实现文档污染接口

当接口文档（例如方法的文档）描述了使用被记录事物不需要的实现细节时，就会出现此危险信号。

现在考虑以下代码，它显示了IndexLookup中isReady方法的第一个版本的文档：

```
/**
 * 检查下一个对象是否为 RESULT_READY。此函数在 DCFT 模块中实现，每次执
 行 isReady() 都会尝试取得 небольшие 进展，并且 getNext() 在 while 循环中调用
   isReady()，直到 isReady() 返回 true。


 *
 * isReady() is implemented in a rule-based approach. 我们检查
 * 通过遵循特定顺序的不同规则，如果满足某些规则，则执行某些操作。


 *
 * \return
 * True 表示下一个对象可用。否则，返回 false。


 */
bool IndexLookup::isReady() { ... }
```

再次强调，这份文档的大部分内容，例如对DCFT的引用和整个第二段，都与实现有关，因此不应该放在这里；这是接口注释中最常见的错误之一。一些实现文档是有用的，但它应该放在方法内部，在那里它将与接口文档清晰地分离。此外，文档的第一句话很隐晦（ RESULT_READY 是什么意思？）

并且缺少一些重要的信息。最后，没有必要描述 getNext 的实现。这是一个更好的注释版本：

```
/*
 * 指示索引读取是否已取得足够的进展，以便
 * getNext 立即返回而不阻塞。此外，此
 * 方法为索引读取完成了大部分实际工作，因此必须
 * 调用（直接或通过调用 getNext 间接调用），以便
 * 索引读取取得进展。
 *
 * \return
 * True 表示下一次调用 getNext 不会阻塞* （至少有一个对象可以返回，或者到达


 *          已达到查找次数限制）；false 表示 getNext 可能会阻塞。
 */
```

此版本的注释提供了关于“就绪”含义的更精确信息，并且提供了重要信息，即如果要使索引检索继续进行，则最终必须调用此方法。

13.6 实现注释：内容和原因，而不是方法

实现注释是出现在方法内部的注释，用于帮助读者理解它们在内部是如何工作的。大多数方法都很短且简单，不需要任何实现注释：给定代码和接口注释，很容易弄清楚方法是如何工作的。

实现注释的主要目标是帮助读者理解代码正在做什么（而不是如何做）。一旦读者知道代码想要做什么，通常很容易理解代码是如何工作的。

对于简短的方法，代码只做一件事，这件事已经在其接口注释中描述过了，所以不需要实现注释。较长的方法有几个代码块，这些代码块作为方法总体任务的一部分来做不同的事情。在每个主要代码块之前添加注释，以提供该代码块所做事情的高级（更抽象的）描述。这是一个例子：

```
// 第一阶段：扫描活动的RPC，查看是否有任何已完成。
```

对于 for 循环，在循环前添加注释来描述每次迭代中发生的事情会很有帮助：

// 以下循环的每次迭代都从请求消息中提取一个请求，增加相应的对象，并且// 将响应附加到响应消息。

请注意，此注释如何在更抽象和直观的层面上描述循环；它没有详细说明如何从请求消息中提取请求或如何增加对象。只有对于较长或更复杂的循环才需要循环注释，因为循环的作用可能不明显；许多循环都很短且简单，以至于它们的行为已经很明显。

除了描述代码正在做什么之外，实现注释对于解释原因也很有用。如果代码中存在一些棘手的地方，从阅读代码中无法明显看出，则应记录它们。例如，如果一个错误修复需要添加一些目的不完全明显的代码，则添加一个注释来描述为什么需要该代码。对于有良好编写的错误报告描述问题的错误修复，注释可以引用错误跟踪数据库中的问题，而不是重复其所有细节（“修复 RAM-436，与 Linux 中的设备驱动程序崩溃有关

2.4.x”）。开发人员可以在错误数据库中查找更多详细信息（这是避免注释中重复的一个例子，这将在第 16 章中讨论）。

对于较长的方法，为一些最重要的局部变量编写注释可能会有所帮助。但是，如果大多数局部变量都有好的名称，则不需要文档。如果一个变量的所有用途在彼此的几行代码中都是可见的，那么通常很容易理解该变量的用途，而无需注释。在这种情况下，可以让读者阅读代码来弄清楚变量的含义。但是，如果变量在大量的代码中使用，那么您应该考虑添加一个注释来描述该变量。在记录变量时，重点关注变量代表什么，而不是它在代码中是如何操作的。

13.7 跨模块设计决策

在一个完美的世界里，每个重要的设计决策都应该封装在一个类中。不幸的是，实际系统不可避免地会遇到影响多个类的设计决策。例如，网络协议的设计将影响发送者和接收者，而这些可能

在不同的地方实现。跨模块决策通常是复杂而微妙的，并且它们导致了许多错误，因此对它们的良好文档至关重要。

跨模块文档的最大挑战是找到一个地方来放置它，以便开发者能够自然地发现它。有时，放置此类文档有一个明显的中心位置。例如，RAMCloud存储系统定义了一个Status值，该值由每个请求返回，以指示成功或失败。为新的错误条件添加Status需要修改许多不同的文件（一个文件将Status值映射到异常，另一个文件为每个Status提供人类可读的消息，等等）。

幸运的是，当开发者添加新的状态值时，有一个明显的地方是他们必须去的，那就是Status枚举的声明。我们利用了这一点，在该枚举中添加注释，以识别所有其他必须修改的地方：

```
typedef enum Status {
    STATUS_OK = 0,
    STATUS_UNKNOWN_TABLET                = 1,
    STATUS_WRONG_VERSION                 = 2,
    ...
    STATUS_INDEX_DOESNT_EXIST            = 29,
    STATUS_INVALID_PARAMETER             = 30,
    STATUS_MAX_VALUE                     = 30,

    // 注意：如果您添加新的状态值，则必须进行以下
    // 附加更新：
    // (1) 修改 STATUS_MAX_VALUE，使其值等于
    // 定义的最大状态值，并确保其定义
    // 是列表中的最后一个。STATUS_MAX_VALUE 用于
    // 主要用于测试。
    // (2) 在 Status.cc 中的表“messages”和“symbols”中添加新条目。

    // (3) 向 ClientException.h 添加一个新的异常类// (4) 向 ClientExcep
    tion::throwException 添加一个新的 "case"，以将状态值映射到// 状态特定的 ClientEx
    ception 子类。
```

```
    // (5) 在 Java 绑定中，为异常添加一个静态类到 ClientException.java

    // (6) 在 ClientException.java 中为异常状态添加一个 case 来抛出异常

    // (7) 将异常添加到 Status.java 中的 Status 枚举中，确保状态位于与其状态代码对应的
    正确位置。

}
```

New status values will be added at the end of the existing list, so the comments 也放在末尾，因为它们最有可能被看到。

不幸的是，在许多情况下，没有一个明显的中心位置来放置跨模块文档。RAMCloud 存储系统中的一个例子是处理僵尸服务器的代码，这些服务器是系统认为已经崩溃但实际上仍在运行的服务器。中和僵尸服务器需要在几个不同的模块中使用代码，并且这些代码相互依赖。这些代码中没有一个是放置文档的明显中心位置。一种可能性是在每个依赖它的位置复制部分文档。然而，这很尴尬，并且很难在系统发展时保持此类文档的更新。或者，文档可以位于需要它的某个位置，但在这种情况下，开发人员不太可能看到文档或知道在哪里查找它。

我最近一直在尝试一种方法，其中跨模块问题记录在一个名为 design Notes 的中心文件中。该文件被分成清晰标记的部分，每个主要主题一个。例如，这是该文件中的摘录：

```
...
僵尸
-------
僵尸是被集群其余部分视为已死的服务器；存储在服务器上的任何数据都已恢复，并
将由其他服务器管理。但是，如果僵尸实际上没有死亡（例如，它只是与其他服务器断开
连接一段时间），则可能会出现两种形式的不一致：


* 一旦替换服务器接管，僵尸服务器不得处理读取请求；否则，它可能会返回过时的不一致数据
```

反映被替换服务器接受的写入。

* 一旦替换服务器在恢复期间开始重放其日志，僵尸服务器就不得接受写入请求；如果它这样做了，这些写入可能会丢失（新值可能不会存储在替换服务器上，因此不会被读取返回）。

RAMCloud 使用两种技术来消除僵尸进程。首先，

...

然后，在任何与这些问题相关的代码中，都会有一段简短的注释指向 designNotes 文件：

```
// 参见设计说明中的“僵尸”
```

使用这种方法，文档只有一个副本，开发人员在需要时可以相对容易地找到它。然而，这种方法的缺点是文档不在任何依赖它的代码附近，因此随着系统的发展，可能难以保持文档的更新。

13.8 结论

注释的目标是确保系统的结构和行为对于读者来说是显而易见的，以便他们能够快速找到他们需要的信息，并有信心地对系统进行修改，确保它们能够工作。其中一些信息可以用代码来表示，这样对于读者来说已经很明显了，但是有大量的信息无法轻易地从代码中推断出来。注释填补了这些信息。

当遵循注释应该描述代码中不明显的内容的规则时，“明显”是从第一次阅读你代码的人（而不是你）的角度来看的。在编写注释时，试着把自己放在读者的心态中，问问自己他或她需要知道的关键是什么。如果你的代码正在接受审查，并且审查人员告诉你某些事情并不明显，不要与他们争论；如果读者认为它不明显，那么它就不明显。与其争论，不如试着理解他们觉得困惑的地方，看看你是否可以用更好的注释或更好的代码来澄清这一点。

13.9 第 13.5 节问题的答案

为了使用IndexLookup类，开发人员是否需要了解以下每条信息？

1. IndexLookup类发送给保存索引和对象的服务器的消息格式。 否：这是一个应该隐藏在类中的实现细节。

2. 用于确定特定对象是否在所需范围内的比较函数（比较是使用整数、浮点数还是字符串完成的？）。 是：该类的用户需要了解此信息。

3. 用于在服务器上存储索引的数据结构。 否：此信息应该封装在服务器上；甚至连IndexLookup的实现
   IndexLookup都不需要知道这一点。
4. IndexLookup是否并发地向不同的服务器发出多个请求。 可能：如果 IndexLookup使用特殊技术来提高性能，那么文档应该提供一些关于此的高级信息，因为用户可能关心性能。

5. 处理服务器崩溃的机制。 否：RAMCloud会自动从服务器崩溃中恢复，因此崩溃对于应用程序级别的软件是不可见的；因此，没有必要在IndexLookup的接口文档中提及崩溃。如果崩溃反映到应用程序，那么接口文档需要描述它们如何表现出来（但不是崩溃恢复如何工作的细节）。

第14章

选择名称

为变量、方法和其他实体选择名称是软件设计中最被低估的方面之一。好的名称是一种文档形式：
它们使代码更容易理解。它们减少了对其他文档的需求，并使其更容易检测错误。相反，糟糕的名称选择会增加代码的复杂性，并产生可能导致错误的歧义和误解。名称选择是复杂性是递增的原则的一个例子。为一个特定的变量选择一个平庸的名称，而不是最好的名称，可能不会对系统的整体复杂性产生太大影响。然而，软件系统有成千上万的变量；为所有这些变量选择好的名称将对复杂性和可管理性产生重大影响。

