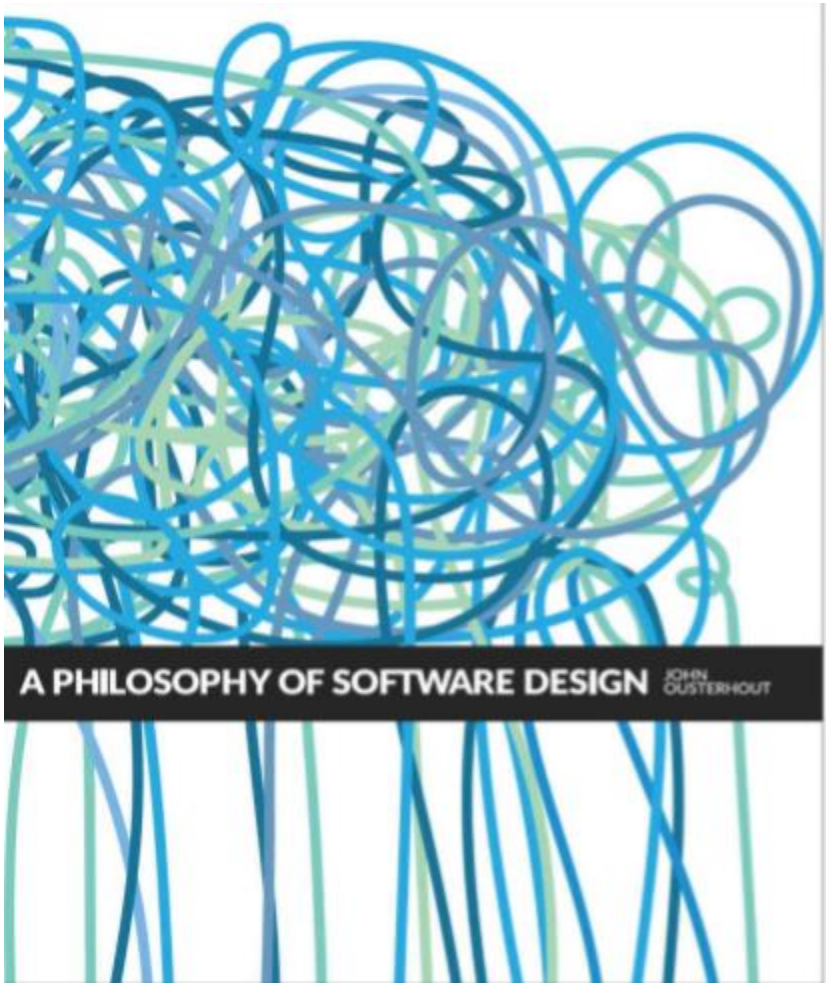

软件设计哲学

A Philosophy of Software Design

[F：John Ousterhout　　T：远方-Qu]

简单的几句话：

经历了疫情，断断续续的持续了大约 1 年半的时间，阅读+翻译了本书，本书完美诠释了软件设计的本质--如何解决复杂性？围绕这个核心不断的抽丝剥茧，深入浅出。

通过阅读本书你将充分认识和理解什么是软件的设计，怎么做设计？

因为个人水平有限，专业述语较多，语言篇幅较大，可能翻译的不够准确，大家可以多多提供建议 To:（mail: qystar@live.cn）；感谢互联网的一些参考资料和英语大辞典~

本翻译读本不以任何商业为目的，只作为学习之用。

2021-03-30

目录

前言

80 多年来人们一直在使用电子计算机编写程序，但是关于如何设计这些程序，以及好的程序看起来应该像什么，都没有很好的定论。关于软件开发的过程（例如敏捷（agile）开发）和关于开发工具（例如调试，版本控制系统，和测试覆盖工具），这些都已经有了大量的讨论。并广泛的分析了这些编程技术，例如面向对象的编程和函数式编程，以及设计模式和算法。所有的这些讨论都是有价值的，但是软件设计的核心问题仍然没有被触及。David Parnas 在他的经典论文(classic paper) "关于将系统分解为模块的准则" 被发表于 1971 年,但是在接下来的 45 年里，软件设计的最新进展并没有超越这篇论文。

在计算机科学中最基本的问题是问题的分解：怎样把一个复杂的问题分解成可以独立解决的几个部分。问题分解是程序员们每天都要面对的核心设计任务，然而，除了这里描述的工作之外，我还没有在任何一所大学里找到一个以问题分解为中心主题的课程。我们教循环和面向对象编程，但是不教软件设计。

此外，程序员之间在质量和生产率上存在着巨大的差异，但我们很少尝试去理解是什么让最好的程序员变得更好或者在我们的课堂上教授这些技能。我曾与一些我认为最优秀的程序员交谈过，但是他们大多数都很难明确表达出能够给他们带来优势的特定技术。很多人认为软件设计技能是一种无法教授的天赋。然而，有相当多的科学证据表明，在许多领域的杰出表现更多地与高质量的实践有关，而不是与生俱来的能力（例如,Geoff Colvin 高估了人的才能）。

多年来，这些问题一直困扰着我。我想知道软件设计是否可以教授，我假设设计技巧是区分好的程序员和一般的程序员的关键。我最终决定，回答这些问题唯一的方法就是尝试教授计算机设计类的课程。结果就是斯坦福大学的 CS 190. 在这套课上，我提出了一系列的软件设计的原则。然后，学生通过一系列的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英语课的写作课。在英语课堂上，学生们用迭代的过程写了一份草稿，获取反馈，然后重写以做出改进。在 CS 190 中，学生们从零开始开发了大量的软件。我们通过广泛的代码评审来识别设计问题，学生修改他们的项目来解决问题。这使学生能够看到如何应用设计原则来改进他们的代码。

我现在已经教了三次软件设计课，这本书是从课堂上基于设计原则而产生的。在哲学上这些原则有着相当高的水准和边界（"脱离存在的错误"），因此学生们理解这些抽象的思想会很难。学生们通过编写代码，犯错误，然后查看他们的错误和随后的修正是如何与原则联系起来的，从而学的更好。

通过这些点你可以很好的弄清楚：为什么软件设计能够使我知道所有的答案？老实说，我不知道.。在我学习编程的时候没有关于软件设计的课程，也没有导师来教我软件设计。在我学习编程期间，代码审查实际上是不存在的。我对关于软件设计的看法来自于编写和阅读代码的个人经验。在我的职业生涯中，我用各种语言写了大约 250000 行代码。在我创建的团队里，从 0 创建了三个操作系统，多个文件系统和存储系统，基础架构的工具集；例如调试器，构建系统，和 GUI 工具套件，一个脚本语言和交互的文本编辑器，画图，演示和集成电路。在此过程中，我亲身经历了这些大型系统的问题，并尝试了各种设计技术。此外，我还阅读了相当多数量别人写的代码，这让我接触到了各种方法，有好有坏。

在这所有的经验中，我试图提取一些常用的思路，包括关于避免失误和使用技巧。 这本书映射了我的个人经验：这里描述的每一个问题都是我个人已经经历过的，每一个技术建议都是在我写代码的过程中成功应用过的。

我不指望这本书在软件设计上成为最后的定论； 我确信这是一个不容错过的有价值的技术书籍，从长远来看我的一些建议可能会变成一些糟糕的主意。然而，我希望这本书将开启关于和软件设计的对话。比较这本书的方法结合你自己的经验，来决定这里描述的方法是否真的能够减少软件的复杂度。这本书是一篇观点文章，因此一些阅读者会不太同意我的一些建议。如果你不同意，就试着理解为什么。我很想听到关于对你有用的建议和没有用的建议，以及你对关于软件设计的其他想法。我期望接下来的对话能提高我们对软件设计共同的理解。我也会把我学到的东西融入到这本书的未来版本中。

第一章　　介绍

(所有的都是关于复杂性)

在人类的历史中，写计算机软件是纯粹的创造性活动之一。程序员们在实际的限制上没有边界，例如：物理定律；我们可以创造出一个令人振奋的虚拟世界，在真实的世界中这些是不可能存在的。写程序不需要很棒的物理技能或协作，像芭蕾或篮球。所有的编程都需要有创造性的思维和组织思维的能力。如果你能构思出一个系统，你就很可能用计算机程序来实现它。

这意味着编写软件的最大限制是理解我们正在创建系统的能力。随着程序的发展和获取更多的特性，它将变得复杂，组件之间存在着微妙的依赖关系。随着时间的推移，复杂性累积变得越来越厉害，程序员在修改系统的时候，将会变得越来越困难。这会减缓开发效率并增加错误率，使开发效率变慢并增加成本。复杂度在任何程序的生命周期中都不可避免的增加了。开发越大的程序，工作人员越多，管理的复杂性就会越困难。

好的开发工具可以帮忙我们处理这些复杂性，并且在过去的几十年中已经创建了很多优秀的工具。但是单靠这些工具我们能做的是有限的。如果我们想更容易的写软件，那么我们可以构建更多便宜而且健壮的系统，我们必须找到使软件更易于理解的方法。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不断增加，在复杂性变得巨大之前，易于理解的设计允许我们构建相对比较大和更多健壮的系统。

这里有两个普通的方法来和复杂性做斗争，它们两个都将在这本书中做讨论。第一个方法是消除复杂性可以使代码比较简单和更利于理解。例如，在一致性的方式中消除特殊情况或使用标识符可以减少复杂性。

复杂性的第二个方法是把它封装起来，这样程序员们就可以在一个系统上工作，而不会立刻暴漏所有的复杂性。这个方法也可以被叫做模块设计。在模块设计中，一个软件系统被分成了几个模块，例如面向对象语言中的类。模块被设计成互相之间可以相对独立，那么程序员就可以在一个模块上工作，就不用理解其它模块的细节。

因为软件是可延展的，软件设计的连续过程是贯穿整个软件系统生命周期的；这使得软件系统的设计不同于物理系统，例如建筑，船舶或桥梁的设计。然而，软件设计并没有总是参考这些方法。在编程的大部分历史中，设计都集中在项目的开始，就像其它的工程学科一样。这种极端的方法被称为瀑布模型。在瀑布模型中的项目被拆分成不同的阶段，例如：需求定义，设计，编码，测试和维护。在瀑布模型中，每个阶段的完成都是下一个阶段的开始；在很多的情况下不同的人负责不同的阶段。在设计阶段期间，整个系统是一起被设计的。这个阶段结束时，设计被冻结，随后的阶段是补充和实现那些设计。

很遗憾，瀑布模型在软件上效果并不明显。软件系统在本质上比物理系统更复杂；在构建任何东西之前，构思设计大的软件系统以满足理解所有的实现是不可能的。因此，最初的设计将会有很多问题。这些问题直到实现正在进行中才会显现出来。然而，瀑布模型的结构不能适应这种主要设计的变更，在这些点上（例如，设计师会被转移到其它项目上）。因此，开发人员试着修补这些问题并不能改变整体的设计。这会导致复杂性的爆发。

因为这些问题，大多数的软件开发项目今天都用递增的方法，例如敏捷开发，初始的设计聚焦在整体功能中的一个小的子集。这个子集是被设计，被实现，和被评估的。 问题在初始设计时就会被发现和纠正，然后更多的特性会被设计，实现和纠正。每一次的迭代都会暴露已存在设计的问题，并会在下一个集合特性被设计之前解决它们。以这种方式展开设计，初始设计的问题可以在系统很小的时候就会被解决；后期特性得益于在早期特性实现期间而获取的经验，因为他们的问题较少。

增量的方法适用于软件，因为软件具有足够的可塑性，允许在部分的实现过程中进行重大的设计更改。相反，重大的设计变更对于物理系统更具有挑战性：如，正在施工中的大桥，改变支撑塔的数量是不现实的。

递增的开发表示软件设计永远不会完成。设计发生在一个系统的生命周期里：程序员需要一直关注设计的问题。递增的开发也代表了持续的重新设计。一个系统或组件在初始设计的时候几乎永远都做不到最好；经验不可避免的能够把做好事情的方法展示出来。作为一个软件开发者，你在工作中应该经常寻找机会来改善系统的设计，并且你应该计划花费一些极少的时间在改善设计上。

软件开发者应该经常思考关于设计的问题，和减少大多数重要元素软件设计的复杂性，那么软件开发者应该经常的思考复杂性。这本书就是关于怎样用复杂性的指标来引导软件设计贯穿整个生命周期。

这本书有两个全局目标。第一个是描述软件复杂性的本质："复杂性"表示什么，为什么重要，你怎么辩识程序在什么时候存在不必要的复杂性？第二，更多的挑战，目标是你可以用现存的技术在软件开发过程期间使复杂性最小化。不幸的是，没有一个简单的方法可以保证软件设计的优秀。相反，我将提出一系列更高层次的概念，和哲学有关系的。例如："复杂性应该隐藏在类中实现" 或 "定义不复存在的错误"。这些概念不能立即马上定义出更好的设计，但是你可以用他们对比设计的方案和引导你探索设计空间。

如何使用本书

这里很多设计原则被描述的有些抽象，因此不看实际的代码很难理解它们。在书中存在的实例都很小，但是大的实例更足以说明真实系统的问题（如果你遇到好的例子，请发送给我）。因此，这本书还不足以让你学会如何灵活运用这些原则。

使用这本书最好的方法是结合代码审核。当你读其他人的代码的时候，应该思考一下这里讨论的概念是否符合，并且如何跟代码的复杂性关联起来。在别人的代码中比在自己的代码中更容易看到设计的问题。在这里你可以用红色来描述定义的问题和改进的建议。审查代码也可以向你展示新的设计方法和编程技术。

改进你的设计技能最好的方法之一是学习辨别红色信号：标记一段代码可能比需要更复杂。在这本书中我将用红色标记出那些建议问题和每个主要设计问题的关联；大多数重要的总结都在书的后面。你可以在你写代码的时候使用它们：当你看到一个红色标记的时候，停止并寻找一个可替代的设计来排除问题。当你首次尝试这个方法的时候，在找到一个排除红色标记之前，你可以尝试几个可替代的设计方案。不要轻易放弃：在问题解决之前你尝试选择的越多，你学到的就会越多。随着时间的推移，你会发现你代码的红色标记越来越少，设计越来越整洁。你的经验也将用于识别其它设计的红色标记（我很高兴听到关于这方面的信息）。

当你运用这本书的观点时，适度和谨慎很重要。每条规则都有例外，每个原则都有局限性。如果你把任何的设计理念都发挥到极致，很可能会陷入一个糟糕的境地。漂亮的设计反射出在构思和方法相互之间冲突的平衡。几个章节有几节标题"太过头了"，描述了当你做一件好事时该如何辨别。

本书中几乎所有的例子都是用 JAVA 和 C++编写的，很多的讨论都是用面向对象的语言设计的。然而，这些想法也适用于其它领域。几乎所有与方法相关联的思想也可以适用于非面向对象特性的函数式编程语言，例如 C 语言。设计思想也适用于除此以外的模块，例如子系统和网络服务。

在此背景下，我们将讨论更详细的复杂性的原因，以及如何使软件系统更简单。

第二章 复杂性的本质

这本书是一本关于如何设计最小复杂性的软件系统。第一阶段是理解敌人。扩展一下什么是"复杂性"？如果系统存在不必要的复杂性你将怎样？什么原因使系统变得复杂？这一章节将演说那些高层次的问题；随后的章节将帮忙你在特殊结构特性的项目中辨别低层次的复杂性。

辨别复杂性的能力是一个极其重要的设计技能。它允许你在投入大量的精力之前识别出问题，并允许你在多种选择中做出正确的选择。判断一个设计是否简单要比创建一个简单的设计要容易的多，但是一旦你意识到一个系统太复杂，你就可以用这种能力引导你的设计理念走向简单化。如果你发现设计很复杂，试着换种不同的方法看一下是否更简单。随着时间的推移，你会注意到确实有些技术可以使其趋向于简单化的设计，而另一些则与复杂性相关。这些允许你能够更快速的产出更简单的设计。

本章还提供了一些基本假设，为本书的其余部分奠定了基石。后面的章节也将依照本章给出的材料作为参考，以用于证明各种改进和结论。

2.1 复杂性的定义

为了描述本书存在的目的，我用了一种实际的方法来定义"复杂性"。<复杂性的任何东西都与软件系统结构相关，这就使得系统的理解和修改很困难>。复杂性可以有多种形式。例如，很难理解一段代码是如何工作的；实现一个小的改进可能都要付出很多的努力，或者可能不清楚系统的哪一部分必须要修改才会使其改进；修复一个错误而不会引入另一个错误可能很困难。如果软件系统很难理解和修改，那么就是太复杂了；如果很容易理解和修改，那么就很简单。

你也可以从成本和效益方面考虑复杂性。在一个复杂的系统中,实现哪怕很小的改进都需要做很多的工作。在一个简单的系统中，较大的改进可以用更少的精力来实现。

复杂性是开发者在试着完成特定目标的时候在特定时间点的经验总结。它不一定与系统的整个大小和功能有关。人们经常用“complex”来描述复杂功能的大型系统，但是如果这个系统很容易操作，那么，就这本书而言，并不复杂。当然，几乎全部大而复杂的软件系统实际上都很难工作，如此，他们也碰到了我对复杂性的定义，但事实并非如此，一个是小而简单的系统可能也会十分复杂。

复杂性是由最通用的活动决定的。如果系统有几个部分很复杂，但是那几个部分几乎不会触及【基本不改动】，那么他们就不会影响整个系统的复杂性。要用粗略的数学方法来描述这一点：

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

模块的复杂性 = 模块认知负担 * 模块开发时间

系统的整体复杂性（C）是由每个部分的复杂性 p(Cp)的权重值乘以开发那部分（tp）所花费的时间决定的。在一个永远看不到的地方孤立复杂性几乎和消除复杂性完全一样好。

对于读者来说，复杂性比作者写的更为显而易见。如果你写了一段代码对于自身来讲看起来很容易，但是其他人认为很复杂，那么它就很复杂。当你发现这种情况的时候，就有必要追问其他的开发人员找出为什么他们看起来代码如此复杂；从你和他们的意见脱节中，也许可以学到一些有趣的教训。作为开发者，你的工作不仅仅是创建容易使用的代码，而是创建其他人也可以容易使用的代码。

2.2 复杂性的症状

复杂性表现为三个常规的方法，在下面的段落里描述。每一种表现形式都使得开发任务难以执行。

散弹式修改（当只需要修改一个功能，但又不得不对许多模块做出改动时，我们称之为霰弹式修改。）：复杂性的第一个症状是看起来一个简单的改变需要更改多个不同地方的代码。例如，考虑到一个包含多个页面的网站，每一个页面都显示一个背景颜色的横幅。在一些早期的网站，每个页面的颜色都显式指定。如 Figure 2.1(a)所示；为了改变网站的背景，开发人员需要手动修改每一个存在的页面；一个大的网站有上千个页面，这几乎是不可能的。幸运的是，现代的网站用的方法类似于图 2.1(b)，横幅颜色在中心位置只指定一次，所有单独的页面都引用共享值。用这个方法，一个单独的页面修改就可以改掉整个网站的横幅颜色。一个好的设计的目标之一就是减少每一个设计决策所影响到的代码量。

认知负担：第二个复杂性的特征是认知负担，它指的是为了完成任务，一个开发人员需要了解多少。一个较高的认知负担代表了开发人员需要花费更多的时间来学习所需的信息，而且因为遗漏了一些重要的信息，因此出现错误的风险更大。例如，假设一个函数在 C 语言中分配了内存，返回了该内存的指针，并假设调用方将内存给释放掉。这增加了开发人员使用该函数的认知负担；如果开发人员释放内存失败，那么将会出现内存泄漏。如果可以重构系统，以便调用者不需要担心内存的释放（负责分配内存的同一个模块也释放内存），这样就减少了认知负担。认知负担以多种方式产生，例如有很多方法的 API，全局变量，不一致，和模块间的依赖。

系统的设计者有时会假设哪些复杂性能够用代码行数来衡量。他们假设如果一个实现的代码行数比另一个短，那么他们一定很简单；如果只需要更改几行代码，那么一定很容易。然而，这种观点忽略了与认知负担相关联的成本。我见过一些只允许用几行代码写的应用的框架，但要理解这些代码是极其困难的。有时一个方法需要更多的代码行数实际上更简单，因为他减少了认知负担。

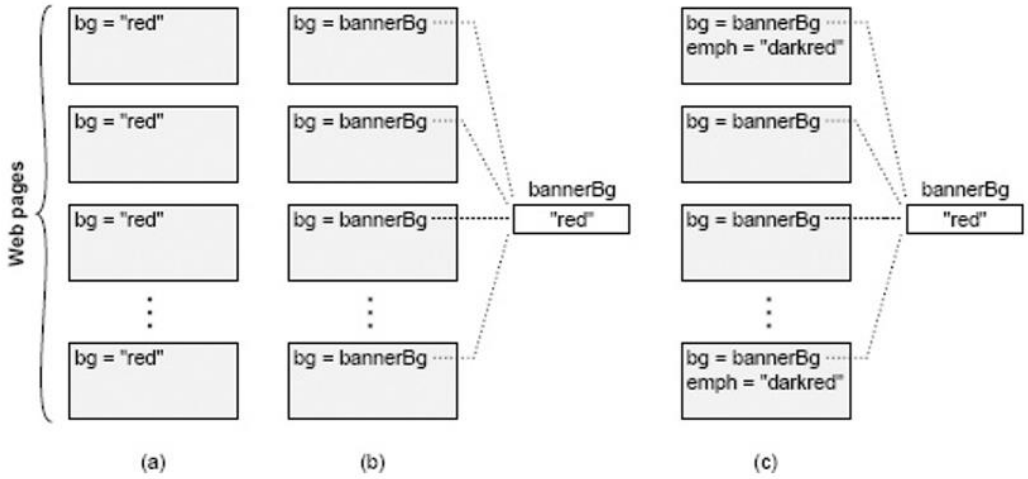

Figure 2.1: 每一个页面在网站中都显示了横幅的颜色。在(a)的背景颜色中，每一个页面横幅的背景颜色都是特别明确的。在（b）中共享了一个变量保留了背景颜色，并且每个页面都引用那个变量。在（c）中一些页面强调显示额外的颜色，即横幅的背景颜色有较深的阴影；如果背景的颜色要改变，那么强调的颜色也必须改变。

不确定性: 第三个复杂性的特征是不清楚哪些代码为了完成任务必须要被修改，或者开发人员必须具备什么信息才能成功的执行任务。Figure 2.1（c）说明了这个问题。网站用一个中心变量来定义横幅背景色，如此看上去改变很容易。无论如何，一些网站页面强调使用深色阴影作为背景颜色，在独立的页面中，那些深颜色是特别明显的。如果背景颜色改变，那么强调的颜色必须做相应的改变。不幸的是，开发者们不太能够意识到这点，因此他们可以在不更新强调颜色的情况下来更改中心 bannerBg 变量。甚至如果开发者意识到了这个问题，他也不清楚哪一个页面使用强调的颜色，因为开发者需要在网站中查找每一个页面。

在复杂性的三种表现中，不确定性是最差的。不确定性代表了你虽然知道一些事情，但是你没有办法弄清楚是为什么，甚至你没有弄清楚是否有问题。当你更改了之后，直到错误的出现你都查不出来原因。

散弹式修改是很烦的。但是只要清楚哪些代码需要被修改，系统就会在更改被完成后工作。类似的，一个高的认知负担将增加更改的成本，但如果清楚要阅读哪些信息，改变仍然可能是正确的。在不确定的情况下，我们不清楚该做什么，并且建议的解决方案是否可行。唯一确定的方法就是阅读系统中每一行代码。这对于任何规模的系统来说是不可能的。即使这样也可能不够。因为一个改变可能依赖于另一个从未记录在案的微妙设计决策。

好的设计最重要的目标之一就是让系统变的易理解。这与高认知负担和不确定性是相反的。在一个易理解的系统中，开发人员可以快速的理解现有的代码是如何工作的，以及做出什么样的改变。易理解的系统是一个开发人员能够快速的猜出该做什么，而不需要很难的思考过程，而且要确信猜测是正确的。Chapter 18 讨论的技术能够使更多的代码易于理解。

2.3 复杂性的原因

既然你知道复杂性的高级特征和复杂性为什么使软件开发变得困难，那么下一步就是要了解复杂性的原因是什么，这样我们就可以设计系统来避免这些问题。复杂性是由两件事情引起的：依赖和隐晦。这一节讨论一些高级要素；随后的章节将讨论如何将他们和低级的设计决策关联起来。

出于本书的目的，当给定的一段代码不能被理解和修改时，就会存在依赖；代码在一定程度上与其他的代码相关，如果给定的代码被改变了那么其他的代码必须考虑用 and/or 来修改。在网站中例如 Figure 2.1(a)，背景颜色的创建跟所有的页面之间都有依赖。所有的页面都需要有相同背景，因此如果一个页面背景被改变了，那么他们全部都必须做更改。另一个依赖的例子出现在网络协议中。通常协议的发送方和接收方都是独立的代码，但是他们必须互相遵守协议；改变发送者的代码几乎总是在接收方也需要做相应的改变，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生成依赖：如果在一个方法里添加了新的参数，所有调用那个方法的参数都必须被修改。

依赖关系是软件的基本组成部分和不能完全被消除。实际上，我们有意将依赖性作为软件设计过程的一部分引入。每次写一个新类时都会围绕着该类的 API 创建依赖。然后，软件设计的目标之一是减少依赖的数量，使依赖尽可能的保持简单和易于理解。

以网站为例。在老的网站中每个页面的背景都是单独说明的，所有的页面都互相依赖。新的网站指定背景颜色这个问题已经被解决了，在一个中心位置和提供了独立页面的 API 检索那些颜色，当他们在生成的时候。新的网站消除了页面之间的依赖，但是为了检索背景颜色在 API 的周围也创建了新的依赖。幸运的是，新依赖多个是易于理解的：在横幅背景颜色上他们清除了每个单独页面依赖，开发者可以很容易的用查找名称变量来找到所有的位置。此外，编译器能够帮助管理 API 的依赖：如果共享变量的名字发生了改变，任何仍然使用旧名字的代码都将出现编译错误。新网站用一个简单和更易理解的依赖关系替代了一个不易理解和管理困难的依赖关系。

复杂性的第二个原因是隐晦。当重要信息不易理解的时候就会出现隐晦。一个简单的实例是一个通用的变量名称它不包含很多有用的信息（例如：时间）。或者，变量的文档标注可能没有指定单位，所以唯一的方法就是扫描代码中使用变量的位置。 易理解性通常与依赖性有关联，一个依赖的存在则是不易理解的。例如，如果在一个系统中添加了一个新的错误状态，则可能必需为每个状态添加一个包含字符串消息的表的条目，但消息表的存在对于查找状态定义的程序员来说可能也不易理解。非一致性也是隐晦主要的贡献者：如果一个相同的变量名被用于两个不同的用途，这些特定变量服务的用途对于开发人员来说是很难理解的。

在很多情况中，因为文档的不足也会导致隐晦；Chapter 13 处理了这个话题。然而，隐晦也是一个设计问题。如果一个系统已经是一个整洁和易理解的设计，那么他将需要很少的文档来描述。对于大量文档的需求经常是一个红色标记，也就是说设计不是十分正确。一个好的方法是简单的系统设计可以减少隐晦。

交互，依赖性和隐晦三个表现，在 Section2.2 复杂性中被描述。依赖性导致了高的认知负担和改变扩大。隐晦造成了不确定性，还有促使认知负担。如果我们能够找到技术设计的最小依赖和隐晦，那么我们就可以减少软件的复杂性。

2.4 复杂性是递增的

复杂性不是由一个单一的灾难性错误引起的；它是由多个小块逐渐累积而成的。单个依赖和隐晦，

对于自身而言，他不太可能对一个软件系统的可维护性产生明显的影响。复杂性的产生是因为随着时间的推移由成百上千的小依赖和隐晦逐步增长的。最终，这些小问题会有很多，系统中每一个可能的改变都会受到他们其中几个问题的影响。

复杂性的递增性质很难被控制。很容易让自己相信，当前的变化带来的一点复杂性并不是什么大事。然后，如果每一个开发者为了每一个改变都用这个方法，复杂性会迅速聚积。一旦复杂性聚积，那么将很难消除，解决掉单个依赖和隐晦，对本身来讲，并没有很大的差异。为了让复杂的增长慢一些，你必须采取“零容忍”的哲学，如 Chapter 3 所讨论的。

2.5 结论

复杂性来自于依赖和隐晦的不断聚积。随着复杂性的递增，导致变化的扩大，高的认知负担和不确定性。因此，它使得实现每一个特性都要修改更多的代码。另外，开发者要花费更多的时间来获取足够的信息来使更改安全，最坏的情况下，他们甚至不能找到他们需要的全部信息。复杂性的底线是使更改现有的代码库将变得困难和危险。

第三章　工作中的代码是不充分的　（系统不仅仅是工作，还要有好的设计）

战略与战术编程

一个好的软件设计最重要的元素之一就是当你在处理一个编程任务时所采用的思维方式。很多组织鼓励战术思维方式，聚焦于让一些功能尽可能快的工作。然而，如果你希望有一个好的设计，就必须使用一种更战略性的方法。在这种方法中，你需要投入时间来生产整洁的设计和解决问题。这一章讨论的是为什么战略方法会产出更好的设计，从长远来看，实际上比战术方法更廉价。

3.1　战术编程

大多数的程序员以一种是被我叫做战术编程的思维方式进行软件开发。在战术方法上，你的主要精力是做些什么，例如一个新的特性和修复一个问题。乍看上去这似乎完全是合理的：还有什么比编写有效代码更重要呢？无论如何，战术编程似乎不太可能产出好的系统设计。

战术编程的问题在于它是短视的。如果你正在使用战术编程，并正在试着尽可能快的完成一个任务。或许你的最后期限很紧。因此，未来的计划并不是当务之急。你不用花太多的时间去寻找最好的设计；你只是想快点做点什么。你告诉自己增加一些复杂性或引入一个或两个小的撇脚的系统也是可以的，如果允许当前的任务能够更快的完成的话。

这就是系统变的复杂的原因。正如前一章所讨论的，复杂性是递增的。并不是一件特别的事情就让系统变得复杂，而是由几十个或上百个的小事情积累而成。在战术编程上，每一个编程任务都将贡献一些复杂性。为了快速的完成当前的任务，他们中的每一个看起来都像是一个折中方案。然而，复杂性迅速累积，特别是如果每一个都是战术编程。

不久以后，一些复杂的问题将会引起一些问题，同时你会希望没有采取这些早期的捷径。但是，你要告诉自己获得下一个特性的工作比回过头去重构已经存在的代码更重要。从长远来看重构可能会有帮助，但是当前的任务肯定会慢下来，因此，你需要找到快速补丁来解决遇到的任何问题。这会增加更多的复杂度，那么就需要打更多的补丁。很快代码就会相当的混乱，而且到目前为止很糟糕，可能要用几个月的时间来清理它。你的日程安排不可能容忍这种延迟，解决一个或两个问题看起来并没有什么不同，所以你只要保持战术编程。

如果你有在一个大型的软件项目上工作很长时间，我觉得你已经在工作上经历了战术编程并总结了一些问题。一旦你开始走向战术编程，就很难再改变。

几乎每个软件开发组织都至少有一个开发人员将战术编程发挥到极致：战术龙卷风。战术龙卷风是一个多产的程序员泵出的代码远远快于其他人，但在一个完全战术的工作方式里。在快速实现功能方面，没有人能比战术龙卷风更快完成。在一些组织，管理层视战术龙卷风为英雄一样。然而，战术龙卷风留下了毁灭的痕迹。工程师们很少认为他们是英雄，因为他们将来必须使用他们的代码。通常，其他工程师必须清理战术龙卷风留下的混乱，这使得这些工程师（真正的英雄）的进度似乎比战术龙卷风慢。

3.2　战略编程

成为一名优秀的软件设计师的第一步是认识到工作代码是不够的。为了更快的完成当前任务而引入不必要的复杂性是不可接受的。最重要的事情是系统的长期架构。在任何系统中大多数的代码都是通过基于扩展已经存在的代码来编写的，所以你的最重的工作是使开发人员促进这些将来的扩展。因此，你不应该把“工作代码”作为你的主要目标，当然你的代码必须工作。你的主要目标是创造一个很棒的设计，也是发生在工作中的。这是战略编程。

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，而不是走最快的路线来完成当前的项目。这些投入在短期内会让你慢一点，但在长期内会让你加快，如图 3.1 所示。

有些投资是积极主动的。例如，花点额外的时间为每个新类找一个简单的设计是值得的；而不是实现第一个想到的想法，尝试两种不同的设计并选择最整洁的一种。试着想象一下未来可能需要对系统进行更改的几种方式，并确保这对你的设计来讲是容易的。编写好的文档是主动投资的另一个例子。

其他的投资都是被动的。不管你前期投入多少，你的设计决策都不可避免地会有错误。随着时间的推移，这些错误将会变得很明显。当你发现一个设计问题时，不要忽略或修补它；花点额外的时间来修复它。如果你的程序是战略性的，你将不断地对系统设计做一些小的改进，这与战术性编程相反，你不断地增加一些在未来引起复杂性的问题。

3.3 投入多少?

