代码大全笔记

张谦

2020 年 8 月 18 日

目录

1 欢迎进入软件构件的世界

1.1 什么是软件构建

如下图所示，构建活动主要是编码与调试，但也涉及详细设计、规划构建、单元测试、集成、集成测试等其他活动。

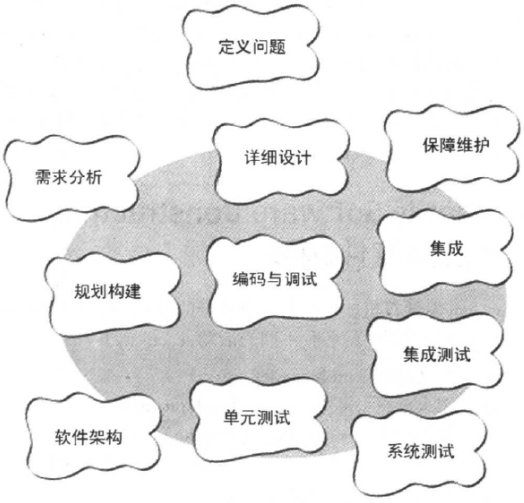


图1-1 构建活动用灰色的椭圆表示。构建活动主要关注于编码与调试，但也包含详细设计、单元测试、集成测试以及其他一些活动

构建活动包含如下具体任务：

- 验证有关的基础工作已经完成，保证构建活动可以顺利进行；
- 确定如何测试所写的代码；
- 设计并编写类和子程序；
- 创建并命名变量和具名常量；
- 选择控制结构，组织语句块；
- 对代码进行单元测试和集成测试，并排除其中的错误；
- 评审开发团队其他成员的底层设计和代码，并让他们评审你的工作；
- 优化代码，仔细进行代码的格式化和注释；
- 将单独开发的多个组件集成为一体；
- 调整代码，让它更快、更省资源。

1.2 构建为什么重要

- 构建活动是软件开发的主要组成部分；
- 构建活动是软件开发中的核心活动；
- 将主要精力集中于构建活动，可以大大提高程序员生产率；
- 构建活动的产物，源代码，往往是对软件的唯一精确描述；
- 构建活动是唯一一项确保完成的工作。

2 用隐喻理解软件开发

隐喻是启示而不是算法，可以将软件开发过程与其他熟悉的活动联系在一起，帮助更好地理解开发过程。相比其他隐喻，例如写作、种植和养殖等，通过将软件的构建过程，比作房屋的建设过程，能够更好地理解软件构建的各个阶段。

2.1 建造隐喻

（1）问题定义 (problem definition)：决定准备建一个什么类型的房子；

（2）架构设计 (architectural design)：和某个建筑师探讨总体设计，并得到批准；

（3）详细设计：画出详细的蓝图，雇一个承包人；

（4）软件构建 (construction)：准备好建造地点，打好地基，搭建房屋框架，砌好边墙，盖好房顶，通好水、电、煤气等；

（5）软件优化：在房子大部分完成后，庭院设计师、油漆匠和装修工还要把新盖的房子以及里面的家什美化一番；

（6）评审和审查 (reviews,inspections)：在整个过程中，还会有各种监察人员来检查工地、地基、框架、布线以及其他需要检查的地方。

2.2 已有组件

当开发软件时，会大量使用高级语言所提供的功能，而不会自己去编写操作系统层次的代码；自己编写那些能买得到的现成程序库是没有意义的，例如一些容器类、科学计算函数、用户界面组件、数据库访问组件等。在建造房子的时候，你也不会去试着建造那些能买得到的东西，例如洗衣机、冰箱、餐桌等。

2.3 定制组件

如果想建造一间拥有一流家具的高档住宅，可能就需要定制的橱柜，以及和橱柜搭配的洗碗机和冰箱等。在软件开发中也有这种订制的情况，例如想要开发一款一流的软件产品，可能会自己编写科学计算函数，以便获得更快的速度和更高的精度。

2.4 防止过度计划

适当的多层次规划对于建造房屋和构建软件都是有好处的，如果按错误的顺序构建软件，那么编码、测试和调试都会很难。精心计划，并不是事无巨细的计划或过度计划，例如你可以把房屋的结构性支撑规划清楚，在日后再决定是用木地板还是瓷砖地板，墙面漆成什么颜色等。

2.5 不同软件项目

建筑业中，盖一间仓库或工具房，或是一座医院或核反应站，在规划、设计和质量保证方面所需达到的程度是不一样的，所用的方法也不相同。同理，在软件开发中，通常只需要用灵活的、轻量级的方法，但有时你就必须用严格的、重量级的开发方法，以达到所需的安全性目标或其他目标。另外，还需要特别关注工作时间，在建造帝国大厦时，每辆运料车运输时都留有 15 分钟的余地，如果某辆车没能在指定的时间到位，则整个工期就会延误。对于超大型的软件项目，就需要比一般规模的项目有更高级的规划设计，如果需要创造在经济规模上可以匹敌帝国大厦的庞大软件项目，那么与之相当水准的技术与管理控制也是必需的。

3 构建前期准备

3.1 前期准备的重要性

准备工作的中心目标就是降低风险，软件开发中最常见的项目风险是糟糕的需求分析和糟糕的项目计划，因此准备工作就倾向于集中改进需求分析和项目规划。高质量的实践方法在项目的初期、中期和末期都强调质量：

（1）如果在项目末期强调质量，那么你会强调系统测试；但是测试只是完整的质量保证策略的一部分，而且不是最有影响的部分；

（2）如果在项目中期强调质量，那么你会强调构建实践；

（3）如果在项目开始阶段强调质量，那么你就会计划、要求并设计一个高质量的产品；例如你用为吉利车做的设计来开始整个生产过程，尽管你可以想尽办法来测试，它也绝对不会变成奔驰；也许你能造出最好的吉利车，但是如果你想要的是奔驰，那么你就得从头开始做设计。

3.2 序列式开发和迭代式开发选择

绝大多数的项目都不会完全使用序列式开发法或完全使用迭代式开发法。预先详细说明 100% 的需求和设计是不切实际的，不过对绝大多数项目来说，尽早把那些最关键的需求要素和架构要素确定下来，时很有价值。

可能因为下列原因选择一个更加迭代的方法：

- 需求并没有被理解透彻，或者出于其他理由你认为它是不稳定的；
- 设计很复杂，或者有挑战性，或者两者兼具；
- 开发团队对于这一应用领域不熟悉；
- 项目包含许多风险；
- “长期可预测性”不重要；
- 后期改变需求、设计和编码的代价很可能比较低。

相反的，你可能需要选择一个更加序列的方法。

3.3 问题定义的先决条件

在开始构建之前，首先要满足的一项先决条件是，对这个系统要解决的问题做出清楚的陈述。问题定义只定义了问题是什么，而不涉及任何可能的解决方案。它是一个很简单的陈述，并且听起来应该像个问题。例如“我们跟不上客户的订单了”听起来就像个问题，而且确实是一个很好的问题定义；而“我们需要优化数据自动采集系统，使之跟上客户的订单”，这种就是糟糕的问题定义，它听起来不像问题，而像解决方案。另外，问题定义应该用客户的语言来书写，而且应该从客户的角度来描述问题。