14.1 示例：糟糕的命名导致错误

有时候，即使是一个命名不佳的变量也会产生严重的后果。
我修复过的最具挑战性的错误，就是因为一个糟糕的名称选择。在 20 世纪 80 年代末和 90 年代初，我的研究生和我创建了一个名为 Sprite 的分布式操作系统。在某个时候，我们注意到文件偶尔会丢失数据：一个数据块突然变成全零，即使该文件没有被用户修改。这个问题并不经常发生，所以很难追踪。一些研究生试图找到这个错误，但他们无法取得进展，最终放弃了。但是，我认为任何未解决的错误都是一种无法容忍的个人侮辱，所以我决定追踪它。

我花了六个月的时间，但最终还是找到了并修复了这个错误。这个问题实际上很简单（就像大多数错误一样，一旦你弄清楚了）。文件系统代码对两个不同的目的使用了变量名 block 。在某些情况下，block 指的是磁盘上的物理块号；在其他情况下，

block 指的是文件中的逻辑块号。不幸的是，在代码中的某一点，有一个 block 变量包含一个逻辑块号，但它被意外地用在了需要物理块号的上下文中；结果，磁盘上一个不相关的块被零覆盖。

在追踪这个错误的过程中，包括我在内的几个人都阅读了错误的代码，但我们从未注意到这个问题。当我们看到变量 block被用作物理块号时，我们下意识地认为它实际上保存的是一个物理块号。我花了很长时间进行检测，最终表明损坏 must 发生在特定的语句中，我才能克服名称造成的思维障碍，并检查其值的来源。如果对不同类型的块使用了不同的变量名，例如 fileBlock 和 diskBlock，那么这个错误就不太可能发生；程序员就会知道 fileBlock 不能在这种情况下使用。

不幸的是，大多数开发人员不会花太多时间思考命名。
他们倾向于使用脑海中出现的第一个名称，只要它与它所命名的事物相当接近即可。例如，block 与磁盘上的物理块和文件中的逻辑块都非常接近；当然，这并不是一个糟糕的名称。即便如此，它还是导致了大量的时间花费来追踪一个微妙的错误。因此，你不应该满足于那些只是“相当接近”的名称。多花一点时间来选择好的名称，这些名称是精确、明确和直观的。额外的关注会很快得到回报，而且随着时间的推移，你将学会快速选择好的名称。

14.2 创建一个图像

在选择名称时，目标是在读者的脑海中创建一个关于被命名事物本质的图像。一个好的名称传达了关于底层实体是什么的大量信息，以及同样重要的是，它不是什么。在考虑一个特定的名称时，问问自己：“如果有人在孤立地看到这个名称，而没有看到它的声明、它的文档或任何使用该名称的代码，他们能多大程度上猜到该名称指的是什么？有没有其他名称可以描绘出更清晰的画面？”当然，你可以放在一个名称中的信息量是有限的；如果名称包含超过两三个单词，就会变得笨拙。因此，挑战在于找到几个能够捕捉实体最重要方面的词语。

名称是一种抽象形式：它们提供了一种简化的方式来思考更复杂的底层实体。与其他抽象形式一样，最好的名称是那些将注意力集中在底层实体最重要的部分，同时省略不太重要的细节的名称。

14.3 名称应精确

好的名称具有两个属性：精确性和一致性。让我们从精确性开始。名称最常见的问题是它们过于通用或模糊；因此，读者很难分辨该名称指的是什么；读者可能会认为该名称指的是与现实不同的事物，如

block bug 以上。考虑以下方法声明：

```
/**
 * 返回此对象正在管理的索引小程序的总数。
 */
int IndexletManager::getCount() {...}
```

术语“count”过于通用：什么的计数？如果有人看到此方法的调用，他们不太可能知道它做什么，除非他们阅读它的文档。更精确的名称，如 getActiveIndexlets 或 numIndexlets会更好：使用这些名称之一，读者可能能够猜出该方法返回的内容，而无需查看其文档。

以下是一些不够精确的名称的其他示例，这些示例取自各种学生项目：

- 一个构建 GUI 文本编辑器的项目使用名称 x 和 y 来指代文件中字符的位置。这些名称过于通用。它们可能意味着很多事情；例如，它们也可能代表屏幕上字符的坐标（以像素为单位）。有人看到孤立的名称 x 不太可能认为它指的是文本行中字符的位置。如果代码使用诸如 charIndex 和 lineIndex之类的名称，代码会更清晰，这些名称反映了代码实现的特定抽象。

- 另一个编辑器项目包含以下代码：

  ```
  // 闪烁状态：光标可见时为 true。
  private boolean blinkStatus = true;
  ```

  名称 `blinkStatus` 没有传达足够的信息。单词“status”（状态）对于布尔值来说太模糊了：它没有给出关于 true

或 false 值意味着什么。单词“blink”（闪烁）也很模糊，因为它没有表明什么在闪烁。以下替代方案更好：

```
// 控制光标闪烁：true 表示光标可见，
// false 表示光标不显示。
private boolean cursorVisible = true;
```

名称 `cursorVisible` 传达了更多信息；例如，它允许读者猜测 true 值意味着什么（作为一般规则，布尔变量的名称应该始终是谓词）。单词“blink”不再出现在名称中，因此如果读者想知道为什么光标不是总是可见的，他们将不得不查阅文档；此信息不太重要。

- 一个实现共识协议的项目包含以下代码：

```
// 代表服务器尚未（在当前选举任期内）投票给
// 任何人的值。
private static final String VOTED_FOR_SENTINEL_VALUE = "null";
```

此值的名称表明它是特殊的，但没有说明特殊含义是什么。更具体的名称，例如 NOT_YET_VOTED 会更好。

- 一个名为 result 的变量在一个没有返回值的方法中使用。这个名称有多个问题。首先，它给人一种误导性的印象，即它将是该方法的返回值。其次，它基本上没有提供关于它实际包含的内容的信息，除了它是一些计算值。该名称应该提供关于结果实际是什么的信息，例如 mergedLine 或 totalChars。在确实有返回值的方法中，使用名称 result 是合理的。这个名字仍然有点通用，但读者可以查看方法文档来了解它的含义，并且知道该值最终将成为返回值是有帮助的。

危险信号：模糊的名称

如果一个变量或方法名称足够宽泛，可以指代许多不同的事物，那么它就不能向开发者传达太多信息，并且底层实体更有可能被误用。

像所有规则一样，关于选择精确名称的规则也有一些例外。例如，可以使用像 `i` 和 `j` 这样的通用名称作为循环迭代变量，只要循环只跨越几行代码。如果你能看到变量的整个使用范围，那么变量的含义很可能从代码中显而易见，所以你不需要一个长名称。例如，考虑以下代码：

```
for  (i = 0; i < numLines; i++) {
    ...
}
```

It's clear from this code that `i` is 被用来迭代某个实体中的每一行。如果循环变得很长，以至于你不能一次看到它，或者如果迭代变量的含义很难从代码中弄清楚，那么一个更具描述性的名称是合适的。

名称也可能过于具体，例如在删除文本范围的方法的声明中：

```
void delete(Range selection) {...}
```

参数名称 selection 过于具体，因为它表明被删除的文本始终在用户界面中被选中。但是，可以在任何文本范围上调用此方法，无论是否选中。因此，参数名称应该更通用，例如 range。

如果你发现很难为一个特定的变量想出一个精确、直观且不太长的名字，这是一个危险信号。这表明该变量可能没有明确的定义或目的。当这种情况发生时，考虑其他的分解方式。例如，也许你试图用一个变量来表示几个事物；如果是这样，将表示分离成多个变量可能会为每个变量带来更简单的定义。选择好名字的过程可以通过识别弱点来改进你的设计。

危险信号：难以选择名称

如果很难为一个变量或方法找到一个简单的名称，以清晰地描绘出底层对象，那么这暗示着底层对象可能没有一个清晰的设计。

14.4 始终如一地使用名称

好名字的第二个重要特性是一致性。在任何程序中，都有一些变量会被反复使用。例如，文件系统会重复操作块号。对于这些常见的用法，为该目的选择一个名称，并在任何地方都使用相同的名称。例如，文件系统可能总是使用 fileBlock 来保存文件中块的索引。一致的命名可以减少认知负荷，就像重用一个通用类一样：一旦读者在一个上下文中看到了这个名称，他们就可以重用他们的知识，并在不同的上下文中看到这个名称时立即做出假设。

一致性有三个要求：首先，对于给定的目的，始终使用通用名称；其次，除了给定的目的之外，永远不要使用通用名称；第三，确保该目的足够狭窄，以便所有具有该名称的变量都具有相同的行为。在本章开头的那个文件系统错误中，第三个要求被违反了。文件系统对具有两种不同行为（文件块和磁盘块）的变量使用了 block ；这导致了对变量含义的错误假设，进而导致了一个错误。

有时你需要多个变量来引用同一类事物。例如，一个复制文件数据的方法将需要两个块号，一个用于源，一个用于目标。当这种情况发生时，对每个变量使用通用名称，但添加一个区分前缀，例如

srcFileBlock 和 dstFileBlock。

循环是另一个一致命名可以提供帮助的领域。如果你对循环变量使用诸如 i 和 j 之类的名称，始终在最外层循环中使用 i ，在嵌套循环中使用 j 。这允许读者在看到给定名称时，立即（安全地）对代码中发生的事情做出假设。

14.5 一种不同的观点：Go 风格指南

并非所有人都赞同我对命名的看法。Go语言的一些开发者认为，名称应该非常短，通常只有一个字符。在一次关于Go语言名称选择的演讲中，Andrew Gerrand指出“长名称会模糊代码的作用。”1他展示了这段代码示例，其中使用了单字母变量名：

```
func RuneCount(b []byte) int {
```

```
    i, n := 0, 0
    for i < len(b) {
        if b[i] < RuneSelf {
            i++
        } else {
            _, size := DecodeRune(b[i:])
            i += size
        }
        n++
    }
    return n
}
```

并认为它比以下使用较长名称的版本更具可读性：

```
func RuneCount(buffer []byte) int {
    index, count := 0, 0
    for index < len(buffer) {
        if buffer[index] < RuneSelf {
            index++
        } else {
            _, size := DecodeRune(buffer[index:])
            index += size
        }
        count++
    }
    返回计数
}
```

就我个人而言，我不觉得第二个版本比第一个版本更难读。如果说有什么区别的话， count 这个名字比 n更能稍微提示变量的行为。对于第一个版本，我最终通读代码，试图弄清楚n 是什么意思，而在第二个版本中我没有这种感觉。但是，如果 n 在整个系统中一致地用于指代计数（而不是其他任何东西），那么短名称对于其他开发人员来说可能很清楚。

Go 文化鼓励对多个不同的事物使用相同的短名称： `ch` 用于字符或通道， `d` 用于数据、差异或距离等等。对我来说，像这样的模棱两可的名称很可能导致混淆和错误，就像 `block` 示例中一样。

总的来说，我认为可读性必须由读者而不是作者来决定。如果你编写的代码使用短变量名，并且阅读它的人觉得很容易理解，那就没问题。如果你开始收到关于你的代码晦涩难懂的抱怨，那么你应该考虑使用更长的名称（在网上搜索“go 语言短名称”会发现一些这样的抱怨）。同样，如果我开始收到关于长变量名使我的代码更难阅读的抱怨，那么我会考虑使用更短的名称。

Gerrand 提出了一个我同意的观点：“名称的声明和使用之间的距离越大，名称应该越长。”前面关于使用名为 `i` 和 `j` 的循环变量的讨论就是这个规则的一个例子。