那么，正确的投入量应该是多少呢？一个巨大的前期投入，例如试图设计整个系统，是不会达到预期结果的。这是瀑布法，并且我们知道这不奏效。随着你对系统的体验，理想的设计往往是逐步显现的。因此，最好的方法是在一个持续的基础上进行大量的小投入。我建议你把总开发时间的大约 10-20%花在投入上。这个数量足够小，不会对你的计划产生重大的影响，但是随着时间的推移会产生足够大的重大的益处。因此你初始化项目将比纯战术方法所花费的时间要长 10-20%。这段额外的时间将带来更好的软件设计，你将在几个月内开始体验这些益处。不久之后你的开发速度至少比战术编程要快 10-20%。此刻你的投入时间会变的自由：你的过去投入收益将节省足够的时间来覆盖未来投入的成本。你将很快收回最初投入的成本。图 3.1 说明了这一现象。

图 3.1：起初，战术编程的方法将比战略方法的进度更快。然后，在战术方法下复杂性积累的更快，从而降低了生产率。 随着时间的推移，战略方法取得了更大的进展。注：此图仅做定性说明；我不知道任何关于曲线精确形状的经验测量。

相反，如果你用战术编程，你会用 10-20%的速度完成你的第一个项目，但是随着时间的推移你的开发速度会随着复杂度的增加而变得慢下来。不久之后，你的编程速度至少会慢 10-20%。你会很快地反思你在一开始节省的所有时间，在系统生命周期的剩余时间里，你的开发速度会比你采取战略方法慢的多。如果你从来没有在一个严重降级的代码库中工作过，请与一个已经经历过的人交谈；他们将会告诉你糟糕的代码质量至少会使开发效率慢 20%。

3.4 创业与投资

在一些环境中，有强大的力量反对这种战略方法。例如，早期初创公司感到巨大的压力而要求他们尽快发布早期版本。在这些公司中，即使是 10-20%的投入也可能无法承受。因此，许多初创公司采取战术方法，在设计上花费很少的精力，在出现问题的时候甚至都较少的整理。他们认为这样是合理的，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事情。

如果你所在的公司正朝着这个方向发展，你应该意识到一旦代码库变成了意大利面，就几乎不可能修复。你可能会为了产品的生命周期支付高昂的研发成本。此外，结果好的（或坏的）设计来的非常快速，所以很有可能战术方法甚至不会加速你的第一个产品发布。

另一个需要考虑的问题是，一个公司成功的最重要的因素之一是其工程师的质量。降低开发成本最好的方法是聘请优秀的工程师：他们的成本不比平庸的工程师高多少，但是确拥有极高的生产效率。然而，最好的工程师非常关心好的设计。如果你的代码库出了问题，消息将很快传出去，这将让你更难招募到新的员工。结果，你很可能会以平庸的工程师收场。这会增加你未来的成本并可能导致系统架构的进一步退化。

Facebook 是一家鼓励战术编程的初创公司的例子。多年来公司的座右铭是“快速行动，破除陈规（Move fast and break things）”。鼓励刚从大学毕业的新工程师立即投入到公司的代码库中；工程师们在工作第一周就投入到产品中是很正常的。从积极的一面来看，Facebook 在一个公司的雇员授权上壮大了声誉。工程师拥有极大的自由度，几乎没有什么规则和限制影响到他们。

Facebook 作为一家公司取得了令人瞩目的成功，但它的代码库却因为公司的战术方法而受到了影响；很多的代码都不稳定和难以理解，很少有注释或测试，而且很痛苦。随着时间的推移，公司意识到这种文化是不可持续的。最终，Facebook 将其座右铭改为“在坚实的基础架构下快速发展（Move fast with solid infrastructure）”以鼓励其工程师们在好的设计中投入更多的时间。Facebook 能否成功解决多年来积累的战术编程问题还有待观察。

公平地说，我应该指出 Facebook 的代码在初创公司中可能并不比平均水平糟糕。战术编程在初创企业之中是很普遍的；Facebook 恰好是一个特别明显的例子。

幸运的是，我们也有可能通过战略方法在硅谷取得成功。谷歌和 VMware 与 Facebook 一样在相同的时间成长，但这家公司都采用了更具战略意义的方法。两家公司都非常重视高质量的代码和良好的设计，两家公司都构建了复杂的产品，用可靠的软件系统解决了复杂的问题。公司强大的技术驱动文化变得人尽皆知在硅谷。很少有其他公司能与他们竞争招聘顶尖的技术人才。

这些案例表明，无论哪种方法，公司都能取得成功。然而，在一家关心软件设计并拥有整洁的代码库的公司工作会更有趣一些。

3.5 结论

解决问题越早越好，它需要你持续的投入，这样小问题才不会积累成大问题。幸运的是，好的设计最终会为你自己带来收益，而且比你想象的要快。

在运用战略方法时保持一致是致关重要的，要把投入当作是今天的事情而不是明天。当你遇到困难的时候，你会很容易把清理工作推迟到困难结束之后。然后，这就是滑坡谬误；在当前困难之后，几乎肯定会有另一个，和另一个的另一个的困难。一旦你开始推迟设计的改进，延迟很容易成为永久性的，并且你的文化也很容易融入到战术方法中。你解决设计问题的时间等待的越长，问题就是越大；解决方案就会越恐怖，这就更容易把问题推迟。最有效的方法是让每一个工程师在良好的设计中持续的进行少量的投入。

第四章　应该深入模块设计(接口应该非常简单)

管理软件的复杂性最重要的技术之一是系统设计，因此开发人员在任何给定的时间只需要面对整体复杂性的一小部分。这种方法被称为模块化设计，本章介绍了其基本原则。

4.1　模块化设计

在模块化设计中，一个软件系统被分解成相对独立的模块集合，模块可以有多种形式，例如类，子系统或服务。在理想的状态下，每个模块都是完全独立的：开发人员可以调用任何模块而不需要知道关于这些模块的任何信息。在这种情况下，一个系统的复杂性就是它最糟糕模块的复杂性。

不幸的是，这一理想是不可能实现的。模块必须通过互相调用其他的函数或方法来协同工作。因此，模块互相之间必须了解一些情况。模块之间会存在依赖：如果一个模块发生了更改，其他的模块也许需要同步更改。例如，方法的参数会在该方法和调用该方法的任何代码之间创建依赖关系。如果所需的参数发生了更改，则必须修改该方法的所有调用以遵守新的签名。依赖项可以采用许多其他形式，并且它们可以非常微秒。模块化设计的目标是使模块之间的依赖最小化。

为了管理依赖关系，我们将每个模块分为两部分：一个*接口*和一个*实现*。接口由在不同模块中工作的开发人员必须知道的所有内容组成，以便使用给定的模块。通常情况下，接口只描述模块*做什么*，而不会包含*怎样做*。完成接口所做出的承诺的代码被称为实现。在一个特定模块内部进行工作的开发者必须知道的信息是：当前模块的接口和实现加上其它被该模块使用的模块的接口。他不需要理解其它模块的实现。

考虑一个实现平衡树的模块。该模块可能包含确保树保持平衡的复杂代码。然后，这种复杂性对于模块的用户来说是不可见的。用户可以看到一个相对简单的接口，用来调用操作对树节点的插入，移除和获取。 若要调用插入操作，调用方只需要提供新节点的键和值；遍历树拆分节点的机制在接口中不可见。

本书的目的，一个模块代码中的任何单元都具有一个接口和实现。在面向对象编程中每一个类都是一个模块。类中的方法或那些非面向对象语言中的函数，也可以看作模块：每一个模块都有接口和实现，模块化设计技术可以把它们应用。更高级的子系统和服务也是模块；它们的接口可以采用不同的形式，例如内核调用或 HTTP 请求。在本书中关于模块化设计的大部分讨论都集中在类的设计上，但是这些技术和概念也适用其他类型的模块。

最好的模块是那些接口比实现简单得多的。例如模块有两个优点。第一，简单的接口最小化了模块施加给系统其余部分的复杂度。第二，如果修改模块时可以不修改它的接口，那么其他模块应不会被修改所影响。如果模块的接口远远比实现简单，那么就更有可能在不改动接口的情况下对模块进行修改。

4.2　接口里有什么?

接口中包含了两种信息：正式的和非正式的。正式的信息在代码中被显式指定，程序语言可以检查其中的部分正确性。比如，方法的签名就是正式的信息，它包含参数的名称和类型，返回值的类型，关于方法异常信息的抛出。很多程序语言都可以保证代码中对方法的调用提供了与方法定义相匹配的参数值。正式接口一个类的签名由全局方法，加上任意全局变量的名称和类型组成。

每个接口也包含非正式元素。非正式部分无法被程序语言理解或强制执行。接口的非正式部分包含了一些高级行为，例如函数会根据某个参数的内容删除具有相应名字的文件。如果一个类的使用存在限制（或许一个方法必须在另一个方法之前调用），那这也属于类接口的一部分。通常，如果开发人员为了使用模块需要知道这些信息，那么这些信息就是模块接口中的一部分。接口的非正式方面只能用注释来描述，程序语言无法确保描述是完整或准确的。大多数接口的非正式信息比正式信息要广泛和更复杂。

清晰的接口定义有助于开发人员了解在使用模块时需要知道的信息，从而有助于消除第 2.2 节中描述“未知的未知”的一些问题。

4.3 抽象

抽象这一术语和模块设计思想的关系相接近。**抽象是实体简化的视图，省略了不重要的细节。**抽象是实用的，因为它们可以使我们对细节的思考和操纵复杂的事情更容易。

在模块化的编程中，每个模块都以接口的形式提供其抽象。接口是模块功能的简化视图；从模块抽象的立场上看实现的细节是不重要的，所以它们从接口上被忽略了。

在抽象的定义中，“不重要”这个词是极其关键的。从抽象中忽略不重要的细节越多越好，然而，如果它不重要，细节就可以从抽象中被忽略掉。抽象可能在两个方面出错。首先，抽象包含的细节可能不是真的重要；当这种情况发生时，会使抽象变得复杂，使用抽象会增加开发人员的认知负担。其次，当抽象真的很重要的时候忽略细节是错误的。这会导致晦涩难懂：只关注抽象的开发人员不会拥有正确使用抽象的所有信息。忽略重要的细节的抽象是错误的抽象：它们可能看起来简单，但实际并不是。关键的抽象设计是要理解什么是重要的，和从那些重要的信息之中寻找设计的最小化。

例如，考虑一个文件系统。文件系统提供的抽象忽略了很多的细节，例如选择存储设备上的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中的用户来说是不重要的（只要系统提供足够的性能）。然而，文件系统中的一些细节的实现对用户来说是重要的。大多数文件系统的缓存数据在主存中，为了改善性能他们可以延迟写入新数据到存储设备。一些应用，例如数据库，需要知道数据何时写入存储，因此在系统崩溃之后他们可以确保数据可保留。因此，将数据刷新到辅助存储的规则在文件系统的接口中必须是是可见的。

我们依赖抽象来管理复杂性不仅仅在编程中，而在我们的日常生活中也很普遍。微波炉包含复杂的电子设备可以将交流电转换为微波射线并将射线分布到整个烹饪腔中。幸运的是，用户看到了一个更简单的抽象，一个有几个按钮组成来控制微波的时间和强度。汽车提供了简单的抽象，它允许我们驾驶但不用理解电机，电池动力管理，防抱死制动系统，定速巡航等等的构造。

4.4 深模块

最好的模块是那些提供强大功能并具有简单接口的模块。我用 *deep* 这个词来描述这些模块。为了深入理解这个构思，假设每个模块用矩形来表示，如图 4.1 所示。每个矩形的面积和模块功能实现成正比。矩形的最顶边等同于模块的接口；那条边的长度表示了接口的复杂性。最好的模块是深的：他们有很多功能都隐藏在一个简单的接口后面。一个深模块是一个好的抽象，因为内部复杂性只有一小部分对用户可见。

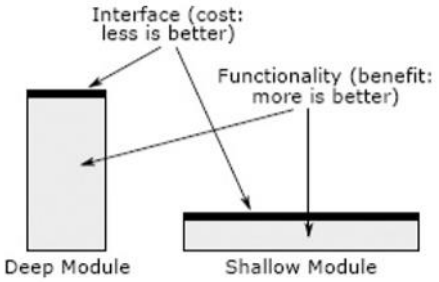

图 4.1：深模块和浅模块。最好的模块是深的：他们允许通过一个简单的接口来访问一些功能。浅模块是一个接口相对复杂但功能不多的模块：它不会隐藏太多的复杂性。

模块深度是一种考虑成本和收益的方式。模块提供的收益是它的功能。模块的成本（从系统复杂度的角度考虑）是它的接口。模块接口代表了模块施加给系统其余部分的复杂度：接口越小越简单，它引入的复杂度越少越好。好的模块是那些成本低和收益高的模块。接口是好的，但更多，或更大，接口未必更好！

Unix 操作系统及其子系统都提供了 I/O 文件机制，例如 Linux，就是一个很精彩的深接口实例。他们只有五个基本系统调用 I/O，包含有简单的签名：

```
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);
```

打开系统调用一个分层的文件名，例如 /a/b/c 和返回一个整型文件描述符，描述符被用于引用打开的文件。打开的其它参数提供了可选信息，例如文件是否在正在读或写中被打开，如果这个文件不存在一个新的文件是否应该被创建， 以及如果一个新的文件被创建，对该文件的访问权限。读和写系统调用应用程序中的缓存区和文件之间的信息传输；关闭结束对文件的访问。大多数文件都是按顺序访问的，因此这是默认的；然而，可以通过援引 lseek 系统调用改变当前访问位置来达到随机访问。

一个 Unix I/O 接口的最新实现需要数十万行代码，这些代码可以解决复杂的问题，例如：

① 如何在磁盘上表示文件以便允许高效的访问？

② 如何存储目录，如何处理分层路径名以查找它们引用的文件？

③ 如何强制权限，以便使一个用户不能修改或删除另一个用户的文件？

④ 如何实现文件访问？例如，如何在中断程序和后台代码之间划分功能，以及如何使这两个元素安全的通信。

⑤ 当并行访问多个文件的时候使用什么调度策略？

⑥　如何将最近访问的文件数据缓存到内存中以便可以减少对磁盘访问的次数?
⑦　如何将各种不同的辅助存储设备，例如磁盘和闪存驱动器，合并到单个文件系统中?

所有的这些问题，以及更多的问题，都由 Unix 文件系统来实现处理。它们对援引系统调用的程序员们是不可见的。Unix I/O 接口的实现多年来有了根本性的发展，但是五项基本内核调用没有被改变。

另一个实例是在一门语言中一个深模块中的垃圾收集，例如 Go 或 Java。这个模块完全没有接口；它在后台默默地回收没有用到的内存。添加垃圾收集到系统实际上缩小了其整体接口，因为它消除了释放对象的接口。垃圾收集的实现十分复杂，但是那些复杂性对于使用开发语言的程序员来说是隐藏的。

深模块，例如 Unix I/O 和垃圾收集提供了强大的抽象，因为它们很容易使用，同时它们对于复杂性实现的隐藏是显著的。

4.5　浅模块

在另一个方面上，与那些提供的功能对比而言浅模块对于接口是相对复杂的。例如，一个链表的实现类是浅的。操作一个链表不需要太多的代码（插入或删除元素只需要几行代码），因此链表抽象不会隐藏太多的细节。链表接口的复杂性几乎与实现的复杂性一样大。浅类有时是不可避免的，但在管理复杂性中它们不会提供更多的帮助。

这里是从软件设计类的项目中，获取浅方法中的一个极端的例子：

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

从管理复杂性的观点来看，这个方法做的事情是很糟糕的，不够好。提供的方法不是抽象的，因为所有的功能通过接口都是可见的。例如，调用者可能需要知道属性将被存储在数据变量中。考虑接口并不比考虑整体的实现简单。如果方法被正确的记录下来，文档将比方法的代码要长。调用该方法所需的击键甚至比调用者直接操作数据变量所需的击键还要多。该方法增加了复杂性（以一种新接口的形式提供给开发者学习）但是没有提供补偿好处。

【Red Flag：浅模块】

浅模块是一个接口中相对于提供的功能来说是比较复杂的模块。浅模块在对抗复杂性的战斗中没有太多的帮助，因为他们提供的益处（不必了解它们内部的工作方式）正在被学习和使用这些接口的成本所抵消。

4.6　Classitis

不幸的是，深类的价值在今天并没有得到广泛的重视。常规看法在编程中那些类应该足够的小，不深。在类的设计上学生们经常被教导那些最重要的东西以此来把比较大的类分解成相对小的类。关于方法通常也会给出相同的建议：“任何长于 N 行的方法都应该分成多个方法”（N 可以低至 10）。这种方法产生了大量的浅类和方法，增加了整个系统的复杂性。

一种极端情况“类应该是小的”方法是一个典型，我把它叫做 *Classitis*，它源于一种错误的观点“类是优质的，如此类就越多越好。”系统的痛苦源于 classitis，在每个新类中鼓励开发者们使功能的数量最小化：如果你需要更多的功能，就需要引入更多的类。Classitis 可以导致类简单独立，但是它也增加了整个系统的复杂度。小的类没有增加太多的功能，因此必须有很多，每一个都有自己的接口。这些接口累积起来在系统级造就了巨大的复杂性，小类也导致了冗长的编程风格，因为每个类都需要样板文件。

4.7 实例：Java and Unix I/O

如今 classitis 中最明显的实例之一就是 Java 类库。Java 语言不需要很多小的类，但是 classitis 文化似乎已经在 Java 编程社区中扎根。例如，要打开文件以便从中读取序列化对象，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 对象只提供基本的 I/O：它不能执行 I/O 缓冲区，也不能读取或写入序列化对象。BufferedInputStream 对象向 FileInputStream 对象中添加缓冲，并且 ObjectInputStream 增加了读取和写入序列化对象的能力。上面代码中的前两个对象，fileStream 和 bufferedStream，文件打开后从来不使用；之后所有的操作都使用 objectStream。

objectStream 必须通过创建单独的 BufferedInputStream 对象来显式请求缓冲，这一点特别烦恼（而且容易出错）；如果开发人员忘记了创建此对象，则不会有缓冲并且 I/O 将会很慢。或许 Java 开发人员将会争辩说不是每一个人都想使用缓冲文件 I/O，因此不应将其内置到基础机制中。他们可能会争辩说最好保持缓冲区分离，这样人们就可以选择是否使用它。提供选择是好的，但是**接口的共同点应尽可能的简单**（见第 6 页的公式）。几乎每个用户在文件 I/O 中都需要缓冲，因此默认情况下应该提供缓冲。对于那些不需要缓冲的少数情况，库可以提供一种机制来禁用它。任何禁用缓冲的机制都应该在接口中清晰的分离（例如，通过为 FileInputStream 提供不同的构造函数，或者通过禁用或替换缓冲机制的方法），因此大多数的开发者甚至不需要知道它的存在。

相对之下，Unix 系统调用的设计者使通用的事情变得简单。例如：他们认识到顺序 I/O 是最常见的，因此他们将此作为默认行为。随机访问做法仍然相对容易，使用 lseek 系统调用，但是开发者只做顺序访问就不需要知道那种机制。如果一个接口有很多特性，但是大多数的开发者只需要知道其中的一些，那么接口的有效复杂性就只是那些经常被使用特性的复杂性。

4.8 结论

通过将模块的接口和实现分离，我们可以从系统的其它部分隐藏实现中的复杂性。模块中的用户只需要理解其接口提供的抽象。在设计类和其它模块中最重要的问题是使它们更深入，以便它们具有用于常见用例的简单接口，然而仍然提供重要的功能。这最大限度的增加了隐藏的复杂性。

第五章　信息的隐藏（和泄漏）

Chapter 4 证明模块应该是深度（尽量通用）的。本章，和接下来几节，讨论和创建深度的模块。

5.1　信息的隐藏

完成深度模块最重要的技术就是*信息隐藏*。这个技术最开始是由 David Parnas 描述的。其基本的思想是每一个模块应该封装一些表示设计决策的知识。这些知识嵌入到模块的实现中，但不会出现在其接口中，因此其它模块看不到这些知识。

隐藏在模块中的信息通常包含有关如何实现某种机制的详细信息。下面是一些可能隐藏在模块中的信息示例：
如何将信息存储在 B-树中，以及如何有效地访问它。
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。
如何分析 JSON 文档。

信息隐藏包含了与该机制相关的数据结构和算法。它还可以包含较低级别的细节，例如页面的大小，也可以包含更抽象较高级别的概念，例如假设大多数的文件都很小。

信息隐藏在两个方面降低了复杂性。首先，它简化了模块的接口。接口映射了模块功能更简单，视图更抽象，并隐藏了细节；这减少了开发人员使用模块的认知负担。例如，一个开发人员使用 B-tree 类不需要担心树中节点的理想扇出或如何保持树的平衡。第二，信息隐藏使系统的进化更容易。如果隐藏了一条信息，则在包含信息的模块外部不存在对该信息的依赖关系，因此与该信息相关的更改将只影响一个模块。例如，如果 TCP 协议发生了变化（例如引入了一种新的拥塞控制机制），则必须修改协议的实现，但使用 TCP 发送和接收数据的更高级别的代码不需要进行任何的更改。

在设计一个新模块的时候，你应该仔细的考虑哪些信息可以隐藏在模块中。如果你可以隐藏更多的信息，你也应该能够简化模块接口，这会使模块更深入。

注意：通过将变量和方法声明为私有来隐藏类中的变量和方法与信息隐藏不同。私有元素可以帮助信息隐藏，因为它们使类项无法从类外部直接访问。但是， 关于私有项的信息仍然可以通过公有的方法例如，getter 和 setter 等方法来暴露。当这种情况发生时，变量的性质和用法就如同变量是公有的一样被公开。信息隐藏的最佳方式是当信息完全隐藏在一个模块中时，这样它对模块中的用户来说是不相关和不可见的。然而，部分信息隐匿也有其价值，例如，如果某个特殊的特性或信息片段仅仅是某个类的少数用户所需要的，并且它是通过单独的方法访问的，因此在最常见的用户案例中是不可见的，那么这些信息基本上是隐藏的。这些信息所产生的依赖性比类中每个用户所见的信息要少。

5.2　信息的泄漏

信息隐藏的反面是*信息泄漏*。当一个设计决策反映在多个模块中时，就会发生信息泄漏。这会在模块之间产生模块依赖关系：对设计决策的任何更改都需要更改所有被调用的模块。如果一条信息反映在模块的接口中，那么根据定义该信息已经被泄漏；因此，简单的接口倾向于更好的与信息隐藏相关联。然而，即使信息没有出现在模块的接口中，它也可能被泄漏。假设两个类都知道某个特定的文件格式（或许一个类在读该格式的文件，而另一个类在写入）。在它接口中的信息即使两个类都不暴漏，它们都依赖于文件格式：如果格式改变，两个类都需要被更改。像这样的后门泄漏比通过接口泄漏更有害，因为它不易理解。

在软件设计中，信息泄漏是最重要的红色标志之一。作为一个软件设计师，你能学到最好的技能之一就是对信息泄漏有高度的敏感性。如果你遇到类之间的信息泄漏，请扪心自问：“如何重新组织这些类，使这些特性的知识只影响单个类？”如果受影响的类相对较少，并且与泄漏的信息密切相关，那么将它们合并到一个单独的类中感觉是有意义的。另一种可能的方法是从所有受影响的类中提取信息，并创建一个只封装这些信息的新类。然而，这种方法只有在你能找到一个简单的接口来抽象细节时才会有效；如果新类通过它的接口公开了大部分知识，那么它就不会提供太多的价值（你只是简单的通过一个接口的泄漏代替了后门泄漏）。

当在多个地方使用相同的知识时，例如两个都理解特定类型文件格式的不同类，就会发生信息泄漏。

【Red Flag：信息泄漏】

当相同的知识在多个地方被使用时就会出现信息泄漏，例如两个不同的类都理解成特定类型的文件格式。

5.3 时间的分解

信息泄漏的一个常见原因是我称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将发生的时间顺序。考虑一个应用程序，该应用程序以特定格式读取文件，修改文件内容，然后再次将文件写出。通过时间分解，该应用程序可以分为三类：一类用于读取文件，另一类用于执行修改，第三类用于写出新版本。文件读取和文件写入步骤都具有有关文件格式的知识，这会导致信息泄漏。解决方案是将用于读写文件的核心机制结合到一个类中。该类将在应用程序的读取和写入阶段使用。很容易陷入时间分解的陷阱，因为在编写代码时通常会想到必须执行操作的顺序。但是，大多数设计决策会在应用程序的整个生命周期中的多个不同时刻表现出来。结果，时间分解常常导致信息泄漏。

顺序通常很重要，因此它将反映在应用程序中的某个位置。但是，除非该结构与信息隐藏保持一致（也许不同阶段使用完全不同的信息），否则不应将其反映在模块结构中。在设计模块时，应专注于执行每个任务所需的知识，而不是任务发生的顺序。

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在不同的方法或类中。如果在执行的不同点使用相同的知识，则会在多个位置对其进行编码，从而导致信息泄漏。

5.4 实例：HTTP 服务

为了说明信息隐藏中的问题，让我们考虑由学生在软件设计课程中实现 HTTP 协议的设计决策。看到他们做得好的事情以及遇到问题的地方都是很有用的。

HTTP 是 Web 浏览器用来与 Web 服务器通信的机制。当用户单击 Web 浏览器中的链接或提交表单时，浏览器使用 HTTP 通过网络将请求发送到 Web 服务器。服务器处理完请求后，会将响应发送回浏览器。该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，两者均以文本形式表示。图 5.1 显示了描述表单提交的 HTTP 请求示例。要求课程中的学生实施一门或多门课程，以使 Web 服务器可以轻松地接收传入的 HTTP 请求并发送响应。

图 5.1：HTTP 协议中的 POST 请求包含通过 TCP 套接字发送的文本。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主体。初始行包含请求类型（POST 用于提交表单数据），指示操作（/注释/创建）和可选参数的 URL（photo_id 的值为 246）以及发送者使用的 HTTP 协议版本。每个标题行由一个名称（例如 Content-Length）及其后的值组成。对于此请求，正文包含其他参数（注释和优先级）。

5.5 实例：太多的类

学生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漏。一个团队使用两种不同的类来接收 HTTP 请求。第一类将来自网络连接的请求读取为字符串，第二类将字符串解析。这是时间分解的一个示例（“首先读取请求，然后解析它”）。发生信息泄漏是因为无法解析大量消息就无法读取 HTTP 请求。例如，Content-Length 标头指定了请求主体的长度，因此必须对标头进行解析才能计算总请求长度。结果，这两个类都需要了解 HTTP 请求的大多数结构，并且解析代码在两个类中都是重复的。

由于这些类共享大量信息，因此最好将它们合并为一个同时处理请求读取和解析的类。由于它将请求格式的所有知识隔离在一个类中，因此它提供了更好的信息隐藏，并且还为调用者提供了一个更简单的接口（只是一种调用方法）。

此示例说明了软件设计中的一般主题：通常可以通过使类稍大一些来改善信息隐藏。这样做的一个原因是将与特定功能相关的所有代码（例如，解析 HTTP 请求）组合在一起，以便生成的类包含与该功能相关的所有内容。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中的每一个步骤使用单独的方法，不如使用一种方法来执行整个计算。这样可以简化界面。这两个优点都适用于上一段的示例：组合类将与解析 HTTP 请求相关的所有代码组合在一起，并且用一个替换了两个外部可见的方法。

当然，可以将较大的类的概念考虑得太远（例如整个应用程序的单个类）。第 9 章将讨论将代码分成多个较小的类的合理条件。

5.6 实例：HTTP 参数处理

服务器收到 HTTP 请求后，服务器需要访问该请求中的某些信息。图 5.1 中处理请求的代码可能需要知道 photo_id 参数的值。参数可以在请求的第一行中指定（图 5.1 中的 photo_id），有时也可以在正文中指定（图 5.1 中的注释和优先级）。每个参数都有一个名称和一个值。参数的值使用一种称为 URL 编码的特殊编码。例如，在图 5.1 中的注释值中，“+”代表空格字符，“%21”代替“！”。为了处理请求，服务器将需要某些参数的值，并且希望它们采用未编码的形式。

关于参数处理，大多数学生项目都做出了两个不错的选择。首先，他们认识到服务器应用程序不在乎是否在标题行或请求的正文中指定了参数，因此他们对调用者隐藏了这种区别，并将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到 Web 服务器之前先对其进行解码，以便图 5.1 中的 comment 参数的值将返回 “What a cute baby!”，而不是“What+a+cute+baby%21”）。在这两种情况下，信息隐藏都使使用 HTTP 模块的代码的 API 更加简单。

但是，大多数学生使用的界面返回的参数太浅，这导致丢失信息隐藏的机会。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 类具有一种类似于以下方法的单个方法来返回参数：

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

该方法不是返回单个参数，而是返回内部用于存储所有参数的映射的引用。这个方法是浅层的，它公开了 HTTPRequest 类用来存储参数的内部表示。对该表示的任何更改都将导致接口的更改，这将需要对所有调用者进行修改。在修改实现时，更改通常涉及关键数据结构表示的更改(例如，为了提高性能)。因此，尽量避免暴露内部数据结构是很重要的。这种方法还为调用者提供了更多的工作:调用者必须首先调用 getParams，然后必须调用另一个方法来从映射中检索特定的参数。最后，调用者必须意识到他们不应该修改 getParams 返回的映射，因为这会影响 HTTPRequest 的内部状态。