3.4 需求的先决条件

“需求”详细描述软件系统应该做什么，这是达成解决方案的第一步。需求明确有如下好处：

- 用户可以自行评审，并进行核准；否则程序员就常常会在编程期间自行决定需求；
- 有助于避免争论，如果你和另外一个程序员有分歧，可以查看书面的需求，已解决分歧；
- 有助于减少开始编程开发之后的系统变更的情况；
- 充分详尽地描述需求，是项目成功的关键，它甚至很可能比有效的构建技术更重要。

在构建期间处理需求变更，有以下一些可以采用的方式：

- 评估需求质量，如果需求不够好，则停止工作，退回去，先做好后再继续前进；
- 确保每一个人都知道需求变更的代价；
- 建立一套变更控制程序；
- 使用能适应变更的开发方法；
- 放弃这个项目；
- 注意项目的商业案例，注重商业价值。

3.5 架构的先决条件

软件架构是软件设计的高层部分，是用于支持更细节设计的框架。架构的质量决定了系统的“概念完整性”，继而决定了系统的最终质量。一个经过慎重考虑的架构，为“从顶层到底层维护系统的概念完整性”，提供了必备的结构和体系，它为程序员提供了指引，其细节程度与程序员的技能和手边的工作相配；它将工作分为几个部分，使多个开发者或多个开发团队可以独立工作。

架构的典型组成部分：

（1）程序组织：

- 系统架构首先要以概括的形式对有关系统做一个综述；
- 在架构中，应该能发现对那些曾经考虑过的，最终组织结构的，替代方案的记叙；找到之所以选用最终的组织结构，而不是其他替代方案的理由；
- 架构应该定义程序的主要构造块，根据程序规模的不同，各个构造块可能是单个类，也可能是由许多类组成的一个子系统；
- 应该明确定义各个构造块的责任，每个构造块应该负责某一个区域的事情，并且对其他构造块负责的区域知道得越少越好，将设计的信息局限在各个构造块之内；
- 应该明确定义每个构造块的通信规则，对于每个构造块，架构应该描述它能直接使用那些构造块，能间接使用哪些构造块，不能使用哪些构造块。

（2）主要的类：

- 架构应该详细定义所用的主要的类，应该指出每个主要的类的责任，以及该类如何与其他类交互；它应该包含对类的继承体系、状态转换、对象持久化等的描述；如果系统足够大，它应该描述如何将这类组织成一个个子系统；
- 架构应该记述曾经考虑过的其他类设计方案，并给出选用当前方案的理由；架构无需详细说明系统中的每一个类，利用 80/20 法则：对那些构成系统 80% 的行为的 20% 的类进行详细说明。

（3）数据设计：

- 架构应该描述所用到的主要文件和数据表的设计。它应该描述曾经考虑过的其他方案，并说明选择当前方案的原因。如果应用程序要维护一个客户 ID 的列表，而架构师决定使用顺序访问的列表来表示该 ID 的列表，那么文档就应该解释为什么顺序访问的列表比随机访问的列表、堆栈、散列表要好。在构建期间，这些信息让你能洞察架构师的思想；在维护阶段，这种洞察力是无价之宝。离开它，你就像看一部没有字幕的外语片；
- 数据通常只应该由一个子系统或一个类直接访问；例外的情况就是通过访问器类或访问器子程序，以受控且抽象的方式来访问数据；
- 架构应该详细定义所用数据库的高层组织结构和内容；架构应该解释为什么单个数据库比多个数据库要好，反之亦然。需要解释为什么不用平坦的文件，而要用数据库，指出与其他访问同一数据的程序的可能交互方式，说明创建哪些数据视图等等。

（4）业务规则：

如果架构依赖于特定的业务规则，那么它就应该详细描述这些规则，并描述这些规则对系统设计的影响。例如，假定要求系统遵循这样一条业务规则：客户信息过时的时间不能超过 30 秒。在此种情况下，架构就应该描述这条规则对架构采用的“保持客户信息及时更新且同步”的方法的影响。

（5）用户界面设计：

- 用户界面常常在需求阶段进行详细说明，如果没有，就应该在软件架构中进行详细说明。架构应该详细定义 Web 页面格式、GUI、命令行接口等主要元素；
- 架构应该模块化，以便在替换为新用户界面时，不影响业务规则和程序的输出部分。例如，架构应该使我们很容易做到：砍掉交互式界面的类，插入一组命令行的类。这种替换能力常常很有用，由其因为命令行界面便于单元级别和子系统级别的软件测试。1

（6）资源管理：

架构应该描述一份管理稀缺资源的计划。稀缺资源包括数据连接、线程、句柄等。在内存受限的应用领域，如驱动程序开发和嵌入式系统中，内存管理是架构应该认真对待的另一个重要领域。架构应该应该估算在正常情况和极端情况下的资源使用量。在简单的情况下，估算数据应该说明：预期的运行环境有能力提供所需的资源，在更复杂的情况下，也许会要求应用程序更主动地管理其拥有的资源。如果是这样，那么资源管理器应该和系统的其他部分一样，进行认真的架构设计。

（7）安全性：

架构应该描述实现设计层面和代码层面的安全性的方法。如果先前尚未建立威胁模型，那么就应该在架构阶段建立威胁模型。在制定编码规范的时候，应该把安全性牢记在心，包括处理缓冲区的方法、处理非受信数据（用户输入数据、cookies、配置数据和其他外部接口输入的数据）的规则、加密、错误信息的细致程度、保护内存中的秘密数据，以及其他事项。

（8）性能：

如果需要关注性能，就应该在需求中详细定义性能目标。性能目标可以包括资源的使用，这时，性能目标也应该详细定义资源（速度、内存、成本）之间的优先顺序。架构应该提供估计的数据，并解释为什么架构师相信能达到性能目标。如果某些部分存在达不到性能目标的风险，那么架构也应该指出来。如果为了满足性能目标，需要在某些部分使用特定的算法或数据类型，架构应该说清楚。架构中也可以包括各个类或各个对象的空间和时间预算。

（9）可伸缩性：

可伸缩性是指系统增长以满足未来需求的能力。架构应该描述系统如何应对用户数量、服务器数量、网络节点数量、数据库记录数、数据库记录的长度、交易量等的增长。如果预计系统不会增长，而且可伸缩性不是问题，那么架构应该明确地列出这一假设。

（10）互用性：

如果预计这个系统会与其他软件或硬件共享数据或资源，架构应该描述如何完成这一任务。

（11）国际化和本地化：

国际化是一项准备让程序支持多个地域的技术活动。国际化常常称为“I18n”，因为国际化的英文单词“Internationalization”首尾两个字符之间有 18 个字母。本地化活动是翻译一个程序，以支持当地特定的语言工作。

（12）输入输出：

输入输出 (I/O) 是架构中值得注意的另一个领域。架构应该详细定义读取策略是先做、后做还是即时做。而且应该描述在哪一层检测 I/O 错误：在字段、记录、流，或者文件的层次。

（13）错误处理：

错误处理已被证实为现代计算机科学中最棘手的问题之一，不能武断地处理它。因为错误处理牵连到整个系统，因此最好在架构层次上对待它：

- 错误处理是进行纠正还是仅仅进行检测？如果是纠正，程序可以尝试从错误中恢复过来。如果仅仅是检测，那么程序可以像没发生任何事一样继续运行，也 wiagua 可以退出。无论哪种情况，都应该通知用户说检测到一个错误；