14.6 结论

精心选择的名称有助于使代码更清晰；当有人第一次遇到变量时，他们未经深思熟虑而对其行为做出的第一猜测将是正确的。选择好的名称是 第 3 章中讨论的投资心态的一个例子：如果你预先花一点额外的时间来选择好的名称，那么将来你更容易处理代码。

此外，你不太可能引入错误。培养命名的技巧也是一种投资。当你第一次决定不再满足于平庸的名称时，你可能会发现想出好的名称令人沮丧且耗时。但是，随着你获得更多经验，你会发现它变得更容易；最终，你会达到几乎不需要额外时间来选择好名称的程度，因此你几乎可以免费获得好处。

[1] https://talks.golang.org/2014/names.slide#1

第 15 章

先写评论

(将注释作为设计过程的一部分)

许多开发人员会将编写文档推迟到开发过程的最后，即在编码和单元测试完成后。这是产生低质量文档的最可靠方法之一。编写注释的最佳时间是在流程的开始，即在编写代码时。首先编写注释使文档成为设计过程的一部分。这不仅可以生成更好的文档，而且可以生成更好的设计，并且使编写文档的过程更加愉快。

15.1 延迟的注释是不好的注释

几乎我见过的每个开发人员都会推迟编写注释。当被问到为什么不早点编写文档时，他们说代码仍在更改。他们说，如果他们提前编写文档，则必须在代码更改时重写它；最好等到代码稳定下来。但是，我怀疑还有另一个原因，那就是他们将文档视为繁琐的工作；因此，他们会尽可能地推迟它。

不幸的是，这种方法有几个负面后果。首先，延迟编写文档通常意味着根本没有编写文档。一旦开始延迟，很容易再延迟一点；毕竟，几周后代码会更加稳定。到代码无可辩驳地稳定下来时，代码已经很多了，这意味着编写文档的任务变得巨大，甚至更没有吸引力。永远没有方便的时间停下来几天并填写所有缺失的注释，并且很容易合理化，即对项目最好的事情是继续前进并修复错误或编写下一个新功能。

这将创建更多没有文档的代码。

即使您确实有自律性返回并编写注释（并且不要自欺欺人：您可能没有），这些注释也不会很好。
在这个过程中的这个时候，您已经在精神上退出了。在您看来，这段代码已经完成；您渴望继续进行下一个项目。您知道编写注释是正确的事情，但它并不有趣。您只想尽快完成它。因此，您快速浏览代码，添加足够的注释以使其看起来体面。到现在为止，自从您设计代码以来已经有一段时间了，因此您对设计过程的记忆变得模糊。您在编写注释时查看代码，因此注释会重复代码。即使您尝试重建代码中不明显的设计思想，也会有一些您不记得的事情。因此，注释缺少一些它们应该描述的最重要的事情。

15.2 首先编写注释

我使用一种不同的方法来编写注释，即在最开始就编写注释：

- 对于一个新的类，我首先编写类接口注释。
- 接下来，我为最重要的公共方法编写接口注释和签名，但方法体留空。
- 我对这些注释进行一些迭代，直到基本结构感觉合适为止。
- 此时，我为类中最重要的类实例变量编写声明和注释。

- 最后，我填写方法体，根据需要添加实现注释。

- 在编写方法体时，我通常会发现需要额外的方法和实例变量。对于每个新方法，我在方法体之前编写接口注释；对于实例变量，我在编写变量声明的同时填写注释。

当代码完成时，注释也完成了。永远不会有未编写的注释积压。

“注释优先”的方法有三个好处。首先，它会产生更好的注释。如果您在设计类时编写注释，那么关键的设计问题将在您的脑海中清晰地呈现，因此很容易记录它们。最好在每个方法的方法体之前编写接口注释，这样您就可以专注于方法的抽象和接口，而不会被其实现分心。在编码和测试过程中，您会注意到并修复

注释的问题。因此，注释会在开发过程中得到改进。

15.3 注释是一种设计工具

第二，也是最重要的好处，就是在开始时写注释可以改进系统设计。注释是唯一能够完整捕捉抽象的方法，而好的抽象是好的系统设计的基础。如果在开始时编写描述抽象的注释，您可以在编写实现代码之前审查和调整它们。

要写出好的注释，你必须识别变量或代码片段的本质：这个东西最重要的方面是什么？在设计过程的早期就做到这一点非常重要；否则你只是在胡乱编写代码。

注释就像煤矿里的金丝雀，可以预警复杂性。如果一个方法或变量需要很长的注释，这是一个危险信号，表明你没有一个好的抽象。请记住第4章的内容，类应该是深入的：最好的类具有非常简单的接口，但实现了强大的功能。判断接口复杂性的最佳方法是从描述它的注释中判断。如果一个方法的接口注释提供了使用该方法所需的所有信息，并且简短明了，则表明该方法具有简单的接口。相反，如果没有冗长而复杂的注释就无法完整地描述一个方法，那么该方法就具有复杂的接口。你可以将方法的接口注释与实现进行比较，以了解该方法的深度：如果接口注释必须描述实现的所有主要特征，那么该方法是浅显的。同样的想法也适用于变量：如果需要很长的注释才能完整地描述一个变量，这是一个危险信号，表明你可能没有选择正确的变量分解。总的来说，编写注释的行为可以让你尽早评估你的设计决策，这样你就可以发现并解决问题。

危险信号：难以描述

描述方法或变量的注释应该简单而完整。如果你发现很难写出这样的注释，这表明你所描述的事物的设计可能存在问题。

当然，注释只有在完整和清晰的情况下才能很好地指示复杂性。如果你编写的方法接口注释没有提供调用该方法所需的所有信息，或者晦涩难懂，那么该注释就不能很好地衡量方法的深度。

15.4 尽早编写注释是件有趣的事

尽早编写注释的第三个也是最后一个好处是，它使编写注释变得更有趣。对我来说，编程中最令人愉快的部分之一是新类的早期设计阶段，在这个阶段，我正在充实类的抽象和结构。我的大部分注释都是在这个阶段编写的，这些注释是我记录和测试我的设计决策质量的方式。我正在寻找可以用最少的文字完整而清晰地表达的设计。注释越简单，我对我的设计感觉越好，所以找到简单的注释是一种自豪感。如果你正在进行战略性编程，你的主要目标是出色的设计，而不仅仅是编写可用的代码，那么编写注释应该是有趣的，因为这就是你识别最佳设计的方式。

15.5 尽早编写注释是否代价高昂？

现在让我们重新审视一下推迟编写注释的理由，即它可以避免随着代码的演进而修改注释的成本。一个简单的粗略计算将表明这并不能节省多少。首先，估计一下你花在输入代码和注释上的总开发时间，包括修改代码和注释的时间；这不太可能超过总开发时间的10%。即使你总代码行数的一半是注释，编写注释可能也不会超过你总开发时间的5%。将注释推迟到最后只会节省其中的一小部分，这并不多。

首先编写注释意味着在开始编写代码之前，抽象会更加稳定。这可能会节省编码时间。
相反，如果你先编写代码，那么抽象可能会随着你的编码而演变，这将比先写注释的方法需要更多的代码修改。当你考虑到所有这些因素时，先写注释可能总体上会更快。

15.6 结论

如果你从未尝试过先写注释，不妨试一试。坚持足够长的时间来适应它。然后思考它如何影响你的注释质量、你的设计质量以及你对软件开发的整体享受。在你尝试了一段时间后，请告诉我你的经验是否与我的相符，以及为什么相符或不相符。

第16章

修改现有代码

第1章描述了软件开发是如何迭代和增量的。一个大型软件系统通过一系列的演化阶段发展，每个阶段都增加新的功能并修改现有的模块。这意味着一个系统的设计在不断演变。一开始就构思出一个系统的正确设计是不可能的；一个成熟系统的设计更多地是由系统演化过程中所做的改变决定的，而不是由任何最初的构想决定的。前面的章节描述了如何在最初的设计和实现过程中挤出复杂性；本章讨论了如何在系统演化过程中防止复杂性蔓延。

16.1 保持战略性

第3章介绍了战术编程和战略编程之间的区别：在战术编程中，主要目标是快速地让某些东西工作起来，即使这会导致额外的复杂性；在战略编程中，最重要的目标是产生一个伟大的系统设计。战术方法很快就会导致混乱的系统设计。如果你想要一个易于维护和增强的系统，那么“工作”并不是一个足够高的标准；你必须优先考虑设计并进行战略性思考。这个想法也适用于你修改现有代码的时候。

不幸的是，当开发人员进入现有代码进行更改，例如修复错误或添加新功能时，他们通常不会进行战略性思考。一种典型的心态是“我可以做的最小的更改是什么，才能满足我的需求？”有时开发人员会以此为理由，因为他们不熟悉正在修改的代码；他们担心较大的更改会带来引入新错误的更大风险。然而，这会导致战术编程。这些最小的更改中的每一个都会引入一些特殊情况、依赖关系或其他形式的复杂性。因此，系统设计会变得稍微糟糕一些，并且问题会随着系统演化的每一步而累积。

如果你想保持一个系统的清晰设计，在修改现有代码时，你必须采取战略性的方法。理想情况下，当你完成每次更改后，系统将具有这样的结构：如果你从一开始就考虑到该更改来设计它，那么它就应该具有这样的结构。为了实现这个目标，你必须抵制快速修复的诱惑。相反，考虑一下当前系统设计是否仍然是最好的，考虑到所需的更改。如果不是，重构系统，以便最终得到最佳的设计。通过这种方法，系统设计会随着每次修改而改进。

这也是第15页介绍的投资心态的一个例子：如果你投入一点额外的时间来重构和改进系统设计，你最终会得到一个更清晰的系统。这将加快开发速度，你将收回你在重构中投入的精力。即使你的特定更改不需要重构，你也应该留意在代码中可以修复的设计缺陷。每当你修改任何代码时，都要尝试找到一种方法，在这个过程中至少稍微改进一下系统设计。如果你没有让设计变得更好，你可能正在让它变得更糟。

正如第3章所讨论的，投资心态有时与商业软件开发的现实相冲突。如果以“正确的方式”重构系统需要三个月，而快速而肮脏的修复只需要两个小时，你可能不得不采取快速而肮脏的方法，特别是如果你正在赶时间。或者，如果重构系统会产生影响许多其他人和团队的不兼容性，那么重构可能不切实际。

尽管如此，你应该尽可能地抵制这些妥协。问问自己“考虑到我目前的限制，这是我能为创建一个干净的系统设计所做的最好的事情吗？”也许有一种替代方法，它几乎和3个月的重构一样干净，但可以在几天内完成？

或者，如果你现在负担不起大规模的重构，让你的老板分配时间让你在当前截止日期之后再回来处理它。每个开发组织都应该计划将其总工作量的一小部分用于清理和重构；从长远来看，这项工作会得到回报。

16.2 维护注释：将注释放在代码附近

当你更改现有代码时，这些更改很有可能会使一些现有注释失效。当你修改代码时，很容易忘记更新注释，这会导致注释不再准确。

不准确的注释会让读者感到沮丧，如果有很多不准确的注释，读者就会开始不信任所有的注释。幸运的是，只要稍加约束和遵循一些指导原则，就可以在不花费大量精力的情况下保持注释的最新状态。本节和接下来的几节将提出一些具体的技术。