这是一个用于检索参数值的更好的接口：

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

getParameter 以字符串形式返回参数值。它提供了一个比上面的 getParams 更深的接口；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，图 5.1 中的 photo_id 参数）。这使调用者不必单独请求字符串到整数的转换，并且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。（如果所需的参数不存在，或者无法将其转换为所请求的类型，则所有这些方法都将引发异常；上面的代码中省略了异常声明）。

5.7 实例：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。学生在该领域中最常见的错误是默认值不足。每个 HTTP 响应必须指定一个 HTTP 协议版本。一个团队要求呼叫者在创建响应对象时明确指定此版本。但是，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将请求作为参数传递（它指示将响应发送到何处）。因此，HTTP 类自动提供响应版本更为有意义。调用者不太可能知道要指定哪个版本，并且如果调用者确实指定了一个值，则可能导致 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。

默认值说明了应该设计接口以使常见情况尽可能简单的原则。它们还是隐藏部分信息的一个示例：在正常情况下，调用者无需知道默认项的存在。在极少数情况下，调用方需要覆盖默认值，它必须知道该值，并且可以调用特殊方法来对其进行修改。

只要有可能，类就应该“做正确的事”，而无需明确要求。默认值就是一个例子。第 26 页上的 Java I/O 示例以负面方式说明了这一点。普遍希望在文件 I/O 中缓冲，以至于没有人需要明确要求它，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。最好的功能是你甚至不知道它们存在的功能。

【Red Flag：Overexposure】

如果常用功能的 API 迫使用户了解很少使用的其他功能，则这会增加不需要很少使用功能的用户的认知负担。

5.8 一个类中隐藏的信息

本章中的示例着重于信息隐藏，因为它与类的外部可见 API 有关，但是信息隐藏也可以应用于系统中的其他级别，例如类内。尝试在一个类中设计私有方法，以便每个方法都封装一些信息或功能，并将其隐藏在类的其余部分中。此外，请尽量减少使用每个实例变量的位置数量。有些变量可能需要在整个类中广泛使用，但是其他变量可能只需要在少数地方使用；如果可以减少使用变量的位置的数量，则将消除类内的依赖关系并降低其复杂性。

5.9 Taking it too far

仅当在其模块外部不需要隐藏信息时，隐藏信息才有意义。如果模块外部需要该信息，则不得隐藏它。假设模块的性能受某些配置参数的影响，并且模块的不同用途将需要对参数进行不同的设置。在这种情况下，将参数暴露在模块的界面中很重要，以便可以对其进行适当的旋转。作为软件设计师，你的目标应该是最大程度地减少模块外部所需的信息量。例如，如果模块可以自动调整其配置，那将比公开配置参数更好。但是，重要的是要识别模块外部需要哪些信息，并确保将其公开。

5.10 结论

信息隐藏和深层模块密切相关。如果模块隐藏了很多信息，则往往会增加模块提供的功能，同时还会减少其接口。这使模块更深。相反，如果一个模块没有隐藏太多信息，则它要么功能不多，要么接口复杂。无论哪种方式，模块都是浅的。

将系统分解为模块时，请尽量不要受运行时操作顺序的影响。这将使你沿着时间分解的路径前进，这将导致信息泄漏和模块浅。相反，请考虑执行应用程序任务所需的不同知识，并设计每个模块以封装这些知识中的一个或几个。这将产生带有深色模块的干净简单的设计。

1 David Parnas，“关于将系统分解为模块的标准”，ACM 通讯，1972 年 12 月。

第六章 模块的通用性

设计新模块时，你将面临的最普遍的决定之一就是是以通用还是专用方式实现它。有人可能会争辩说，你应该采用通用方法，在这种方法中，你将实现一种可用于解决广泛问题的机制，而不仅是当今重要的问题。在这种情况下，新机制可能会在将来发现意外用途，从而节省时间。通用方法似乎与第 3 章中讨论的投资思路一致，在这里你花了更多时间在前面，以节省以后的时间。

另一方面，我们知道很难预测软件系统的未来需求，因此通用解决方案可能包含从未真正需要的功能。此外，如果你实现的东西过于通用，那么可能无法很好地解决你今天遇到的特定问题。结果，有些人可能会争辩说，最好只关注当今的需求，构建你所知道的需求，并针对你今天打算使用的方式进行专门化处理。如果你采用特殊用途的方法并在以后发现更多用途，则始终可以对其进行重构以使其通用。专用方法似乎与软件开发的增量方法一致。

6.1 使类具有一定的通用性

以我的经验，最有效的方法是以某种通用的方式实现新模块。短语“有点通用”表示该模块的功能应反映你当前的需求，但其接口则不应。相反，该接口应该足够通用以支持多种用途。该界面应易于使用，以满足当今的需求，而不必专门与它们联系在一起。“有点”这个词很重要：不要被带走并建造通用的东西，以致于很难满足当前的需求。

通用方法最重要的（也许是令人惊讶的）好处是，与专用方法相比，它导致更简单，更深入的界面。如果你将该类用于其他目的，则通用方法还可以节省将来的时间。但是，即使该模块仅用于其原始用途，由于其简单性，通用方法仍然更好。

6.2 实例：为编辑器存储文本

让我们考虑一个软件设计课程的示例，其中要求学生构建简单的 GUI 文本编辑器。编辑器必须显示一个文件，并允许用户指向，单击并键入以编辑该文件。编辑者必须在不同的窗口中支持同一文件的多个同时视图。他们还必须支持多级撤消和重做以修改文件。

每个学生项目都包括一个管理文件的基础文本的类。文本类通常提供以下方法：将文件加载到内存，读取和修改文件的文本以及将修改后的文本写回到文件。

许多学生团队为文本课实现了专用的 API。他们知道该类将在交互式编辑器中使用，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，如果编辑者的用户键入了退格键，则编辑者会立即删除光标左侧的字符；如果用户键入删除键，则编辑器立即删除光标右侧的字符。知道这一点后，一些团队在文本类中创建了一个方法来支持以下每个特定功能：

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

这些方法中的每一个都以光标位置作为参数。特殊类型的光标表示此位置。编辑器还必须支持可以复制或删除的选择。学生通过定义选择类并在删除过程中将该类的对象传递给文本类来解决此问题：
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，则将更易于实现用户界面。但是，实际上，这种专业化对用户界面代码几乎没有好处，并且为使用用户界面或文本类的开发人员带来了很高的认知负担。文本类以大量浅层方法结束，每种浅层方法仅适用于一个用户界面操作。许多方法（例如delete）仅在单个位置调用。结果，在用户界面上工作的开发人员必须学习大量有关文本类的方法。

这种方法在用户界面和文本类之间造成了信息泄漏。与用户界面有关的抽象（例如选择或退格键）反映在文本类中；这增加了从事文本课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方法，因此使用该用户界面的开发人员也可能最终也要使用该文本类。类设计的目标之一是允许每个类独立开发，但是专用方法将用户界面和文本类联系在一起。

6.3 更多通用性的接口

更好的方法是使文本类更通用。仅应根据基本文本功能定义其 API，而不应反映将用其实现的更高级别的操作。例如，只需两种方法即可修改文本：
void insert(Position position, String newText);
void delete(Position start, Position end);

第一种方法在文本内的任意位置插入任意字符串，第二种方法删除大于或等于开始但小于结束的位置处的所有字符。此 API 还使用了更通用的 Position 类型来代替 Cursor，它反映了特定的用户界面。文本类还应该提供用于操纵文本中位置的通用工具，例如：
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字符数。如果 numChars 参数为正，则新位置在文件中比位置晚；如果 numChars 为负，则新位置在位置之前。必要时，该方法会自动跳到下一行或上一行。使用这些方法，可以使用以下代码来实现删除键（假定 cursor 变量保留当前光标的位置）：
text.delete(cursor, text.changePosition(cursor, 1));

同样，可以按以下方式实现退格键：
text.delete(text.changePosition(cursor, -1), cursor);

使用通用文本 API，实现用户界面功能（如删除和退格）的代码比使用专用文本 API 的原始方法要长一些。但是，新代码比旧代码更明显。在用户界面模块中工作的开发人员可能会关心由 Backspace 键删除哪些字符。使用新代码，这是显而易见的。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/或代码以验证行为。此外，通用方法总体上比专用方法具有更少的代码，因为它用较少数量的通用方法代替了文本类中的大量专用方法。

使用通用接口实现的文本类除交互式编辑器外，还可以用于其他目的。作为一个示例，假设你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文件。专用文本类中的方法（例如，退格键和 Delete）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

程序所需的大多数功能。缺少的只是一种搜索给定字符串的下一个匹配项的方法，例如：
Position findNext(Position start, String string);

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

6.4 通用性导致更好的信息隐藏

通用方法在文本和用户界面类之间提供了更清晰的分隔，从而可以更好地隐藏信息。文本类不需要知道用户界面的详细信息，例如如何处理退格键。这些细节现在封装在用户界面类中。可以添加新的用户界面功能，而无需在文本类中创建新的支持功能。通用界面还减轻了认知负担：使用用户界面的开发人员只需要学习一些简单的方法，就可以将其重复用于各种目的。

文本类原始版本中的 backspace 方法是错误的抽象。它旨在隐藏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。用户界面开发人员可能会阅读退格方法的代码，以确认其精确的行为。将方法放在文本类中只会使用户界面开发人员更难获得所需的信息。软件设计最重要的元素之一就是确定谁需要知道什么以及何时知道。当细节很重要时，最好使它们明确且尽可能明显，例如修订的 Backspace 操作实现。将这些信息隐藏在界面后面只会产生晦涩感。

6.5 问自己一些问题

识别干净的通用类设计要比创建一个简单。你可以问自己一些问题，这将帮助你在接口的通用和专用之间找到适当的平衡。

满足我当前所有需求的最简单的界面是什么？如果减少 API 中的方法数量而不降低其整体功能，则可能正在创建更多通用的方法。专用文本 API 至少具有三种删除文本的方法：退格，删除和 deleteSelection。通用性更强的 API 只有一种删除文本的方法，可同时满足所有三个目的。仅在每种方法的 API 保持简单的前提下，减少方法的数量才有意义。如果你必须引入许多其他参数以减少方法数量，那么你可能并没有真正简化事情。

在多少情况下会使用此方法？如果一种方法是为特定用途而设计的，例如退格方法，那是一个危险信号，它可能太特殊了。看看是否可以用一个通用方法替换几种专用方法。

这个 API 是否易于使用以满足我当前的需求？这个问题可以帮助你确定何时使 API 变得简单而通用。如果你必须编写许多其他代码才能将类用于当前用途，那么这是一个危险信号，即该接口未提供正确的功能。例如，针对文本类的一种方法是围绕单字符操作进行设计：insert 插入单个字符，而 delete 删除单个字符。该 API 既简单又通用。但是，对于文本编辑器来说并不是特别容易使用：更高级别的代码将包含许多循环，用于插入或删除字符范围。单字符方法对于大型操作也将是低效的。

6.6 结论

通用接口比专用接口具有许多优点。它们往往更简单，使用的方法更少。它们还提供了类之间的更清晰的分隔，而专用接口则倾向于在类之间泄漏信息。使模块具有某种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

软件系统由层组成，其中较高的层使用较低层提供的功能。在设计良好的系统中，每一层都提供与其上，下两层不同的抽象。如果你通过调用方法遵循单个操作在层中上下移动，则每个方法调用的抽象都会改变。例如：

在文件系统中，最上层实现文件抽象。文件由可变长度的字节数组组成，可以通过读写可变长度的字节范围来更新该字节。文件系统的下一个下一层在固定大小的磁盘块的内存中实现了高速缓存。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。最低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 的网络传输协议中，最顶层提供的抽象是从一台机器可靠地传递到另一台机器的字节流。此级别在较低级别上构建，该级别可以尽最大努力在计算机之间传输有限大小的数据包：大多数数据包将成功交付，但某些数据包可能会丢失或乱序交付。

如果系统包含具有相似抽象的相邻层，则这是一个红色标记，表明类分解存在问题。本章讨论了发生这种情况的情况，导致的问题以及如何重构以消除问题。

7.1 方法传递

当相邻的层具有相似的抽象时，问题通常以直通方法的形式表现出来。直通方法是一种很少执行的方法，除了调用另一个方法（其签名与调用方法的签名相似或相同）之外。例如，一个实施 GUI 文本编辑器的学生项目包含一个几乎完全由直通方法组成的类。这是该类的摘录：

```
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);
        }
    }
```

```
    ...
}
```

该类别中 15 个公共方法中的 13 个是直通方法。

【Red Flag：直通方法】

直通方法是一种不执行任何操作的方法，只是将其参数传递给另一个方法，通常使用与直通方法相同的 API。这通常表示各类之间没有明确的职责划分。

直通方法使类变浅：它们增加了类的接口复杂性，从而增加了复杂性，但是并没有增加系统的整体功能。在上述四个方法中，只有最后一个具有任何功能，甚至没有什么功能：该方法检查一个变量的有效性。直通方法还会在类之间创建依赖关系：如果针对 TextArea 中的 insertString 方法更改了签名，则必须更改 TextDocument 中的 insertString 方法以进行匹配。

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 insertString 方法，但是用于插入文本的功能完全在 TextArea 中实现。这通常是一个坏主意：某个功能的接口应该在实现该功能的同一类中。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功能和抽象？” 你可能会注意到，各类之间的职责重叠。

解决方案是重构类，以使每个类都有各自不同且连贯的职责。图 7.1 说明了几种方法。一种方法，如图 7.1（b）所示，是将较低级别的类直接暴露给较高级别的类的调用者，而从较高级别的类中删除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，如图 7.1（c）所示。最后，如果无法解开这些类，最好的解决方案可能是如图 7.1（d）所示合并它们。

在上面的示例中，职责交织的三个类为：TextDocument，TextArea 和 TextDocumentListener。学生通过在类之间移动方法并将三个类缩减为两个类来消除直通方法，这两个类的职责更加明确。

7.2 什么时候可以复制接口?

具有相同签名的方法并不总是不好的。重要的是，每种新方法都应贡献重要的功能。直通方法很糟糕，因为它们不提供任何新功能。

分派器是一个示例，该示例对于一种方法调用具有相同签名的另一种方法很有用。调度程序是一种使用其参数选择要调用的其他方法之一的方法。然后将其大部分或所有参数传递给所选方法。调度程序的签名通常与其调用的方法的签名相同。即便如此，调度程序仍提供有用的功能：它可以选择其他几种方法中的哪一种来执行每个任务。

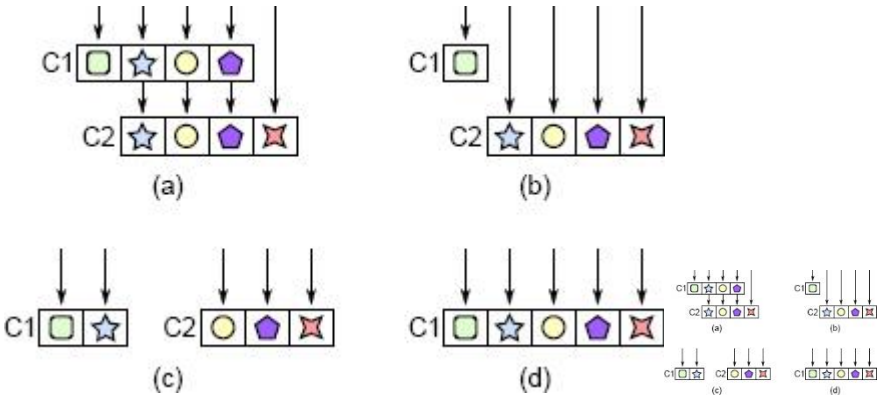

图 7.1：直通方法。在（a）中，类 C1 包含三个直通方法，这些方法只调用 C2 中具有相同签名的方法（每个符号代表一个特定的方法签名）。可以通过使 C1 的调用方像在（b）中那样直接调用 C2，通过在 C1 和 C2 之间重新分配功能以避免在（c）中的类之间进行调用，或者通过组合在（d）中的类来消除直通方法。 。

例如，当 Web 服务器从 Web 浏览器接收到传入的 HTTP 请求时，它将调用一个调度程序，该调度程序检查传入请求中的 URL 并选择一种特定的方法来处理该请求。某些 URL 可以通过返回磁盘上文件的内容来处理。其他人则可以通过调用诸如 PHP 或 JavaScript 之类的语言的过程来处理。分发过程可能非常复杂，通常由与传入 URL 匹配的一组规则来驱动。

只要每种方法都提供有用且独特的功能，几种方法都应具有相同的签名。调度程序调用的方法具有此属性。另一个示例是具有多种实现方式的接口，例如操作系统中的磁盘驱动程序。每个驱动程序都支持不同类型的磁盘，但是它们都有相同的接口。当几种方法提供同一接口的不同实现时，它将减少认知负担。使用其中一种方法后，与其他方法一起使用会更容易，因为你无需学习新的接口。像这样的方法通常位于同一层，并且它们不会相互调用。

7.3 装饰者

装饰器设计模式(也称为“包装器”)是一种鼓励跨层复制 API 的模式。装饰对象接受现有对象并扩展其功能;它提供一个与底层对象相似或相同的 API，它的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，它提供了相同的 API，但是引入了缓冲。例如，当它的 read 方法被调用来读取单个字符时，它会调用底层 InputStream 上的 read 来读取更大的块，并保存额外的字符来满足未来的 read 调用。另一个例子出现在窗口系统中:Window 类实现了一个不能滚动的窗口的简单形式，而 ScrollableWindow 类通过添加水平和垂直滚动条来装饰窗口类。

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，装饰器类往往很浅：它们引入了大量的样板，以实现少量的新功能。装饰器类通常包含许多直通方法。过度使用装饰器模式很容易，为每个小的新功能创建一个新类。这导致诸如 Java I/O 示例之类的浅层类激增。

创建装饰器类之前，请考虑以下替代方法：

你能否将新功能直接添加到基础类，而不是创建装饰器类？如果新功能是相对通用的，或者在逻辑上与基础类相关，或者如果基础类的大多数使用也将使用新功能，则这是有意义的。例如，几乎每个创建 Java InputStream 的人都会创建一个 BufferedInputStream，并且缓冲是 I/O 的自然组成部分，因此应该合并这些类。

如果新功能专用于特定用例，将其与用例合并而不是创建单独的类是否有意义？
你可以将新功能与现有的装饰器合并，而不是创建新的装饰器吗？这将导致一个更深的装饰器类，而不是多个浅的装饰器类。

最后，问问自己新功能是否真的需要包装现有功能：是否可以将其实现为独立于基类的独立类？在窗口示例中，滚动条可能与主窗口分开实现，而无需包装其所有现有功能。

有时装饰者很有意义，但通常有更好的选择。

7.4 接口与实现

“不同层，不同抽象”规则的另一个应用是，类的接口通常应与其实现不同：内部使用的表示形式应与接口中出现的抽象形式不同。如果两者具有相似的抽象，则该类可能不是很深。例如，在第 6 章讨论的文本编辑器项目中，大多数团队都以文本行的形式实现了文本模块，每行分别存储。一些团队还使用 getLine 和 putLine 之类的方法围绕行设计了文本类的 API。但是，这使文本类使用起来较浅且笨拙。在较高级别的用户界面代码中，通常在行中间插入文本（例如，当用户键入内容时）或删除跨行的文本范围。通过用于文本类的面向行的 API，调用者被迫拆分和合并行以实现用户界面操作。这段代码很简单，并且在用户界面的实现中被复制和散布。

文本类提供面向字符的接口时，使用起来要容易得多，例如，insert 方法可在文本的任意位置插入任意文本字符串（可能包括换行符），而 delete 方法则删除文本在文本中的两个任意位置之间。在内部，文本仍以行表示。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，这使文本类更深，并简化了使用该类的高级代码。通过这种方法，文本 API 与面向行的存储机制大不相同。差异表示该类提供的有价值的功能。

7.5 变量传递

跨层 API 复制的另一种形式是传递变量，该变量是通过一长串方法向下传递的变量。图 7.2（a）显示了数据中心服务的示例。命令行参数描述用于安全通信的证书。只有底层方法 m3 才需要此信息，该方法调用一个库方法来打开套接字，但是该信息会通过 main 和 m3 之间路径上的所有方法向下传递。cert 变量出现在每个中间方法的签名中。

传递变量增加了复杂性，因为它们强制所有中间方法知道它们的存在，即使这些方法对变量没有用处。此外，如果存在一个新变量（例如，最初构建的系统不支持证书，但是你后来决定添加该支持），则可能必须修改大量的接口和方法才能将变量传递给所有相关路径。

消除传递变量可能具有挑战性。一种方法是查看最顶层和最底层方法之间是否已共享对象。在图 7.2 的数据中心服务示例中，也许存在一个对象，其中包含有关网络通信的其他信息，这对于 main 和 m3 都

是可用的。如果是这样，main 可以将证书信息存储在该对象中，因此不必通过通往 m3 的路径上的所有干预方法来传递证书（请参见图 7.2（b））。但是，如果存在这样的对象，则它本身可能是传递变量（m3 还将如何访问它？）。

另一种方法是将信息存储在全局变量中，如图 7.2（c）所示。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，全局变量使得不可能在同一过程中创建同一系统的两个独立实例，因为对全局变量的访问会发生冲突。在生产中似乎不太可能需要多个实例，但是它们通常在测试中很有用。

我最常使用的解决方案是引入一个上下文对象，如图 7.2（d）所示。上下文存储应用程序的所有全局状态（否则将是传递变量或全局变量的任何状态）。大多数应用程序在其全局状态下具有多个变量，这些变量表示诸如配置选项，共享子系统和性能计数器之类的内容。每个系统实例只有一个上下文对象。上下文允许系统的多个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，在许多地方可能都需要上下文，因此它有可能成为传递变量。为了减少必须意识到的方法数量，可以将上下文的引用保存在系统的大多数主要对象中。在图 7.2（d）的示例中，包含 m3 的类将对上下文的引用作为实例变量存储在其对象中。创建新对象时，创建方法将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。使用这种方法，上下文随处可见，但在构造函数中仅作为显式参数出现。

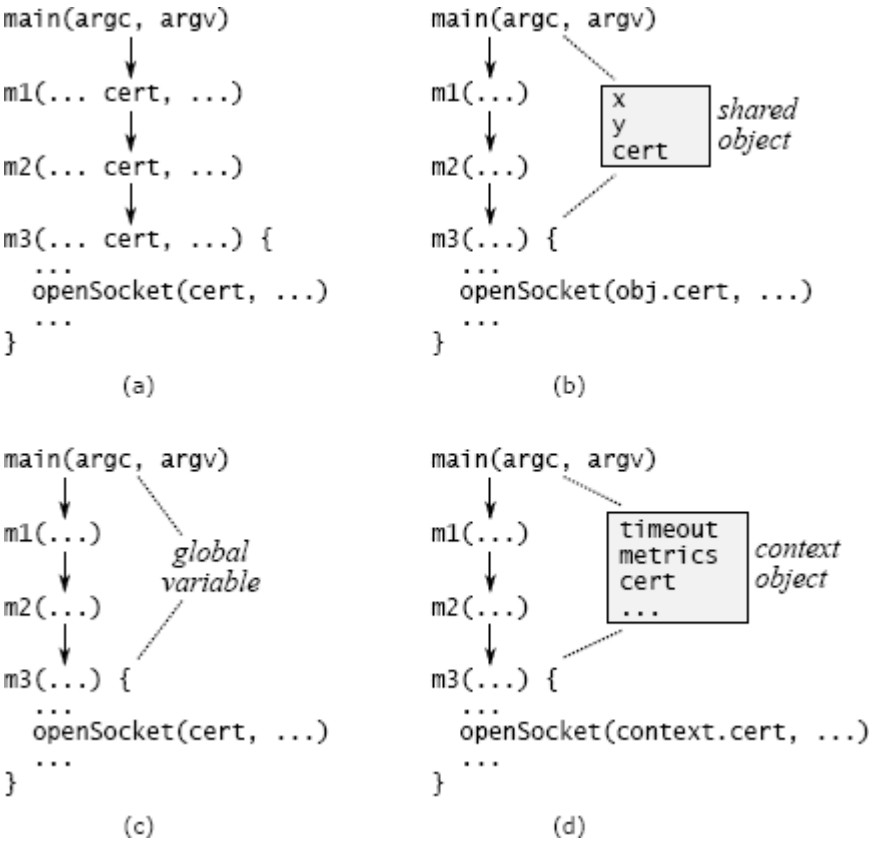

图 7.2：处理传递变量的可能技术。在（a）中，证书通过方法 m1 和 m2 传递，即使它们不使用它也是如此。在（b）中，main 和 m3 具有对一个对象的共享访问权，因此可以将变量存储在此处，而不用

将其传递给 m1 和 m2。在（c）中，cert 存储为全局变量。在（d）中，证书与其他系统范围的信息（例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其方法需要访问它的所有对象中。

上下文对象统一了所有系统全局信息的处理，并且不需要传递变量。如果需要添加新变量，则可以将其添加到上下文对象；除了上下文的构造函数和析构函数外，现有代码均不受影响。由于上下文全部存储在一个位置，因此上下文可以轻松识别和管理系统的全局状态。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，则实施此类更改将更加困难。

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多数缺点。例如，为什么存在特定变量或在何处使用特定变量可能并不明显。没有纪律，上下文会变成巨大的数据抓包，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；避免问题的最佳方法是使上下文中的变量不可变。不幸的是，我没有找到比上下文更好的解决方案。

7.6 结论

接口，参数，函数，类或定义之类的添加到系统中的每个设计基础架构都会增加复杂性，因为开发人员必须了解该元素。为了使元素能够提供相对于复杂性的净收益，它必须消除在没有设计元素的情况下会出现的一些复杂性。否则，最好不要使用该特定元素来实施系统。例如，一个类可以通过封装功能来降低复杂性，以使该类的用户无需意识到这一点。

“不同的层，不同的抽象”规则只是此思想的一种应用：如果不同的层具有相同的抽象，例如直通方法或装饰器，则很有可能它们没有提供足够的利益来补偿它们代表的其他基础结构。类似地，传递参数要求几种方法中的每一种都知道它们的存在（这增加了复杂性），而又不提供其他功能。

第八章 降低复杂性

本章介绍了有关如何创建更深层类的另一种思考方式。假设你正在开发一个新模块，并且发现了一个不可避免的复杂性。哪个更好：应该让模块用户处理复杂性，还是应该在模块内部处理复杂性？如果复杂度与模块提供的功能有关，则第二个答案通常是正确的答案。大多数模块拥有的用户多于开发人员，因此开发人员遭受的苦难要大于用户。作为模块开发人员，你应该努力使模块用户的生活尽可能轻松，即使这对你来说意味着额外的工作。表达此想法的另一种方法是，模块具有简单的接口比简单的实现更为重要。

作为开发人员，很容易以相反的方式行事：解决简单的问题，然后将困难的问题推给其他人。如果出现不确定如何处理的条件，最简单的方法是引发异常并让调用方处理它。如果不确定要实施什么策略，则可以定义一些配置参数来控制该策略，然后由系统管理员自行确定最佳策略。

这样的方法短期内会使你的生活更轻松，但它们会加剧复杂性，因此许多人必须处理一个问题，而不仅仅是一个人。例如，如果一个类抛出异常，则该类的每个调用者都必须处理该异常。如果一个类导出配置参数，则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 实例：类文本编辑器

考虑为 GUI 文本编辑器管理文件文本的类，这在第 6 章和第 7 章中讨论过。该类提供了将文件从磁盘读入内存、查询和修改文件在内存中的副本以及将修改后的版本写回磁盘的方法。当学生必须实现这个类时，他们中的许多人选择了一个面向行的接口，该接口具有读取、插入和删除整行文本的方法。这导致了类的简单实现，但也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，击键会导致在现有行中插入单个字符;复制或删除选择项可以修改几个不同行的部分。使用面向行的文本界面，为了实现用户界面，高级软件必须分割和连接行。

面向字符的界面（如 6.3 节中所述）降低了复杂性。用户界面软件现在可以插入和删除任意范围的文本，而无需分割和合并行，因此变得更加简单。文本类的实现可能会变得更加复杂：如果内部将文本表示为行的集合，则必须拆分和合并行以实现面向字符的操作。这种方法更好，因为它封装了在文本类中拆分和合并的复杂性，从而降低了系统的整体复杂性。

8.2 实例：参数配置

配置参数是提高复杂度而不是降低复杂度的一个示例。类可以在内部输出一些控制其行为的参数，而不是在内部确定特定的行为，例如高速缓存的大小或在放弃之前重试请求的次数。然后，该类的用户必须为参数指定适当的值。在当今的系统中，配置参数已变得非常流行。有些系统有数百个。

拥护者认为配置参数不错，因为它们允许用户根据他们的特定要求和工作负载来调整系统。在某些情况下，低级基础结构代码很难知道要应用的最佳策略，而用户则对其域更加熟悉。例如，用户可能知道某些请求比其他请求更紧迫，因此用户为这些请求指定更高的优先级是有意义的。在这种情况下，配置参数可以在更广泛的域中带来更好的性能。

但是，配置参数还提供了一个轻松的借口，可以避免处理重要问题并将其传递给其他人。在许多情况下，用户或管理员很难或无法确定参数的正确值。在其他情况下，可以通过在系统实现中进行一些额外的工作来自动确定正确的值。考虑必须处理丢失数据包的网络协议。如果它发送请求但在一定时间内未收到响应，则重新发送该请求。确定重试间隔的一种方法是引入配置参数。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，自己计算出一个合理的值。这种方法降低了复杂性，使用户不必找出正确的重试间隔。它具有动态计算重试间隔的其他优点，因此，如果操作条件发生变化，它将自动进行调整。相反，配置参数很容易过时。

因此，你应尽可能避免使用配置参数。在导出配置参数之前，请问自己：“用户（或更高级别的模块）是否能够确定比我们在此确定的更好的值？” 当你创建配置参数时，请查看是否可以自动计算合理的默认值，因此用户仅需在特殊情况下提供值即可。理想情况下，每个模块都应完全解决问题。配置参数导致解决方案不完整，从而增加了系统复杂性。

8.3 太过头了

降低复杂性时要谨慎处理；这个想法很容易被夸大。一种极端的方法是将整个应用程序的所有功能归为一个类，这显然没有意义。如果（a）被降低的复杂度与该类的现有功能密切相关，（b）降低复杂度将导致应用程序中其他地方的许多简化，则降低复杂度最有意义。简化了类的界面。请记住，目标是最大程度地降低整体系统复杂性。

第六章介绍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，因为它可以降低复杂性。但是，将用户界面的知识添加到文本类中并不会大大简化高层代码，并且用户界面的知识与文本类的核心功能无关。在这种情况下，降低复杂度只会导致信息泄漏。