- 错误检测时主动的还是被动的？系统可以主动地预测错误，例如，通过检查用户输入的有效性，也可以在不能避免错误的时候，被动地响应错误，例如，当用户输入的组合产生了一个数值溢出错误时。前者可以扫清障碍，后者可以清除混乱。同样，无论采用哪种方案，都与用户界面有影响；
- 程序如何传播错误？程序一旦检测到错误，它可以立刻丢弃引发错误的数据；也可以把这个错误当成一个错误，并进入错误处理状态；或者可以等到所有处理完成，再通知用户说在某个地方发现了错误；
- 错误消息的处理有什么约定？如果架构没有详细定义一个一致的处理策略，那么用户界面看起来就像“令人困惑的乱七八糟的抽象拼贴画”，由程序的不同部分的各种界面拼接而成。要避免这种外观体验，架构应该建立一套有关错误消息的约定；
- 如何处理异常？架构应该规定代码何时能够抛出异常，在什么地方捕获异常，如何记录这些异常，以及如何在文档中描述异常等等；
- 在程序中，在什么层次上处理错误？你可以在发现错误的地方处理，可以将错误传递到专门处理错误的类进行处理，或者沿着函数调用链往上传递错误；
- 每个类在验证其输入数据的有效性方面需要负何种责任？是每个类负责验证自己的数据有效性，还是有一组类负责验证整个系统的数据的有效性？某个层次上的类是否能假设它接收的数据是干净的？
- 你是希望用运行环境中内建的错误处理机制，还是想建立自己的一套机制？事实上，运行环境所拥有的某种特定的错误处理方法，并不是符合你需求的最佳方法。

（14）容错性：

架构还应该详细定义所期望的容错种类。容错是增强系统可靠性的一组技术，包括检测错误：如果可能的话，从错误中回复；如果不能从错误中回复，则包容其不利影响。例如，为了计算某数的平方根，系统的容错策略有以下几种：

- 系统在检测到错误的时候退回去，再试一次。如果第一次的结果是错误的，那么系统可以退回到之前一切正常的时刻，然后从该点继续运行；
- 系统拥有一套辅助代码，以备在主代码出错时使用。在本例中，如果发现第一次的答案似乎错误，系统就切换到另一个计算平方根的子程序，以取而代之；
- 系统使用一种表决算法。它可以有三个计算平方根的类，每一个都使用不同的计算方法；每个类分别计算平方根，然后系统对结果进行比较；根据系统内建的容错机制的种类，系统可以以三个结果的均值、中值或众数作为最终结果；
- 系统使用某个不会对系统其余部分产生危害的虚假值代替这个错误的值；
- 其他容错方法包括，在遇到错误时，让系统转入某种部分运转状态，或者转入某种功能退化状态；系统可以自动关闭或重启。

（15）架构的可行性：

设计师关注系统的各种能力，例如是否能达到性能目标，能够在有限的资源下运转，运行环境是否有足够的支持。架构应该论证系统的技术可行性。如果在任何一个方面不可行，都会导致项目无法实施；那么架构应该说明“这些问题是如何经过研究的”，通过验证概念的原型、研究或其他手段，必须在全面开展构建之前解决掉这些风险。

（16）过度工程：

健壮性 (robustness) 是指系统在检测到错误后，继续运行的能力。通常架构详细描述的系统，会比需求详细描述的系统更健壮。理由之一为，如果组成系统的各个部分都只能在最低限度上，满足健壮性要求，那么系统整体上是达不到所有要求的健壮程度的。在软件中，链条的强度不是取决于最薄弱的一环，而是等于所有薄弱环节的乘积。架构应该清楚地指出程序员应该“为了谨慎起见，宁可进行过度工程 (overengineering)”，还是应该做出最简单的能工作的东西。

详细定义一种过度工程的方法尤其重要，因为许多程序员会出于专业自豪感，对自己编写的类做过度工程。通过在架构中明确地设立期望目标，就能避免出现“某些类异常健壮，而其他类勉强够健壮”的现象。

（17）关于“买”还是“造”的决策：

如果架构不采用现货供应的组件，那么就应该说明“自己定制的组件，应该在哪些方面胜过现成的程序库和组件”。

（18）关于复用的决策：

如果开发计划提倡使用业已存在的软件、测试用例、数据格式或其他原料，架构应该说明：如何对复用的软件进行加工，使之符合其他架构目标（如果需要使之符合的话）。

（19）变更策略：

面对变更，软件架构师面临的一个主要挑战，是让架构足够灵活，能够适应可能出现的变化。

- 架构应当清楚地描述处理变更的策略。架构应该列出已经考虑过的可能会有所增强的功能，并说明“最有可能增强的功能，同样也是最容易实现的”。如果变更很可能出现在输入输出格式、用户交互的风格、需求的处理等方面，那么架构就应该说明：这些变更已经被预料到了，并且任何单一的变更都只会影响少数几个类。架构应该对变更的计划可以很简单，比如在数据文件中放入版本号、保留一些供将来使用的字段、或者将文件设计成能够添加新的表格。如果使用了代码生成器，那么架构应该说明，可预见的变更都不会超出该代码生成器的能力范围；
- 架构应该指出“延迟提交”所用的策略。比如说，架构也许规定使用表驱动技术。它也许还规定“表”中的数据是保存在外部文件中，而非直接写在代码中，这样就能做到在不重新编译的情况下修改程序。

（20）架构的总体质量：

优秀的架构规格书的特点在于，讨论了系统中的类、讨论了每个类背后的隐藏信息、讨论了“采纳或排斥所有可能的设计替代方案”的根本理由。

- 架构应该是带有少许特别附加物的，精炼且完整的概念体系。好的架构设计，应该与待解决的问题和谐一致；
- 在架构开发过程中的多种变更方式，每一项变更，都应该干净地融入整体概念；
- 架构的目标应该清楚地表述；
- 架构应该描述所有主要决策的动机；
- 优秀的软件架构，很大程度上是与机器和编程语言无关的；要尽可能地独立于环境；如果程序的用途就是去试验某种特定的机器或语言，那么这条指导原则就不适用了；
- 架构应该处于对系统，“欠描述”和“过度描述”之间的那条分界线上；设计者不应该将注意力放在某个部件上，而损害其他部件；
- 架构应该明确地指出有风险的区域；它应该解释为什么这些区域是有风险的，并说明已经采取了哪些步骤以使风险最小化；
- 架构应该包含多个视角，包括暴露隐藏的错误和不一致的情况，以及帮助程序员完整地理解系统的设计；
- 最后，架构不应该包含任何对你而言，很难理解的东西。

3.6 花费在前期准备上的时间

花费在问题定义、需求分析、软件架构上的时间，依据项目的需要而变化。一般说来，一个运作良好的项目，会在需求、架构以及其他前期计划方面，投入 $10\% - 20\%$ 的工作量，和 $20\% - 30\%$ 的时间。这些时间不包括详细设计的时间，因为详细设计是构建活动的一部分。