确保注释得到更新的最好方法是将它们放在靠近它们所描述的代码的位置，这样开发人员在更改代码时就会看到它们。注释离其关联代码越远，它就越不可能被正确更新。例如，方法接口注释的最佳位置是在代码文件中，紧挨着方法的主体。对方法的任何更改都将涉及此代码，因此开发人员很可能会看到接口注释并在需要时更新它们。

对于像C和C++这样具有单独的代码和头文件的语言，另一种方法是将接口注释放在.h文件中方法声明的旁边。然而，这离代码很远；开发人员在修改方法的主体时不会看到这些注释，并且需要额外的工作来打开不同的文件并找到接口注释来更新它们。有些人可能会争辩说，接口注释应该放在头文件中，以便用户可以在不必查看代码文件的情况下学习如何使用抽象。但是，用户不应该需要阅读代码或头文件；他们应该从Doxygen或Javadoc等工具编译的文档中获取信息。

此外，许多集成开发环境（IDE）会提取文档并将其呈现给用户，例如在键入方法名称时显示方法的文档。
有了这些工具，文档应该放在对开发人员处理代码最方便的地方。

在编写实现注释时，不要将整个方法的注释都放在方法的顶部。将它们分散开来，将每个注释向下推到包含注释所引用的所有代码的最小范围。例如，如果一个方法有三个主要阶段，不要在方法的顶部写一个注释，详细描述所有阶段。

相反，为每个阶段编写单独的注释，并将该注释放在该阶段的第一行代码之上。另一方面，它也可能是有帮助的

在方法实现的顶部添加一个注释，描述整体策略，像这样：

```
// 我们分三个阶段进行：
// 第一阶段：寻找可行的候选者
// 第二阶段：为每个候选者分配一个分数
// 第三阶段：选择最佳候选者并将其移除
```

更多细节可以在每个阶段的代码上方进行记录。

一般来说，注释离它所描述的代码越远，它就应该越抽象（这降低了注释因代码更改而失效的可能性）。

16.3 注释属于代码，而不是提交日志

修改代码时一个常见的错误是，将关于更改的详细信息放在源代码库的提交消息中，但不在代码中记录它。虽然将来可以通过扫描存储库的日志来浏览提交消息，但需要信息的开发人员不太可能想到扫描存储库日志。即使他们扫描日志，找到正确的日志消息也会很乏味。

在编写提交信息时，问问自己开发者将来是否需要使用这些信息。如果是，那么将这些信息记录在代码中。一个例子是描述一个细微问题的提交信息，这个问题促使了代码的更改。如果这没有记录在代码中，那么开发者稍后可能会在没有意识到他们已经重新创建了一个错误的情况下撤销更改。如果你也想在提交信息中包含一份此信息的副本，那也可以，但最重要的是将其放入代码中。

这说明了将文档放在开发者最有可能看到的地方的原则；提交日志很少是那个地方。

16.4 维护注释：避免重复

保持注释更新的第二种技术是避免重复。如果文档被复制，开发人员就更难找到并更新所有相关的副本。相反，尽量只记录每个设计决策一次。如果代码中有多个地方受到特定决策的影响，不要在每个地方重复文档。相反，找到放置文档最明显的单个位置。例如，假设有

与变量相关的棘手行为，会影响使用该变量的几个不同地方。您可以在变量声明旁边的注释中记录该行为。如果开发人员在理解使用该变量的代码时遇到问题，这里是他们可能会检查的自然位置。

如果对于某个特定的文档，没有一个“显而易见”的单一位置可以让开发者找到它，那么就创建一个 designNotes 文件，如第 13.7 节所述。或者，选择现有位置中最好的一个，并将文档放在那里。此外，在其他位置添加简短的注释，指向中心位置：“有关下面代码的解释，请参见 xyz 中的注释。”如果由于主注释被移动或删除而导致引用过时，这种不一致性将是不言而喻的，因为开发者将无法在指定位置找到注释；他们可以使用版本控制历史来找出注释发生了什么，然后更新引用。

相反，如果文档被复制，并且某些副本没有得到更新，那么开发者将无法得知他们正在使用过时的信息。

不要在一个模块中重复记录另一个模块的设计决策。例如，不要在方法调用前添加注释来解释被调用方法中发生的事情。如果读者想知道，他们应该查看该方法的接口注释。好的开发工具通常会自动提供这些信息，例如，如果您选择方法的名称或将鼠标悬停在其上，则会显示该方法的接口注释。尽量让开发人员容易找到合适的文档，但不要通过重复文档来实现。

如果信息已经记录在程序之外的某个地方，不要在程序内部重复该文档；只需引用外部文档即可。例如，如果您编写一个实现 HTTP 协议的类，则无需在代码中描述 HTTP 协议。Web 上已经有大量关于此文档的来源；只需在您的代码中添加一个简短的注释，其中包含指向其中一个来源的 URL。另一个例子是用户手册中已经记录的功能。假设您正在编写一个程序，该程序实现了一系列命令，其中一个方法负责实现每个命令。如果有一个用户手册描述了这些命令，则无需在代码中复制此信息。相反，在每个命令方法的接口注释中包含如下简短的注释：

// 实现 Foo 命令；详情请参阅用户手册。

重要的是，读者能够轻松找到理解你的代码所需的所有文档，但这并不意味着你必须编写所有这些文档。

16.5 维护注释：检查差异

确保文档保持更新的一个好方法是在提交更改到修订控制系统之前花几分钟时间扫描该提交的所有更改；确保每个更改都已正确反映在文档中。这些提交前扫描还将检测到其他几个问题，例如意外地将调试代码留在系统中或未能修复TODO项。

16.6 更高级别的注释更容易维护

关于维护文档的最后一个想法：如果注释比代码更高级别和更抽象，则更容易维护。这些注释不反映代码的细节，因此它们不会受到细微代码更改的影响；只有总体行为的更改才会影响这些注释。当然，正如第13章中所讨论的，有些注释确实需要详细而精确。但总的来说，最有用的注释（它们不仅仅是重复代码）也最容易维护。

第17章

一致性

一致性是降低系统复杂性并使其行为更明显的强大工具。如果一个系统是一致的，这意味着相似的事情以相似的方式完成，而不同的事情以不同的方式完成。
一致性创造了认知杠杆：一旦你学会了如何在某个地方完成某件事，你就可以利用这些知识立即理解使用相同方法的其他地方。如果一个系统没有以一致的方式实现，开发人员必须分别了解每种情况。这将花费更多时间。

一致性减少了错误。如果一个系统不一致，两种情况可能看起来相同，但实际上它们是不同的。开发人员可能会看到一个看起来很熟悉的模式，并根据之前遇到该模式的情况做出不正确的假设。另一方面，如果系统是一致的，那么基于看起来熟悉的情况所做的假设将是安全的。一致性使开发人员能够更快地工作，减少错误。

17.1 一致性的例子

一致性可以应用于系统中的许多级别；以下是一些例子。

名称。第 14 章已经讨论了以一致的方式使用名称的好处。

编码风格。现在，开发组织通常都有风格指南，除了编译器强制执行的规则之外，还限制程序结构。现代风格指南涉及一系列问题，例如缩进、大括号位置、声明顺序、命名、注释以及对被认为危险的语言功能的限制。风格指南使代码更易于阅读，并可以减少某些类型的错误。

接口。具有多个实现的接口是另一个一致性的例子。一旦你理解了接口的一个实现，任何其他的实现都会变得更容易理解，因为你已经知道它必须提供的功能。

设计模式。设计模式是针对某些常见问题的普遍接受的解决方案，例如用户界面设计的模型-视图-控制器方法。如果你可以使用现有的设计模式来解决问题，那么实现将进行得更快，它更有可能工作，并且你的代码对于读者来说将更加明显。设计模式将在第19.5节中更详细地讨论。

不变性。不变性是变量或结构的始终为真的属性。例如，存储文本行的数据结构可以强制执行一个不变性，即每行都以换行符结尾。不变性减少了代码中必须考虑的特殊情况的数量，并使推理代码的行为更容易。

17.2 确保一致性

一致性很难维护，尤其是在许多人长时间在一个项目上工作时。一个小组的人可能不知道另一个小组建立的约定。新手不知道规则，因此他们无意中违反了约定并创建了与现有约定冲突的新约定。以下是一些建立和维护一致性的技巧：

文档。创建一个文档，列出最重要的总体约定，例如编码风格指南。将文档放在开发人员可能看到的地方，例如项目Wiki上的显眼位置。鼓励加入该小组的新人阅读该文档，并鼓励现有人员不时对其进行审查。来自各个组织的几个风格指南已在Web上发布；考虑从其中一个开始。

对于更局部的约定，例如不变性，请在代码中找到适当的位置来记录它们。如果你不写下约定，其他人不太可能遵循它们。

强制执行。即使有良好的文档，开发人员也很难记住所有的约定。强制执行约定的最佳方法是编写一个工具

检查违规行为，并确保除非代码通过检查器，否则无法将其提交到存储库。自动检查器对于低级语法约定特别有效。

我最近的一个项目遇到了行终止符的问题。
一些开发人员在Unix上工作，那里的行以换行符结束；另一些开发人员在Windows上工作，那里的行通常以回车符后跟换行符结束。如果一个系统上的开发人员对先前在另一个系统上编辑过的文件进行了少量编辑，则编辑器有时会用适用于该系统的行终止符替换所有行终止符。这给人的印象是文件的每一行都已被修改，这使得跟踪有意义的更改变得困难。我们建立了一个约定，即文件应仅包含换行符，但很难确保每个开发人员使用的每个工具都遵循该约定。每次有新的开发人员加入项目时，当该开发人员适应该约定时，我们都会遇到一系列行终止问题。

我们最终通过编写一个简短的脚本解决了这个问题，该脚本在更改提交到源代码存储库之前自动执行。该脚本检查所有已修改的文件，如果其中任何文件包含回车符，则中止提交。该脚本也可以手动运行，通过将回车/换行序列替换为换行符来修复损坏的文件。

这立即消除了问题，并且还有助于培训新的开发人员。

代码审查提供了另一个执行约定和教育新开发人员了解约定的机会。代码审查员越挑剔，团队中的每个人学习约定的速度就越快，代码就越干净。

入乡随俗 ... 最重要的约定是每个开发人员都应遵循古老的格言“入乡随俗”。在新文件中工作时，请环顾四周，看看现有代码是如何构建的。是否所有公共变量和方法都在私有变量和方法之前声明？这些方法是否按字母顺序排列？变量是否使用“驼峰式命名”，如firstServerName，或“蛇形命名”，如first_server_name？当你看到任何看起来可能是一种约定的东西时，请遵循它。在做出设计决策时，问问自己是否有可能在项目的其他地方做出类似的决策；如果是这样，请找到一个现有的例子，并在你的新代码中使用相同的方法。

不要更改现有约定。抵制“改进”现有约定的冲动。拥有一个“更好的主意”不足以成为引入不一致性的理由。你的新想法可能确实更好，但相对于不一致性而言，一致性的价值几乎总是大于一种方法相对于另一种方法的价值。在引入不一致的行为之前，问自己两个问题。首先，你是否有重要的新信息来证明你的方法是合理的，而这些信息在建立旧约定时尚未获得？其次，新方法是否好到值得花时间更新所有旧用法？如果你的组织同意这两个问题的答案都是“是”，那么继续进行升级；当你完成后，应该没有旧约定的迹象。但是，你仍然面临其他开发人员可能不知道新约定的风险，因此他们将来可能会重新引入旧方法。总的来说，重新考虑已建立的约定很少能很好地利用开发人员的时间。