8.4 结论

在开发模块时，为了减少用户的痛苦，要找机会给自己多吃一点苦。

第九章 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，它们应该在同一位置一起实现，还是应该分开实现？这个问题适用于系统中的所有级别，例如功能，方法，类和服务。例如，应该在提供面向流的文件 I/O 的类中包括缓冲，还是应该在单独的类中？HTTP 请求的解析应该完全在一种方法中实现，还是应该在多个方法（甚至多个类）之间划分？本章讨论做出这些决定时要考虑的因素。这些因素中的一些已经在前面的章节中进行了讨论，但是为了完整起见，这里将对其进行重新讨论。

在决定是合并还是分开时，目标是降低整个系统的复杂性并改善其模块化。看来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越简单。但是，细分的行为会带来额外的复杂性，而这在细分之前是不存在的：

- 一些组件的复杂性仅来自组件的数量：组件越多，就越难以追踪所有组件，也就越难在大型集合中找到所需的组件。细分通常会导致更多接口，并且每个新接口都会增加复杂性。
- 细分可能会导致附加代码来管理组件。例如，在细分之前使用单个对象的一段代码现在可能必须管理多个对象。
- 细分产生分离：细分后的组件将比细分前的组件相距更远。例如，在细分之前位于单个类中的方法可能在细分之后位于不同的类中，并且可能在不同的文件中。分离使开发人员更难于同时查看这些组件，甚至很难知道它们的存在。如果组件真正独立，那么分离是好的：它使开发人员可以一次专注于单个组件，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性，则分离是不好的：开发人员最终将在组件之间来回翻转。更糟糕的是，他们可能不了解依赖关系，这可能导致错误。
- 细分可能导致重复：细分之前的单个实例中存在的代码可能需要存在于每个细分的组件中。

如果它们紧密相关，则将代码段组合在一起是最有益的。如果各部分无关，则最好分开。以下是两个代码相关的一些提示：

- 它们共享信息；例如，这两段代码都可能取决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都可能同时使用另一段代码。这种关系形式只有在双向关系中才具有吸引力。作为反例，磁盘块高速缓存几乎总是包含哈希表，但是哈希表可以在许多不涉及块高速缓存的情况下使用。因此，这些模块应该分开。
- 它们在概念上重叠，因为存在一个简单的更高级别的类别，其中包括这两段代码。例如，搜索子字符串和大小写转换都属于字符串操作类别。流控制和可靠的交付都属于网络通信的范畴。
- 不看其中的一段代码就很难理解。

本章的其余部分使用更具体的规则以及示例来说明何时将代码段组合在一起以及何时将它们分开是有意义的。

9.1 如果信息是共享的则聚合起来

5.4 节在实现 **HTTP** 服务器的项目上下文中介绍了此原理。在其第一个实现中，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

放置在字符串对象中。第二种方法解析字符串以提取请求的各个组成部分。经过这种分解，这两种方法最终都对 HTTP 请求的格式有了相当的了解：第一种方法只是尝试读取请求，而不是解析请求，但是如果不执行大多数操作，就无法确定请求的结束解析它的工作（例如，它必须解析标头行才能识别包含整个请求长度的标头）。由于此共享信息，最好在同一位置读取和解析请求；当两个类合而为一时，代码变得更短，更简单。

9.2 如果它能简化接口则聚合起来

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解决方案的一部分时，通常会发生这种情况。在上一部分的 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。当这些方法结合在一起时，这些接口就被淘汰了。

另外，将两个或更多类的功能组合在一起时，可能会自动执行某些功能，因此大多数用户无需了解它们。Java I/O 库说明了这种机会。如果将 FileInputStream 和 BufferedInputStream 类组合在一起，并且默认情况下提供了缓冲，则绝大多数用户甚至都不需要知道缓冲的存在。组合的 FileInputStream 类可能提供禁用或替换默认缓冲机制的方法，但是大多数用户不需要了解它们。

9.3 聚合起来消除重复

如果发现反复重复相同的代码模式，请查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为一个单独的方法，并用对该方法的调用替换重复的代码段。如果重复的代码段很长并且替换方法具有简单的签名，则此方法最有效。如果代码段只有一两行，那么用方法调用替换它可能不会有太多好处。如果代码段与其环境以复杂的方式进行交互（例如，通过访问多个局部变量），则替换方法可能需要复杂的签名（例如，许多“按引用传递”参数），这会降低其价值。

消除重复的另一种方法是重构代码，使相关代码段仅需要在一个地方执行。假设你正在编写一种方法，该方法需要在几个不同的点返回错误，并且在返回之前需要在每个这些点执行相同的清除操作（示例请参见图 9.1）。如果编程语言支持 goto，则可以将清除代码移到方法的最后，然后在需要返回错误的每个点处转到该片段，如图 9.2 所示。Goto 语句通常被认为是一个坏主意，如果不加选择地使用它们，可能会导致无法识别的代码，但是在诸如此类的情况下，它们可用于从嵌套代码中转义，因此它们非常有用。

9.4 单独通用的和专用的代码

如果模块包含可用于多种不同目的的机制，则它应仅提供一种通用机制。它不应包含专门针对特定用途的机制的代码，也不应包含其他通用机制。与通用机制关联的专用代码通常应放在不同的模块中（通常是与特定用途关联的模块）。第 6 章中的 GUI 编辑器讨论阐明了这一原理：最佳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（例如删除所选内容）则在用户界面模块中实现。

【Red Flag：重复】

如果相同的代码（或几乎相同的代码）一遍又一遍地出现，那是一个危险信号，你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而言太短，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 中的代码进行了重新组织，因此只有 LOG 语句的一个副本。

通常，系统的下层倾向于更通用，而上层则更专用。例如，应用程序的最顶层包含完全特定于该应用程序的功能。将专用代码与通用代码分开的方法是将专用代码向上拉到较高的层，而将较低的层保留为通用。当你遇到同时包含通用功能和专用功能的同一类的类时，请查看该类是否可以分为两个类，一个包含通用功能，另一个在其上分层以提供特殊功能

9.5 实例：插入光标和选择

下一节将通过三个示例说明上述原理。在两个示例中，最好的方法是分离相关的代码段。在第三个示例中，最好将它们结合在一起。

第一个示例由插入光标和第 6 章的 GUI 编辑器项目中的选择组成。编辑器显示闪烁的垂直线，指示用户键入的文本将出现在文档中的何处。它还显示了一个突出显示的字符范围，称为选择，用于复制或删除文本。插入光标始终可见，但是有时可能没有选择文本。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相关。例如，光标始终位于所选内容的一端，并且倾向于将光标和所选内容一起操作：单击并拖动鼠标将它们都设置，然后插入文本会首先删除所选的文本（如果有），然后在光标位置插入新文本。因此，使用单个对象管理选择和光标似乎合乎逻辑，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，以及布尔值，它们指示光标的哪一端以及选择是否存在。

但是，合并的对象很尴尬。它对高级代码没有任何好处，因为高级代码仍然需要将选择和游标视为不同的实体，并且对它们进行单独操作（在插入文本期间，它首先在组合对象上调用一个方法来删除选定的文本；然后调用另一个方法来检索光标位置，以插入新文本）。实际上，组合对象比单独的对象实现起来要

复杂得多。它避免了将光标位置存储为单独的实体，而是不得不存储一个布尔值，该布尔值指示选择的哪一端是光标。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结尾。

【Red Flag：Special-General Mixture】

当通用机制还包含专门用于该机制的特定用途的代码时，就会出现此红色标志。这使该机制更加复杂，并在该机制与特定用例之间造成了信息泄漏：对用例的未来修改也可能需要对基础机制进行更改。

在这种情况下，选择和光标之间的关联度不足以将它们组合在一起。当修改代码以分隔选择和光标时，用法和实现都变得更加简单。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。游标的实现也变得更加简单，因为游标的位置是直接表示的，而不是通过选择和布尔值间接表示的。实际上，在修订版中，没有特殊的类用于选择或游标。相反，引入了一个新的 Position 类来表示文件中的位置（行号和行内的字符）。选择用两个位置表示，光标用一个位置表示。职位还在项目中找到了其他用途。

9.6 实例：用户记录日志的单独类

第二个示例涉及学生项目中的错误记录。一个类包含几个代码序列，如下所示：

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

而不是在检测到错误时记录错误，而是调用特殊错误记录类中的单独方法。错误记录类是在同一源文件的末尾定义的：

```
private static class NetworkErrorLogger {

    /**
    *  Output information relevant to an error that occurs when trying
    *  to open a connection to send an RPC.
    *
    *  @param req
    *   The RPC request that would have been sent through the connection
    *  @param dest
    *      The destination of the RPC
    *  @param e
    *      The caught error
    */
```

```
public static void logRpcOpenError(RpcRequest req, AddrPortTuple dest, Exception e) {
        logger.log(Level.WARNING, "Cannot send message: " + req + ". \n" + "Unable to
        find or open connection to " + dest + " :" + e);

}
...

}
```

NetworkErrorLogger 类包含几个方法，例如 logRpcSendError 和 logRpcReceiveError，每个方法都记录了不同类型的错误。

这种分离增加了复杂性，没有任何好处。日志记录方法很浅：大多数只包含一行代码，但是它们需要大量的文档。每个方法仅在单个位置调用。日志记录方法高度依赖于它们的调用：读取调用的人很可能会切换到日志记录方法，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转到调用站点以了解该方法的目的。

在此示例中，最好消除日志记录方法，并将日志记录语句放置在检测到错误的位置。这将使代码更易于阅读，并消除了日志记录方法所需的接口。

9.7 实例：编辑器的撤消机制

在 6.2 节的 GUI 编辑器项目中，要求之一是支持多级撤消/重做，不仅要更改文本本身，还要更改选择，插入光标和视图。例如，如果用户选择了一些文本，将其删除，滚动到文件中的其他位置，然后调用 undo，则编辑器必须将其状态恢复为删除前的状态。这包括还原已删除的文本，再次选择它，并使所选的文本在窗口中可见。

一些学生项目将整个撤消机制实现为文本类的一部分。文本类维护所有不可撤消更改的列表。每当更改文本时，它将自动将条目添加到此列表中。为了更改选择，插入光标和视图，用户界面代码调用了文本类中的其他方法，然后将这些更改的条目添加到撤消列表中。当用户请求撤消或重做时，用户界面代码将调用文本类中的方法，该方法然后处理撤消列表中的条目。对于与文本相关的条目，它更新了文本类的内部。对于与其他事物（例如选择）相关的条目，将调用返回到用户界面代码的文本类来执行撤消或重做。

这种方法在文本类中导致了一系列尴尬的功能。撤消/重做的核心由通用机制组成，用于管理已执行的动作列表，并在撤消和重做操作期间逐步执行这些动作。核心与专用处理程序一起位于 text 类中，该专用处理程序对诸如文本和选择之类的特定内容实现了撤消和重做。用于选择和光标的专用撤消处理程序与文本类中的任何其他内容均无关。它们导致文本类和用户界面之间的信息泄漏，以及每个模块中来回传递撤消信息的额外方法。如果将来将新的可撤消实体添加到系统中，则将需要更改文本类，包括特定于该实体的新方法。

通过提取撤消/重做机制的通用核心并将其放在单独的类中，可以解决这些问题：

```
public class History {
    public interface Action {
```

```
        public void redo();
        public void undo();
    }
    History() {...}
    void addAction(Action action) {...}
    void addFence() {...}
    void undo() {...}
    void redo() {...}
}
```

在此设计中，History 类管理实现接口 History.Action 的对象的集合。每个 History.Action 描述一个操作，例如插入文本或更改光标位置，并且它提供了可以撤消或重做该操作的方法。History 类对操作中存储的信息或它们如何实现其撤消和重做方法一无所知。历史记录维护一个历史记录列表，该列表描述了应用程序整个生命周期中执行的所有操作，并且它提供了撤消和重做方法，以响应用户请求的撤消和重做而在列表中前后移动，并在应用程序中调用撤消和重做方法。历史动作。

历史。动作是特殊目的的对象：每个人都了解一种特殊的不可操作。它们在 History 类之外的模块中实现，这些模块可以理解特定类型的可撤销操作。文本类可能实现 UndoableInsert 和 UndoableDelete 对象，以描述文本的插入和删除。每当插入文本时，文本类都会创建一个描述该插入的新 UndoableInsert 对象，并调用 History.addAction 将其添加到历史列表中。编辑器的用户界面代码可能会创建 UndoableSelection 和 UndoableCursor 对象，这些对象描述对选择和插入光标的更改。

History 类还允许对操作进行分组，例如，来自用户的单个撤消请求可以恢复已删除的文本，重新选择已删除的文本以及重新放置插入光标。有多种将动作分组的方法。历史记录类使用围栏，围栏是放置在历史记录列表中的标记，用于分隔相关动作的组。每次对 History.redo 的调用都会向后浏览历史记录列表，撤消操作，直到到达下一个栅栏。围栏的位置由更高级别的代码通过调用 History.addFence 确定。

这种方法将撤消功能分为三类，每类都在不同的地方实现：
一种用于管理和分组动作以及调用撤消/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，每个类都了解少量的操作类型）。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施。历史课不知道要撤消哪种操作；它可以用于多种应用。每个动作类仅理解一种动作，并且历史记录类和动作类都不需要知道将动作分组的策略。

关键的设计决策是将撤消机制的通用部分与专用部分分开，然后将通用部分单独放在一个类中的决定。一旦完成，其余的设计就会自然消失。

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，特殊用途的撤消代码(例如撤消文本插入的代码)应该与通用用途的撤消代码(例如管理历史记录列表的代码)分开。然而，将一种机制的专用代码与另一种机制的通用代码组合起来通常是有意义的。text 类就是这样一个例子:它实现了一种管理文本的通用机制，但是它包含了与撤销相关的专用代码。撤消代码是专用的，因为它只处理文本修改的撤消操作。将这段代码与 History 类中通用的 undo 基础结构结合在一起是没有意义的，但是将它放在 text 类中是有意义的，因为它与其他文本函数密切相关。

9.8 拆分和合并方法

何时细分的问题不仅适用于类，而且还适用于方法：是否有时最好将现有方法分为多个较小的方法？还是应该将两种较小的方法合并为一种较大的方法？长方法比短方法更难于理解，因此许多人认为仅长度是分解方法的一个很好的理由。课堂上的学生通常会获得严格的标准，例如“拆分超过 20 行的任何方法！”

但是，长度本身很少是拆分方法的一个很好的理由。通常，开发人员倾向于过多地分解方法。拆分方法会引入其他接口，从而增加了复杂性。它还将原始方法的各个部分分开，如果这些部分实际上是相关的，则使代码更难阅读。你不应该分解一种方法，除非它使整个系统更加简单；我将在下面讨论这种情况。

长方法并不总是坏的。例如，假设一个方法包含按顺序执行的五个 20 行代码块。如果这些块是相对独立的，则可以一次读取并理解该方法的一个块。将每个块移动到单独的方法中并没有太大的好处。如果这些块具有复杂的交互作用，则将它们保持在一起就显得尤为重要，这样读者就可以一次看到所有代码。如果每个块使用单独的方法，则读者将不得不在这些扩展方法之间来回切换，以了解它们如何协同工作。如果方法具有简单的签名并且易于阅读，则包含数百行代码的方法就可以了。这些方法很深入（很多功能，简单的接口），很好。

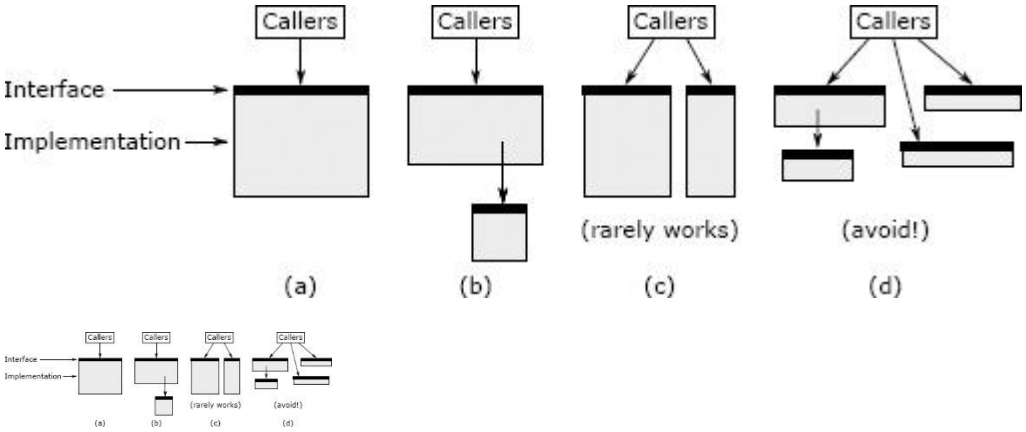

图 9.3：方法（a）可以通过提取子任务（b）或将其功能划分为两个单独的方法（c）进行拆分。如果方法导致浅层方法，则不应拆分该方法，如（d）所示。

设计方法时，最重要的目标是提供简洁的抽象。每种方法都应该做一件事并且完全做到这一点。该方法应该具有简洁的接口，以便用户无需费神就可以正确使用它。该方法应该很深：其接口应该比其实现简单得多。如果一个方法具有所有这些属性，那么它的长短与否可能无关紧要。

总体而言，拆分方法只有在其导致更抽象的抽象时才有意义。有两种方法可以做到这一点，如图 9.3 所示。最佳方法是将子任务分解为单独的方法，如图 9.3（b）所示。该细分产生一个包含该子任务的子方法和一个包含原始方法其余部分的父方法；父级调用子级。新的父方法的接口与原始方法的接口相同。如果存在一个与原始方法的其余部分完全可分离的子任务，则这种细分形式是有意义的，这意味着（a）读取子方法的某人不需要了解有关父方法的任何信息，以及（b）某人在阅读父方法不需要了解子方法的实现。通常，这意味着子方法是相对通用的：可以想象除父方法外，其他方法也可以使用它。如果你对这种

形式进行拆分，然后发现自己在父母和孩子之间来回翻转以了解他们如何一起工作，那是一个红色标记（“联合方法”），表明拆分可能不是一个好主意。

分解方法的第二种方法是将其拆分为两个单独的方法，每个方法对原始方法的调用者可见，如图 9.3（c）所示。如果原始方法的接口过于复杂，这是有道理的，因为该接口试图执行不密切相关的多项操作。在这种情况下，可以将方法的功能划分为两个或更多个较小的方法，每个方法仅具有原始方法功能的一部分。如果进行这样的拆分，则每个结果方法的接口应该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之一即可；如果调用者必须同时调用这两个新方法，则将增加复杂性，从而降低拆分是个好主意的可能性。新方法将更加专注于它们的工作。如果新方法比原始方法更具通用性，那么这是一个好兆头（例如，你可以想象在其他情况下单独使用它们）。

图 9.3（c）所示形式的拆分并不是很有意义，因为它们导致调用者不得不处理多个方法而不是一个方法。当你以这种方式拆分时，你可能会遇到几种浅层方法的风险，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，则拆分不是一个好主意。如果你正在考虑像图 9.3（c）所示的拆分，则应基于它是否简化了呼叫者的情况来进行判断。

在某些情况下，通过将方法结合在一起可以简化系统。例如，连接方法可以用一种更深的方法代替两种浅的方法。它可以消除重复的代码；它可以消除原始方法或中间数据结构之间的依赖关系；它可能导致更好的封装，从而使以前在多个位置存在的知识现在被隔离在一个位置；否则可能会导致接口更简单，如 9.2 节所述。

应该有可能独立地理解每种方法。如果你不能不理解另一种方法的实现而无法理解一种方法的实现，那就是一个危险信号。该危险信号也可以在其他情况下发生：如果两段代码在物理上是分开的，但是只有通过查看另一段代码才能理解它们，这就是危险信号。

9.9 结论

拆分或加入模块的决定应基于复杂性。选择一种结构，它可以隐藏最佳的信息，最少的依赖关系和最深的接口。

第十章 定义不存在的错误

异常处理是软件系统中最糟糕的复杂性来源之一。处理特殊情况的代码在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定义异常时不考虑异常的处理方式。本章讨论了为什么异常对复杂性的贡献不成比例，然后说明了如何简化异常处理。本章总的主要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，以便正常行为可以处理所有情况，并且没有要报告的特殊条件（因此，本章标题）。

10.1 为什么异常会增加复杂性

我使用“异常”一词来指代任何会改变程序中正常控制流程的不常见条件。许多编程语言都包含一种正式的异常机制，该机制允许异常由低级代码引发并由封闭代码捕获。但是，即使不使用正式的异常报告机制，异常也可能发生，例如，当某个方法返回一个特殊值指示其未完成其正常行为时。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：

- 调用方可能会提供错误的参数或配置信息。
- 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。
- 该代码可能会检测到错误，内部不一致或未准备处理的情况。

大型系统必须应对许多特殊情况，特别是在它们是分布式的或需要容错的情况下。异常处理可以占系统中所有代码的很大一部分。

异常处理代码天生就比正常情况下的代码更难写。异常中断了正常的代码流;它通常意味着某事没有像预期的那样工作。当异常发生时，程序员可以用两种方法处理它，每种方法都很复杂。第一种方法是向前推进并完成正在进行的工作，尽管存在例外。例如，如果一个网络数据包丢失，它可以被重发;如果数据损坏了，也许可以从冗余副本中恢复数据。第二种方法是中止正在进行的操作，向上报告异常。但是，中止可能很复杂，因为异常可能发生在系统状态不一致的地方(数据结构可能已经部分初始化);异常处理代码必须恢复一致性，例如通过撤销发生异常之前所做的任何更改。

此外，异常处理代码为更多异常创造了机会。考虑重新发送丢失的网络数据包的情况。也许该数据包实际上并没有丢失，但是只是被延迟了。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；这引入了对等方必须处理的新的例外条件。或者，考虑从冗余副本恢复丢失的数据的情况：如果冗余副本也丢失了怎么办？在恢复期间发生的次要异常通常比主要异常更加微妙和复杂。如果通过中止正在进行的操作来处理异常，则必须将此异常作为另一个异常报告给调用方。为了防止无休止的异常级联，开发人员最终必须找到一种在不引入更多异常的情况下处理异常的方法。

语言对异常的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，该代码使用 Java 对对象序列化和反序列化的支持从文件中读取 tweet 的集合：

```
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) {
    // Not a problem: not all tweet files have full
    // set of tweets.
}
catch (IOException e) {
    ...
}
catch (ClassCastException e) {
    ...
}
```

只是基本的 try-catch 样板代码比正常情况下的操作代码所占的代码行更多，甚至没有考虑实际处理异常的代码。很难将异常处理代码与普通情况代码相关联：例如，每个异常的生成位置都不明显。另一种方法是将代码分解为许多不同的 try 块。在极端情况下，可能会尝试尝试每行可能产生异常的代码。这样可以清楚地说明异常发生的位置，但是 try 块本身会破坏代码流，并使代码难以阅读。此外，某些异常处理代码可能最终会在多个 try 块中重复。

确保异常处理代码真正起作用是困难的。某些异常（例如 I/O 错误）在测试环境中不易生成，因此很难测试处理它们的代码。异常在运行的系统中很少发生，因此异常处理代码很少执行。错误可能会长时间未被发现，并且当最终需要异常处理代码时，它很有可能无法正常工作（我最喜欢的一句话是：“未执行的代码无效”） 。最近的一项研究发现，分布式数据密集型系统中超过 90%的灾难性故障是由错误的错误处理引起的 1。当异常处理代码失败时，很难调试该问题，因为它很少发生。

10.2 异常太多

程序员通过定义不必要的异常加剧了与异常处理有关的问题。告诉大多数程序员，检测和报告错误很重要。他们通常将其解释为“检测到的错误越多越好”。这导致了一种过分防御的风格，其中任何看起来甚至有点可疑的东西都被拒绝，并带有异常，这导致了不必要的异常的泛滥，从而增加了系统的复杂性。

在设计 Tcl 脚本语言时，我自己就犯了这个错误。Tcl 包含一个未设置的命令，可用于删除变量。我定义了 unset 以便在变量不存在时抛出错误。当时我认为，如果有人试图删除一个不存在的变量，那么它一定是一个 bug，所以 Tcl 应该报告它。然而，unset 最常见的用途之一是清理以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，特别是在操作中途中止的情况下。因此，最简单的方法是删除可能已经创建的所有变量。unset 的定义使得这种情况很尴尬:开发人员最终会在 catch 语句中封装对 unset 的调用，以捕获并忽略 unset 抛出的错误。回顾过去，unset 命令的定义是我在 Tcl 设计中犯下的最大错误之一。

试图使用异常来避免处理困难的情况很诱人：与其想出一种干净的方法来处理它，不如抛出一个异常并将问题平移给调用者。有人可能会争辩说，这种方法可以赋予调用者权力，因为它允许每个调用者以不同的方式处理异常。但是，如果你在确定特定情况下该怎么做时遇到困难，则呼叫者很可能都不知道该怎么办。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，并且比具有较少异常的类浅。异常是接口中特别复杂的元素。它可以在被捕获之前通过多个堆栈级别向上传播，因此它不仅影响方法的调用者，而且还可能影响更高级别的调用者（及其接口）。

抛出异常很容易；处理它们很困难。因此，异常的复杂性来自异常处理代码。减少由异常处理引起的复杂性破坏的最佳方法是减少必须处理异常的位置的数量。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 定义未知的错误

消除异常处理复杂性的最好方法是定义你的 API，以便没有异常要处理：定义错误而已。这似乎是牺牲品，但在实践中非常有效。考虑上面讨论的 Tcl unset 命令。而不是在要求 unset 删除未知变量时引发错误，它应该只是返回而无需执行任何操作。我应该稍微修改一下 unset 的定义：与其删除一个变量，不应该删除 unset 来确保一个变量不再存在。根据第一个定义，如果变量不存在，则 unset 不能执行其工作，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 是很自然的。在这种情况下，它的工作已经完成，因此可以简单地返回。

10.4 实例：在 windows 中删除文件

文件删除提供了如何定义错误的另一个示例。Windows 操作系统不允许删除文件（如果已在进程中打开文件）。对于开发人员和用户来说，这是不断沮丧的根源。为了删除正在使用的文件，用户必须在系统中搜索以找到已打开文件的进程，然后终止该进程。有时用户放弃并重新启动系统，只是为了删除文件。

Unix 操作系统更优雅地定义了文件删除。在 Unix 中，如果在删除文件时打开了文件，则 Unix 不会立即删除该文件。而是将文件标记为删除，然后删除操作成功返回。该文件名已从其目录中删除，因此其他进程无法打开该旧文件，并且可以创建具有相同名称的新文件，但现有文件数据将保留。已经打开文件的进程可以继续读取和正常写入文件。一旦所有访问进程都关闭了文件，便释放其数据。

Unix 方法定义了两种不同的错误。首先，如果文件当前正在使用中，则删除操作不再返回错误；删除成功，该文件最终将被删除。其次，删除正在使用的文件不会为使用该文件的进程创建例外。解决此问题的

一种可能方法是立即删除文件并标记文件的所有打开以禁用它们。其他进程读取或写入已删除文件的任何尝试均将失败。但是，此方法将为那些要处理的过程创建新的错误。相反，Unix 允许他们继续正常访问文件。延迟文件删除将定义错误不存在。

Unix 允许进程继续读取和写入已损坏的文件可能看起来很奇怪，但是我从未遇到过引起严重问题的情况。对于开发人员和用户，Unix 删除文件的定义比 Windows 定义要容易得多。

10.5 实例：Java substring 方法

作为最后一个示例，请考虑 Java String 类及其子字符串方法。给定一个字符串中的两个索引，substring 返回该子字符串，该字符串从第一个索引给定的字符开始，以第二个索引之前的字符结束。但是，如果两个索引中的任何一个都超出字符串的范围，则子字符串将引发 IndexOutOfBoundsException。此异常是不必要的，并且会使此方法的使用复杂化。我经常发现自己处于一个或两个索引可能不在字符串范围内的情况，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，这要求我检查每个索引并将它们向上舍入为零或向下舍入到字符串的末尾。现在，单行方法调用变成 5-10 行代码。

如果 Java 子字符串方法自动执行此调整，则将更易于使用，因此它实现了以下 API：“返回索引大于或等于 beginIndex 且小于 endIndex 的字符串的字符（如果有）。” 这是一个简单自然的 API，它定义了 IndexOutOfBoundsException 异常。现在，即使一个或两个索引均为负，或者 beginIndex 大于 endIndex，该方法的行为也已明确定义。这种方法简化了方法的 API，同时增加了其功能，因此使方法更深。许多其他语言都采用了无错误的方法。例如，Python 对于超出范围的列表切片返回空结果。

当我主张定义错误而不再存在时，人们有时会反驳说抛出错误会捕获错误。如果错误定义不存在，那会不会导致 Buggier 软件出现？也许这就是 Java 开发人员决定子字符串应引发异常的原因。错误的方法可能会捕获一些错误，但也会增加复杂性，从而导致其他错误。在错误有效的方法中，开发人员必须编写额外的代码来避免或忽略错误，这增加了发生错误的可能性。或者，他们可能会忘记编写其他代码，在这种情况下，运行时可能会引发意外错误。相反，定义错误而不存在将简化 API，并减少必须编写的代码量。

总体而言，减少错误的最好方法是简化软件。

10.6 掩藏异常

减少必须处理异常的地方数量的第二种技术是异常屏蔽。使用这种方法，可以在系统的较低级别上检测和处理异常情况，因此，更高级别的软件无需知道该情况。异常屏蔽在分布式系统中尤其常见。例如，在诸如 TCP 的网络传输协议中，由于各种原因（例如损坏和拥塞），可能会丢弃数据包。TCP 通过在其实现中重新发送丢失的数据包来掩盖数据包的丢失，因此所有数据最终都将通过，并且客户端不知道丢失的数据包。

NFS 网络文件系统中出现了一个更具争议性的屏蔽示例。如果 NFS 文件服务器由于任何原因崩溃或无法响应，客户端将一遍又一遍地向服务器发出请求，直到问题最终得到解决。客户端上的低级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（以及因此的应用程序）只是挂起，直到操作可以

成功完成。如果挂起持续的时间不超过一小段时间，则 NFS 客户端将在用户控制台上以“ NFS 服务器 xyzzy 无法响应仍在尝试响应”的形式打印消息。

NFS 用户经常抱怨这样的事实，即他们的应用程序在等待 NFS 服务器恢复正常运行时挂起。许多人建议 NFS 应该异常终止操作而不是挂起。但是，报告异常会使情况更糟，而不是更好。如果应用程序无法访问其文件，则无能为力。一种可能性是应用程序重试文件操作，但这仍然会使应用程序挂起，并且在 NFS 层中的一个位置执行重试会比在每个应用程序中的每个文件系统调用处执行重试更容易（编译器应不必为此担心！）。另一种选择是让应用程序中止并将错误返回给调用者。呼叫者不太可能知道该怎么做，因此他们也将中止，导致用户工作环境崩溃。用户在文件服务器关闭时仍然无法完成任何工作，并且一旦文件服务器恢复工作，他们将不得不重新启动所有应用程序。

因此，最好的替代方法是让 NFS 掩盖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运行，它们就可以无缝恢复。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，但是在它起作用的情况下它是一个强大的工具。它导致了更深的类，因为它减少了类的界面（用户需要注意的异常更少）并以掩盖异常的代码形式添加了功能。异常屏蔽是降低复杂性的一个例子。

10.7 异常聚合

减少与异常相关的复杂性的第三种技术是异常聚合。异常聚合的思想是用一个代码段处理许多异常。与其为多个单独的异常编写不同的处理程序，不如用一个处理程序将它们全部处理在一个地方。

考虑如何处理 Web 服务器中缺少的参数。Web 服务器实现 URL 的集合。服务器收到传入的 URL 时，将分派到特定于 URL 的服务方法来处理该 URL 并生成响应。该 URL 包含用于生成响应的各种参数。每个服务方法都将调用一个较低层的方法（将其称为 getParameter）以从 URL 中提取所需的参数。如果 URL 不包含所需的参数，则 getParameter 会引发异常。

当参加软件设计课程的学生实现这样的服务器时，他们中的许多人将对 getParameter 的每个不同调用包装在单独的异常处理程序中以捕获 NoSuchParameter 异常，如图 10.1 所示。这导致大量的处理程序，所有这些处理程序基本上都执行相同的操作（生成错误响应）。

Dispatcher:

```
...
if (...) {
  handleUrl1(...);
} else if (...) {
  handleUrl2(...);
} else if (...) {
  handleUrl3(...);
} else if (...)
  ...
}
...
```

handleUrl1:

```
...
try {
  ... getParameter("photo_id")
  ...
} catch (NoSuchParameter e) {
  ...
}
...
try {
  ... getParameter("message")
  ...
} catch (NoSuchParameter e) {
  ...
}
...
```

handleUrl2:

```
...
try {
  ... getParameter("user_id")
  ...
} catch (NoSuchParameter e) {
  ...
}
...
```

handleUrl3:

```
...
try {
  ... getParameter("login")
  ...
} catch (NoSuchParameter e) {
  ...
}
...
try {
  ... getParameter("password")
  ...
} catch (NoSuchParameter e) {
  ...
}
...
```

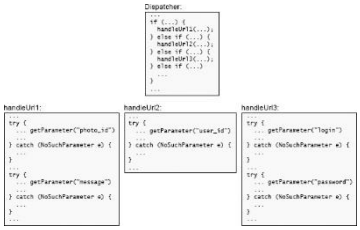

图 10.1：顶部的代码将分派给 Web 服务器中的几种方法之一，每种方法都处理一个特定的 URL。每个方法（底部）都使用传入 HTTP 请求中的参数。在此图中，每个对 getParameter 的调用都有一个单独的异常处理程序。这导致重复的代码。

更好的方法是汇总异常。让它们传播到 Web 服务器的顶级调度方法，而不是在单个服务方法中捕获异常，如图 10.2 所示。此方法中的单个处理程序可以捕获所有异常，并为丢失的参数生成适当的错误响应。

在 Web 示例中甚至可以采用聚合方法。处理网页时，除了缺少参数外，还有许多其他错误；例如，参数可能没有正确的语法（服务方法应为整数，但值为“xyz”），或者用户可能无权执行所请求的操作。在每种情况下，错误都应导致错误响应。错误仅在响应中包含的错误消息中有所不同（“URL 中不存在参数'数量'”或“'数量'参数的错误值'xyz'；必须为正整数”）。因此，所有导致错误响应的条件都可以使用单个顶级异常处理程序进行处理。错误消息可以在引发异常时生成，并作为变量包含在异常记录中。例如，getParameter 将生成“URL 中不存在的参数'数量'”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。

Dispatcher:

```
...
try {
  if (...) {
    handleUrl1(...);
  } else if (...) {
    handleUrl2(...);
  } else if (...) {
    handleUrl3(...);
  } else if (...)
    ...
  }
} catch (NoSuchParameter e) {
  send error response;
}
...
```

handleUrl1:

```
... getParameter("photo_id")
... getParameter("message")
...
```

handleUrl2:

```
... getParameter("user_id")
...
```

handleUrl3:

```
... getParameter("login")
... getParameter("password")
...
```

Figure 10.2: This code is functionally equivalent to Figure 10.1, but exception handling has been aggregated: a single exception handler in the dispatcher catches all of the NoSuchParameter exceptions from all of the URL-specific methods.