4 关键的构建决策

4.1 选择编程语言

研究表明，编程语言的选择从多个方面，影响生产率和代码质量。程序员使用熟悉的语言时，生产率比使用不熟悉的语言时要高。使用高级语言的程序员，能比使用较低级语言的程序员，达到更好的生产率和质量。每种编程语言都有其优点和弱点，要知道你使用的语言的明确优点和弱点。

4.2 编程约定

在高质量的软件中，可以看到“架构的概念完整性”，与“底层实现”之间的关系。“实现”必须与指导该实现的“架构”保持一致，并且这种一致性是内在的、固有的。这正是变量名称、类的名称、子程序名称、格式约定、注释约定等这些针对“构建活动”的指导方针的关键所在。在“构建”开始之前，讲清楚你使用的编程约定，编码约定的细节，要达到这样的精确度：在编写完软件之后，几乎不可能改变软件所遵循的编码约定。

4.3 深入一种语言去编程

需要理解“在一种语言上编程”和“深入一种语言去编程”的区别。大多数重要的编程原则，并不依赖于特定的语言，而依赖于你使用语言的方式。如果你使用的语言缺乏你希望的构件，或者倾向于出现其他种类的问题，那就应该试着去弥补它，发明你自己的编码约定、标准、类库以及其他改进措施。

5 软件构建中的设计

软件设计是指构思、创造或发明一套方案，把一份计算机软件的规格说明书要求，转变为可实际运行的软件。设计就是把需求分析和编码调试连接在一起的活动。好的高层设计能提供一个可以稳妥容纳多个较低层次设计的结构。

5.1 设计中的挑战

（1）设计是一个 Wicked 问题：

Wicked 问题是指那种只能通过解决或部分解决才能被明确的问题。你必须首先把这个问题“解决”一遍，以便能够明确地定义它，然后再次解决该问题，从而形成一个可行的方案。例子，有一座桥，设计时主要考虑的问题为是否足够结实，以承受设计负荷；没有意识到大风带来的横向谐波，最终导致大桥坍塌。

（2）设计是一个了无章法的过程：

软件设计的成果应该是组织良好、干净利落的，然而形成这个设计的过程，却并非如此清爽。

- 在设计过程中你会采取很多错误的步骤，多次误入歧途。事实上，犯错正是设计的关键所在，在设计阶段犯错并加以改正，其代价要比在编码后才发现同样的错误，并彻底修改低得多；
- 优劣设计之间的差异，往往非常微妙；
- 很难判断设计何时算是“足够好”了。

（3）设计是确定取舍和调整顺序的过程：

设计的一个关键内容，是去衡量彼此冲突的各项设计特性，例如存储空间、占用的网络带宽、时间成本等。

（4）设计涉及到诸多限制

设计的要点，一部分是在创造可能发生的事情，而另外一部分又是在限制可能发生的事情。如果人们在建造房屋时，拥有无限的时间、资源和空间，那么你会看到房屋不可思议地随意蔓延，每幢楼都有上百间屋子，一只鞋就可以占用一间屋子。

（5）设计是不确定的：

如果让三个人去设计一套同样的程序，可能会有三套截然不同的设计。

（6）设计是一个启发式过程：

因为设计充满了不确定性，因此设计也就趋于具有探索性，而不是保证能产生预期结果的课重复过程，设计过程中总会有试验和犯错误。

（7）设计是自然而然形成的：

设计是在不断地设计评估、非正式讨论、写试验代码以及修改试验代码中演化和完善的。

5.2 关键的设计概念

好的设计源于对一小批关键设计概念的理解。这一节将会讨论：复杂度所扮演的角色、设计应具有的特征、以及设计层次。

（1）复杂度管理：

- 本质问题和偶然问题：本质属性是一件事物必须具备、如果不具备就不再是该事物的属性；例如，汽车必须具有发动机、轮子和车门，否则就不能称其为汽车。偶然属性是指一件事物恰巧具有的属性，有没有这些属性，并不影响这件事物本身；例如，一辆汽车可能有不同的发动机，但是都是一辆汽车。软件开发中，大部分的偶然性难题，很久以前就得到解决了，例如，由笨拙的语法相关的偶然问题，大多已经从汇编语言到第三代编程语言的演进过程中解决了；集成编程环境更是进一步解决了由于开发工具之间，无法很好地协作而带来的效率问题。软件开发剩下的那些本质性难题，将会变得相对缓慢；究其原因，是因为从本质上说，软件开发就是不断地去发掘错综复杂、相互连接的整套概念的所有细节。即使我们能发明出一种与现实中，亟待解决的问题，有着相同术语的编程语言，但是人们想清楚地认清现实世界到底如何运作，仍然有很多挑战，因此编程仍会十分困难。当软件要解决更大规模的现实问题时，现实的实体之间的交互行为，就变得更为复杂，这些转而又增加软件解决方案的本质性问题。所有这些本质性困难的根源，都在于复杂性，不论是本质的，还是偶然的；

- 管理复杂度的重要性：在对导致软件项目失败的原因进行调查时，人们很少把技术原因归为项目失败的首要因素。项目的失败，大多数都是由于差强人意的需求、规划和管理所导致的。但是，当项目由技术因素导致失败时，其原因通常就是失控的复杂度。当没人知道对一处代码的改动，会对其他代码带来什么影响时，项目也就块停止进展了。因此管理负责度，是软件开发中最为重要的技术话题。

- 如何应对复杂度：作为软件开发人员，我们不应该试着在同一时间，把整个程序都塞进自己的大脑，而应该试着以某种方式去组织程序，以便能够在一个时刻，可以专注于一个特定的部分。这么做的目的是尽量减少在任一时间段内，所要考虑的程序量。在软件架构的层次上，可以通过把整个系统分解为多个子系统，来降低问题的复杂度。人类更容易理解许多项简单的信息，而不是一项复杂的信息。所有软件设计技术的目标，都是把复杂问题分解为简单的部分。子系统的相互依赖越少，你就越容易在同一时间里，专注问题的一小部分。精心设计的对象关系，使关注点相互分离，从而使你能在每个时刻，只关注一件事情。保持子程序（函数）的短小精悍，也能帮助你减少思考的负担。高代价、低效率的设计源于下面三种根源：