17.3 过犹不及

一致性不仅意味着应该以类似的方式完成类似的事情，而且意味着应该以不同的方式完成不同的事情。如果你对一致性过于热心，并试图将不同的事物强行采用相同的方法，例如对真正不同的事物使用相同的变量名，或者对不适合该模式的任务使用现有的设计模式，你将制造复杂性和混乱。只有当开发人员确信“如果它看起来像一个x，它实际上就是一个x”时，一致性才能提供好处。

17.4 结论

一致性是投资心态的另一个例子。确保一致性需要额外的工作：决定约定，创建自动化检查器，寻找类似的情况在新代码中模仿，并在代码审查中教育团队。这项投资的回报是你的代码会更清晰。开发人员能够更快、更准确地理解代码的行为，这将使他们能够更快地工作，减少错误。

第 18 章

代码应该显而易见

晦涩是第 2.3 节中描述的复杂性的两个主要原因之一。当系统中重要的信息对于新开发人员来说不明显时，就会出现晦涩。解决晦涩问题的方法是以一种显而易见的方式编写代码；本章讨论了一些使代码或多或少显而易见的因素。

如果代码是显而易见的，这意味着有人可以快速阅读代码，而无需过多思考，并且他们对代码的行为或含义的第一个猜测将是正确的。如果代码是显而易见的，则读者无需花费大量时间和精力来收集他们使用代码所需的所有信息。如果代码不明显，那么读者必须花费大量时间和精力来理解它。这不仅降低了他们的效率，而且还增加了误解和错误的几率。显而易见的代码比不明显的代码需要的注释更少。

“显而易见”存在于读者的脑海中：更容易注意到别人的代码不明显，而不是看到自己代码的问题。因此，确定代码是否显而易见的最佳方法是通过代码审查。如果有人阅读你的代码说它不明显，那么它就不明显，无论它对你来说多么清晰。通过尝试理解是什么使代码不明显，你将学会如何在将来编写更好的代码。

18.1 使代码更明显的因素

使代码更明显的两个最重要的技术已经在前面的章节中讨论过了。第一个是选择好的名称（第 1 4章）。精确且有意义的名称可以阐明代码的行为并减少对文档的需求。如果名称含糊不清，那么读者将不得不通读代码才能推断出命名实体的含义；这既耗时又容易出错。第二种技术是一致性

（第17章）。如果类似的事情总是以类似的方式完成，那么读者可以识别他们以前见过的模式，并立即得出（安全的）结论，而无需详细分析代码。

以下是一些使代码更清晰的其他通用技术：

明智地使用空白。代码的格式会影响代码的可理解程度。请看下面的参数文档，其中空白被压缩了：

```
/**
 *  ...
 * @param numThreads 该管理器应启动的线程数，* 以便管理正在进行的连接。MessageManager* 为每个打开的连接至少启动一个线程，因此这* 应该至少等于您期望同时打开的连接数。* 如果您希望在短时间内发送大量消息，则这应该是该数字的倍数。

 * @param handler 用作回调，以便处理此 MessageManager 的开放连接上的传入消息。请参阅
 * 有关详细信息，请参阅 {@code MessageHandler} 和 {@code handleMessage}。
 */
```

很难看出一个参数的文档在哪里结束，下一个参数在哪里开始。甚至不清楚有多少个参数，或者它们的名称是什么。如果添加一些空白，结构会突然变得清晰，文档也更容易扫描：

```
/**
 * @param numThreads（线程数）
 *                该管理器应启动的线程数，*                以便管理正在进行的连接。MessageManager会启动

 * 每个打开的连接至少启动一个线程，因此这应该至少等于您期望同时打开的连接数。这应该是that的倍数

 * 如果您希望在短时间内发送大量消息，请使用数字。
```

```
*  @param handler
* 用作回调，以便处理此MessageManager的开放连接上的传入消息。请参阅

* {@code MessageHandler}和{@code handleMessage}了解详情。

*/
```

空行也可用于分隔方法中的主要代码块，如下例所示：

```
void* Buffer::allocAux(size_t numBytes)
{
    // 将长度向上舍入为8字节的倍数，以确保
        对齐。
    uint32_t numBytes32 =  (downCast<uint32_t>(numBytes) + 7) & ~0x7;
    assert(numBytes32 != 0);

    // 如果firstAvailable有足够的内存，请使用它。从顶部
        向下工作
    // 因为保证此内存已对齐
    //（底部的内存可能已用于可变大小的
        块）。
    如果 (availableLength >= numBytes32) {
        availableLength -= numBytes32;
        return firstAvailable + availableLength;
    }

    // 接下来，看看最后一个块的末尾是否有额外的空间。
    如果 (extraAppendBytes >= numBytes32) {
        extraAppendBytes -= numBytes32;
        return lastChunk->data + lastChunk->length + extraAppendBytes;
    }

    // 必须创建一个新的空间分配；在其中分配空间。
    uint32_t allocatedLength;
    firstAvailable = getNewAllocation(numBytes32, &allocatedLength);
    availableLength = allocatedLength numBytes32;
```

```
    return firstAvailable + availableLength;
}
```

如果每个空白行后的第一行是描述下一个代码块的注释，那么这种方法特别有效：空白行使注释更可见。

语句中的空格有助于阐明语句的结构。
比较以下两个语句，其中一个有空格，另一个没有：

```
for(int pass=1;pass>=0&&!empty;pass--) {
```

```
for (int pass = 1; pass >= 0 && !empty; pass--) {
```

注释。有时，不可避免地会出现一些不明显的代码。当发生这种情况时，重要的是使用注释来弥补，提供缺失的信息。为了做好这一点，你必须设身处地为读者着想，弄清楚什么可能会让他们感到困惑，以及什么信息可以消除这种困惑。下一节将展示几个例子。

18.2 使代码不太明显的事情

有很多事情会使代码变得不明显；本节提供了一些示例。其中一些，例如事件驱动编程，在某些情况下很有用，因此您最终可能会使用它们。当发生这种情况时，额外的文档可以帮助最大限度地减少读者的困惑。

事件驱动编程。在事件驱动编程中，应用程序响应外部事件，例如网络数据包的到达或鼠标按钮的按下。一个模块负责报告传入的事件。应用程序的其他部分通过要求事件模块在这些事件发生时调用给定的函数或方法来注册对某些事件的兴趣。

事件驱动编程使得跟踪控制流变得困难。事件处理函数永远不会被直接调用；它们是由事件模块间接调用的，通常使用函数指针或接口。即使您在事件模块中找到了调用点，仍然无法确定将调用哪个特定函数：这将取决于在运行时注册了哪些处理程序。因此，很难推理事件驱动的代码或说服自己它可以工作。

为了弥补这种模糊性，请使用每个处理程序的接口注释来指示何时调用它，如以下示例所示：

```
/**
 * 如果传输层错误阻止 RPC 完成，则此方法由传输在调度线程中调用。

 */
void
Transport::RpcNotifier::failed() {
    ...
}
```

危险信号：不明显的代码

如果代码的含义和行为不能通过快速阅读来理解，这是一个危险信号。通常这意味着有些重要的信息对于阅读代码的人来说并不立即清楚。

通用容器。许多语言都提供了通用类，用于将两个或多个项目分组到一个对象中，例如 Java 中的 Pair 或 C++ 中的 std::pair 。
这些类很诱人，因为它们可以很容易地用一个变量传递多个对象。最常见的用途之一是从方法返回多个值，如以下 Java 示例所示：

```
return new Pair<Integer, Boolean>(currentTerm, false);
```

不幸的是，通用容器会导致代码不明显，因为分组的元素具有通用的名称，掩盖了它们的含义。在上面的例子中，调用者必须使用 result.getKey() 和 result.getValue() 引用两个返回值，这没有提供关于值的实际含义的线索。

因此，最好不要使用通用容器。如果需要容器，请定义一个新的类或结构，该类或结构专门用于特定用途。然后，您可以为元素使用有意义的名称，并且可以在声明中提供额外的文档，这对于通用容器是不可能的。

这个例子说明了一个普遍的规则：软件的设计应该便于阅读，而不是便于编写。通用容器对于

编写代码的人来说很方便，但它们会给所有后续的读者带来困惑。编写代码的人最好多花几分钟来定义一个特定的容器结构，这样生成的代码会更清晰。

声明和分配的不同类型。考虑以下Java
示例：

```
private List<Message> incomingMessageList;
...
incomingMessageList = new ArrayList<Message>();
```

该变量被声明为List，但实际值是一个ArrayList。这段代码是合法的，因为List是ArrayList的超类，但它可能会误导只看到声明而没有看到实际分配的读者。实际类型可能会影响变量的使用方式（ArrayLists具有与其他List子类不同的性能和线程安全性属性），因此最好将声明与分配相匹配。

违反读者期望的代码。考虑以下代码，它是Java应用程序的主程序

```
public static void main(String[] args) {
    ...
    new RaftClient(myAddress, serverAddresses);
}
```

大多数应用程序在主程序返回时退出，因此读者很可能会认为这里也会发生这种情况。然而，事实并非如此。RaftClient的构造函数RaftClient创建额外的线程，即使应用程序的主线程完成，这些线程也会继续运行。这种行为应该在RaftClient构造函数的接口注释中进行记录，但这种行为非常不明显，因此值得在main的末尾添加一个简短的注释。该注释应表明应用程序将继续在其他线程中执行。如果代码符合读者期望的约定，那么它就是最明显的；如果不符合，那么重要的是记录这种行为，以免读者感到困惑。

18.3 结论

另一种思考显而易见性的方式是从信息的角度出发。如果代码不明显，那通常意味着关于代码有一些重要的信息

读者没有掌握：在 RaftClient 示例中，读者可能不知道 RaftClient 构造函数创建了新的线程；在 Pair 示例中，读者可能不知道 result.getKey() 返回当前任期的编号。

为了使代码显而易见，您必须确保读者始终拥有理解代码所需的信息。您可以通过三种方式做到这一点。最好的方法是减少所需的信息量，使用诸如抽象和消除特殊情况之类的设计技术。其次，您可以利用读者已经在其他上下文中获得的信息（例如，通过遵循约定和符合期望），这样读者就不必为您的代码学习新的信息。第三，您可以使用诸如好的命名和战略性注释之类的技术，在代码中向他们展示重要的信息。

第19章

软件趋势

作为说明本书中讨论的原则的一种方式，本章考虑了在过去几十年中在软件开发中变得流行的几种趋势和模式。对于每种趋势，我将描述该趋势与本书中的原则有何关系，并使用这些原则来评估该趋势是否能利用软件的复杂性。

19.1 面向对象编程和继承

面向对象编程是过去30-40年来软件开发中最重要的新思想之一。它引入了诸如类、继承、私有方法和实例变量等概念。如果小心使用，这些机制可以帮助产生更好的软件设计。例如，私有方法和变量可以用于确保信息隐藏：类外部的任何代码都不能调用私有方法或访问私有变量，因此不能存在对它们的任何外部依赖。