图 10.2：此代码在功能上等效于图 10.1，但是异常处理已聚合：分派器中的单个异常处理程序从所有特定于 URL 的方法中捕获所有 NoSuchParameter 异常。

从封装和信息隐藏的角度来看，上一段中描述的聚合具有良好的属性。顶级异常处理程序封装了有关如何生成错误响应的知识，但对特定错误一无所知。它仅使用异常中提供的错误消息。getParameter 方法封装了有关如何从 URL 提取参数的知识，并且还知道如何以人类可读的形式描述提取错误。这两个信息密切相关，因此将它们放在同一位置是很有意义的。但是，getParameter 对 HTTP 错误响应的语法一无所知。随着向 Web 服务器中添加了新功能，可能会创建具有类似自身错误的新方法，如 getParameter。

此示例说明了用于异常处理的通用设计模式。如果系统处理一系列请求，则定义一个异常以中止当前请求，清除系统状态并继续下一个请求非常有用。异常被捕获在系统请求处理循环顶部附近的单个位置。在处理中止请求的任何时候都可以抛出该异常。可以为不同的条件定义异常的不同子类。应该将这种类型的异常与对整个系统致命的异常区分开来。

如果异常在处理之前在堆栈中传播了多个级别，则异常聚集最有效。这样可以在同一位置处理更多方法的更多异常。这与异常屏蔽相反：如果使用低级方法处理异常，则屏蔽通常效果最好。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，允许传播异常会增加处理该异常的位置数。掩码和聚合的相似之处在于，这两种方法都将异常处理程序置于可以捕获最多异常的位置，从而消除了许多本来需要创建的处理程序。

RAMCloud 存储系统中发生异常聚集的另一个示例是崩溃恢复。RAMCloud 系统由一组存储服务器组成，这些存储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其所有数据，RAMCloud 会使用存储在其他服务器上的副本来重建丢失的数据。错误也可能在较小的范围内发生。例如，服务器可能发现单个对象已损坏。

对于每种不同类型的错误，RAMCloud 没有单独的恢复机制。相反，RAMCloud 将许多较小的错误“提升”为较大的错误。原则上，RAMCloud 可以通过从备份副本中恢复一个损坏的对象来处理这个损坏的对象。然而，它并不这样做。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，而且这种方法最小化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，而且这还意味着服务器崩溃恢复将更频繁地被调用。因此，恢复中的 bug 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，它大大增加了恢复成本。这在 RAMCloud 中不是问题，因为对象损坏非常罕见。但是，错误升级对于经常发生的错误可能没有意义。举一个例子，在服务器的任何网络数据包丢失时使服务器崩溃是不切实际的。

考虑异常聚合的一种方法是，它用可以处理多种情况的单个通用机制替换了几种针对特定情况而量身定制的特殊用途的机制。这再次说明了通用机制的好处。

10.8 崩溃了吗?

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，有些错误是不值得尝试的。通常，这些错误很难或不可能处理，而且很少发生。针对这些错误的最简单的操作是打印诊断信息，然后中止应用程序。

一个示例是在存储分配期间发生的“内存不足”错误。考虑一下 C 语言中的 malloc 函数，如果它无法分配所需的内存块，则该函数将返回 NULL。这是一个不幸的行为，因为它假定 malloc 的每个调用者都将检查返回值并在没有内存的情况下采取适当的措施。应用程序包含许多对 malloc 的调用，因此在每次调用后检查结果将增加相当大的复杂性。如果程序员忘记了检查（这很有可能），那么如果内存用完，应用程序将取消引用空指针，从而导致崩溃，从而掩盖了实际问题。

此外，当应用程序发现内存已用完时，它无能为力。原则上，应用程序可以寻找不需要的内存以释放它，但是如果应用程序有不需要的内存，它可以已经释放它，这首先可以防止内存不足错误。当今的系统具有如此大的内存，以至于内存几乎永远不会耗尽。如果是这样，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几乎没有道理。这会带来太多的复杂性，而带来的收益却太少。

更好的方法是定义一个新的 ckalloc 方法，该方法调用 malloc，检查结果，并在内存耗尽时通过错误消息中止应用程序。该应用程序从不直接调用 malloc。它总是调用 ckalloc。

在较新的语言（例如 C ++和 Java）中，如果内存耗尽，则 new 运算符将引发异常。捕获此异常没有什么意义，因为异常处理程序很有可能还会尝试分配内存，这也会失败。动态分配的内存是任何现代应用程序中的基本元素，如果内存耗尽，则继续应用程序是没有意义的。最好在检测到错误后立即崩溃。

还有许多其他错误示例，这些错误会使应用程序崩溃很有意义。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，则应用程序无济于事，因此中止了操作。清除错误消息是一种明智的方法。这些错误很少发生，因此它们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（如数据结构不一致），则错误消息中止也是合适的。这样的条件可能表明程序中存在错误。

在特定错误上崩溃是否可以接受取决于应用程序。对于复制的存储系统，不适合因 I/O 错误而中止。相反，系统必须使用复制的数据来恢复丢失的任何信息。恢复机制将给程序增加相当大的复杂性，但是恢复丢失的数据是系统为用户提供的价值的重要组成部分。

10.9 设计未知的特殊情况

出于同样的原因，定义不存在的错误是有意义的，而定义其他不存在的特殊情况也是有意义的。特殊情况可能导致代码中混入 if 语句，这使代码难以理解并导致错误。因此，应尽可能消除特殊情况。做到这一点的最佳方法是设计一种普通情况，这种方式可以自动处理特殊情况而无需任何额外的代码。

在第 6 章中描述的文本编辑器项目中，学生必须实现一种选择文本以及复制或删除所选内容的机制。大多数学生在他们的选择实现中引入了状态变量，以表明选择是否存在。他们之所以选择这种方法，是因为有时屏幕上看不到任何选择，因此在实现中似乎很自然地代表了这一概念。但是，这种方法导致大量检查以检测“无选择”条件并进行特殊处理。

通过消除“不选择”的特殊情况，可以简化选择处理代码，从而使选择始终存在。当屏幕上没有可见的选择时，可以在内部用空的选择表示，其开始和结束位置相同。使用这种方法，可以编写选择管理代码，而无需对“不选择”进行任何检查。复制所选内容时，如果所选内容为空，则将在新位置插入 0 字节（如果正确实现，则在特殊情况下无需检查 0 字节）。同样，应该有可能设计用于删除选择的代码，以便无需任何特殊情况检查就可以处理空情况。在一行上考虑所有选择。要删除选择，提取选择之前的行的一部分，并将其与选择之后的行的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。

此示例还说明了第 7 章中的“不同的层，不同的抽象”概念。“无选择”的概念在用户对应用程序界面的看法方面很有意义，但这并不意味着必须明确表示它在应用程序内部。选择总是存在的，但有时是空的，因此是不可见的，这样可以简化实现。

10.10 Talking is too far

定义异常或将其屏蔽在模块内部，仅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 子字符串方法，都是如此。在极少数情况下，呼叫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。

但是，有可能使这个想法太过分。在用于网络通信的模块中，一个学生团队掩盖了所有网络异常：如果发生网络错误，则模块将其捕获，丢弃并继续进行，就好像没有问题一样。这意味着使用该模块的应用程序无法确定消息是否丢失或对等服务器是否发生故障；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。

与软件设计中的许多其他领域一样，你必须确定哪些是重要的，哪些是不重要的。不重要的事物应该被隐藏起来，它们越多越好。但是，当某件事很重要时，必须将其暴露出来。

10.11 结论

任何形式的特殊情况都使代码更难以理解，并增加了发生错误的可能性。本章重点讨论异常，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。做到这一点的最佳方法是重新定义语义以消除错误条件。对于无法定义的异常，你应该寻找机会将它们掩盖到较低的水平，以免影响有限，或者将多个特殊情况的处理程序聚合到一个更通用的处理程序中。总之，这些技术可能会对整体系统复杂性产生重大影响。

1 丁元等 等人，“简单的测试可以防止最关键的故障：对分布式数据密集型系统中的生产故障的分析”，2014 USENIX 操作系统设计和实施大会。

第十一章 设计多个方案

设计软件非常困难，因此你对如何构造模块或系统的初步思考不太可能会产生最佳的设计。如果为每个主要设计决策考虑多个选项，最终将获得更好的结果：***设计两次***。

假设你正在设计用于管理 GUI 文本编辑器文件文本的类。第一步是定义该类将呈现给编辑器其余部分的接口。与其选择想到的第一个想法，不如考虑几种可能性。一种选择是面向行的界面，该界面具有插入，修改和删除整行文本的操作。另一个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，该接口可对可能跨越线边界的任意范围的字符进行操作。你无需确定每个替代方案的每个功能；在这一点上，勾勒出一些最重要的方法就足够了。

尝试选择彼此根本不同的方法；这样你将学到更多。即使你确定只有一种合理的方法，无论你认为有多糟糕，都应该考虑第二种设计。考虑该设计的弱点并将它们与其他设计的特征进行对比将很有启发性。

在对备选方案进行粗略设计之后，列出每个方案的优缺点。接口最重要的考虑因素是高级软件的易用性。在上面的示例中，面向行的界面和面向字符的界面都需要使用文本类的软件中的额外工作。面向行的界面将需要更高级别的软件来在部分行和多行操作（例如剪切和粘贴所选内容）期间拆分和合并行。面向字符的接口将需要循环来实现修改多个字符的操作。还值得考虑其他因素：

- 一种选择是否具有比另一种更简单的界面？在文本示例中，所有文本界面都相对简单。
- 一个接口比另一个接口更通用吗？
- 一个接口是否比另一个接口更有效地实现？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。

比较了备选设计之后，你将可以更好地确定最佳设计。最佳选择可能是这些选择之一，或者你可能发现可以将多个选择的功能组合到一个比任何原始选择都要好的新设计中。

有时，没有其他选择特别有吸引力。发生这种情况时，请查看是否可以提出其他方案。使用你在原始替代方案中发现的问题来推动新设计。如果你在设计文本类并且仅考虑面向行和面向字符的方法，则可能会注意到每个替代方案都比较笨拙，因为它需要更高级别的软件来执行其他文本操作。那是一个危险信号：如果要有一个文本类，它应该处理所有文本操作。为了消除其他文本操作，文本界面需要更紧密地匹配高级软件中发生的操作。这些操作并不总是对应于单个字符或一行。

两次设计原则可以在系统的许多级别上应用。对于模块，你可以首先使用此方法来选择接口，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，你可以考虑实现这些实现，例如行的链接列表，固定大小的字符块或“间隙缓冲区”。实现的目标与接口的目标是不同的：对于实现，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或将系统分解为主要模块时。在每种情况下，如果你可以比较几种选择，则更容易确定最佳方法。

对其进行两次设计不需要花费很多额外的时间。对于较小的模块（如课程），你可能不需要一两个小时就能考虑替代方法。与你将花费数天或数周时间来实施该课程相比，这是很少的时间。最初的设计实验可能

会导致明显更好的设计，这将比花两次设计时间所花的时间多。对于较大的模块，你将花费更多的时间进行初始设计探索，但是实现也将花费更长的时间，并且更好的设计所带来的好处也会更高。

我已经注意到，真正聪明的人有时很难接受两次设计原则。当他们长大后，聪明的人会发现，他们对任何问题的第一个快速构想就足以取得良好的成绩。无需考虑第二种或第三种可能性。这使得容易养成不良的工作习惯。但是，随着这些人变老，他们将被提升到越来越困难的环境中。最终，每个人 都达到了你的第一个想法不再足够好的地步。如果你想获得非常好的结果，那么无论你多么聪明，都必须考虑第二种可能性，或者第三种可能性。大型软件系统的设计属于此类：没有人能很好地在首次尝试时就将其正确。

不幸的是，我经常看到聪明的人坚持要实现第一个想到的想法，这会使他们无法发挥其真正的潜力（这也使他们沮丧地工作）。也许他们下意识地相信“聪明的人第一次就能做到”，因此，如果他们尝试多种设计，那将意味着他们毕竟并不聪明。不是这种情况。不是说你不聪明；问题真的很难解决！此外，这是一件好事：处理一个必须认真思考的难题比处理一个根本不需要思考的难题更有趣。

“两次设计”方法不仅可以改善你的设计，而且可以提高你的设计技能。设计和比较多种方法的过程将教你使设计更好或更坏的因素。随着时间的流逝，这将使你更容易排除不良的设计并磨练真正的出色设计。

第十二章 为什么写注释？四个原因

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和有效工作至关重要，但是注释的作用不止于此。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，***编写注释的过程（如果正确完成）将实际上改善系统的设计***。相反，如果没有很好的文档记录，那么好的软件设计会失去很多价值。

不幸的是，这种观点并未得到普遍认同。生产代码的很大一部分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。其他人则看到了注释中的价值，但不知何故从不动手编写它们。幸运的是，许多开发团队认识到了文档的价值，并且感觉这些团队的普及率正在逐渐提高。但是，即使在鼓励文档的团队中，注释也经常被视为繁琐的工作，而且许多开发人员也不了解如何编写注释，因此生成的文档通常是平庸的。文档不足会给软件开发带来巨大且不必要的拖累。

在本章中，我将讨论开发人员避免写注释的借口，以及注释真正重要的原因。然后，第 13 章将描述如何编写好的注释，其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。我希望这些章节能使你相信三件事：好的注释可以对软件的整体质量产生很大的影响；写好注释并不难；并且（可能很难相信）写注释实际上很有趣。

当开发人员不写注释时，他们通常会以以下一种或多种借口为自己的行为辩护：

- “好的代码可以自我记录。”

- “我没有时间写注释。”
- “注释过时，并会产生误导。”
- “我所看到的注释都是毫无价值的；何必？” 在以下各节中，我将依次讨论这些借口。

12.1 好的代码本身就是文档

有人认为，如果代码编写得当，那么显而易见，不需要注释。这是一个美味的神话，就像谣言说冰淇淋对你的健康有益：我们真的很想相信！不幸的是，事实并非如此。可以肯定的是，在编写代码时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（请参阅第 14 章）。尽管如此，仍有大量设计信息无法用代码表示。例如，只能在代码中正式指定类接口的一小部分，例如其方法的签名。接口的非正式方面，例如对每种方法的作用或其结果含义的高级描述，只能在注释中描述。

一些开发人员认为，如果其他人想知道某个方法的作用，那么他们应该只阅读该方法的代码：这将比任何注释都更准确。读者可能会通过阅读其代码来推断该方法的抽象接口，但这既费时又痛苦。另外，如果在编写代码时期望用户会阅读方法实现，则将尝试使每个方法尽可能短，以便于阅读。如果该方法执行了一些重要操作，则将其分解为几个较小的方法。这将导致大量浅层方法。此外，它并没有真正使代码更易于阅读：为了理解顶层方法的行为，读者可能需要了解嵌套方法的行为。

此外，注释是抽象的基础。回顾第四章，抽象的目的是隐藏复杂性：抽象是实体的简化视图，该实体保留必要的信息，但忽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，则没有任何抽象：方法的所有复杂性都将暴露出来。没有注释，方法的唯一抽象就是其声明，该声明指定其名称以及其参数和结果的名称和类型。该声明缺少太多基本信息，无法单独提供有用的抽象。例如，提取子字符串的方法可能有两个参数，开始和结束，表示要提取的字符范围。仅凭宣言，无法确定提取的子字符串是否将包含 end 指示的字符，或者如果 start> end 会发生什么。注释使我们能够捕获调用者所需的其他信息，从而在隐藏实现细节的同时完成简化的视图。用人类语言（例如英语）写注释也很重要；这使它们不如代码精确，但提供了更多的表达能力，因此我们可以创建简单直观的描述。如果要使用抽象来隐藏复杂性，则注释必不可少。

12.2 我没有时间写注释

优先考虑低于其他开发任务的注释是很诱人的。在添加新功能和记录现有功能之间做出选择之后，选择新功能似乎合乎逻辑。但是，软件项目几乎总是处于时间压力之下，并且总会有比编写注释优先级更高的事情。因此，如果你允许取消对文档的优先级，则最终将没有文档。

与该借口相反的是第 15 页上讨论的投资思路。如果你想要一个干净的软件结构，可以长期有效地工作，那么你必须花一些额外的时间才能创建该结构。好的注释对软件的可维护性有很大的影响，因此花费在它们上面的精力将很快收回成本。此外，撰写注释不需要花费很多时间。询问自己，假设你不包含任何注释，那么你花费了多少开发时间来键入代码（与设计，编译，测试等相对）。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与输入代码所花费的时间一样多。这应该是一个安全的上限。基于这些假设，撰写好的注释不会增加你的开发时间约 10%。拥有良好文档的好处将迅速抵消这一成本。

此外，许多最重要的注释是与抽象有关的注释，例如类和方法的顶级文档。第 15 章认为，这些注释应作为设计过程的一部分编写，并且编写文档的行为是改善整体设计的重要设计工具。这些注释立即付诸行动。

12.3 注释过时，容易误导

注释有时确实会过时，但这实际上并不是主要问题。使文档保持最新状态并不需要付出巨大的努力。仅当对代码进行了较大的更改时才需要对文档进行大的更改，并且代码更改将比文档的更改花费更多的时间。第 16 章讨论了如何组织文档，以便在修改代码后尽可能容易地对其进行更新（主要思想是避免重复的文档并使文档与相应的代码保持一致）。代码审查提供了一种检测和修复陈旧注释的强大机制。

12.4 我看到所有的注释都毫无价值

在这四个借口中，这可能是最有价值的借口。每个软件开发人员都看到没有提供有用信息的注释，并且大多数现有文档充其量都是这样。幸运的是，这个问题是可以解决的。一旦知道了如何编写可靠的文档并不难。下一章将为如何编写良好的文档并随时间进行维护提供一个框架。

12.5 写好注释的益处

既然我已经讨论了（并希望揭穿了这些）反对撰写注释的论点，让我们考虑一下从良好注释中将获得的好处。***注释背后的总体思想是捕获设计者所想但不能在代码中表示的信息。***这些信息从低级详细信息（例如，激发特殊代码的硬件怪癖）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，这些注释将使他们能够更快，更准确地工作。没有文档，未来的开发人员将不得不重新编写或猜测开发人员的原始知识。这将花费额外的时间，并且如果新开发者误解了原始设计者的意图，则存在错误的风险。

第 2 章介绍了在软件系统中表现出复杂性的三种方式：
变更放大：看似简单的变更需要在许多地方进行代码修改。
认知负荷：为了进行更改，开发人员必须积累大量信息。
未知的未知：尚不清楚需要修改哪些代码，或必须考虑哪些信息才能进行这些修改。

好的文档可以帮助解决最后两个问题。通过为开发人员提供他们进行更改所需的信息，并使开发人员容易忽略不相关的信息，文档可以减轻认知负担。没有足够的文档，开发人员可能必须阅读大量代码才能重构设计人员的想法。文档还可以通过阐明系统的结构来减少未知的未知数，从而可以清楚地了解与任何给定更改相关的信息和代码。

第 2 章指出，导致复杂性的主要原因是依赖性和隐晦性。好的文档可以阐明依赖关系，并且可以填补空白以消除隐晦性。

接下来的几章将向你展示如何编写好的文档。他们还将讨论如何将文档编写集成到设计过程中，从而改善软件设计。

第十三章 注释应该描述那些不明显的事情在代码中

编写注释的原因是，使用编程语言编写的语句无法捕获编写代码时开发人员想到的所有重要信息。注释记录了这些信息，以便后来的开发人员可以轻松地理解和修改代码。注释的指导原则是，***注释应描述代码中不明显的内容。***

从代码中看不到很多事情。有时，底层细节并不明显。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之内还是之外并不明显。有时不清楚为什么需要代码，或者为什么要以特定方式实现代码。有时，开发人员遵循一些规则，例如“总是在 b 之前调用 a”。你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。注释可以使规则清晰明了。

注释的最重要原因之一是抽象，其中包括许多从代码中看不到的信息。抽象的思想是提供一种思考问题的简单方法，但是代码是如此详细，以至于仅通过阅读代码就很难看到抽象。注释可以提供一个更简单，更高级的视图（“调用此方法后，网络流量将被限制为每秒 maxBandwidth 字节”）。即使可以通过阅读代码推断出此信息，我们也不想强迫模块用户这样做：阅读代码很耗时，并且迫使他们考虑很多不需要使用的信息模块。***开发人员应该能够理解模块提供的抽象，而无需阅读其外部可见声明以外的任何代码。***

本章讨论需要在注释中描述哪些信息以及如何编写良好的注释。就像你将看到的那样，好的注释通常以与代码不同的详细程度来解释事物，在某些情况下，注释会更详细，而在某些情况下，代码则较不抽象（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，例如你要注释的内容和注释的格式。如果你正在使用存在文档编译工具的语言进行编程，例如 Java 的 Javadoc，C ++的 Doxygen 或 Go！的 GoDoc，请遵循工具的约定。这些约定都不是完美的，但是这些工具可提供足够的好处来弥补这一缺点。如果在没有现有约定可遵循的环境中进行编程，请尝试从其他类似的语言或项目中采用这些约定；这将使其他开发人员更容易理解和遵守你的约定。

约定有两个目的。首先，它们确保一致性，这使得注释更易于阅读和理解。其次，它们有助于确保你实际编写评论。如果你不清楚要发表的评论以及发表评论的方式，那么很容易最终根本不发表评论。

大多数注释属于以下类别之一：
接口：在模块声明（例如类，数据结构，函数或方法）之前的注释块。注释描述模块的接口。对于一个类，注释描述了该类提供的整体抽象。对于方法或函数，注释描述其整体行为，其参数和返回值（如果有），其生成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，它描述代码在内部的工作方式。
跨模块注释：描述跨模块边界的依赖项的注释。

最重要的注释是前两个类别中的注释。每个类都应有一个接口注释，每个类变量应有一个注释，每个方法都应有一个接口注释。有时，变量或方法的声明是如此明显，以至于在注释中没有添加任何有用的东西

（getter 和 setter 有时都属于此类），但这很少见。评论所有内容要比花精力担心是否需要评论要容易得多。实施注释通常是不必要的（请参阅下面的 13.6 节）。跨模块注释是最罕见的，而且编写起来很成问题，但是当需要它们时，它们就很重要。第 13.7 节将更详细地讨论它们。

13.2 不要重复代码

不幸的是，许多注释并不是特别有用。最常见的原因是注释重复了代码：可以轻松地从注释旁边的代码中推断出注释中的所有信息。这是最近研究论文中出现的代码示例：

```
ptr_copy = get_copy(obj) #Get pointer copy
if is_unlocked(ptr_copy): #Is obj free?
return obj #return current obj
if is_copy(ptr_copy): #Already a copy?
return obj #return obj
thread_id = get_thread_id(ptr_copy)
if thread_id == ctx.thread_id: #Locked by current ctx
return ptr_copy #Return copy
```

这些注释中没有任何有用的信息，但“ Locked by”注释除外，该注释暗示了有关线程的某些信息可能在代码中并不明显。请注意，这些注释的详细程度与代码大致相同：每行代码有一个注释，用于描述该行。这样的注释很少有用。

以下是重复代码的注释的更多示例：

```
//Add a horizontal scroll bar
hScrollBar = new JScrollBar(JScrollBar.HORIZONTAL);
add(hScrollBar, BorderLayout.SOUTH);
//Add a vertical scroll bar
vScrollBar = new JScrollBar(JScrollBar.VERTICAL);
add(vScrollBar, BorderLayout.EAST);
//Initialize the caret-position related values
caretX = 0;
caretY = 0;
caretMemX = null;
```

这些注释均未提供任何价值。对于前两个注释，代码已经很清楚了，它实际上不需要注释。在第三种情况下，注释可能有用，但是当前注释没有提供足够的细节来提供帮助。

编写注释后，请问自己以下问题：从未看过代码的人能否仅通过查看注释旁边的代码来编写注释？如果答案是肯定的（如上述示例所示），则注释不会使代码更易于理解。像这样的注释是为什么有些人认为毫无价值的原因。

另一个常见的错误是在注释中使用与要记录的实体名称相同的词：

```
/*
*  Obtain  a  normalized  resource  name  fromREQ.
*/
private  static  String[]  getNormalizedResourceNames(
    HTTPRequest  req) ...


/*
*  Downcast  PARAMETER  to  TYPE.
*/
private  static  Object  downCastParameter(String  parameter,  String  type) ...

/*
*  The  horizontal  padding  of  eachline  in  the  text.
*/
private  static  final int  textHorizontalPadding  =  4;
```

这些注释只是从方法或变量名中提取单词，或者从参数名称和类型中添加几个单词，然后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”！再说一次，这些注释可以仅通过查看声明来编写，而无需任何了解变量的方法。结果，它们没有价值。

【Red Flag：Comment Repeats Code】

如果注释旁边的代码中的注释信息已经很明显，则注释无济于事。这样的一个例子是，当注释使用与所描述事物名称相同的单词时。

同时，注释中缺少一些重要信息：例如，什么是“标准化资源名称”，以及 getNormalizedResourceNames 返回的数组的元素是什么？“贬低”是什么意思？填充的单位是什么，填充是在每行的一侧还是在两者的两侧？在注释中描述这些内容将很有帮助。

编写良好注释的第一步是在注释中使用与所描述实体名称不同的词。为注释选择单词，以提供有关实体含义的更多信息，而不仅仅是重复其名称。例如，以下是针对 textHorizontalPadding 的更好注释：

```
/*
* The amount of blank space to leave on the left and
* right sides of each line of text, in pixels.
*/
private static final int  =  4;
```

该注释提供了从声明本身不明显的其他信息，例如单位（像素）以及填充适用于每行两边的事实。如果读者不熟悉该术语，则注释将解释什么是填充，而不是使用术语“填充”。

13.3 低级的注释增加精确度

现在你知道了不应该做的事情，让我们讨论应该在注释中添加哪些信息。***注释通过提供不同详细程度的信息来增强代码。***一些注释提供了比代码更低，更详细的信息。这些注释通过阐明代码的确切含义来增加精度。其他注释提供了比代码更高，更抽象的信息。这些注释提供了直觉，例如代码背后的推理，或者更简单，更抽象的代码思考方式。与代码处于同一级别的注释可能会重复该代码。本节将更详细地讨论下层方法，而下一节将讨论上层方法。

在注释变量声明（例如类实例变量，方法参数和返回值）时，精度最有用。变量声明中的名称和类型通常不是很精确。注释可以填写缺少的详细信息，例如：

- 此变量的单位是什么？
- 边界条件是包容性还是排他性？
- 如果允许使用空值，则意味着什么？
- 如果变量引用了最终必须释放或关闭的资源，那么谁负责释放或关闭该资源？
- 是否存在某些对于变量始终不变的属性（不变量），例如“此列表始终包含至少一个条目”？

通过检查使用该变量的所有代码，可以潜在地了解某些信息。但是，这很耗时且容易出错。声明的注释应清晰，完整，以免不必要。当我说声明的注释应描述代码中不明显的内容时，“代码”是指注释（声明）旁边的代码，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太模糊。这是两个不够精确的注释示例：

```
// Current offset in resp Buffer
// Contains all line-widths inside the document and
// number of appearances.
private TreeMap<Integer, Integer>  lineWidths;;
```