  – 用复杂的方法，解决简单的问题；
  – 用简单但错误的方法，解决复杂的问题；
  – 用不恰当的复杂方法，解决复杂的问题。

  现代的软件本身就很复杂，无论你多努力，最终都会与存于现实世界问题本身的，某种程度的复杂性不期而遇。这就意味着要用下面这两种方法来管理复杂度：

  – 把任何人在同一时间，需要处理的本质复杂度的量，减到最少；
  – 不要让偶然的复杂度无谓地快速增长。

  一旦你能理解软件开发中，任何其他技术目标，都不如管理复杂度重要时，众多设计上的考虑，就都变得直截了当了。

（2）良好的设计特征：

- 最小的复杂度：应该做简单且易于理解的设计，如果你的设计方案，不能让你在专注于程序的一部分时，安心地忽视其他部分，这一设计就没有什么作用了；

- 易于维护：请时刻想着维护程序员，可能对你的代码提出的问题，把维护程序员当成你的听众，进而设计出能自明的系统来；

- 松散耦合：在设计时，让程序的各个组成部分之间，关联最小。通过应用类接口中的合理抽象、封装性以信息隐藏等原则，设计出相互关联尽可能最少的类。减少关联也就减少了集成、测试与维护时的工作量；
- 可扩展性：能增强系统的功能，而无须破坏其底层结构。你可以改动系统的某一部分，而不会影响其他部分；
- 可重用性：所这几的系统的组成部分，能在其他系统中重复利用；
- 高扇入：让大量的类，使用某个给定的类；设计出的系统很好地利用了，在较低层次上的工具类；
- 低扇出：让一个类里，少量或适中使用其他的类；高扇出（超过约 7 个），说明一个类使用了大量其他的类，因此可能变得过于复杂；
- 可移植性：能方便移植到其他环境中；
- 精简性：设计出的系统没有多余部分；伏尔泰曾说，一本书的完成，不在它不能再加入任何内容的时候，而在不能再删去任何内容的时候。任何多余的代码，需要开发、复审和测试，并且当修改了其他代码之后，还要重新考虑它们；
- 层次性：尽量保持系统各个分解层的层次性，使你能在任意的层面上观察系统，并且得到某种具有一致性的看法，设计出来的系统应该能在任意层次上观察，而不需要进入其他层次；例如，假设你在编写一个新系统，其中用到很多设计不佳的旧代码，这是你就应该为新系统编写一个，负责同旧代码交互的层。层次化设计的溢出有：(a) 将低劣代码的烂泥潭禁闭起来；(b) 如果你最终能抛弃或重构旧代码，那是就不必修改除交互层之外的任何新代码；
- 标准技术：一个系统所依赖的外来的、古怪的东西越多，别人在第一次想要理解它的时候就越是头疼；要尽量用标准化的、常用的方法，让整个系统给人一种熟悉的感觉。

（3）设计层次：
如下图所示，一个软件系统包含有多个设计层次。

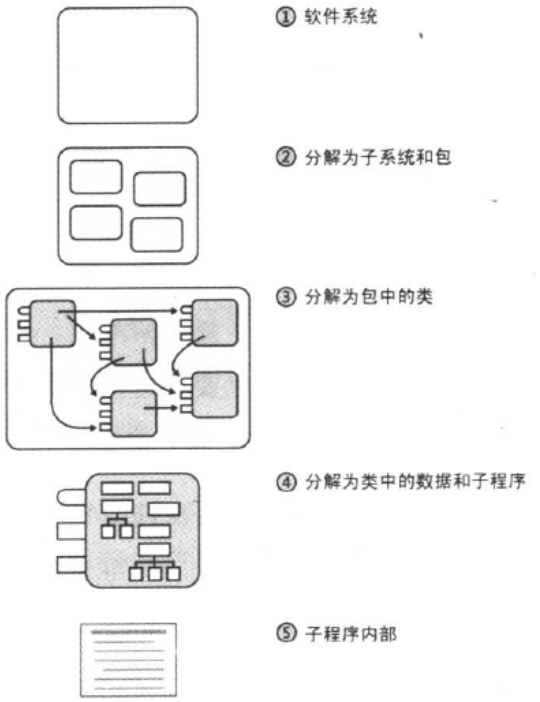


图 5-2　一个程序中的设计层次。系统①首先被组织为子系统②。子系统被进一步分解为类③，然后类又被分解为子程序和数据④。每个子程序的内部也需要进行设计⑤

- 第 1 层：软件系统。第一个层次就是整个系统，需要分解为子系统或包。

- 第 2 层：子系统或包。这一层的主要设计活动，就是确定如何把整个系统分为主要的子系统，并且定义清楚允许各子系统，如何使用其他子系统。这些子系统可能会很大，比如数据库、用户界面、业务规则、命令解释器、报表引擎等。在这一层的设计中，子系统之间的相互通信规则特别重要。如果所有的子系统都能和其他子系统通信，就完全失去了把它们分开所带来的好处。因此，应该通过限制子系统之间的通信，来让每个子系统更有存在的意义。

  例如，如下图所示，假设将系统分为 6 个子系统。在没有定义任何规则时，热力学第二定律就会发生作用，整个系统将会熵增。熵增的一种原因是，如果不对子系统间的通信加以任何限制，那么它们之间的通信就会肆意发生。这里的每个子系统，最终都会直接与所有其他子系统进行通信，如果改动某一个子系统，则其他所有和其通信的

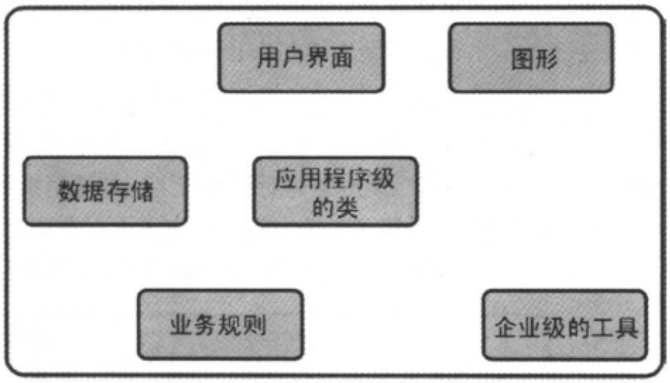


图 5-3　一个有六个子系统的系统示例

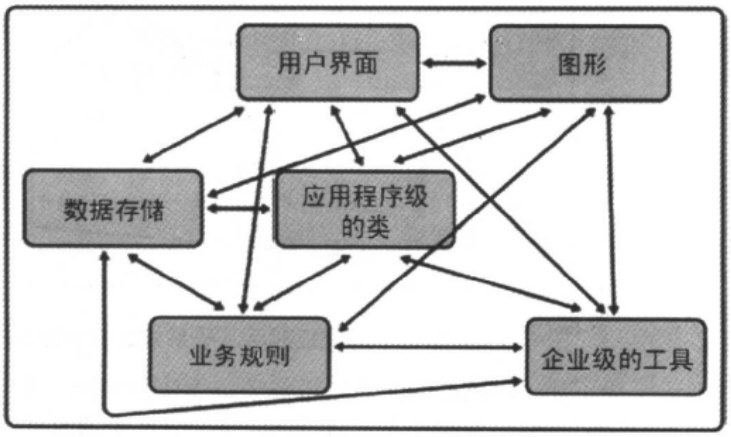


图 5-4　当子系统之间的通信没有任何限制时就会像这个样子

子系统，都需要修改，这样是不合理的。因此需要限制子系统之间的通信，如下图所示，为施加了少量通信规则后的系统，另外，为了让子系统之间的连接简单、易懂、且易于维护，就要尽量简化子系统之间的交互关系。最

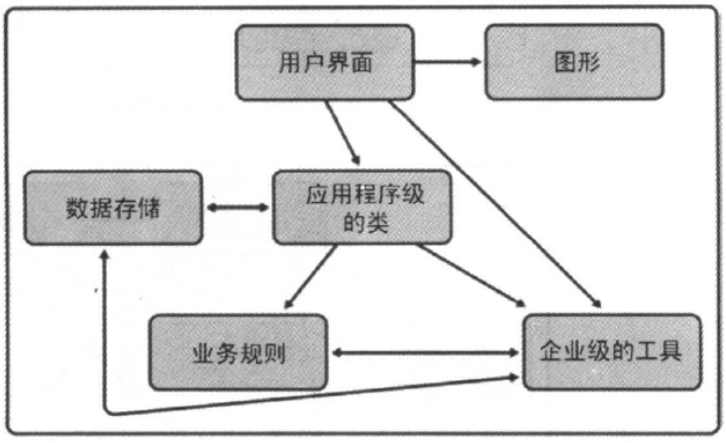


图 5-5　施加若干通信规则后，子系统之间的交互得以显著地简化

简单的交互关系，是让一个子系统，去调用另一个子系统中的子程序；稍微复杂一点的交互，是在一个子系统中，包含另一个子系统中的类；而最复杂的交互关系，是让一个子系统中的类，继承另一个子系统中的类。