面向对象编程的关键要素之一是继承。
继承有两种形式，它们对软件复杂性有不同的影响。第一种继承形式是接口继承，其中父类定义一个或多个方法的签名，但不实现这些方法。每个子类都必须实现这些签名，但不同的子类可以用不同的方式实现相同的方法。例如，接口可以定义用于执行I/O的方法；一个子类可以实现用于磁盘文件的I/O操作，另一个子类可以实现用于网络套接字的相同操作。

接口继承通过为多个目的重用相同的接口来利用复杂性。它允许将在解决一个问题（例如如何使用I/O接口来读取和写入磁盘文件）中获得的知识用于解决其他问题（例如通过网络套接字进行通信）。

另一种思考这个问题的方式是从深度的角度来看：一个接口的不同实现越多，这个接口就越深入。为了使一个接口有许多实现，它必须捕捉所有底层实现的基本特征，同时避开实现之间不同的细节；这个概念是抽象的核心。

第二种继承形式是实现继承。在这种形式中，父类不仅定义了一个或多个方法的签名，还定义了默认实现。子类可以选择继承父类的方法实现，或者通过定义具有相同签名的新方法来覆盖它。

如果没有实现继承，相同的方法实现可能需要在多个子类中重复，这将在这些子类之间创建依赖关系（修改需要在方法的所有副本中重复）。因此，实现继承减少了随着系统发展需要修改的代码量；换句话说，它减少了第2章中描述的变更放大问题。

然而，实现继承在父类和它的每个子类之间创建了依赖关系。父类中的类实例变量通常被父类和子类访问；这导致继承层次结构中的类之间的信息泄漏，并且使得在不查看其他类的情况下很难修改层次结构中的一个类。例如，对父类进行更改的开发人员可能需要检查所有子类，以确保更改不会破坏任何内容。类似地，如果子类覆盖了父类中的一个方法，子类的开发人员可能需要检查父类中的实现。在最坏的情况下，程序员需要完全了解父类下的整个类层次结构，才能对任何类进行更改。广泛使用实现继承的类层次结构往往具有很高的复杂性。

因此，应该谨慎使用实现继承。在使用实现继承之前，请考虑基于组合的方法是否可以提供相同的好处。例如，可以使用小型辅助类来实现共享功能。原始类可以各自构建辅助类的特性，而不是从父类继承函数。

如果没有可行的替代实现继承方案，请尝试将父类管理的状态与子类管理的状态分开。一种方法

这样做的方法是，某些实例变量完全由父类中的方法管理，子类仅以只读方式或通过父类中的其他方法使用它们。这在类层次结构中应用了信息隐藏的概念，以减少依赖关系。

尽管面向对象编程提供的机制可以帮助实现清晰的设计，但它们本身并不能保证良好的设计。例如，如果类很浅，或者具有复杂的接口，或者允许外部访问其内部状态，那么它们仍然会导致高复杂性。

19.2 敏捷开发

敏捷开发是一种软件开发方法，它于 20 世纪 90 年代末从一系列关于如何使软件开发更轻量、更灵活和更增量的想法中产生；它在 2001 年的一次从业者会议上被正式定义。敏捷开发主要关注软件开发的过程（组织团队、管理进度、单元测试的角色、与客户互动等），而不是软件设计。

尽管如此，它与本书中的一些设计原则有关。

敏捷开发中最重要的要素之一是开发应该是增量和迭代的。在敏捷方法中，软件系统是通过一系列迭代开发的，每次迭代都会添加和评估一些新功能；每次迭代都包括设计、测试和客户输入。总的来说，这与这里提倡的增量方法类似。正如第1章中提到的，在项目开始时，不可能充分地可视化一个复杂的系统，从而确定最佳设计。获得良好设计的最佳方法是以增量方式开发系统，其中每个增量都会添加一些新的抽象，并根据经验重构现有的抽象。这与敏捷开发方法类似。

敏捷开发的风险之一是它可能导致战术编程。敏捷开发倾向于让开发者关注功能，而不是抽象，并且它鼓励开发者推迟设计决策，以便尽快生成可用的软件。例如，一些敏捷实践者认为你不应该立即实现通用的机制；首先实现一个最小的专用机制，然后在你知道需要它之后，将其重构为更通用的东西。

虽然这些论点在某种程度上是有道理的，但它们反对

投资方法，并且它们鼓励一种更具战术性的编程风格。这可能导致复杂性的快速积累。

增量开发通常是一个好主意，但是开发的增量应该是抽象，而不是功能。在某个功能需要之前，推迟对特定抽象的所有想法是可以的。一旦你需要抽象，就花时间来干净地设计它；遵循第6章的建议，并使其具有一定的通用性。

19.3 单元测试

过去，开发者很少编写测试。如果编写了测试，通常是由一个单独的QA团队编写的。然而，敏捷开发的宗旨之一是测试应该与开发紧密结合，程序员应该为自己的代码编写测试。这种做法现在已经很普遍了。测试通常分为两种：单元测试和系统测试。单元测试是开发者最常编写的。它们很小且专注：每个测试通常验证单个方法中的一小段代码。单元测试可以独立运行，而无需为系统设置生产环境。单元测试通常与测试覆盖率工具一起运行，以确保应用程序中的每一行代码都经过测试。

每当开发者编写新代码或修改现有代码时，他们都有责任更新单元测试，以保持适当的测试覆盖率。

第二种测试包括系统测试（有时称为集成测试），它确保应用程序的不同部分都能协同工作。它们通常涉及在生产环境中运行整个应用程序。系统测试更可能由单独的QA或测试团队编写。

测试，特别是单元测试，在软件设计中扮演着重要的角色，因为它们有助于重构。如果没有测试套件，对系统进行重大的结构性更改是很危险的。没有简单的方法来发现错误，因此很可能错误在部署新代码之前不会被发现，而在部署后发现和修复它们的成本要高得多。因此，开发人员在没有良好测试套件的系统中会避免重构；他们试图最小化每个新功能或错误修复的代码更改数量，这意味着复杂性会累积，设计错误得不到纠正。

有了良好的一组测试，开发人员在重构时可以更加自信，因为测试套件会发现大多数引入的错误。这鼓励

开发人员对系统进行结构性改进，从而产生更好的设计。单元测试尤其有价值：它们比系统测试提供更高的代码覆盖率，因此它们更有可能发现任何错误。

例如，在开发 Tcl 脚本语言期间，我们决定通过用字节码编译器替换 Tcl 的解释器来提高性能。这是一个巨大的变化，几乎影响了核心 Tcl 引擎的每个部分。幸运的是，Tcl 有一个出色的单元测试套件，我们在新的字节码引擎上运行了它。现有的测试在发现新引擎中的错误方面非常有效，以至于在字节码编译器的 alpha 版本发布后，只出现了一个错误。

19.4 测试驱动开发

测试驱动开发是一种软件开发方法，程序员在编写代码之前先编写单元测试。在创建新类时，开发人员首先根据其预期行为为该类编写单元测试。

由于没有类的代码，因此没有一个测试通过。然后，开发人员一次完成一个测试，编写足够的代码以使该测试通过。
当所有测试都通过时，该类就完成了。

虽然我强烈支持单元测试，但我并不喜欢测试驱动开发。测试驱动开发的问题在于，它将注意力集中在使特定功能正常工作上，而不是找到最佳设计。这纯粹是战术编程，具有所有缺点。测试驱动开发过于增量式：在任何时间点，都容易仅仅为了使下一个测试通过而加入下一个功能。没有明显的时间进行设计，因此很容易最终得到一团糟。

正如在第 19.2节中提到的，开发的单元应该是抽象，而不是功能。一旦你发现需要一个抽象，不要随着时间的推移逐步创建抽象；一次性设计它（或者至少足以提供一套相当全面的核心功能）。这更有可能产生一个干净的设计，其各个部分能够很好地组合在一起。

在修复错误时，首先编写测试的一个合理之处。
在修复错误之前，编写一个由于该错误而失败的单元测试。然后修复错误，并确保单元测试现在通过。这是确保你真正修复了错误的最佳方法。如果你在编写测试之前修复了错误，那么新的单元测试可能实际上不会触发该错误，在这种情况下，它不会告诉你是否真的解决了问题。

19.5 设计模式

设计模式是一种常用的方法，用于解决特定类型的问题，例如迭代器或观察者。设计模式的概念因《设计模式：可复用面向对象软件的元素》一书而普及，该书由 Gamma、Helm、Johnson 和 Vlissides 撰写，现在设计模式已广泛应用于面向对象软件开发中。

设计模式代表了设计的一种替代方案：与其从头开始设计一种新的机制，不如直接应用一种广为人知的设计模式。在大多数情况下，这是好的：设计模式的出现是因为它们解决了常见的问题，并且因为人们普遍认为它们提供了清晰的解决方案。如果一种设计模式在特定情况下运行良好，那么你可能很难想出一种更好的不同方法。

设计模式的最大风险是过度应用。并非每个问题都可以用现有的设计模式干净利落地解决；当自定义方法更清晰时，不要试图将问题强行塞进设计模式中。使用设计模式并不能自动改进软件系统；只有当设计模式适用时才能做到这一点。与软件设计中的许多想法一样，设计模式是好的这一概念并不一定意味着更多的设计模式就更好。

19.6 Getters 和 setters

在 Java 编程社区中，getter 和 setter 方法是一种流行的设计模式。getter 和 setter 与类的实例变量相关联。它们的名字类似于 getFoo 和 setFoo，其中 Foo 是变量的名字。getter 方法返回变量的当前值，而 setter 方法修改该值。

Getters 和 setters 并非绝对必要，因为实例变量可以被公开。支持 getters 和 setters 的理由是，它们允许在获取和设置时执行额外的功能，例如在变量更改时更新相关值、通知更改的监听器或强制执行值的约束。即使最初不需要这些功能，也可以在以后添加它们，而无需更改接口。

虽然如果你必须公开实例变量，那么使用 getters 和 setters 可能是有意义的，但最好不要首先公开实例变量。
暴露的实例变量意味着类的部分实现
在外部可见，这违反了信息隐藏的思想并增加了

类的接口的复杂性。Getter 和 setter 是浅层方法（通常只有一行），因此它们会增加类接口的混乱，而不会提供太多功能。最好尽可能避免使用 getter 和 setter（或任何实现数据的暴露）。

建立设计模式的风险之一是开发人员认为该模式很好并尝试尽可能多地使用它。这导致了 Java 中 getter 和 setter 的过度使用。

19.7 结论

每当您遇到新的软件开发范例的提议时，都要从复杂性的角度对其提出质疑：该提议是否真的有助于最大限度地降低大型软件系统中的复杂性？许多提议表面上听起来不错，但如果您更深入地观察，您会发现其中一些提议会使复杂性变得更糟，而不是更好。

第 20 章

性能设计

到目前为止，软件设计的讨论主要集中在复杂性上；
目标是使软件尽可能简单易懂。
但是，如果您正在处理需要快速运行的系统，该怎么办？
性能考虑因素应该如何影响设计过程？本章讨论了
如何在不牺牲简洁设计的情况下实现高性能。最
重要的思想仍然是简单性：简单性不仅可以改善系统的
设计，而且通常可以使系统更快。

20.1 如何考虑性能

要解决的第一个问题是“在正常的开发过程中，您应该对性能有多担心？”
如果您尝试优化每个语句以获得最大速度，它将减慢开发速度并产生大量
不必要的复杂性。此外，许多“优化”实际上并不能提高性能。另一方面，
如果您完全忽略性能问题，很容易最终在代码中出现大量重大低效率；由
此产生的系统很容易达到 5–