在第一个示例中，尚不清楚“当前”的含义。在第二个示例中，尚不清楚 TreeMap 中的键是线宽，值是出现次数。另外，宽度是以像素或字符为单位测量的吗？以下修订后的注释提供了更多详细信息：

```
//  Position in this buffer of the first object that hasn't
//  been returned to the client.
uint32_t  offset;
//  Holds statistics about line lengths of the form <length, count>
//  where length is the number of characters in a line (including
//  the newline), and count is the number of lines with
//  exactly that many characters. If there are no lines with
//  a particular length, then there is no entry for that length.
private TreeMap<Integer, Integer> numLinesWithLength;
```

第二个声明使用一个较长的名称来传达更多信息。它还将“宽度”更改为“长度”，因为该术语更可能使人们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，还记录了缺少条目的含义。

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

```
/* FOLLOWER VARIABLE: indicator variable that allows the Receiver and the
* PeriodicTasks thread to communicate about whether a heartbeat has been
* received within the follower's election timeout window.
* Toggled to TRUE when a valid heartbeat is received.
* Toggled to FALSE when the election timeout window is reset.  */

private boolean receivedValidHeartbeat;
```

本文档描述了如何通过类中的几段代码来修改变量。如果注释描述变量代表什么而不是镜像代码结构，则注释将更短且更有用：

```
/* True means that a heartbeat has been received since the last time
* the election timer was reset. Used for communication between the
* Receiver and PeriodicTasks threads.  */
private boolean receivedValidHeartbeat;
```

根据本文档，很容易推断出，当接收到心跳信号时，变量必须设置为 true；而当重置选举计时器时，则必须将变量设置为 false。

13.4 高级的注释增强直觉

注释可以增加代码的第二种方法是提供直觉。这些注释是在比代码更高的级别上编写的。它们忽略了细节，并帮助读者理解了代码的整体意图和结构。此方法通常用于方法内部的注释以及接口注释。例如，考虑以下代码：

```
// If there is a LOADING readRpc using the same session
// as PKHash pointed to by assignPos, and the last PKHash
// in that readRPC is smaller than current assigning
// PKHash, then we put assigning PKHash into that readRPC.

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;
        break;
    }
}
```

该注释太底层和太详细。一方面，它部分重复了代码：“如果有 LOADING readRPC”仅重复测试 readRpc[i].status == LOADING。另一方面，注释不能解释此代码的总体目的，也不能解释其如何适合包含此代码的方法。如此一来注释不能帮助读者理解代码。

这是一个更好的注释：

```
// Try to append the current key hash onto an existing
// RPC to the desired server that hasn't been sent yet.
```

此注释不包含任何详细信息。相反，它在更高级别上描述了代码的整体功能。有了这些高级信息，读者就可以解释代码中几乎发生的所有事情：循环必须遍历所有现有的远程过程调用（RPC）；会话测试可能用于查看特定的 RPC 是否发往正确的服务器；LOADING 测试表明 RPC 可以具有多个状态，在某些状态下添加更多的哈希值是不安全的；MAX-PKHASHES_PERRPC 测试表明在单个 RPC 中可以发送多少个哈希值是有限制的。注释中唯一没有解释的是 maxPos 测试。此外，新注释为读者判断代码提供了基础：它可以完成将密钥哈希添加到现有 RPC 所需的一切吗？原始注释并未描述代码的整体意图，因此，读者很难确定代码是否行为正确。

高级别的注释比低级别的注释更难编写，因为你必须以不同的方式考虑代码。问问自己：这段代码要做什么？你能以何种最简单方式来解释代码中的所有内容？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，善于管理其中的许多细节；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退后一步，从更高层次考虑系统。这意味着要确定系统的哪些方面最重要，并且能够忽略底层细节，仅根据系统的最基本特征来考虑系统。这是抽象的本质（找到一种思考复杂实体的简单方法），这也是编写高级注释时必须执行的操作。一个好的高层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”。使用该框架，可以很容易地看到特定的代码语句与总体目标之间的关系。

这是另一个代码示例，具有较高层次的注释：

```
if (numProcessedPKHashes < readRpc[i].numHashes) {
    // Some of the key hashes couldn't be looked up in
    // this request (either because they aren't stored
    // on the server, the server crashed, or there
    // wasn't enough space in the response message).
    // Mark the unprocessed hashes so they will get
    // reassigned to new RPCs.
    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;
            }
```

```
        }
    }
}
```

此注释有两件事。第二句话提供了代码功能的抽象描述。第一句话是不同的：它以高级的方式解释了为什么执行代码。“如何到达这里”形式的注释对于帮助人们理解代码非常有用。例如，在记录方法时，描述最有可能在什么情况下调用该方法的条件（特别是仅在异常情况下调用该方法的情况）会非常有帮助。

13.5 接口文档

注释最重要的作用之一就是定义抽象。回想一下第四章，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；它的级别太低，它包含实现细节，这些细节在抽象中不应该看到。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，则必须用注释记录这些抽象。

记录抽象的第一步是将接口注释与实现注释分开。接口注释提供了使用类或方法时需要知道的信息。他们定义了抽象。实现注释描述了类或方法如何在内部工作以实现抽象。区分这两种注释很重要，这样接口的用户就不会暴露于实现细节。此外，这两种形式最好有所不同。如果接口注释也必须描述实现，则该类或方法很浅。这意味着撰写注释的行为可以提供有关设计质量的线索；第 15 章将回到这个想法。

类的接口注释提供了该类提供的抽象的高级描述，例如：

```
/**
* This class implements a simple server-side interface to the HTTP
* protocol: by using this class, an application can receive HTTP
* requests, process them, and return responses. Each instance of
* this class corresponds to a particular socket used to receive
* requests. The current implementation is single-threaded and
* processes one request at a time.
*/
public class Http {...}
```

该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。它还描述了该类的每个实例代表什么。最后，注释描述了该类的限制（它不支持从多个线程的并发访问），这对于考虑是否使用它的开发人员可能很重要。

方法的接口注释既包括用于抽象的高层信息，又包括用于精度的低层细节：

- 注释通常以一两个句子开头，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。副作用是该方法的任何结果都会影响系统的未来行为，但不属于结果的一部分。例如，如果该方法将一个值添加到内部数据结构中，可以通过将来的方法调用来检索该值，则这是副作用。写入文件系统也是一个副作用。
- 方法的接口注释必须描述该方法可能产生的任何异常。
- 如果在调用某个方法之前必须满足任何前提条件，则必须对其进行描述（也许必须先调用其他方法；对于二进制搜索方法，必须对要搜索的列表进行排序）。尽量减少前提条件是一个好主意，但是任何保留的条件都必须记录在案。

这是从 Buffer 对象复制数据的方法的接口注释：

```
/**
* Copy a range of bytes from a buffer to an external location.
*
* \param offset
*       Index within the buffer of the first byte to copy.
* \param length
*       Number of bytes to copy.
* \param dest
*       Where to copy the bytes: must have room for at least
*       length bytes.
*
* \return
*       The return value is the actual number of bytes copied,
*       which may be less than length if the requested range of
*       bytes extends past the end of the buffer. 0 is returned
*       if there is no overlap between the requested range and
*       the actual buffer.
*/
uint32_t
Buffer::copy(uint32_t offset, uint32_t length, void* dest)
...
```

此注释的语法 {例 \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为 Web 页。注释的目的是提供开发人员调用该方法所需的所有信息，包括特殊情况的处理方式 { 请注意，此方法如何遵循第 10 章的建议并定义与范围规范相关的任何错误。 } 开发人员不必为了调用它而阅读方法的主体，并且接口注释不提供有关如何实现该方法的信息，例如它如何扫描其内部数据结构以查找所需的数据。

对于更扩展的示例，让我们考虑一个称为 IndexLookup 的类，该类是分布式存储系统的一部分。存储系统拥有一个表集合，每个表包含许多对象。另外，每个表可以具有一个或多个索引；每个索引都基于对象的特定字段提供对表中对象的有效访问。例如，一个索引可以用于根据对象的名称字段查找对象，而另一个索引可以用于根据对象的年龄字段查找对象。使用这些索引，应用程序可以快速提取具有特定名称的所有对象，或具有给定范围内的年龄的所有对象。

IndexLookup 类为执行索引查找提供了一个方便的接口。这是一个如何在应用程序中使用的示例：

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

应用程序首先构造一个类型为 IndexLookup 的对象，并提供用于选择表，索引和索引内范围的参数（例如，如果索引基于年龄字段，则 key1 和 key2 可以指定为 21 和 65 选择年龄介于这些值之间的所有对象）。然后，应用程序重复调用 getNext 方法。每次调用都返回一个位于所需范围内的对象。一旦返回所有匹配的对象，getNext 将返回 NULL。因为存储系统是分布式的，所以此类的实现有些复杂。表中的对象可以分布在多个服务器上，每个索引也可以分布在一组不同的服务器上。

现在，让我们考虑该类的接口注释中需要包含哪些信息。对于下面给出的每条信息，问自己一个开发人员是否需要知道该信息才能使用该类（我对问题的回答在本章的结尾）：

1. IndexLookup 类发送给包含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在所需范围内的比较功能（使用整数，浮点数或字符串进行比较吗？）。
3. 用于在服务器上存储索引的数据结构。
4. IndexLookup 是否同时向多个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；摘录还包括类定义的几行内容，在注释中进行了引用：

```
/*
* This class implements the client side framework for index range
* lookups. It manages a single LookupIndexKeys RPC and multiple
* IndexedRead RPCs. Client side just includes "IndexLookup.h" in
* its header to use IndexLookup class. Several parameters can be set
* in the config below:
* - The number of concurrent indexedRead RPCs
* - The max number of PKHashes a indexedRead RPC can hold at a time
* - The size of the active PKHashes
*
* To use IndexLookup, the client creates an object of this class by
* providing all necessary information. After construction of
* IndexLookup, client can call getNext() function to move to next
* available object. If getNext() returns NULL, it means we reached
* the last object. Client can use getKey, getKeyLength, getValue,
* and getValueLength to get object data of current object.
*/
```

```
class IndexLookup {
    ...
  private:
    /// Max number of concurrent indexedRead RPCs
    static const uint8_t NUM_READ_RPC = 10;
    /// Max number of PKHashes that can be sent in one
    /// indexedRead RPC
    static const uint32_t MAX_PKHASHES_PERRPC = 256;
    /// Max number of PKHashes that activeHashes can
    /// hold at once.
    static const size_t MAX_NUM_PK = (1 << LG_BUFFER_SIZE);
}
```

在进一步阅读之前，请先查看你是否可以使用此注释确定问题所在。这是我发现的问题：

- 第一段的大部分与实现有关，而不是接口。举一个例子，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。在第一段的后半部分中提到的配置参数都是所有私有变量，它们仅与类的维护者相关，而与类的用户无关。所有这些实现信息都应从注释中省略。
- 该评论还包括一些显而易见的事情。例如，不需要告诉用户包括 IndexLookup.h：任何编写 C ++ 代码的人都可以猜测这是必要的。另外，“通过提供所有必要的信息”一词毫无意义，因此可以省略。

对此类的简短注释就足够了（并且更可取）：

```
/*
* This class is used by client applications to make range queries
* using indexes. Each instance represents a single range query.
*
* To start a range query, a client creates an instance of this
* class. The client can then call getNext() to retrieve the objects
* in the desired range. For each object returned by getNext(), the
* caller can invoke getKey(), getKeyLength(), getValue(), and
* getValueLength() to get information about that object.
*/
```

此注释的最后一段不是严格必需的，因为它主要针对单个方法复制了注释中的信息。但是，在类文档中提供示例来说明其方法如何协同工作可能会有所帮助，特别是对于使用模式不明显的深层类尤其如此。注意，新注释未提及 getNext 的 NULL 返回值。此注释无意记录每种方法的每个细节；它只是提供高级信息，以帮助读者了解这些方法如何协同工作以及何时可以调用每种方法。有关详细信息，读者可以参考接口注释中的各个方法。此注释也没有提到服务器崩溃；这是因为此类服务器的用户看不到服务器崩溃（系统会自动从中恢复）。

【Red Flag：Implementation Documentation Contaminates Interface】

当接口文档（例如方法的文档）描述了不需要使用要记录的事物的实现详细信息时，就会出现此红色标记。

现在考虑以下代码，该代码显示 IndexLookup 中 isReady 方法的文档的第一版：

```
/**
* Check if the next object is RESULT_READY. This function is
* implemented in a DCFT module, each execution of isReady() tries
* to make small progress, and getNext() invokes isReady() in a
* while loop, until isReady() returns true.
*
* isReady() is implemented in a rule-based approach. We check
* different rules by following a particular order, and perform
* certain actions if some rule is satisfied.
*
* \return
*       True means the next Object is available. Otherwise, return
*       false.
*/
bool IndexLookup::isReady() { ... }
```

再一次，本文档中的大多数内容，例如对 DCFT 的引用以及整个第二段，都与实现有关，因此不属于此处。这是接口注释中最常见的错误之一。某些实现文档很有用，但应放在方法内部，在该方法中应将其与接口文档明确分开。此外，文档的第一句话是含糊的（RESULT_READY 是什么意思？），并且缺少一些重要信息。最后，无需在此处描述 getNext 的实现。这是注释的更好版本：

```
/*
* Indicates whether an indexed read has made enough progress for
* getNext to return immediately without blocking. In addition, this
* method does most of the real work for indexed reads, so it must
* be invoked (either directly, or indirectly by calling getNext) in
* order for the indexed read to make progress.
*
* \return
*       True means that the next invocation of getNext will not block
*       (at least one object is available to return, or the end of the
*       lookup has been reached); false means getNext may block.
*/
```

此注释版本提供了有关“就绪”含义的更精确信息，并且提供了重要信息，如果要继续进行索引检索，则必须最终调用此方法。

13.6 注释实现：什么和为什么，而不是如何

实现注释是出现在方法内部的注释，以帮助读者了解它们在内部的工作方式。大多数方法是如此简短，简单，以至于它们不需要任何实现注释：有了代码和接口注释，就很容易弄清楚方法的工作原理。

实现注释的主要目的是帮助读者理解代码在做什么（而不是代码如何工作）。一旦读者知道了代码要做什么，通常就很容易理解代码的工作原理。对于简短的方法，代码只做一件事，该问题已在其接口注释中进行了描述，因此不需要实现注释。较长的方法具有多个代码块，这些代码块作为方法的整体任务的一部分执行不同的操作。在每个主要块之前添加注释，以提供对该块的作用的高级（更抽象）描述。这是一个例子：

// Phase 1: Scan active RPCs to see if any have completed.

对于循环，在循环前加一个注释来描述每次迭代中发生的事情是有帮助的：

// Each iteration of the following loop extracts one request from
// the request message, increments the corresponding object, and
// appends a response to the response message.

请注意，此注释如何更抽象和直观地描述循环。它没有详细介绍如何从请求消息中提取请求或对象如何递增。仅对于更长或更复杂的循环才需要循环注释，在这种情况下，循环的作用可能并不明显。许多循环足够短且简单，以至于其行为已经很明显。

除了描述代码在做什么之外，实现注释还有助于解释原因。如果代码中有些棘手的方面从阅读中看不出来，则应将它们记录下来。例如，如果一个错误修复程序需要添加目的不是很明显的代码，请添加注释以说明为什么需要该代码。对于错误修复，其中有写得很好的错误报告来描述问题，该注释可以引用错误跟踪数据库中的问题，而不是重复其所有详细信息（“修复 RAM-436，与 Linux 2.4 中的设备驱动程序崩溃有关。” X”）。开发人员可以在 bug 数据库中查找更多详细信息（这是一个避免注释重复的示例，这将在第 16 章中进行讨论）。

对于更长的方法，为一些最重要的局部变量写注释会很有帮助。但是，如果大多数局部变量具有好名字，则不需要文档。如果变量的所有用法在几行之内都是可见的，则通常无需注释即可轻松理解变量的用途。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，如果在大量代码中使用了该变量，则应考虑添加注释以描述该变量。在记录变量时，应关注变量表示的内容，而不是代码中如何对其进行操作。

13.7 跨模块设计决策

在理想环境中，每个重要的设计决策都将封装在一个类中。不幸的是，真实的系统不可避免地最终会影响到多个类的设计决策。例如，网络协议的设计将影响发送方和接收方，并且它们可以在不同的地方实现。跨模块决策通常是复杂而微妙的，并且会导致许多错误，因此，为它们提供良好的文档至关重要。

跨模块文档的最大挑战是找到一个放置它的位置，以便开发人员自然地发现它。有时，放置此类文档的中心位置很明显。例如，RAMCloud 存储系统定义一个状态值，每个请求均返回该状态值以指示成功或失败。为新的错误状况添加状态需要修改许多不同的文件（一个文件将状态值映射到异常，另一个文件为每个状态提供人类可读的消息，依此类推）。幸运的是，添加新的状态值（即 Status 枚举的声明）时，开发人员必须去一个明显的地方。我们通过在该枚举中添加注释来标识所有其他必须修改的地方，从而利用了这一点：在理想环境中，每个重要的设计决策都将封装在一个类中。不幸的是，真实的系统不可避免地最终会影响到多个类的设计决策。例如，网络协议的设计将影响发送方和接收方，并且它们可以在不同的地方实现。跨模块决策通常是复杂而微妙的，并且会导致许多错误，因此，为它们提供良好的文档至关重

要。

```
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,
// Note: if you add a new status value you must make the following
// additional updates:
// (1)  Modify STATUS_MAX_VALUE to have a value equal to the
//      largest defined status value, and make sure its definition
//      is the last one in the list. STATUS_MAX_VALUE is used
//      primarily for testing.
// (2)  Add new entries in the tables "messages" and "symbols" in
//      Status.cc.
// (3)  Add a new exception class to ClientException.h
// (4)  Add a new "case" to ClientException::throwException to map
//      from the status value to a status-specific ClientException
//      subclass.
// (5)  In the Java bindings, add a static class for the exception
//      to ClientException.java
// (6)  Add a case for the status of the exception to throw the
//      exception in ClientException.java
// (7)  Add the exception to the Status enum in Status.java, making
//      sure the status is in the correct position corresponding to
//      its status code.
}
```

新状态值将添加到现有列表的末尾，因此注释也将放置在最有可能出现的末尾。

不幸的是，在许多情况下，并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，僵尸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 zombie server 需要几个不同模块中的代码，这些代码都相互依赖。没有一段代码明显是放置文档的中心位置。一种可能性是在每个依赖文档的位置复制文档的部分。然而，这是令人尴尬的，并且随着系统的发展，很难使这样的文档保持最新。或者，文档可以位于需要它的位置之一，但是在这种情况下，开发人员不太可能看到文档或者知道在哪里查找它。

我最近一直在尝试一种方法，该方法将跨模块问题记录在一个名为 designNotes 的中央文件中。该文件分为清楚标记的部分，每个主要主题一个。例如，以下是该文件的摘录：

```
...
Zombies
```

A zombie is a server that is considered dead by the rest of the cluster; any data stored on the server has been recovered and will be managed by other servers. However, if a zombie is not actually dead (e.g., it was just disconnected from the other servers for a while) two forms of inconsistency can arise: * A zombie server must not serve read requests once replacement servers have taken over; otherwise it may return stale data that does not

reflect writes accepted by the replacement servers. * The zombie server must not accept write requests once replacement servers have begun replaying its log during recovery; if it does, these writes may be lost (the new values may not be stored on the replacement servers and thus will not be returned by reads).

RAMCloud uses two techniques to neutralize zombies. First,

...

然后，在与这些问题之一相关的任何代码段中，都有一条简短的注释引用了 designNotes 文件：

// See "Zombies" in designNotes.

使用这种方法，文档只有一个副本，因此开发人员在需要时可以相对容易地找到它。但是，这样做的缺点是，文档离它依赖的任何代码段都不近，因此随着系统的发展，可能难以保持最新。

13.8 结论

注释的目的是确保系统的结构和行为对读者来说是显而易见的，因此他们可以快速找到所需的信息，并有信心对其进行修改，以对系统进行修改。这些信息中的某些信息可以以对读者来说显而易见的方式表示在代码中，但是有大量信息无法从代码中轻易推导出。注释将填写此信息。

当遵循注释应描述代码中不明显的内容的规则时，“明显”是从第一次读取你的代码的人（不是你）的角度出发。在撰写注释时，请尝试使自己进入读者的心态，并问自己他或她需要知道哪些关键事项。如果你的代码正在接受审核，并且审核者告诉你某些不明显的内容，请不要与他们争论。如果读者认为它不明显，那么它就不明显。不用争论，而是尝试了解他们发现的令人困惑的地方，并查看是否可以通过更好的注释或更好的代码来澄清它们。

13.9 对 13.5 节的问题回答

开发人员是否需要了解以下每条信息才能使用 IndexLookup 类?

1. IndexLookup 类发送给包含索引和对象的服务器的消息格式。否：这是应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功能（使用整数，浮点数或字符串进行比较吗？）。是：该课程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：此信息应封装在服务器上；甚至 IndexLookup 的实现都不需要知道这一点。

4. IndexLookup 是否同时向多个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，则文档应提供有关此问题的一些高级信息，因为用户可能会在意性能。
5. 处理服务器崩溃的机制。否：RAMCloud 可从服务器崩溃中自动恢复，因此崩溃对于应用程序级软件不可见；因此，在 IndexLookup 的接口文档中无需提及崩溃。如果崩溃反映到应用程序中，则接口文档将需要描述它们如何表现出来（而不是崩溃恢复如何工作的详细信息）。

第十四章 命名

为变量，方法和其他实体选择名称是软件设计中被低估的方面之一。良好的名字是一种文档形式：它们使代码更易于理解。它们减少了对其他文档的需求，并使检测错误更加容易。相反，名称选择不当会增加代码的复杂性，并造成可能导致错误的歧义和误解。名称选择是复杂度是递增的原理的一个示例。为特定变量选择一个平庸的名称，而不是最好的名称，这可能不会对系统的整体复杂性产生太大影响。但是，软件系统具有数千个变量。为所有这些选择好名字将对复杂性和可管理性产生重大影响。

14.1 实例：坏的命名引起的漏洞

有时，即使是一个名称不正确的变量也会产生严重的后果。我曾经修复过的最具挑战性的错误是由于名称选择不当造成的。在 1980 年代末和 1990 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：即使用户未修改文件，数据块之一突然变为全零。该问题并不经常发生，因此很难追踪。一些研究生试图找到该错误，但他们未能取得进展，最终放弃了。但是，我认为任何未解决的错误都是无法忍受的个人侮辱，因此我决定对其进行跟踪。

花了六个月的时间，但我最终找到并修复了该错误。这个问题实际上很简单（就像大多数错误一样，一旦找出它们）。文件系统代码将变量名块用于两个不同的目的。在某些情况下，块是指磁盘上的物理块号。在其他情况下，块是指文件中的逻辑块号。不幸的是，在代码的某一点上有一个包含逻辑块号的块变量，但是在需要物理块号的情况下意外地使用了它。结果，磁盘上无关的块被零覆盖。

在跟踪该错误时，包括我自己在内的几个人阅读了错误的代码，但我们从未注意到问题所在。当我们看到可变块用作物理块号时，我们反身地假设它确实拥有物理块号。经过很长时间的检测，最终显示出腐败一定是在特定的语句中发生的，然后我才能越过该名称所创建的思维障碍，并查看其价值的确切来源。如果对不同种类的块（例如 fileBlock 和 diskBlock）使用了不同的变量名，则错误不太可能发生；程序员会知道在那种情况下不能使用 fileBlock。

不幸的是，大多数开发人员没有花太多时间在思考名字。他们倾向于使用想到的名字，只要它与匹配的名字相当接近即可。例如，块与磁盘上的物理块和文件内的逻辑块非常接近；这肯定不是一个可怕的名字。即使这样，它仍然要花费大量时间来查找一个细微的错误。因此，你不应该只选择“合理接近”的名称。花一些额外的时间来选择准确，明确且直观的好名字。额外的注意力将很快收回成本，随着时间的流逝，你将学会快速选择好名字。

14.2 创建图像

选择名称时，目标是在读者的脑海中创建一幅关于被命名事物的性质的图像。一个好名字传达了很多有关底层实体是什么，以及同样重要的是，不是什么的信息。在考虑特定名称时，请问自己：“如果有人孤立地看到该名称，而没有看到其声明，文档或使用该名称的任何代码，他们将能够猜到该名称指的是什么？还有其他名称可以使画面更清晰吗？” 当然，一个名字可以输入多少信息是有限制的。如果名称包含两个或三个以上的单词，则会变得笨拙。因此，面临的挑战是仅找到捕获实体最重要方面的几个单词。

名称是一种抽象形式：名称提供了一种简化的方式来考虑更复杂的基础实体。像其他形式的抽象一样，最好的名字是那些将注意力集中在对底层实体最重要的东西上，而忽略那些次要的细节。

14.3 名字应该要准确

良好名称具有两个属性：精度和一致性。让我们从精度开始。名称最常见的问题是名称太笼统或含糊不清。结果，读者很难说出这个名字指的是什么。读者可能会认为该名称所指的是与现实不符的事物，如上面的代码错误所示。考虑以下方法声明：

```
/**
* Returns the total number of indexlets this object is managing.
*/
int IndexletManager::getCount() {...}
```

术语“计数”太笼统了：计数什么？如果有人看到此方法的调用，除非他们阅读了它的文档，否则他们不太可能知道它的作用。像 getActiveIndexlets 或 numIndexlets 这样的更精确的名称会更好：使用这些名称之一，读者可能无需查看其文档就能猜测该方法返回的内容。

以下是来自其他学生项目的一些名称不够精确的示例：

- 建立 GUI 文本编辑器的项目使用名称 x 和 y 来引用字符在文件中的位置。这些名称太笼统了。他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）。单独看到名称 x 的人不太可能会认为它是指字符在一行文本中的位置。如果使用诸如 charIndex 和 lineIndex 之类的名称来反映代码实现的特定抽象，该代码将更加清晰。
- 另一个编辑器项目包含以下代码：
  ```
  // Blink state: true when cursor visible.
  private boolean blinkStatus = true;
  ```
  名称 blinkStatus 无法传达足够的信息。“状态”一词对于布尔值来说太含糊了：它不提供关于真值或假值含义的任何线索。“闪烁”一词也含糊不清，因为它并不表示闪烁的内容。以下替代方法更好：
  ```
  // Controls cursor blinking: true means the cursor is visible,
  // false means the cursor is not displayed.
  private boolean cursorVisible = true;
  ```
  名称 cursorVisible 传达了更多信息；例如，它允许读者猜测一个真值的含义（通常，布尔变量的名称应始终为谓词）。名称中不再包含“ blink”一词，因此，如果读者想知道为什么光标不总是可见，则必须查阅文档。此信息不太重要。
- 一个实施共识协议的项目包含以下代码：
  ```
  // Value representing that the server has not voted (yet) for
  // anyone for the current election term.
  private static final String VOTED_FOR_SENTINEL_VALUE = "null";
  ```
  此值的名称表示它是特殊的，但没有说明特殊含义是什么。使用更具体的名称（例如 NOT_YET_VOTED）会更好。
- 在没有返回值的方法中使用了名为 result 的变量。这个名字有多个问题。首先，它会产生误导性的印象，即它将作为方法的返回值。其次，除了它是一些计算值外，它实际上不提供有关其实际持有

内容的任何信息。该名称应提供有关实际结果是什么的信息，例如 mergedLine 或 totalChars。在实际上确实具有返回值的方法中，使用名称结果是合理的。该名称仍然有点通用，但是读者可以查看方法文档以了解其含义，这有助于知道该值最终将成为返回值。

【Red Flag：Vague Name】

如果变量或方法的名称足够广泛，可以引用许多不同的事物，那么它不会向开发人员传达太多信息，因此底层实体很可能会被滥用。

像所有规则一样，有关选择精确名称的规则也有一些例外。例如，只要循环仅跨越几行代码，就可以将通用名称（如 i 和 j）用作循环迭代变量。如果你可以看到一个变量的整个用法范围，那么该变量的含义在代码中就很明显了，因此你不需要长名称。例如，考虑以下代码：

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

从这段代码中很明显，i 正被用来迭代某个实体中的每一行。如果循环太长，以至于你无法一次看到全部内容，或者如果很难从代码中找出迭代变量的含义，那么应该使用更具描述性的名称。

名称也可能太具体，例如在此声明中删除一个文本范围的方法：

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

参数名称的选择过于具体，因为它建议始终在用户界面中选择要删除的文本。但是，可以在任意范围的文本（无论是否选中）上调用此方法。因此，参数名称应更通用，例如范围。

如果你发现很难为精确，直观且时间不长的特定变量命名，那么这是一个危险信号。这表明该变量可能没有明确的定义或目的。发生这种情况时，请考虑其他因素。例如，也许你正在尝试使用单个变量来表示几件事；如果是这样，将表示形式分成多个变量可能会导致每个变量的定义更简单。选择好名字的过程可以通过识别弱点来改善你的设计。

【Red Flag：Hard to Pick Name】

如果很难为创建基础对象清晰图像的变量或方法找到简单的名称，则表明基础对象可能没有简洁的设计。

14.4 名称使用的一致性

好的名称的第二个重要属性是一致性。在任何程序中，都会反复使用某些变量。例如，文件系统反复操作块号。对于每种常见用法，请选择一个用于该目的的名称，并在各处使用相同的名称。例如，文件系统可能总是使用 fileBlock 来保存文件中块的索引。一致的命名方式与重用普通类的方式一样，可以减轻认知负担：一旦读者在一个上下文中看到了该名称，他们就可以重用其知识并在不同上下文中看到该名称时立即做出假设。

一致性具有三个要求：首先，始终将通用名称用于给定目的；第二，除了给定目的外，切勿使用通用名称；第三，确保目的足够狭窄，以使所有具有名称的变量都具有相同的行为。在本章开头的文件系统错误中违反了此第三项要求。文件系统使用块来表示具有两种不同行为的变量（文件块和磁盘块）；这导致对变量含义的错误假设，进而导致错误。

有时你将需要多个变量来引用相同的一般事物。例如，一种复制文件数据的方法将需要两个块号，一个为源，一个为目标。发生这种情况时，请对每个变量使用通用名称，但要添加一个可区分的前缀，例如 srcFileBlock 和 dstFileBlock。

循环是一致性命名可以提供帮助的另一个领域。如果将诸如 i 和 j 之类的名称用于循环变量，则始终在最外层循环中使用 i，而在嵌套循环中始终使用 j。这使读者可以在看到给定名称时对代码中发生的事情做出即时（安全）假设。

14.5 一个不同的观点：Go 风格引导

并非所有人都同意我对命名的看法。一些使用 Go 语言的开发人员认为，名称应该非常简短，通常只能是一个字符。在关于 Go 的名称选择的演示中，Andrew Gerrand 指出“长名称模糊了代码的作用。”[1] 他介绍了此代码示例，该示例使用单字母变量名：

```
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更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++
    }
    return count
}
```

就个人而言，我不觉得第二版比第一版更难读。如果有的话，与 n 相比，名称计数为变量的行为提供了更好的线索。在第一个版本中，我最终通读了代码，试图弄清楚 n 的含义，而第二个版本中我并没有这

种需要。但是，如果在整个系统中一致地使用 n 来引用计数（而没有其他内容），那么其他开发人员可能会清楚知道该短名称。

Go 文化鼓励在多个不同的事物上使用相同的短名称：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乱和错误，就像在示例中一样。