设计子系统，有一条很好的基本原则，即系统层设计图，应该是无环图；亦即程序中不应该有任何环形关系，比如说 A 类使用了 B 类、B 类使用了 C 类、而 C 类又实用了 A 类这种情况。

有些种类的子系统，会在不同的系统中反复出现，例如：

- 业务规则：指那些在计算机系统中，编入的法律、规则、政策以及过程；
- 用户界面：应创建一个子系统，把用户界面组件，同其他部分分隔开，以便用户界面的演化不会破坏程序的其余部分；在大多数情况下，用户界面子系统会使用多个附属的子系统或类，来处理用户界面、命令行接口、菜单操作、窗体管理、帮助系统等等；
- 数据库访问：可以将对数据库访问的实现细节隐藏起来，让程序的绝大部分，可以不必关心处理底层结构的繁琐细节，并能像在业务层次一样处理数据；
- 对系统的依赖性：把对操作系统的依赖因素，归到一个子系统里，就如同把对硬件的依赖因素，封装起来一样。例如，开发的程序不仅能在 windows 上运行，也应该可以方便地移植到 linux 或 Mac OS 上，且只需要修改接口子系统就可以了。

- 第 3 层：分解为类。这一层的主要设计任务，是把所有的子系统，进行适当的分解，并确保分解出的细节都恰到好处，能够用单个的类实现。当定义子系统中的类时，也就同时定义了这些类与系统其余部分打交道的细节，尤其是要确定好类的接口。例如，数据库子系统可能会被进一步划分成数据库访问类、持久优化框架类、以及数据库元数据。

  类与对象的比较：面向对象设计的一个核心概念，就是对象 (object) 与类 (class) 的区分。对象是指运行期间，在程序中实际存在的具体实体，而类是指在程序源码中，存在的静态事物。对象是动态的，它拥有你在程序运行期间所能得到的具体的值和属性。例如，你可以定义一个名为 Person 的类，它具有姓名、年龄、性别等属性，在程序运行期间，你可以有 nancy、hank、tony 等对象，它们是类的具体实例。

- 第 4 层：分解成子程序。这一层的设计，包括把每个类细分为子程序（函数）。在第 3 层中，定义出类的接口，已经定义了其中一些子程序，而该层的设计，将细化出类的其他子程序。当你查看类里面子程序的细节时，就会发现很多子程序都很简单，但也有些子程序，是由更多层次的子程序所组成，这就需要更多的设计工作了。这一层次的分解和设计，通常是留给程序员个人来完成的。

- 第 5 层：子程序内部的设计。这里的设计工作，包括编写伪代码、选择算法、组织子程序内部的代码块，以及用何种编程语言编写代码。

5.3 启发式设计方法

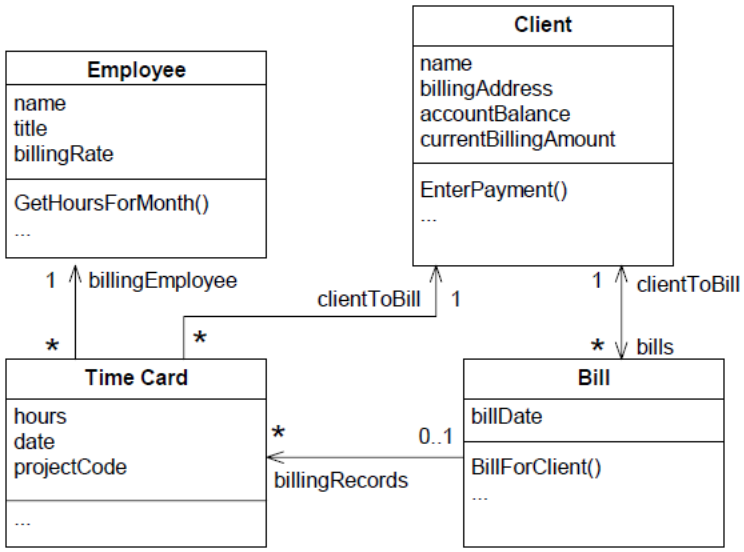
由于软件设计是非确定性的，因此，灵活熟练地运用一组有效的启发式方法，便成了合理的软件设计核心工作。

（1）找出现实世界中的对象：

在确定设计方案时，首选且最流行的是面向对象的设计方法。此方法的要点是辨明现实世界中的对象，以及人造的对象。使用对象设计的步骤：

- 辨识对象及其属性：计算机程序通常都是基于现实世界的实体。例如，如下图所示，可以基于现实世界中的雇员 (Employee)、顾客 (Client)、工作时间记录 (Timecard)、以及账单 (Bill) 等实体，来开发一套按时间计费的系统；辨识对象的属性，并不比辨识对象本身更困难。每个对象都有一些与计算机程序相关的特征。例如，在这个收费系统里，每个雇员对象都具有名字 (name)、职务 (title) 和费率 (billingRate) 等属性；而顾客对象则具有名字 (name)、账单寄送地址 (billingAddress)、以及账户余额 (accountBalance) 的属性；账单对象具有收费金额、顾客名字、支付日期 (billDate) 等等。

- 确定可以对各个对象进行的操作：在每个对象上，都可以执行多种操作；例如，雇员对象可能需要修改职务或者费率，顾客对象可能需要修改名字，或者账单寄送地址等等；



- 确定各个对象能对其他对象进行的操作：对象之间最常见的两种关系是包含和继承；一个 Timecard 对象可以包含一个 Employee 对象和一个 Client 对象，一个 Bill 对象可以包含一个或多个 Timecard 对象；另外，一份账单可以标示是否已经给某位顾客开过账单了，而顾客也可以签付一份账单；
- 确定对象的哪些部分，对其他对象可见：哪些部分是 public 的，哪些是 private 的；
- 定义每个对象的 public 接口：在编程语言的层次上，为每个对象定义具有正式语法的接口。对象对其他对象暴露的数据及方法，都被称为该对象的“public 接口”，而对象通过继承关系，向其派生对象暴露的部分，则被称为“protected 接口”。

经过上述这些步骤得到一个高层次的、面向对象的系统组织结构之后，你可以用这两种方法来迭代：在高层次的系统组织结构上进行迭代，以便更好地组织类的结构；或者在每个已经定义好的类上进行迭代，把每个类的设计详细化。

（2）形成一致的抽象：

抽象是一种能让你在专注某一概念的同时，可以放心地忽略其中一些细节的能力，即在不同的层次处理不同的细节。任何时候，当你在对一个聚合物品操作时，就是在用抽象了。例如，当你把一个东西称为“房子”，而不是由玻璃、木材和钉子构成的组合体时，就是在用抽像了。