1 比实际需要的速度慢 0x。在这种“千刀万剐”的情况下，
以后很难再回来提高性能，因为没有哪一项改进会产生很大的影响。

最好的方法是介于这两种极端之间，即您利用性能的基本知识来选择“
自然高效”且干净简单的设计方案。关键是要意识到哪些操作从根本上来说
是昂贵的。以下是一些当今相对昂贵的操作示例：

- 网络通信：即使在数据中心内，一次往返消息交换也可能需要 10
  -50 微秒，这是数万条指令的时间。
  广域网往返可能需要 10-100 毫秒。
- 二级存储的 I/O：磁盘 I/O 操作通常需要 5-10 毫秒，这

是数百万条指令的时间。闪存存储需要 10-100 微秒。新兴的非易失性存储器可能快至 1 微秒，但这仍然大约是 2000 条指令的时间。

- 动态内存分配（C 中的 malloc，C++ 或 Java 中的 new）通常涉及分配、释放和垃圾回收的大量开销。
- 缓存未命中：从 DRAM 中获取数据到片上处理器缓存中需要数百条指令的时间；在许多程序中，整体性能的决定因素是缓存未命中，而不是计算成本。

了解哪些东西昂贵的最好方法是运行微基准测试（测量单个操作隔离成本的小程序）。在RAMCloud项目中，我们创建了一个简单的程序，该程序提供了一个微基准测试框架。创建该框架花了几天时间，但是该框架使得在五到十分钟内添加新的微基准测试成为可能。这使我们能够积累数十个微基准测试。我们使用这些方法来了解RAMCloud中使用的现有库的性能，并测量为RAMCloud编写的新类的性能。

一旦你对什么是昂贵的，什么是便宜的有一个大致的了解，你就可以利用这些信息尽可能地选择廉价的操作。在许多情况下，一种更有效的方法会和一种较慢的方法一样简单。例如，当存储一个可以使用键值查找的大型对象集合时，你可以使用哈希表或有序映射。两者在库包中都很常见，而且使用起来都很简单和干净。

然而，哈希表可以很容易地快 5-10 倍。因此，除非你需要映射提供的排序属性，否则你应该始终使用哈希表。

再举一个例子，考虑在 C 或 C++ 等语言中分配一个结构体数组。有两种方法可以做到这一点。一种方法是让数组保存指向结构体的指针，在这种情况下，你必须首先为数组分配空间，然后为每个单独的结构体分配空间。更有效的方法是将结构体存储在数组本身中，这样你只需要为所有内容分配一个大的块。

如果提高效率的唯一方法是增加复杂性，那么选择就更加困难。如果更有效的设计只增加少量复杂性，并且如果复杂性是隐藏的，因此它不会影响任何接口，那么它可能是值得的（但要注意：复杂性是递增的）。如果更快的设计增加了大量的实现复杂性，或者如果它导致更多

更复杂的接口，那么最好从更简单的方法开始，如果性能出现问题，以后再进行优化。但是，如果你有明确的证据表明性能在特定情况下很重要，那么你最好立即实现更快的方法。

在 RAMCloud 项目中，我们的总体目标之一是为通过数据中心网络访问存储系统的客户端机器提供尽可能低的延迟。因此，我们决定使用特殊的硬件进行联网，这使得 RAMCloud 能够绕过内核并直接与网络接口控制器通信以发送和接收数据包。我们做出这个决定，即使它增加了复杂性，因为我们从之前的测量中得知，基于内核的网络速度太慢，无法满足我们的需求。在 RAMCloud 系统的其余大部分中，我们都能够设计得简单；把这一个大问题“解决好”使得许多其他事情变得更容易。

一般来说，更简单的代码往往比复杂的代码运行得更快。如果你已经定义了特殊情况和异常，那么就不需要代码来检查这些情况，系统运行得更快。深类比浅类更有效，因为它们每次方法调用都能完成更多的工作。浅类导致更多的层交叉，并且每个层交叉都会增加开销。

20.2 修改前先测量

但是，假设你的系统仍然太慢，即使你已经按照上述方式设计了它。人们很容易冲动地开始进行性能调整，基于你对什么慢的直觉。不要这样做！程序员对性能的直觉是不可靠的。即使对于经验丰富的开发人员来说也是如此。如果你开始根据直觉进行更改，你会在实际上没有提高性能的事情上浪费时间，并且你可能会使系统在此过程中变得更加复杂。

在进行任何更改之前，测量系统现有的行为。这有两个目的。首先，测量将确定性能调整将产生最大影响的地方。仅仅测量顶层系统性能是不够的。这可能会告诉你系统太慢，但它不会告诉你为什么。你需要更深入地测量，以详细识别导致整体性能的因素；目标是识别出少量非常具体的地方，系统目前在这些地方花费大量时间，并且你对改进有想法。测量的第二个目的是提供一个基线，以便你可以重新测量性能

在进行更改后，请确保性能确实得到了提升。如果这些更改没有对性能产生可衡量的影响，那么请撤销它们（除非它们使系统更简单）。保留复杂性是没有意义的，除非它能提供显著的加速。

20.3 围绕关键路径进行设计

现在，让我们假设您已经仔细分析了性能，并且已经
识别出一段代码，这段代码的速度足够慢，足以影响整个系统
性能。提高其性能的最佳方法是进行“根本性”的
更改，例如引入缓存，或使用不同的算法方法
（例如，平衡树与列表）。我们在RAMCloud中绕过内核进行
网络通信的决定就是一个根本性修复的例子。如果
您可以确定一个根本性的修复，那么您可以使用设计
前几章中讨论的技术来实现它。

不幸的是，有时会出现没有根本性解决方案的情况。这就引出了本章的核心问题，即如何重新设计一段现有的代码，使其运行得更快。这应该是你的最后手段，而且不应该经常发生，但在某些情况下，它可以产生很大的影响。关键的想法是围绕关键路径设计代码。

首先问问自己，在常见情况下，执行所需任务必须执行的最小代码量是多少。忽略任何现有的代码结构。想象一下，您正在编写一个新方法，该方法仅实现关键路径，这是在最常见情况下必须执行的最小代码量。当前的代码可能充斥着特殊情况；在本练习中忽略它们。当前的代码可能会在关键路径上通过多个方法调用；想象一下，您可以将所有相关代码放在一个方法中。当前的代码也可能使用各种变量和数据结构；仅考虑关键路径所需的数据，并假设最方便关键路径的数据结构。例如，将多个变量合并为一个值可能是有意义的。假设您可以完全重新设计系统，以最大程度地减少必须为关键路径执行的代码。让我们将此代码称为“理想”。

理想的代码可能与你现有的类结构冲突，而且可能不实用，但它提供了一个很好的目标：这代表了代码可能达到的最简单和最快状态。下一步是寻找一种新的设计，它

尽可能接近理想状态，同时保持清晰的结构。你可以应用本书前几章的所有设计理念，但要额外约束自己，保持理想代码（大部分）的完整性。你可能需要在理想代码中添加一些额外的代码，以便实现清晰的抽象；例如，如果代码涉及哈希表查找，可以引入对通用哈希表类的额外方法调用。根据我的经验，几乎总是可以找到一种既干净又简单的设计，而且非常接近理想状态。

在这个过程中，最重要的事情之一是从关键路径中移除特殊情况。当代码运行缓慢时，通常是因为它必须处理各种各样的情况，并且代码的结构是为了简化所有不同情况的处理。每种特殊情况都会以额外的条件语句和/或方法调用的形式向关键路径添加少量代码。

这些新增内容都会使代码运行速度变慢。在重新设计以提高性能时，尽量减少必须检查的特殊情况的数量。
理想情况下，在开始时会有一个单一的 if 语句，它会检测所有特殊情况，只需一次测试。在正常情况下，只需要进行这一个测试，之后就可以执行关键路径，而无需对特殊情况进行额外的测试。如果初始测试失败（这意味着发生了特殊情况），代码可以分支到关键路径之外的单独位置来处理它。
对于特殊情况，性能并不那么重要，因此您可以为了简单起见而不是为了性能来构建特殊情况代码。

20.4 一个例子：RAMCloud 缓冲区

让我们考虑一个例子，其中 RAMCloud 存储系统的 Buffer 类经过优化，使其在最常见的操作中实现了大约 2 倍的加速。

RAMCloud 使用 Buffer 对象来管理可变长度的内存数组，例如远程过程调用的请求和响应消息。Buffer 的设计目的是减少内存复制和动态存储分配的开销。Buffer 存储的是一个看起来像是线性排列的字节数组，但为了提高效率，它允许将底层存储划分为多个不连续的内存块，如图 20.1 所示。Buffer 通过追加数据块来创建。每个块可以是外部的或内部的。如果一个块是外部的，则其存储由调用者拥有；Buffer 保留对该存储的引用。外部块通常用于大型块，以避免内存复制。如果一个块是内部的，则 Buffer 拥有该块的存储；调用者提供的数据会被复制到 Buffer 的内部存储中。

如果一个块是内部的，则 Buffer 拥有该块的存储；调用者提供的数据会被复制到 Buffer 的内部存储中。
每个 Buffer 都包含一个小的内置分配，这是一个可用于存储内部块的内存块。如果此空间耗尽，则 Buffer 会创建额外的分配，这些分配必须在 Buffer 被销毁时释放。
内部块对于内存复制成本可忽略不计的小块来说很方便。图 20.1 显示了一个包含 5 个块的 Buffer：第一个块是内部的，接下来的两个是外部的，最后两个块是内部的。

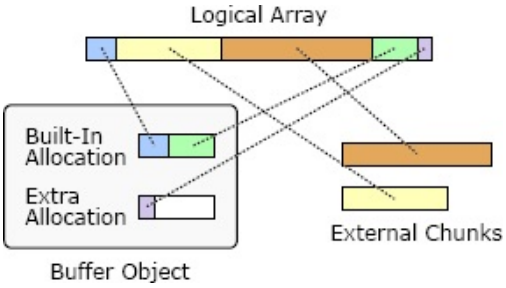

图 20.1：Buffer 对象使用内存块的集合来存储看起来像是线性排列的字节数组。内部块由 Buffer 拥有，并在 Buffer 被销毁时释放；外部块不归 Buffer 所有。

Buffer 类本身代表了一种“根本性的修复”，因为它消除了没有它所必需的昂贵的内存复制。例如，当组装包含短标头和 RAMCloud 存储系统中大型对象内容的响应消息时，RAMCloud 使用带有两个块的 Buffer。第一个块是包含标头的内部块；第二个块是引用 RAMCloud 存储系统中对象内容的外部块。响应可以在 Buffer 中收集，而无需复制大型对象。

除了允许不连续块的根本方法之外，我们没有尝试在原始实现中优化 Buffer 类的代码。然而，随着时间的推移，我们注意到 Buffer 被用于越来越多的情况；例如，在每次远程过程调用的执行过程中，至少会创建四个 Buffer。最终，很明显，加速 Buffer 的实现可能会对整体系统性能产生显著影响。我们决定看看是否可以提高 Buffer 类的性能。

Buffer 最常见的操作是使用内部块为少量新数据分配空间。例如，当

为请求和响应消息创建标头。我们决定使用此操作作为优化的关键路径。在最简单的情况下，可以通过扩大缓冲区中最后一个现有块来分配空间。

然而，这只有在最后一个现有块是内部块，并且其分配中有足够的空间来容纳新数据时才有可能。理想的代码将执行一次检查，以确认简单方法是可行的，然后它将调整现有块的大小。