总的来说，我认为可读性必须由读者而不是作家来决定。如果你使用简短的变量名编写代码，并且阅读该代码的人很容易理解，那么很好。如果你开始抱怨代码很含糊，那么你应该考虑使用更长的名称（在网络上搜索“ go language short name”（使用语言简称）会识别出几种此类抱怨）。同样，如果我开始抱怨长变量名使我的代码难以阅读，那么我会考虑使用较短的变量名。

Gerrand 发表一个我同意的评论：“名称声明与使用之间的距离越大，名称就应该越长。” 前面有关使用名为 i 和 j 的循环变量的讨论是此规则的示例。

14.6 结论

精心选择的名称有助于使代码更明显。当某人第一次遇到该变量时，他们对行为的第一次猜测是正确的。选择好名字是第 3 章讨论的投资思维方式的一个示例：如果你花一些额外的时间来选择好名字，那么将来你将更容易处理代码。此外，你不太可能引入错误。培养命名技巧也是一项投资。当你第一次决定停止为平庸的名字定居时，你会发现想出好名字的过程既令人沮丧又耗时。但是，随着你获得更多的经验，你会发现它变得更加容易。最终，你将几乎不需要花费额外的时间来选择好名字，因此你几乎可以免费获得好处。

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

第十五章 先写注释

在完成编码和单元测试之后，许多开发人员推迟编写文档，直到开发过程结束。这是产生质量差的文档的最可靠方法之一。编写注释的最佳时间是在过程开始时。首先编写注释使文档成为设计过程的一部分。这不仅可以产生更好的文档，还可以产生更好的设计，并使编写文档的过程更加愉快。

15.1 延迟的注释不是好注释

我见过的几乎每个开发人员都会推迟编写注释。当被问及为什么不更早编写文档时，他们说代码仍在更改。他们说，如果他们尽早编写文档，则必须在代码更改时重新编写文档。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，他们尽可能地推迟了。

不幸的是，这种方法有几个负面影响。首先，延迟文档通常意味着根本无法编写文档。一旦开始延迟，就容易再延迟一些。毕竟，代码将在几周后变得更加稳定。到了代码毫无疑问地稳定下来的时候，代码已经很多了，这意味着编写文档的任务变得越来越庞大，吸引力也越来越小。从来没有一个方便的时间可以停下来几天并填写所有遗漏的注释，并且很容易使该项目的最佳选择合理化，那就是继续并修复错误或编写下一个新功能。这将创建更多未记录的代码。

即使你有自律性回去写注释(不要欺骗你自己:你可能没有)，注释也不会很好。在这个过程的这个时候，你已经在精神上离开了。在你的脑海中，这段代码已经完成了;你急于开始下一个项目。你知道写注释是正确的事情，但它没有乐趣。你只想尽快度过难关。因此，你可以快速地浏览代码，添加足够的注释以使其看起来令人满意。到目前为止，你已经有一段时间没有设计代码了，所以你对设计过程的记忆变得模糊了。你在编写注释时查看代码，因此注释重复了代码。即使你试图重构代码中不明显的设计思想，也会有你不记得的事情。因此，这些注释忽略了他们应该描述的一些最重要的事情。

15.2 先写注释

我使用一种不同的方法来编写注释，在开始时就写注释：

- 对于新类，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，但将方法主体保留为空。
- 我对这些注释进行了迭代，直到基本结构感觉正确为止。
- 在这一点上，我为类中最重要的类实例变量编写了声明和注释。
- 最后，我填写方法的主体，并根据需要添加实现注释。
- 在编写方法主体时，我通常会发现需要其他方法和实例变量。对于每个新方法，我在方法主体之前编写接口注释。例如变量，我在编写变量声明的同时填写了注释。

代码完成后，注释也将完成。从来没有积压的书面注释。

注释优先的方法具有三个好处。首先，它会产生更好的注释。如果你在设计课程时写注释，那么关键的设计问题将在你的脑海中浮现，因此很容易记录下来。最好在每个方法的主体之前编写接口注释，这样你就

可以专注于方法的抽象和接口，而不会因其实现而分心。在编码和测试过程中，你会注意到并修复注释问题。结果，注释在开发过程中得到了改善。

15.3 注释是一种设计工具

在开始时编写注释的第二个也是最重要的好处是可以改善系统设计。注释提供了完全捕获抽象的唯一方法，好的抽象是好的系统设计的基础。如果你在一开始就写了描述抽象的注释，则可以在编写实现代码之前对其进行检查和调整。要写一个好的注释，你必须确定一个变量或一段代码的本质：这件事最重要的方面是什么？在设计过程的早期进行此操作很重要；否则，你只是在破解代码。

注释是复杂煤矿中的金丝雀。如果方法或变量需要较长的注释，则它是一个危险信号，表明你没有很好的抽象。请记住，在第 4 章中，类应该很深：最好的类具有非常简单的接口，但可以实现强大的功能。判断接口复杂性的最佳方法是从描述接口的注释中进行。如果某个方法的接口注释提供了使用该方法所需的所有信息，并且又简短又简单，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无法完全描述一个方法，则该方法具有复杂的接口。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功能，则该方法很浅。同样的想法也适用于变量：如果要花很长的时间来完整描述一个变量，那是一个危险信号，表明你可能没有选择正确的变量分解。总体而言，编写注释的行为使你可以及早评估设计决策，以便发现并解决问题。

【Red Flag：Hard to Describe】

描述方法或变量的注释应该简单而完整。如果你发现很难写这样的注释，则表明你所描述的内容的设计可能存在问题。

当然，如果注释完整而清晰，那么它们仅是复杂性的良好指标。如果编写的方法接口注释未提供调用该方法所需的全部信息，或者编写的注释过于神秘以至于难以理解，则该注释不能很好地衡量该方法的深度。

15.4 早期的注释是有趣的注释

尽早编写注释的第三个也是最后一个好处是，它使编写注释更加有趣。对我来说，编程中最有趣的部分之一是新类的早期设计阶段，在那里，我将充实该类的抽象和结构。我的大部分注释都是在此阶段编写的，这些注释是我记录和测试设计决策质量的方式。我正在寻找可以用最少的词来完整而清晰地表达的设计。注释越简单，我对设计的感觉就越好，因此找到简单的注释是一种自豪感。如果你是策略性编程，而你的主要目标是一个出色的设计，而不仅仅是编写有效的代码，那么编写注释应该很有趣，因为这是你确定最佳设计的方式。

15.5 早期的注释贵吗?

现在，让我们重新讨论延迟注释的参数，这是因为它避免了在代码演变时重新处理注释的开销。一个简单的信封计算将显示这并不能节省很多。首先，估算你一起键入代码和注释所花费的开发时间的总和，包括修改代码和注释的时间；这不太可能超过所有开发时间的 10%。即使你的全部代码行中有一半是注释，

编写注释也可能不会占开发总时间的 5%以上。将注释延迟到最后只会节省其中的一小部分，这不是很多。

首先编写注释将意味着在开始编写代码之前，抽象将更加稳定。这可能会节省编码时间。相反，如果你首先编写代码，则抽象可能会随代码的发展而变化，与注释优先方法相比，将需要更多的代码修订。当你考虑所有这些因素时，可能首先整体编写注释可能会更快。

15.6 结论

如果你从未尝试过先编写注释，请尝试一下。坚持足够长的时间来习惯它。然后考虑它如何影响你的注释质量，设计质量以及软件开发的整体乐趣。在尝试了一段时间之后，让我知道你的经历是否与我的相符，以及为什么或为什么不这样。

第十六章 修改已经存在的（老的）代码

第 1 章介绍了软件开发是如何迭代和增量的。大型软件系统是通过一系列演化阶段开发的，其中每个阶段都添加了新功能并修改了现有模块。这意味着系统的设计在不断发展。一开始就不可能为系统设计正确的设计。一个成熟的系统的设计更多地取决于系统演化过程中所做的更改，而不是任何初始概念。前面的章节描述了如何在初始设计和实现过程中降低复杂性。本章讨论如何防止随着系统的发展而增加复杂性。

16.1 保持战略编程

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，主要目标是使某些事物快速工作，即使这会导致额外的复杂性；在战略编程中，最重要的目标是进行出色的系统设计。战术方法很快导致系统设计混乱。如果你想要一个易于维护和增强的系统，那么“工作”还不够高。你必须优先考虑设计并进行战略思考。当你修改现有代码时，此想法也适用。

不幸的是，当开发人员进入现有代码以进行更改（例如错误修复或新功能）时，他们通常不会从战略角度进行思考。一个典型的心态是“我能做出我需要做的最小的改变是什么？” 有时，开发人员证明这是合理的，因为他们对修改的代码不满意。他们担心较大的更改会带来更大的引入新错误的风险。但是，这导致了战术编程。这些最小的变化中的每一个都会引入一些特殊情况，依赖性或其他形式的复杂性。结果，系统设计变得更糟，并且问题随着系统发展的每个步骤而累积。

如果要维护系统的简洁设计，则在修改现有代码时必须采取战略性方法。理想情况下，当你完成每次更改时，如果你从一开始就考虑到更改就设计了系统，那么系统将具有它应该具有的结构。为了实现此目标，你必须抵制诱惑以快速解决问题。相反，请根据所需的更改来考虑当前的系统设计是否仍然是最佳的。如果不是，请重构系统，以便最终获得最佳设计。通过这种方法，每次修改都会改善系统设计。

这也是第 15 页介绍的投资心态的一个示例：如果你花费一些额外的时间来重构和改善系统设计，你将得到一个更干净的系统。这将加快开发速度，你将收回在重构方面投入的精力。即使你的特定更改不需要重构，你仍然应该注意在代码中可以修复的设计缺陷。每当你修改任何代码时，都尝试在该过程中至少找到一点方法来改进系统设计。如果你没有使设计更好，则可能会使它变得更糟。

如第 3 章所述，投资心态有时与商业软件开发的现实相冲突。如果“正确的方式”重构系统需要三个月，而快速且肮脏的修复仅需两个小时，则你可能必须采取快速而肮脏的方法，尤其是在紧迫的期限内工作时。或者，如果重构系统会造成影响许多其他人员和团队的不兼容性，则重构可能不切实际。

但是，你应尽可能抵制这些妥协。问问自己：“考虑到我目前的限制，这是否是我能做的最好的工作来创建一个干净的系统设计？” 也许有一种替代方法几乎可以像 3 个月的重构一样干净，但是可以在几天内完成？或者，如果你现在负担不起大型重构，请让你的老板为你分配时间，让你在当前截止日期之后恢复到原来的水平。每个开发组织都应计划将其全部工作的一小部分用于清理和重构；从长远来看，这项工作将收回成本。

16.2 注释维护：将注释保留在代码附近

当你更改现有代码时，更改很有可能会使某些现有注释无效。修改代码时，很容易忘记更新注释，从而导致注释不再准确。不准确的注释使读者感到沮丧，如果注释太多，读者就会开始不信任所有注释。幸运的是，只要有一点纪律和一些指导规则，就可以在不付出巨大努力的情况下使注释保持最新。本节及随后的部分提出了一些特定的技术。

确保注释更新的最佳方法是将注释放置在它们描述的代码附近，以便开发人员在更改代码时可以看到它们。注释离其关联的代码越远，正确更新的可能性就越小。例如，方法接口注释的最佳位置是在代码文件中，紧靠该方法主体的位置。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，并在需要时进行更新。

对于 C 和 C ++等具有单独的代码和头文件的语言，一种替代方法是将接口注释放在.h 文件中方法声明的旁边。但是，这距离代码还有很长的路要走。开发人员在修改方法的主体时将看不到这些注释，因此需要打开其他文件并查找接口注释来更新它们，这需要额外的工作。有人可能会争辩说接口注释应该放在头文件中，以便用户可以不必看代码文件就可以学习如何使用抽象。但是，用户无需读取代码或头文件；他们应该从由 Doxygen 或 Javadoc 等工具编译的文档中获取信息。此外，许多 IDE 都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。给定诸如此类的工具，文档应位于对开发人员进行代码开发最方便的位置。

在编写实现注释时，不要将整个方法的所有注释放在方法的顶部。展开它们，将每个注释推到最狭窄的范围，其中包括该注释所引用的所有代码。例如，如果一种方法具有三个主要阶段，则不要在方法的顶部写一个详细描述所有阶段的注释。而是为每个阶段编写一个单独的注释，并将该注释放置在该阶段的第一行代码的正上方。另一方面，在描述总体策略的方法实现的顶部添加注释也可能会有所帮助，例如：

```
// We proceed in three phases:
// Phase 1: Find feasible candidates
// Phase 2: Assign each candidate a score
// Phase 3: Choose the best, and remove it
```

每个阶段的代码上方都可以记录其他详细信息。
通常，注释离描述的代码越远，注释应该越抽象（这减少了注释因代码更改而无效的可能性）。

16.3 注释属于代码，而不是提交日志

修改代码时，常见的错误是将有关更改的详细信息放入源代码存储库的提交消息中，而不是将其记录在代码中。尽管将来可以通过扫描存储库的日志来浏览提交消息，但是需要该信息的开发人员不太可能考虑扫描存储库的日志。即使他们确实扫描了日志，也很难找到正确的日志消息。

在编写提交消息时，请问自己将来开发人员是否需要使用该信息。如果是这样，则在代码中记录此信息。一个示例是提交消息，描述了导致代码更改的细微问题。如果代码中未对此进行记录，则开发人员可能会稍后再提出并撤消更改，而不会意识到他们已经重新创建了错误。如果你也想在提交消息中包含此信息的副本，那很好，但是最重要的是在代码中获取它。这说明了将文档放置在开发人员最有可能看到它的地方的原理；提交日志很少在那个地方。

16.4 注释维护：避免重复

保持注释最新的第二种技术是避免重复。如果文档重复，那么开发人员将很难找到并更新所有相关副本。相反，请尝试仅一次记录每个设计决策。如果代码中有多个地方受某个特定决定的影响，请不要在所有这些地方重复文档。相反，找到放置文档最明显的位置。例如，假设存在与变量相关的棘手行为，这会影响使用变量的几个不同位置。你可以在变量声明旁边的注释中记录该行为。这是很自然的地方，开发人员可能会检查他们是否在理解使用该变量的代码时遇到麻烦。

如果没有一个“明显的”地方来放置特定的文档，开发人员可以找到它，那么创建一个 designNotes 文件，如第 13.7 节所述。或者，选择最好的地方，把文档放在那里。另外，在引用中心位置的其他地方添加简短的注释:“查看 xyz 中的注释以了解下面代码的解释。 “如果引用因为主注释被移动或删除而变得过时，这种不一致性将是不言而喻的，因为开发人员将无法在指定的位置找到注释;他们可以使用修订控制历史记录来查找注释发生了什么，然后更新引用。相反，如果文档是重复的，并且一些副本没有得到更新，那么开发人员就不会知道他们使用的是陈旧的信息。

不要在另一个模块中记录一个模块的设计决策。例如，不要在方法调用前添加注释，以解释被调用方法中发生的情况。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此信息，例如，如果你选择了方法的名称或将鼠标悬停在该方法的名称上，则将显示该方法的接口注释。尝试使开发人员容易找到合适的文档，但是不要重复文档。

如果信息已经在程序之外的某个地方记录了，不要在程序内部重复记录;只需参考外部文档。例如，如果你编写一个实现 HTTP 协议的类，那么就不需要在代码中描述 HTTP 协议。在网上已经有很多关于这个文档的来源;只需在你的代码中添加一个简短的注释，并为其中一个源添加一个 URL。另一个例子是已经在用户手册中记录的特性。假设你正在编写一个实现命令集合的程序，其中有一个负责实现每个命令的方法。如果有描述这些命令的用户手册，就不需要在代码中重复这些信息。相反，在每个命令方法的接口注释中包含如下简短说明:

```
// Implements the Foo command; see the user manual for details.
```

读者可以轻松找到理解代码所需的所有文档，这一点很重要，但这并不意味着你必须编写所有这些文档。

16.5 注释维护：检查差异

确保文档保持最新状态的一种好方法是，在将更改提交到修订控制系统之前需要花费几分钟，以扫描该提交的所有更改。确保文档中正确反映了每个更改。这些预先提交的扫描还将检测其他一些问题，例如意外地将调试代码留在系统中或无法修复 TODO 项目。

16.6 更高级别注释更易于维护

关于维护文档的最后一个想法：如果注释比代码更高级，更抽象，则注释更易于维护。这些注释不反映代码的详细信息，因此它们不会受到代码更改的影响；只有整体行为的变化才会影响这些评论。当然，正如第 13 章所讨论的那样，某些注释的确需要详细和精确。但总的来说，最有用的注释（它们不只是重复代码）也最容易维护。

第十七章 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。如果系统是一致的，则意味着相似的事情以相似的方式完成，而不同的事情则以不同的方式完成。一致性会产生认知影响力：一旦你了解了某个地方的工作方式，就可以使用该知识立即了解其他使用相同方法的地方。如果系统的实施方式不一致，则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。如果系统不一致，则实际上两种情况可能不同，但两种情况可能看起来相同。开发人员可能会看到一个看起来很熟悉的模式，并根据以前对该模式的遭遇做出错误的假设。另一方面，如果系统是一致的，则基于熟悉情况的假设将是安全的。一致性允许开发人员以更少的错误来更快地工作。

17.1 一致性示例

一致性可以应用于系统中的许多级别。这里有一些例子。

名字。第 14 章已经讨论了以一致的方式使用名称的好处。
编码样式。如今，开发组织通常拥有样式指南，这些样式指南将程序结构限制在编译器所强制执行的规则之外。现代风格指南解决了一系列问题，例如缩进，大括号放置，声明顺序，命名，注释以及对认为危险的语言功能的限制。样式指南使代码更易于阅读，并且可以减少某些类型的错误。
接口。具有多个实现的接口是一致性的另一个示例。一旦了解了接口的一种实现，其他任何实现都将变得更易于理解，因为你已经知道它将必须提供的功能。
设计模式。设计模式是某些常见问题的普遍接受的解决方案，例如用于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，则实现会更快地进行，更有可能起作用，并且你的代码对读者来说也会更明显。设计模式将在 19.5 节中详细讨论。
不变量。不变式是始终为真的变量或结构的属性。例如，存储文本行的数据结构可能会强制要求每行以换行符终止。不变式减少了代码中必须考虑的特殊情况的数量，并使推理行为的方式变得更加容易。

17.2 确保一致性

一致性很难维护，尤其是当许多人长时间从事一个项目时。一组人可能不了解另一组中建立的约定。新来者不了解规则，因此他们无意间违反了约定并创建了与现有约定冲突的新约定。以下是建立和保持一致性的一些技巧：

文档。创建一个列出最重要的总体约定的文档，例如编码样式准则。将文档放置在开发人员可能会看到的位置，例如项目 Wiki 上的显眼位置。鼓励新成员加入小组阅读文档，并鼓励现有人员不时审阅该文档。Web 上已经发布了来自各个组织的一些样式指南；考虑从其中之一开始。
对于局部性更强的约定，例如不变式，请在代码中找到合适的位置进行记录。如果你不写下约定，那么其他人不太可能会遵循它们。
执行。即使有好的文档，开发人员也很难记住所有约定。实施约定的最佳方法是编写一个检查违规的工具，并确保除非通过检查程序，否则代码无法提交到存储库。自动检查器对于底层语法约定特别有用。

我最近的一个项目有行终止字符的问题。一些开发人员在 Unix 上工作，行被换行终止;其他的工作在 Windows 上，行通常由一个 carriage-return 后跟一个换行符来结束。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小的编辑，那么编辑器有时会将所有行终止符替换为适合该系统的行终止符。这给人的感觉是文件的每一行都被修改了，这使得跟踪有意义的更改变得很困难。我们建立了一个约定，即文件应该只包含换行，但是很难确保每个开发人员使用的每个工具都遵循这个约定。每当一个新的开发人员加入这个项目，我们就会经历一连串的线路终止问题，而那个开发人员就会适应这个约定。

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，如果其中任何一个包含回车符，则中止提交。还可以通过用换行符替换回车/换行符序列来手动运行脚本以修复损坏的文件。这立即消除了问题，并且还帮助培训了新开发人员。

代码审查为实施约定和向新开发者提供有关约定的教育提供了另一个机会。代码审阅者越挑剔，团队中的每个人都将更快地学习约定，并且代码越干净。

在罗马时……最重要的约定是每个开发人员都应遵循古老的格言“在罗马时，就像罗马人一样。” 在处理新文件时，请环顾四周以了解现有代码的结构。是否在私有变量和方法之前声明了所有公共变量和方法？方法是否按字母顺序排列？变量是否使用 firstServerName 中的“ camel case”或使用 first_server_name 中的“ snake case”？当你看到任何看起来可能是约定的内容时，请遵循该约定。在做出设计决策时，请问自己是否有可能在项目的其他地方做出了类似的决策；如果是这样，请找到一个现有示例，并在新代码中使用相同的方法。

不要更改现有约定。抵制“改善”现有公约的冲动。拥有一个“更好的主意”不足以引起矛盾。***你的新想法可能确实更好，但是一致性胜于不一致的价值几乎总是大于一种方法胜过另一种方法的价值。***在引入不一致的行为之前，请问自己两个问题。首先，你是否拥有大量的新信息来证明你的方法在建立旧约定时是不可用的？其次，新方法是否好得多，值得花时间更新所有旧用法？如果你的组织同意对两个问题的回答均为“是”，则继续进行升级；否则，请进行升级。完成后，应该没有旧约定的迹象。然而，你仍然冒着其他开发人员不了解新约定的风险，因此他们将来可能会重新引入旧方法。总体而言，重新考虑已建立的约定很少会很好地利用开发人员时间。

17.3 走得太远

一致性不仅意味着相似的事情应该以相似的方式完成，而且不同的事情也应该以不同的方式完成。如果你对一致性过于热衷，并试图将不同的事物强制采用相同的方法，例如对确实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，那么会造成复杂性和混乱。一致性只有在开发人员确信“如果看起来像 x 时，它确实是 x”时才有好处。

17.4 结论

一致性是投资心态的另一个例子。确保一致性的工作将需要一些额外的工作：确定约定，创建自动检查程序，寻找类似情况以模仿新代码，以及进行代码审查以教育团队。这项投资的回报是你的代码将更加明显。开发人员将能够更快，更准确地了解代码的行为，这将使他们能够以更少的错误来更快地工作。

第十八章 代码应该是易于理解的

晦涩难懂是第 2.3 节中描述的导致复杂性的两个主要原因之一。当有关系统的重要信息对于新开发人员而言并不明显时，就会发生模糊。解决晦涩问题的方法是以显而易见的方式编写代码。本章讨论使代码或多或少变得显而易见的一些因素。

如果代码很明显，则意味着某人可以不加思索地快速阅读该代码，并且他们对代码的行为或含义的最初猜测将是正确的。如果代码很明显，那么读者就不需要花费很多时间或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。这不仅降低了它们的效率，而且还增加了误解和错误的可能性。明显的代码比不明显的代码需要更少的注释。

读者的想法是“显而易见”：注意到别人的代码不明显比发现自己的代码有问题要容易得多。因此，确定代码是否显而易见的最佳方法是通过代码审查。如果有人在阅读你的代码时说它并不明显，那么无论你看起来多么清晰，它也不是显而易见。通过尝试理解什么使代码变得不明显，你将学习如何在将来编写更好的代码。

18.1 使代码变得显而易见

在前面的章节中已经讨论了使代码显而易见的两种最重要的技术。首先是选择好名字（第 14 章）。精确而有意义的名称可以阐明代码的行为，并减少对文档的需求。如果名称含糊不清或含糊不清，那么读者将通读代码以推论命名实体的含义；这既费时又容易出错。第二种技术是一致性（第 17 章）。如果总是以相似的方式完成相似的事情，那么读者可以识别出他们以前所见过的模式，并立即得出（安全）结论，而无需详细分析代码。

以下是使代码更明显的其他一些通用技术：

明智地使用空白。代码格式化的方式会影响其理解的容易程度。考虑以下参数文档，其中空格已被压缩：

```
/**
* ...
* @param numThreads The number of threads that this manager should
* spin up in order to manage ongoing connections. The MessageManager
* spins up at least one thread for every open connection, so this
* should be at least equal to the number of connections you expect
* to be open at once. This should be a multiple of that number if
* you expect to send a lot of messages in a short amount of time.
* @param handler Used as a callback in order to handle incoming
* messages on this MessageManager's open connections. See
* {@code MessageHandler} and {@code handleMessage} for details.
*/
```

很难看到一个参数的文档在哪里结束而下一个参数的文档在哪里开始。甚至不知道有多少个参数或它们的名称是什么。如果添加了一些空白，结构会突然变得清晰，文档也更容易扫描：

```
/**
*  @param numThreads
*          The number of threads that this manager should spin up in
*          order to manage ongoing connections. The MessageManager spins
*          up at least one thread for every open connection, so this
*          should be at least equal to the number of connections you
*          expect to be open at once. This should be a multiple of that
*          number if you expect to send a lot of messages in a short
*          amount of time.
*  @param handler
*          Used as a callback in order to handle incoming messages on
*          this MessageManager's open connections. See
*          {@code MessageHandler} and {@code handleMessage} for details.
*/
```

空行也可用于分隔方法中的主要代码块，例如以下示例：

```
void* Buffer::allocAux(size_t numBytes) {
    //  Round up the length to a multiple of 8 bytes, to ensure alignment.
    uint32_t numBytes32 =  (downCast<uint32_t>(numBytes) + 7) & ~0x7;
    assert(numBytes32 != 0);
    //  If there is enough memory at firstAvailable, use that. Work down
    //  from the top, because this memory is guaranteed to be aligned
    //  (memory at the bottom may have been used for variable-size chunks).
    if  (availableLength >= numBytes32) {
        availableLength -= numBytes32;
        return firstAvailable + availableLength;
    }
    //  Next, see if there is extra space at the end of the last chunk.
    if  (extraAppendBytes >= numBytes32) {
        extraAppendBytes -= numBytes32;
        return lastChunk->data + lastChunk->length + extraAppendBytes;
    }
    //  Must create a new space allocation; allocate space within it.
    uint32_t allocatedLength;
    firstAvailable = getNewAllocation(numBytes32, &allocatedLength);
    availableLength = allocatedLength numBytes32;
    return firstAvailable + availableLength;
}
```

如果每个空白行之后的第一行是描述下一个代码块的注释，则此方法特别有效：空白行使注释更可见。

语句中的空白有助于阐明语句的结构。比较以下两个语句，其中之一具有空格，而其中一个没有空格：

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


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

注释。有时无法避免非显而易见的代码。发生这种情况时，重要的是使用注释来提供缺少的信息以进行补偿。要做到这一点，你必须使自己处于读者的位置，弄清楚什么可能会使他们感到困惑，以及哪些信息可以消除这种混乱。下一部分显示了一些示例。

18.2 使代码变得不那么明显

有很多事情可以使代码变得不明显。本节提供了一些示例。其中某些功能（例如事件驱动的编程）在某些情况下很有用，因此你可能最终还是要使用它们。发生这种情况时，额外的文档可以帮助最大程度地减少读者的困惑。

事件驱动的编程。在事件驱动的编程中，应用程序对外部事件做出响应，例如网络数据包的到来或按下鼠标按钮。一个模块负责报告传入事件。应用程序的其他部分通过在事件发生时要求事件模块调用给定的函数或方法来注册对某些事件的兴趣。

事件驱动的编程使其很难遵循控制流程。永远不要直接调用事件处理函数。它们是由事件模块间接调用的，通常使用函数指针或接口。即使你在事件模块中找到了调用点，也仍然无法确定将调用哪个特定功能：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相信它是可行的。
为了弥补这种隐晦性，请为每个处理程序函数使用接口注释，以指示何时调用该函数，如以下示例所示：

```
/**
* This method is invoked in the dispatch thread by a transport if a
* transport-level error prevents an RPC from completing.
*/
void Transport::RpcNotifier::failed() {
    ...
}
```

img Red Flag: Nonobvious Code img

【Red Flag：Nonobvious Code】

如果无法通过快速阅读来理解代码的含义和行为，则它是一个危险标记。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能立即清除。

通用容器。许多语言提供了用于将两个或多个项目组合到一个对象中的通用类，例如 Java 中的 Pair 或 C ++中的 std :: pair。这些类很诱人，因为它们使使用单个变量轻松传递多个对象变得容易。最常见的用途之一是从一个方法返回多个值，如以下 Java 示例所示：

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

不幸的是，通用容器导致代码不清晰，因为分组后的元素的通用名称模糊了它们的含义。在上面的示例中，调用者必须使用 result.getKey（）和 result.getValue（）引用两个返回的值，而这两个值都不提供这些值的实际含义。

因此，最好不要使用通用容器。如果需要容器，请定义专门用于特定用途的新类或结构。然后，你可以为元素使用有意义的名称，并且可以在声明中提供其他文档，而对于常规容器而言，这是不可能的。

此示例说明了一条通用规则：***软件应设计为易于阅读而不是易于编写。***通用容器对于编写代码的人来说是很方便的，但是它们会使随后的所有读者感到困惑。对于编写代码的人来说，花一些额外的时间来定义特定的容器结构是更好的选择，以便使生成的代码更加明显。

不同类型的声明和分配。考虑以下 Java 示例：

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

将该变量声明为 List，但实际值为 ArrayList。这段代码是合法的，因为 List 是 ArrayList 的超类，但是它会误导看到声明但不是实际分配的读者。实际类型可能会影响变量的使用方式（ArrayList 与 List 的其他子类相比，具有不同的性能和线程安全属性），因此最好将声明与分配匹配。

违反读者期望的代码。考虑以下代码，这是 Java 应用程序的主程序

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

大多数应用程序在其主程序返回时退出，因此读者可能会认为这将在此处发生。但是，事实并非如此。RaftClient 的构造函数创建其他线程，即使应用程序的主线程完成，该线程仍可继续运行。应该在 RaftClient 构造函数的接口注释中记录此行为，但是该行为不够明显，因此值得在 main 末尾添加简短注释。该注释应指示该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惯例，那么它是最明显的。如果没有，那么记录该行为很重要，以免使读者感到困惑。

18.3 结论

关于显而易见性的另一种思考方式是信息。如果代码不是显而易见的，则通常意味着存在有关读者所不具备的代码的重要信息：在 RaftClient 示例中，读者可能不知道 RaftClient 构造函数创建了新线程；在“配对”示例中，读者可能不知道 result.getKey（）返回当前项的编号。

为了使代码清晰可见，你必须确保读者始终拥有理解它们所需的信息。你可以通过三种方式执行此操作。最好的方法是使用抽象等设计技术并消除特殊情况，以减少所需的信息量。其次，你可以利用读者在其他情况下已经获得的信息（例如，通过遵循约定并符合期望），从而使读者不必为代码学习新的信息。第三，你可以使用诸如好名和战略注释之类的技术在代码中向他们提供重要信息。

第十九章 软件趋势/动向

为了说明本书中讨论的原理，本章考虑了过去几十年来在软件开发中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理之间的关系，并使用这些原理来评估该趋势是否提供了针对软件复杂性的杠杆作用。