基类也是一种抽像，它使你能集中精力关注一组派生类所具有的共同特性，并在基类的层次上，忽略各个具体派生类的细节；一个好的接口也是一种抽象，它能让你关注于接口本身，而不是类的内部工作方式。

如下图所示，抽象是我们用来处理现实世界复杂度的一种重要手段；软件开发人员有时就是在木材纤维、油漆分子，以及铁原子这一层来构建系统，因此就变得异常复杂，难以通过人的智力去管理；优秀的程序员会在子程序接口的层次上、在类接口层次上，以及包接口的层次上进行抽象，这样才能更快、更稳妥地进行开发。

（3）封装实现细节：

封装填补了抽象留下的空白；抽象是指，可以让你从高层的细节，来看待一个对象；而封装则是指，除此之外，你不能看到对象的任何其他细节层次。例如，如下图所示，封装是指，你可以从房屋的外面看，但不能靠得太近，去将门的细节都看清楚；可以让你知道哪里有门，门是开还是关，但不能让你知道门是木质的还是钢质的。

（4）当继承能简化设计时就继承：

在设计软件系统时，经常会发现一些大同小异的对象。例如，在一套账务系统中，包含全职员工和兼职员工，两者的大多数数据是相同的，只是某些数据不同。在面向对象编程时，可以定义一个代表普通员工的通用类型 (general)，然后把全职员工定义为普通员工，除了有一些不同之处；同样，把兼职员工也定义为普通员工，除了一些不同之处；当一项针对员工的操作，与具体的员工类别无关时，这一操作就可以针对通用员工类型来进行。当该操作需要区别全职员工和兼职员工时，就需要按照不同的方法来处理了。定义这种对象之间的相同点和不同点，就叫“继承”，因为全职员工和兼职员工，都从基本员工类型继承了某些特征。

继承的好处在于，它能很好地辅佐抽象的概念，并且能简化编程；因为你可以写一个基本的子程序，来处理只依赖于门的基本属性的事项，另外写一些特定的子程序，来处理依赖特定种类门的特定操作。例如，有些操作，如 Open() 或 Close()，对于任何种类的门都能用，无论是防盗门还是玻璃门；编程语言如果能支持像 Open() 或 Close() 这种，在运行期间才能确定所针对的对象的实际类型的操作，这种能力叫做“多态”。

（5）信息隐藏：

信息隐藏是降低软件复杂度的一种格外重要的启发式方法，因为它强调的就是隐藏复杂度。

- 隐私权：当信息被隐藏后，每个类或子程序都代表了，某种对其他类保密的设计或构建决策。隐藏起来的秘密，可能是某个易变的区域，或者某种文件格式，或某种数据类型的实现方式，或某个需要隔离的区域，在这个区域中发生的错误，不会给程序其余部分带来太大损失。在这里，类的职责就是把部分信息隐藏起来，并保护自己的隐私权。对系统的非重大改动，可能会影响到某个类中的几个子程序，但它们不应该波及到类接口的外面。

  在设计类的时候，一项关键的决策，就是确定类的哪些信息应该对外可见，而哪些信息应该隐藏起来。如下图所示，类的接口应该尽可能少地暴露其内部工作机制。设计类的接口与设计其他环节一样，都是一个迭代的过程；如果你第一次没有得到合适的接口，那么就多试几次，知道设计稳定下来；如果设计仍不稳定，那就需要换种方法再尝试。

- 信息隐藏的一个例子：假设你有一个程序，其中的每个对象，都是通过一个名为 id 的成员变量来保存一种唯一的 ID。一种设计方法，是用一个整数来表示 ID，同时用一个名为 g_maxId 的全局变量，来保存目前已分配的 ID 的最大值。每当创建新的对象时，你只要在该对象的构造函数里，简单地使用 id=++g_maxId 这条语句，就可以获得一个唯一的 ID 值，这种做法会让对象在创建时，执行的代码量最少。可这样设计可能会出错：如果你像把某些范围的 ID 留作它用该怎么办？如果想用非连续 ID 来提高安全性又该怎么办？如果你想重新使用已销毁对象的 ID 呢？如果你想增加一个断言，来确保所分配的 ID 不会超过预期的最大范围呢？如果程序中到处都是 id=++g_maxId 这种语句，一旦上面说的任何一种情况出现，就需要修改所有这些语句。另外如果程序是多线程的，这种方法也不是线程安全的。

图 5-9　好的类接口就像是冰山的尖儿一样，让类的大部分内容都不会暴露出来

创建新 ID 的方法就是一种你应该隐藏信息的设计决策。如果你在程序中到处使用 ++g_maxId 的话，就暴露了创建新 ID 的方法，即通过简单递增的方式；想法，如果你在程序中，使用语句 id=NewId()，那就把创建新 ID 的方法隐藏起来了。你可以在 NewId() 子程序中仍然只用一行代码，return (++g_maxId)，或者其他与之等价的方法。如果想修改，只需修改 NewId() 即可。

现假设需要把 ID 的类型由 int 改为字符串，如果在程序中大量使用了针对 int 的操作，例如 >、<、= 等等，这些操作并不适用字符串，那么即使改用 NewId() 子程序，也无济于事。因此，另一个需要隐藏的信息，就是 ID 的类型。在 C++ 里，可以简单地使用 typedef 来把 ID 定义为 IdType，也可以创建一个简单的 IdType 类。

隐藏设计决策，对于减少“改动所影响的代码量”，是至关重要的。信息隐藏在设计的所有层次上，都有很大作用，从使用具名常量替代字面量，到创建数据类型，再到类的设计、子程序的设计以及子系统的设计等等。

- 两种信息：信息隐藏中所说的信息主要分为两大类，复杂度和变化源；
- 信息隐藏的障碍：
  - 信息过度分散。例如，将 100 这个数字直接写到程序各个地方，会导致对它的引用过度分散；最好将它写入 MAX_EMPLOYEES 的常量中，如果需要改动，只需要改动一处即可；
  - 循环依赖。例如，A 类中的子程序，调用了 B 类的中的子程序；然后 B 类中的子程序，又调用 A 类中的子程序；
  - 将类内数据误认为全局数据。为了避免全局数据可能带来的问题，将类内数据误认为全局数据，并避免使用它。全局数据通常会受困于两类问题：一种是子程序在全局数据上执行操作，却不知道还有其他子程序也在用这些全局数据进行操作；另一种是子程序知道其他子程序也在用全局数据进行操作，但却无法明确地知道都进行了哪些操作。而类内数据就不会有这种问题，因为只有类内部的少数子程序才能直接访问这些数据。这些子程序不但知道有其他子程序在操纵这些数据，而且也明确知道具体是哪些子程序在执行这些操作。但如果设计的类包含很多体积庞大的众多子程序，那么类数据和全局数据之间的区别就变得模糊起来，类内数据也将开始受困于全局数据所面临的那些问题了。
  - 可以察觉的性能损耗。如果在架构层按照信息隐藏的目标去设计系统，并不会与按照性能目标去设计想冲突，因此在系统架构层和编码层均避免性能上的损耗。
- 信息隐藏的价值：运用了信息隐藏技术的大型项目，与没有应用这一技术的项目，修改起来大约容易 4 倍；而且信息隐藏还是结构化程序设计和面向对象设计的根基之一。