图 20.2显示了关键路径的原始代码，该代码以方法Buffer::alloc开始。在最快的情况下，Buffer::alloc调用Buffer::
allocateAppend，它调用Buffer::Allocation::allocateAppend。从性能的角度来看，此代码有两个问题。第一个问题是，许多特殊情况都是单独检查的：

- Buffer::allocateAppend检查缓冲区当前是否有任何分配。
- 代码检查两次，以查看当前分配是否有足够的空间容纳新数据：一次在Buffer::Allocation::allocateAppend中，另一次在其返回值被Buffer::allocateAppend测试时。
- Buffer::alloc测试来自Buffer::allocAppend的返回值，以再次确认分配成功。

此外，代码没有尝试直接扩展最后一个块，而是分配了新的空间，而没有考虑最后一个块。然后
Buffer::alloc检查该空间是否恰好与最后一个块相邻，如果是，则将新空间与现有块合并。这导致额外的检查。总的来说，此代码在关键路径中测试了 6 个不同的条件。

原始代码的第二个问题是它有太多的层，所有这些层都很浅。这既是性能问题，也是设计问题。
除了最初调用Buffer::alloc之外，关键路径还进行了两次额外的方法调用。每次方法调用都会花费额外的时间，并且每次调用的结果都必须由其调用者检查，这导致需要考虑更多特殊情况。第7章讨论了当您从一层传递到另一层时，抽象通常应该如何改变，但是图20.2中的所有三个方法都具有相同的签名，并且它们提供基本相同的抽象；这是一个危险信号。Buffer::allocateAppend几乎是一个传递方法；它唯一的贡献是在需要时创建一个新的分配。额外的层使代码更慢且更复杂。

为了解决这些问题，我们重构了 Buffer 类，使其设计围绕性能最关键的路径展开。我们不仅考虑了上面的分配代码，还考虑了其他几个常用的执行路径，例如检索当前存储在 Buffer 中的数据总字节数。对于每个关键路径，我们都试图确定在常见情况下必须执行的最少量代码。然后，我们围绕这些关键路径设计了类的其余部分。我们还将本书中的设计原则应用于简化整个类。例如，我们消除了浅层，并创建了更深层次的内部抽象。重构后的类比原始版本小 20%（1476 行代码，而原始版本为 1886 行）。

```
char* Buffer::alloc(int numBytes)
{
    char* data = allocateAppend(numBytes);
    Buffer::Chunk* lastChunk = this->chunksTail;
    if ((lastChunk != NULL && lastChunk->isInternal()) &&
            (data - lastChunk->length == lastChunk->data)) {
        // Fast path: grow the existing Chunk.
        lastChunk->length += numBytes;
        this->totalLength += numBytes;
    } else {
        // Creates a new Chunk out of the allocated data.
        append(data, numBytes);
    }
    return data;
}

// Allocates new space at the end of the Buffer; uses space at the end
// of the last current allocation, if possible; otherwise creates a
// new allocation. Returns a pointer to the new space.
char* Buffer::allocateAppend(int size) {
    void* data;
    if (this->allocations != NULL) {
        data = this->allocations->allocateAppend(size);
        if (data != NULL) {
            // Fast path
            return data;
        }
    }
    data = newAllocation(0, size)->allocateAppend(size);
    assert(data != NULL);
    return data;
}

// Tries to allocate space at the end of an existing allocation. Returns
// a pointer to the new space, or NULL if not enough room.
char* Buffer::Allocation::allocateAppend(int size) {
    if ((this->chunkTop - this->appendTop) < size)
        return NULL;
    char *retVal = &data[this->appendTop];
    this->appendTop += size;
    return retVal;
}
```

图 20.2：使用内部块在 Buffer 末尾分配新空间的原始代码。

```
char* Buffer::alloc(int numBytes)
    if (this->extraAppendBytes >= numBytes) {
        // There is extra space at the end of the current
        // last chunk, so we can just allocate the new
        // region there.
        Buffer::Chunk* chunk = this->lastChunk;
        char* result = chunk->data + chunk->length;
        chunk->length += numBytes;
        this->extraAppendBytes -= numBytes;
        this->totalLength += numBytes;
        return result;
    }

    // We're going to have to create a new chunk.
    ...
}
```

图 20.3：在 Buffer 的内部块中分配新空间的新代码。

图 20.3显示了在 Buffer 中分配内部空间的新关键路径。新代码不仅更快，而且更易于阅读，因为它避免了浅层抽象。整个路径在单个方法中处理，并且它使用单个测试来排除所有特殊情况。新代码引入了一个新的实例变量，extraAppendBytes，以简化关键路径。此变量跟踪 Buffer 中最后一个块之后有多少未使用的空间可用。如果没有可用空间，或者 Buffer 中的最后一个块不是内部块，或者 Buffer 根本不包含任何块，则extraAppendBytes为零。图 20.3中的代码表示处理此常见情况的最少代码量。

注意：可以通过在需要时从各个块重新计算总 Buffer 长度来消除对totalLength的更新。
但是，对于具有许多块的大型 Buffer 来说，这种方法会很昂贵，并且获取总 Buffer 长度是另一个常见操作。因此，我们选择在alloc中添加少量额外开销，以确保 Buffer 长度始终立即可用。

新代码的速度大约是旧代码的两倍：使用内部存储将
1 字节字符串附加到 Buffer 的总时间从 8.8 纳秒降至 4.75 纳秒。由于修订，许多其他 Buffer 操作也加快了速度。例如，构造新 Buffer、在内部存储中附加一个小块并销毁 Buffer 的时间从 24 纳秒降至 12 纳秒。

20.5 结论

本章最重要的总体教训是，清晰的设计和高性能是兼容的。Buffer类的重写将其性能提高了2倍，同时简化了其设计，并将代码大小减少了20%。

复杂的代码往往运行缓慢，因为它会执行多余或冗余的工作。另一方面，如果你编写的是干净、简洁的代码，你的系统可能已经足够快了，以至于你根本不必担心性能问题。在少数确实需要优化性能的情况下，关键仍然是简洁：找到对性能至关重要的关键路径，并使其尽可能简单。

第21章

结论

本书是关于一件事的：复杂性。处理复杂性是软件设计中最重要的挑战。它使得系统难以构建和维护，并且常常也使得它们运行缓慢。在本书中，我试图描述导致复杂性的根本原因，例如依赖性和模糊性。我讨论了一些可以帮助你识别不必要复杂性的危险信号，例如信息泄露、不必要的错误条件或过于通用的名称。我提出了一些你可以用来创建更简单软件系统的一般性想法，例如努力创建既深刻又通用的类，将错误定义为不存在，以及将接口文档与实现文档分离。最后，我讨论了产生简单设计所需的投资心态。

所有这些建议的缺点是，它们会在项目的早期阶段增加额外的工作。此外，如果你不习惯考虑设计问题，那么在学习好的设计技巧时，你的速度会更慢。如果对你来说，唯一重要的是尽快让当前的代碼工作，那么考虑设计就会显得像是繁重的工作，阻碍你实现真正的目标。

另一方面，如果良好的设计对你来说是一个重要的目标，那么本书中的想法应该会让编程更有趣。设计是一个引人入胜的谜题：
如何用最简单的结构解决一个特定的问题？探索不同的方法很有趣，发现一个既简单又强大的解决方案是一种很棒的感觉。一个干净、简单和明显的设计是一件美好的事情。

此外，您对优秀设计的投资将很快获得回报。
你在项目开始时精心定义的模块将在以后节省你的时间，因为你可以一遍又一遍地重复使用它们。六个月前你编写的清晰文档将在你返回代码添加新功能时节省你的时间。你花在磨练设计技能上的时间也将得到回报：

随着你的技能和经验的增长，你会发现你能越来越快地产生好的设计。一旦你知道方法，好的设计实际上并不比快速而粗糙的设计花费更多的时间。

成为一名优秀设计师的回报是，你可以在设计阶段花费更多的时间，这很有趣。糟糕的设计师大部分时间都在复杂而脆弱的代码中追逐错误。如果你提高了你的设计技能，你不仅能更快地生产出更高质量的软件，而且软件开发过程也会更加愉快。

索引

设计原则总结

以下是本书讨论的最重要的软件设计原则：

1. 复杂性是递增的：你必须关注小细节（见p. 11）。
2. 可运行的代码是不够的（见p. 14）。
3. 不断进行小额投资以改进系统设计（参见第15页）。
4. 模块应该深入（参见第22页）
5. 接口的设计应使最常见的用法尽可能简单（参见第27页）。
6. 对于一个模块来说，拥有一个简单的接口比拥有一个简单的实现更重要（参见第55页，第71页）。
7. 通用模块更深入（参见第39页）。
8. 分离通用代码和专用代码（参见第62页）。
9. 不同的层应该有不同的抽象（参见第45页）。
10. 将复杂性向下传递（参见第55页）。
11. 将错误（和特殊情况）定义为不存在（参见第79页）。
12. 设计两次（参见第91页）。
13. 注释应该描述代码中不明显的内容（参见第页）。101).
14. 软件的设计应该易于阅读，而不是易于编写（参见p）。149).
15. 软件开发的增量应该是抽象，而不是功能（参见p. 154）。

红旗总结

以下是本书中讨论的一些最重要的红旗。系统中出现以下任何症状都表明系统设计存在问题：

浅模块：类或方法的接口并不比它的实现简单多少（参见pp. 25, 110）。

信息泄露：一个设计决策反映在多个模块中（参见p。31).

时间分解：代码结构基于
操作的执行顺序，而不是信息隐藏（参见p. 32）。

过度暴露：一个API迫使调用者了解很少使用的特性，以便使用常用特性（参见p. 36）。

传递方法：一种几乎什么都不做的方法，除了将其参数传递给另一个具有类似签名的方法（参见p. 46）。

重复：一段重要的代码被一遍又一遍地重复（参见p. 62）。

特殊-通用混合：专用代码没有与通用代码完全分离（参见p. 65）。

连接方法：两种方法有太多的依赖关系，以至于很难理解其中一种方法的实现，而不理解另一种方法的实现（参见p. 72）。

注释重复代码：注释中的所有信息都可以从注释旁边的代码中立即明显地看出（参见p. 104）。

实现文档污染接口：接口注释描述了事物用户不需要的实现细节（参见p. 114）。

模糊的名称：变量或方法的名称非常不精确，以至于没有传达太多有用的信息（参见p. 123）。

难以选择名称：很难为实体想出一个精确而直观的名称（参见p. 125）。

难以描述：为了完整起见，变量或方法的文档必须很长。（参见p. 131）。
不明显的代码：一段代码的行为或含义不容易理解。（参见p. 148）。

关于作者

约翰·奥斯特豪特是斯坦福大学的博萨克·勒纳计算机科学教授。他是 Tcl 脚本语言的创建者，并且以其在分布式操作系统和存储系统方面的工作而闻名。奥斯特豪特获得了耶鲁大学的物理学学士学位和卡内基梅隆大学的计算机科学博士学位。他是美国国家工程院院士，并获得了众多奖项，包括 ACM 软件系统奖、ACM 格蕾丝·穆雷·霍珀奖、国家科学基金会总统青年研究员奖和加州大学

伯克利杰出教学奖。