19.1 面向对象编程和继承

在过去的 30-40 年中，面向对象编程是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之类的概念。如果仔细使用，这些机制可以帮助产生更好的软件设计。例如，私有方法和变量可用于确保信息隐藏：类外的任何代码都不能调用私有方法或访问私有变量，因此它们上没有任何外部依赖关系。

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含义。继承的第一种形式是接口继承，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签名，但是不同的子类可以以不同的方式实现相同的方法。例如，该接口可能定义用于执行 I/O 的方法。一个子类可能对磁盘文件实现 I/O 操作，而另一个子类可能对网络套接字实现相同的操作。

接口继承通过出于多种目的重用同一接口，从而提供了针对复杂性的杠杆作用。它使解决一个问题（例如如何使用 I/O 接口读取和写入磁盘文件）中获得的知识可以用于解决其他问题（例如通过网络套接字进行通信）。关于深度的另一种思考方式是：接口的实现越不同，接口就越深入。为了使接口具有许多实现，它必须捕获所有基础实现的基本功能，同时避免实现之间的差异。这个概念是抽象的核心。

继承的第二种形式是实现继承。以这种形式，父类不仅定义了一个或多个方法的签名，而且还定义了默认实现。子类可以选择继承方法的父类实现，也可以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，则可能需要在几个子类中复制相同的方法实现，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复制）。因此，实现继承减少了随着系统的发展而需要修改的代码量。换句话说，它减少了第 2 章中描述的变化放大问题。

但是，实现继承会在父类及其每个子类之间创建依赖关系。父类和子类通常都访问父类中的类实例变量。这会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，以确保所做的更改不会破坏任何内容。同样，如果子类覆盖父类中的方法，则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，程序员将需要完全了解父类下的整个类层次结构，以便对任何类进行更改。

因此，应谨慎使用实现继承。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，可以使用小型帮助程序类来实现共享功能。原始类可以从辅助类的功能构建，而不是从父类继承函数。

如果没有实现继承的可行选择，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。这适用于隐藏在类层次结构中的信息的概念，以减少依赖性。

尽管面向对象编程提供的机制可以帮助实现干净的设计，但是它们本身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，那么它们仍将导致很高的复杂性。

19.2 敏捷开发

敏捷开发是一种软件开发方法，它是在 1990 年代末期出现的，其思想涉及如何使软件开发更加轻量，灵活和增量。它是在 2001 年的一次从业者会议上正式定义的。敏捷开发主要是关于软件开发的过程（组织团队，管理进度表，单元测试的角色，与客户交互等），而不是软件设计。但是，它与本书中的某些设计原则有关。

敏捷开发中最重要的元素之一是开发应该是渐进的和迭代的概念。在敏捷方法中，软件系统是通过一系列迭代开发的，每个迭代都添加并评估了一些新功能。每个迭代都包括设计，测试和客户输入。通常，这类似于此处提倡的增量方法。如第 1 章所述，在项目开始时就不可能对复杂的系统进行充分的可视化以决定最佳设计。最终获得良好设计的最佳方法是逐步开发一个系统，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。这类似于敏捷开发方法。

敏捷开发的风险之一是它可能导致战术编程。敏捷开发倾向于使开发人员专注于功能，而不是抽象，它鼓励开发人员推迟设计决策，以便尽快生产可运行的软件。例如，一些敏捷的从业者认为，你不应该立即实施通用机制。实现一个最小的特殊用途机制，从此开始，并在以后知道需要时重构为更通用的东西。尽管这些论点在一定程度上是合理的，但它们反对投资方法，并鼓励采用更具战术性的编程风格。这可以导致复杂性的快速累积。

渐进式开发通常是一个好主意，但是***渐进式开发应该是抽象的，而不是功能***。可以推迟对特定抽象的所有想法，直到功能需要它为止。一旦需要抽象，就要花一些时间进行简洁的设计。遵循第六章的建议并使其具有通用性。

19.3 单元测试

过去，开发人员很少编写测试。如果测试是由一个独立的 QA 团队编写的，那么它们就是由一个独立的 QA 团队编写的。然而，敏捷开发的原则之一是测试应该与开发紧密集成，程序员应该为他们自己的代码编写测试。这种做法现在已经很普遍了。测试通常分为两类:单元测试和系统测试。单元测试是开发人员最常编写的测试。它们很小，而且重点突出:每个测试通常在单个方法中验证一小段代码。单元测试可以独立运行，而不需要为系统设置生产环境。单元测试通常与测试覆盖工具一起运行，以确保应用程序中的每一行代码都经过了测试。每当开发人员编写新代码或修改现有代码时，他们都要负责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），这些测试可确保应用程序的不同部分都能正常协同工作。它们通常涉及在生产环境中运行整个应用程序。系统测试更有可能由独立的质量检查或测试小组编写。

测试，尤其是单元测试，在软件设计中起着重要作用，因为它们有助于重构。没有测试套件，对系统进行重大结构更改很危险。没有容易找到错误的方法，因此在部署新代码之前，很可能将无法检测到错误，因为在新代码中查找和修复它们的成本要高得多。结果，开发人员避免在没有良好测试套件的系统中进行重

构。他们尝试将每个新功能或错误修复的代码更改次数减至最少，这意味着复杂性会累积，而设计错误不会得到纠正。

有了一套很好的测试，开发人员可以在重构时更有信心，因为测试套件将发现大多数引入的错误。这鼓励开发人员对系统进行结构改进，从而获得更好的设计。单元测试特别有价值：与系统测试相比，它们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该套件。现有测试在发现新引擎中的错误方面是如此有效，以至于在字节码编译器的 alpha 版本发布之后仅出现了一个错误。

19.4 测试驱动开发

测试驱动开发是一种软件开发方法，程序员可以在编写代码之前先编写单元测试。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。没有测试通过，因为该类没有代码。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。所有测试通过后，该类结束。

尽管我是单元测试的坚决拥护者，但我不喜欢测试驱动的开发。***测试驱动开发的问题在于，它将注意力集中在使特定功能起作用，而不是寻找最佳设计上。***这是一种纯净而简单的战术编程，具有所有缺点。测试驱动的开发过于增量：在任何时间点，很容易破解下一个功能以进行下一个测试通过。没有明显的时间进行设计，因此很容易陷入混乱。

如第 19.2 节所述，开发单位应该是抽象的，而不是功能。一旦发现需要抽象，就不要随着时间的流逝而逐步创建抽象。一次设计所有功能（或至少足以提供一组合理全面的核心功能）。这样更有可能产生干净的设计，使各个部分很好地契合在一起。

首先编写测试的地方是修复错误。修复错误之前，请编写由于该错误而失败的单元测试。然后修复该错误，并确保现在可以通过单元测试。这是确保你已真正修复该错误的最佳方法。如果你在编写测试之前就已修复了该错误，则新的单元测试很可能实际上不会触发该错误，在这种情况下，它不会告诉你是否确实修复了该问题。

19.5 设计模式

设计模式是解决特定类型问题（例如迭代器或观察器）的常用方法。设计模式的概念在 Gamma，Helm，Johnson 和 Vlissides 的《设计模式：可重用的面向对象软件的元素》一书中得到了普及，现在设计模式已广泛用于面向对象的软件开发中。

设计模式代表了设计的替代方法：与其从头设计新的机制，不如应用一种众所周知的设计模式。在大多数情况下，这是件好事：出现设计模式是因为它们解决了常见的问题，并且因为它们被普遍同意提供干净的解决方案。如果设计模式在特定情况下运作良好，那么你可能很难想出另一种更好的方法。

设计模式的最大风险是过度使用。使用现有的设计模式并不能完全解决所有问题。当自定义方法更加简洁时，请勿尝试将问题强加到设计模式中。使用设计模式并不能自动改善软件系统。只有在设计模式合适的情况下才这样做。与软件设计中的许多想法一样，设计模式良好的概念并不一定意味着更多的设计模式会更好。

19.6 Getter 和 Setters

在 Java 编程社区中，getter 和 setter 方法是一种流行的设计模式。一个 getter 和一个 setter 与一个类的实例变量相关联。它们具有类似 getFoo 和 setFoo 的名称，其中 Foo 是变量的名称。getter 方法返回变量的当前值，setter 方法修改该值。

由于实例变量可以公开，因此不一定必须使用 getter 和 setter 方法。getter 和 setter 的论点是，它们允许在获取和设置时执行其他功能，例如在变量更改时更新相关值，通知更改的侦听器或对值实施约束。即使最初不需要这些功能，也可以稍后添加它们而无需更改界面。

尽管如果必须公开实例变量，则可以使用 getter 和 setter 方法，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实现的一部分在外部是可见的，这违反了信息隐藏的思想，并增加了类接口的复杂性。Getter 和 Setter 是浅层方法（通常只有一行），因此它们在不提供太多功能的情况下为类的接口增加了混乱。最好避免使用 getter 和 setter（或任何暴露的实现数据）。

建立设计模式的风险之一是，开发人员认为该模式是好的，并尝试尽可能多地使用它。这导致 Java 中的 getter 和 setter 的过度使用。

19.7 结论

每当你遇到有关新软件开发范例的提案时，就必须从复杂性的角度对其进行挑战：该提案确实有助于最大程度地降低大型软件系统的复杂性吗？从表面上看，许多建议听起来不错，但是如果你深入研究，你会发现其中一些会使复杂性恶化，而不是更好。

第二十章 设计性能

到目前为止，关于软件设计的讨论都集中在复杂性上。目标是使软件尽可能简单易懂。但是，如果你在需要快速的系统上工作，该怎么办？性能方面的考虑应如何影响设计过程？本章讨论如何在不牺牲简洁设计的情况下实现高性能。最重要的想法仍然是简单性：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“你在正常的开发过程中应该为性能多少担心？” 如果你尝试优化每条语句以获得最大速度，则它将减慢开发速度并产生许多不必要的复杂性。此外，许多“优化”实际上对性能没有帮助。另一方面，如果你完全忽略了性能问题，则很容易导致遍及整个代码的大量效率低下。结果系统很容易比所需的速度慢 5–10 倍。在这种“千刀砍死”的情况下，以后很难再回来提高性能了，因为没有单一的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，在这种极端情况下，你可以使用性能的基本知识来选择“自然高效”但又干净又简单的设计替代方案。关键是要了解哪些操作根本是昂贵的。以下是一些今天相对昂贵的操作示例：

- 网络通信：即使在数据中心内，往返消息交换也可能花费 10–50 µs，这是数以万计的指令时间。广域往返可能需要 10 到 100 毫秒。
- I/O 到辅助存储：磁盘 I/O 操作通常需要 5 到 10 毫秒，这是数百万条指令时间。闪存存储需要 10–100 µs。新出现的非易失性存储器的速度可能高达 1 µs，但这仍约为 2000 条指令时间。
- 动态内存分配（C 语言中的 malloc，C ++或 Java 中的新增功能）通常涉及分配，释放和垃圾回收的大量开销。
- 高速缓存未命中：将数据从 DRAM 提取到片上处理器高速缓存中需要数百条指令时间；在许多程序中，整体性能取决于缓存未命中和计算成本。

了解哪些东西最昂贵的最好方法是运行微基准测试（小型程序，这些程序单独测量单个操作的成本）。在 RAMCloud 项目中，我们创建了一个简单的程序，该程序提供了微基准测试的框架。创建该框架花了几天时间，但是该框架使在五到十分钟内添加新的微基准成为可能。这使我们积累了几十个微基准。我们既可以使用它们来了解 RAMCloud 中使用的现有库的性能，也可以衡量为 RAMCloud 编写的新类的性能。

一旦对什么是昂贵和什么便宜有了一般的认识，就可以使用该信息尽可能地选择便宜的业务。在许多情况下，更有效的方法将与较慢的方法一样简单。例如，当存储将使用键值查找的大量对象时，可以使用哈希表或有序映射。两者都通常在库包中提供，并且都简单易用。但是，哈希表可以轻松地快 5-10 倍。因此，除非需要映射提供的排序属性，否则应始终使用哈希表。

作为另一个示例，请考虑使用诸如 C 或 C ++之类的语言分配结构数组。有两种方法可以执行此操作。一种方法是让数组保留指向结构的指针，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分配空间。将结构存储在数组本身中效率要高得多，因此你只为所有内容分配一个大块。

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。如果更高效的设计仅增加了少量复杂性，并且复杂性是隐藏的，因此它不影响任何接口，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很多实现复杂性，或者导致更复杂的接口，那么最好是从更简单的方法开始，然后在性能出现问题时进行优化。但是，如果你有明确的证据表明性能在特定情况下很重要，那么你最好立即实施更快的方法。

在 RAMCloud 项目中，我们的总体目标之一是为客户端计算机通过数据中心网络访问存储系统提供尽可能低的延迟。结果，我们决定使用特殊的硬件进行联网，从而使 RAMCloud 绕过内核并直接与网络接口控制器进行通信以发送和接收数据包。即使增加了复杂性，我们还是做出了这个决定，因为我们从先前的测量中知道，基于内核的网络太慢了，无法满足我们的需求。在其余的 RAMCloud 系统中，我们能够进行简单设计。解决这个大问题“对”使其他事情变得更加容易。

通常，较简单的代码往往比复杂的代码运行更快。如果你定义了特殊情况和例外，则无需代码即可检查这些情况，并且系统运行速度更快。深层类比浅层类更有效，因为它们为每个方法调用完成了更多工作。浅类会导致更多的层交叉，并且每个层交叉都会增加开销。

20.2 修改前的度量

但是，即使你如上所述进行设计，也请假设你的系统仍然太慢。根据你对慢速运动的直觉，急于着手开始进行性能调整。不要这样！程序员对性能的直觉是不可靠的。即使对于有经验的开发人员也是如此。如果你开始根据直觉进行更改，则会浪费时间在实际上无法提高性能的事情上，并且可能会使系统变得更加复杂。

进行任何更改之前，请测量系统的现有行为。这有两个目的。首先，这些测量将确定性能调整将产生最大影响的地方。仅仅测量顶级系统性能是不够的。这可能会告诉你系统速度太慢，但不会告诉你原因。你需要进行更深入的衡量，以详细确定影响整体绩效的因素；目标是确定系统当前花费大量时间的少量非常具体的地方，以及你有改进想法的地方。测量的第二个目的是提供基线，以便你可以在进行更改后重新测量性能，以确保性能得到实际改善。如果这些更改并未在效果上带来可衡量的变化，然后将其退出（除非它们使系统更简单）。除非能够显着提高速度，否则保持复杂性毫无意义。

20.3 围绕关键路径设计

在这一点上，我们假设你已经仔细分析了性能，并确定了一段缓慢的代码来影响整个系统的性能。改善其性能的最佳方法是进行“根本”更改，例如引入缓存，或使用其他算法方法（例如，平衡树与列表）。我们决定绕过内核进行 RAMCloud 中的网络通信的决定是一个基本修补程序的示例。如果你可以确定基本修复程序，则可以使用前面各章中讨论的设计技术来实施它。

不幸的是，有时会出现一些根本无法解决的情况。这将我们带到本章的核心问题，即如何重新设计现有代码，使其运行更快。这应该是你的不得已的方法，并且不应该经常发生，但是在某些情况下它可能会带来很大的不同。关键思想是围绕关键路径设计代码。

首先，问自己在通常情况下执行所需任务必须执行的最少代码量是多少。忽略任何现有的代码结构。相反，想象一下你正在编写一个仅实现关键路径的新方法，这是在最常见的情况下必须执行的最少代码量。

当前的代码可能充满特殊情况。在此练习中，请忽略它们。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，你可以将所有相关代码放在一个方法中。当前代码还可以使用各种变量和数据结构。仅考虑关键路径所需的数据，并假定最适合关键路径的任何数据结构。例如，将多个变量合并为一个值可能很有意义。假设你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。我们将此代码称为“理想”。

理想的代码可能会与你现有的类结构冲突，并且可能不切实际，但它提供了一个很好的目标：这代表了代码可能是最简单，最快的。下一步是寻找一种新设计，使其尽可能接近理想状态，同时又要保持干净的结构。你可以应用本书前面各章中的所有设计思想，但要保持（最好）保持理想代码的附加约束。你可能需要在理想情况下添加一些额外的代码，以允许使用简洁的抽象。例如，如果代码涉及哈希表查找，则可以向通用哈希表类引入额外的方法调用。以我的经验，几乎总是可以找到干净简洁的设计，但非常接近理想。

在此过程中发生的最重要的事情之一是从关键路径中除去特殊情况。当代码运行缓慢时，通常是因为它必须处理各种情况，并且代码经过结构化以简化所有不同情况的处理。每个特殊情况都以额外的条件语句和/或方法调用的形式向关键路径添加了一些代码。这些添加中的每一个都会使代码变慢。重新设计性能时，请尝试减少必须检查的特殊情况的数量。理想情况下，开头应该有一个 if 语句，该语句可以通过一个测试检测所有特殊情况。在正常情况下，只需要进行一项测试，之后就可以执行关键路径，而对于特殊情况则无需进行其他测试。如果初始测试失败（这意味着发生了特殊情况），则代码可以分支到关键路径之外的单独位置以进行处理。对于特殊情况，性能并不是那么重要，因此你可以为简化而不是性能来构造特殊情况的代码。

20.4 例如：RAMCloud 缓冲区

让我们考虑一个示例，其中 RAMCloud 存储系统的 Buffer 类经过优化，以使大多数常见操作的速度提高约 2 倍。

RAMCloud 使用 Buffer 对象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。缓冲区旨在减少内存复制和动态存储分配的开销。缓冲区存储看似线性的字节数组，但是为了提高效率，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。通过附加数据块来创建缓冲区。每个块都是外部的或内部的。如果块在外部，则其存储由调用方拥有；缓冲区保留对此存储的引用。外部块通常用于大型块，以避免内存复制。如果内部有块，则 Buffer 拥有该块的存储；调用者提供的数据将被复制到缓冲区的内部存储器中。每个缓冲区包含一个小的内置分配，这是一个内存块，可用于存储内部块。如果此空间已用完，则缓冲区将创建其他分配，销毁缓冲区时必须释放这些分配。内部块对于内存复制成本可忽略不计的小块很方便。图 20.1 显示了具有 5 个块的 Buffer：第一个块是内部的，接下来的两个块是外部的，最后两个块是内部的。

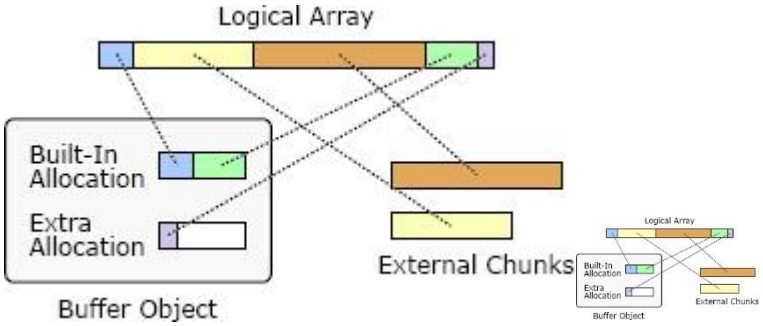

图 20.1：Buffer 对象使用内存块的集合来存储看似线性字节数组。内部块由 Buffer 拥有，并在 Buffer 销毁时释放；外部块不属于缓冲区。

Buffer 类本身代表“根本性的修补程序”，因为它消除了没有它就需要的昂贵的内存副本。例如，在 RAMCloud 存储系统中组装包含短标头和大对象内容的响应消息时，RAMCloud 使用带有两个块的 Buffer。第一个块是包含头的内部块；第二个块是一个外部块，它引用 RAMCloud 存储系统中的对象内容。可以在不复制大对象的情况下将响应收集到缓冲区中。

除了允许不连续块的基本方法外，我们没有尝试在原始实现中优化 Buffer 类的代码。但是，随着时间的流逝，我们注意到缓冲区越来越多地被使用。例如，在每个远程过程调用的执行期间，至少创建四个缓冲区。最终，很明显，加速 Buffer 的实现可能会对整体系统性能产生显着影响。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，在为请求和响应消息创建标题时，就会发生这种情况。我们决定将此操作用作优化的关键路径。在最简单的情况下，可以通过扩大 Buffer 中最后存在的块来分配空间。但是，只有在最后一个现有块位于内部，并且其分配中有足够的空间来容纳新数据时，才有可能这样做。理想的代码将执行一次检查以确认简单方法是否可行，然后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 Buffer :: alloc 方法开头。在最快的情况下，Buffer :: alloc 调用 Buffer :: allocateAppend，后者调用 Buffer :: Allocation :: allocateAppend。从性能的角度来看，此代码有两个问题。第一个问题是要单独检查许多特殊情况：

- 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，一次在其返回值由 Buffer::allocateAppend 测试时。
- Buffer::alloc 测试 Buffer::allocAppend 的返回值，以再次确认分配成功。

此外，该代码没有尝试直接扩展最后一个块，而是在不考虑最后一个块的情况下分配了新空间。然后，Buffer::alloc 检查该空间是否恰好与最后一块相邻，在这种情况下，它将新空间与现有块合并。这导致其他检查。总体而言，此代码测试关键路径中的 6 种不同条件。

原始代码的第二个问题是它具有太多的层，所有层都很浅。这既是性能问题，也是设计问题。关键路径除了对 Buffer::alloc 的原始调用之外，还进行了另外两个方法调用。每个方法调用花费额外的时间，并且

每个调用的结果必须由其调用者检查，这导致需要考虑更多特殊情况。第七章讨论了当你从一层传递到另一层时，抽象通常应该如何变化，但是图 20.2 中的所有三种方法都具有相同的签名，并且它们提供了基本相同的抽象。这是一个危险信号。Buffer::allocateAppend 几乎是一个传递方法；它的唯一作用是在需要时创建新的分配。额外的层使代码既慢又复杂。

为了解决这些问题，我们重构了 Buffer 类，使其设计围绕最关键性能的路径进行。我们不仅考虑了上面的分配代码，还考虑了其 他几种常用的执行路径，例如检索当前存储在 Buffer 中的数据的字节总数。对于这些关键路径中的每一个，我们试图确定在通常情况下必须执行的最少代码量。然后，我们围绕这些关键路径设计了课程的其余部分。我们还应用了本书中的设计原则来简化整个类。例如，我们消除了浅层并创建了更深的内部抽象。重构的类比原始版本小 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 的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：用于在 Buffer 的内部块中分配新空间的新代码。

图 20.3 显示了用于在 Buffer 中分配内部空间的新关键路径。新代码不仅速度更快，而且更容易阅读，因为它避免了浅层抽象。整个路径使用单一方法处理，并且使用单一测试排除所有特殊情况。新代码引入了新的实例变量 extraAppendBytes，以简化关键路径。此变量跟踪缓冲区中最后一个块之后立即有多少未使用空间可用。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 中的代码表示处理这种常见情况的最少代码量。

注意：只要需要，就可以通过重新计算各个块的总缓冲区长度来消除对 totalLength 的更新。但是，这种方法对于具有许多块的大型 Buffer 而言将是昂贵的，并且获取 Buffer 的总长度是另一种常见的操作。因此，我们选择添加少量额外的开销来分配，以确保 Buffer 长度始终立即可用。

新代码的速度约为旧代码的两倍：使用内部存储将 1 字节字符串附加到缓冲区的总时间从 8.8 ns 降低到 4.75 ns。由于修订，许多其他缓冲区操作也加快了速度。例如，构建新缓冲区，在内部存储中附加一小块并销毁缓冲区所需的时间从 24 ns 降至 12 ns。

20.5 结论

本章最重要的总体教训是，干净的设计和高性能是兼容的。重写 Buffer 类可将其性能提高 2 倍，同时简化其设计并将代码大小减少 20%。复杂的代码通常会很慢，因为它会执行多余或多余的工作。另一方面，如果你编写干净，简单的代码，则系统可能会足够快，因此你一开始就不必担心性能。在少数需要优化性能的情况下，关键再次是简单性：找到对性能最重要的关键路径并使它们尽可能简单。

第二十一章 结论

这本书是描述关于一件事的：复杂性。处理复杂性是软件设计中最重要的挑战。这是使系统难以构建和维护的原因，并且通常也使它们变慢。在本书的整个过程中，我试图描述导致复杂性的根本原因，例如依赖性和隐晦性。我已经讨论了可以帮助你识别不必要的复杂性的危险标记，例如信息泄漏，不必要的错误情况或名称过于笼统。我已经提出了一些通用的思想，可以用来创建更简单的软件系统，例如，努力研究更深和更通用的类，定义不存在的错误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思路。

所有这些建议的缺点是它们会在项目的早期阶段创建额外的工作。此外，如果你不习惯于思考设计问题，那么当你学习良好的设计技巧时，你甚至会放慢脚步。如果对你而言唯一重要的事情就是尽快使当前代码工作，那么思考设计就好像是在费劲工作，而这实际上妨碍了你实现真正的目标。

另一方面，如果良好的设计对你来说是重要的目标，那么本书中的思想应使编程更有趣。设计是一个令人着迷的难题：如何用最简单的结构解决特定问题？探索不同的方法很有趣，找到一种既简单又强大的解决方案是一种很好的感觉。干净，简单和明显的设计是一件美丽的事情。

此外，你对优质设计的投资将很快获得回报。在项目开始时仔细定义的模块将为你节省时间，因为你一遍又一遍地重复使用它们。你六个月前编写的清晰文档将为你节省返回代码添加新功能的时间。花在磨练设计技能上的时间也将有所回报：随着技能和经验的增长，你会发现可以越来越快地制作出好的设计。一旦知道了什么，一个好的设计实际上并不会比一个简单的设计花费更多的时间。

成为优秀设计师的好处是，你可以在设计阶段花费大部分时间，这很有趣。可怜的设计师花费大量时间在复杂而脆弱的代码中寻找错误。如果提高设计技能，不仅可以更快地生产出更高质量的软件，而且软件开发过程也将变得更加愉快。

Summary of Design Principles

1, Complexity is incremental : you have to sweat the small stuff 【软件的复杂度是慢慢的积累起来的，你必须锱铢必较。】
解释：

软件复杂度都是慢慢累积起来的。初始设计多少都是能看的，但是真实开发中有太多的场景，我们没有搞懂原有设计，或者是搞懂了也不敢改，而是通过打补丁的方式支持一个新的功能，因此迭代几轮之后几乎都变得面目全非无法理解了。这个是作者所谓“战术编程” 的方式。

而要达成“战略编程”，就要求我们必须在平日的每一次迭代中都“锱铢必较”，维护整体的设计，在完整的框架下来扩展功能。当然这要求设计要有前瞻性，能够满足一般的功能扩展需求。更多也是对于我们软件工程的挑战，要求整个团队都清晰认识到软件质量的问题，并且持续投入。

2, Working code isn't enough
3, Make continual small investments to improve system design 【通过持续的小投入来改进系统设计】
解释：

和第 1 条的要求是一样的，但是出发点不一样。这条说的是，别想着一次搞定完整的设计，而是在开发过程中逐步改进。书里说了，一次搞定就是瀑布流的思想，逐步改进就是敏捷思想。如果我们发现原有的框架不能很好的支持新功能，应当是持续的去优化原有设计框架，而不是打补丁。

4, Modules should be deep
5, Interfaces should be designed to make the most common usage as simple as possible 【接口设计应当使最常用的路径越简单越好】
6, It's more important for a module to have a simple interface than a simple implementation 【相比起接口实现上的简单，一个模块的接口的简单更加重要】

7, General-purpose modules are deeper
解释：

模块应当尽量设计的通用，但因为需求的多变性，也不要盲目的追求通用。一个最佳实践是在基本不影响当前需求的实现复杂度的同时，把接口设计的尽量通用。

8, Separate general-purpose and special-purpose code【分离通用的代码和特定需求的代码】
解释：

往小了说就是抽取公共函数或者公共类，往大了说就是建立共享服务或者中台系统

9, Different layers should have different abstractions 【不同的层次应当有不同的抽象】
解释：

如果不同层次的抽象是一样的，那必然会导致大量的重复代码和没有意义的类。

我觉得作者在书中比较反对 pass-through 方法（没什么逻辑而只是调用其他方法）和装饰者模式，还是有一些适用的场景。有一种情况就是虽然操作是一样的，但是处在不同层次的抽象级别；另外一种是虽然当前没什么操作，但是有可能会在将来有变动。当然肯定不能滥用，尽量保持简单还是第一前提。

10, Pull complexity downward【把复杂性下沉到底层】
11, Define errors (and special cases) out of existence 【通过定义让错误和特殊的场景不存在（没有机会发生）】
解释:

很有启发性的观点。其实就是因为错误处理非常困难，最好就是让接口的设计更加包容，从而在定义上就无需处理错误。我觉得这个还是对于正确性和易用性的权衡，对于非常重要的接口，还是要对使用条件检查的更加严格，避免严重的错误。而不太重要的接口应当尽量能够处理宽泛的场景，减少使用者的心智负担。

作者特别指出了自己在 Tcl 语言中对 unset 操作的设计，要求只有在 set 状态下才能调用，这就导致了很多额外处理的代码和报错，其实完全可以在没有 set 的情况下忽略这个操作。出发点也是担心使用者可能会用错这个方法。但实践证明这个设计导致了很多额外的检查代码却没什么好处，因此是得不偿失的。

12, Design it twice
解释:

感觉这个更多是一种态度和工作方式。我们不要拘泥于初始想法，而是要考虑多个方案，从中选择或者整合最好的方案，并且应当在项目开发之后进行复盘。这肯定是提高软件设计质量的好方法，恐怕也是提升任何能力的好方法。

13, Comments should describe things that are not obvious from the code 【注释要描述那些在代码中不明显的内容】

作者在书中花了大量篇幅来描述注释，认为注释应当是接口重要的一部分，而且普通用户应当可以通过注释来完整的理解接口。我悲观的认为在大部分实际应用开发中难以推行，尤其是快速迭代的应用中。相比较而言我觉得保持代码整洁可能是更加实用的。当然一些设计文档肯定还是必须的。

14, Software should be designed for ease of reading, not ease of writing. 【软件的设计应该便于阅读，而不是便于书写。】
解释:

就是要尽量保持代码清晰可读，哪怕写的时候会稍微麻烦一点。完全同意。而且，代码可读重点是针对人，而不是针对机器。

15, The increments of software development should be abstractions, not features. 【软件开发中的增量应该是抽象，还不是特性】

DESC:
我理解就是要在设计的过程中尽量采用通用的设计，不但可以满足当前特性的需求，也应当能够满足（可预见的）相关特性的需求。
如果每次的增量都只是一些特性开发，没有努力的建立统一模型，那么系统中肯定就是会充满各种补丁，也就是作者所说的战术编程。

关于作者：

John Ousterhout 是斯坦福大学计算机科学教授。他是 Tcl 脚本语言的创建者，并且以在分布式操作系统和存储系统中的工作而闻名。John Ousterhout 在耶鲁大学获得了物理学学士学位，并在卡内基梅隆大学获得了计算机科学博士学位。他是美国国家工程院院士，并获得了无数奖项，包括 ACM 软件系统奖，ACM Grace Murray Hopper 奖，美国国家科学基金会总统年轻研究者奖和 UC Berkeley 杰出教学奖。

英文版（豆瓣）地址参阅：https://book.douban.com/subject/30218046/