（6）找出容易改变的区域：

程序设计面临的最重要挑战之一，就是适应变化。需要将不稳定的区域隔离出来，从而把变化所带来的影响，限制在一个子程序、类或包的内部。可采取的应对各种变动的措施：

- 找出看起来容易变化的项目。如下是一些容易发生变化的地方：

- 业务规则：业务规则很容易成为软件频繁变化的根源。国会改变了税率结构，保险公司改变了它的税率表等等；如果你遵循信息隐藏的原则，那么基于这些业务规则的逻辑，就不应该遍布于整个程序，而是仅仅隐藏在系统的某个角落，知道需要对它进行改动，才会把它拎出来；
- 对硬件的依赖：与屏幕、键盘、鼠标设施以及通信设计等之间的接口，都是硬件依赖的例子。请把对硬件的依赖，隔离在它们自身的子系统或类中。这种隔离非常有利于把你的程序移植到新的硬件环境下。另外，当你为可能变化的硬件开发程序时，这样做也会有很大帮助。你可以写软件来模拟与特定硬件的交互，在硬件尚不稳定，或者不可用的时候，让硬件接口子系统使用该模拟器，当硬件可用的时候，把硬件接口子系统与模拟器切断，最终连接到真正的硬件设备上；
- 输入和输出：在做比纯硬件接口层稍高一些层面上的设计时，输入输出也是一个容易变化的区域。如果你的程序创建了自己的数据文件，那么该文件格式就可能会随软件开发的不断深化而变化。用户层的输入和输出格式也会变化：输出页面上字段位置、数量和排列顺序等都可能会变。因此，检查所有的外部接口，看看有哪些可能的变化，通常是个不错的主意；
- 非标准的语言特性：大多数编程语言的实现中，都包含了一些便利的、非标准的扩展。这些扩展可能在其他的环境中不可用；因此需要将这样的扩展单独隐藏在某个类里，以便当你转移到新的环境后，可以用自己写的代码区取代。与此类似，如果你使用了并非所有环境中都可用的函数库，请把这些子程序库隐藏在一个接口的后面，为新环境做好准备；
- 困难的设计区域和构建区域：将困难的设计区域和构建区域隐藏起来，也是很好的想法，因为这些代码可能因为设计得很差，而需要重新做；
- 状态变量：状态变量用于表示程序的状态，与大多数其他的数据相比，这种东西更容易改变；在一个典型的应用场景里，你可能一开始用布尔变量，来定义出错状态，然后又发现用具有 ErrorType_None、ErrorType_Warning 和 ErrorType_Fatal 等值的枚举类型，来表示该状态更好。可以在使用状态变量时，增加至少两层的灵活性和可读性：
  - 不要使用布尔变量作为状态变量，而是用枚举类型；
  - 使用访问器子程序检查状态变量，而不是直接检测；
- 隐藏数据量：例如用具名常量 MAX_EMPLOYEES 来隐藏 100 这样的数字。

- 把容易变化的项目分离出来。将容易变化的组件，单独划分成类，或者和其他容易同时发生变化的组件，分到同一个类中。

- 把容易变化的项目隔离开来。设计类之间的接口，使其对潜在的变化不敏感。

找出容易发生变化区域的一个好办法：首先找出程序中可能对用户有用的最小子集。这一子集构成了系统的核心，不容易发生变化。接下来，用微小的步伐扩充这个系统。这里的增量可以非常微小，小到看似微不足道。当你考虑功能上的改变时，同时也要考虑质的变化：比如让程序变成线程安全，使程序能够本地化等。这些潜在的改进区域，就构成了系统中的潜在变化。请依照信息隐藏的原则，来设计这些区域。通过首先定义清楚核心，你可以认清哪些组件属于附加功能，这是就可以把它们提取出来，并把它们的可能改进隐藏起来。

（7）保持松散耦合：

耦合度表示类与类之间，或子程序与子程序之间的关系紧密程度。耦合度设计的目标，是创建出小的、直接的、清晰的类或子程序，使它们与其他类或子程序之间，关系尽可能地灵活，这就被称作“松散耦合”。模块之间良好的耦合关系，需要松散到恰好能使一个模块，能够很容易地被其他模块使用，确保模块之间的连接关系尽可能的简单。尽量使创建的模块，不依赖或很少依赖其他模块。例如 sin() 这样的子程序是松耦合的，因为它需要知道的东西，也就是一个传入的、代表角度的数值。而像 InitVars(var1,...,varN) 这样的子程序，则耦合得过于紧密，因为在调用端，必须传入各个参数，调用它的模块实际上知道在 InitVars() 内部会做些什么。如果两个类都依赖于同一个全局变量的使用，那么它们之间的耦合关系就更紧密了。

- 耦合标准：下面是一些在衡量模块之间耦合度使，可采用的标准
  - 规模。指模块之间的连接数。只有 1 个参数的子程序，相比有 6 个参数的子程序，耦合度更高；

– 可见性。指两个模块之间，连接的显著程度。在程序开发过程中，需要把模块之间的连接关系，变得广为人知而获取信任。通过参数表传递数据，则是一种明显连接，值得提倡；而通过修改全局数据，进而使另一个模块能使用该数据，则是一种不好的设计；

– 灵活性。指模块之间的连接，是否容易改动。一个模块越容易被其他模块调用，那么它们之间的耦合关系，就会越松散，并且更易于维护。因此，在创建系统架构时，应按照“尽可能缩减相互连接”的准则，来分解程序。

- 耦合种类：几种常见的几种耦合，

– 简单数据类型的参数耦合。当两个模块之间，通过参数来传递数据，并且所有的数据都是简单数据类型时，这两个模块之间的耦合关系，就是简单数据参数耦合的。这种耦合关系是正常且可以接受的。

– 简单对象耦合。如果一个模块实例化一个对象，那么它们之间的耦合关系，就是简单对象耦合的。这种耦合关系也很不错。

– 对象参数耦合。如果 Object1 要求 Object2 传给它一个 Object3，那么这两个模块就是对象参数耦合的。与 Object1 仅要求 Object2 传递给他简单数据类型相比，这种耦合关系要更紧密些，因为它要求 Object2 了解 Object3。

– 语义上的耦合。最难缠的耦合关系是这样发生的：一个模块不仅使用了另一个模块的语法元素，而且还使用了有关那个模块内部工作细节的语义知识。例如 Module1 向 Module2 传递了一个控制标志，用它告诉 Module2 该做什么，这种方法要求 Module1 对 Module2 的内部工作细节有所了解，也就是说需要了解 Module2 对控制标志的使用。语义上的耦合是非常危险的，因为更改被调用模块中的代码，可能会破坏调用它的模块，破坏的方式是编译器完全无法检查的。类似这样的代码崩溃时，其方式是非常微妙的，看起来与被使用的模块中的代码更改毫无关系，因此会使得调试工作变得无比困难。

松散耦合的关键之处在于，一个有效的模块提供了一层附加的抽象：一旦你写好了它，就可以想当然地去用它。这样就降低了整体系统的复杂度，使得你可以在同一时间，只关注一件事。如果对一个模块的使用，要求你同时关注好几件事：其内部工作的细节、对全局数据的修改、不确定的功能点等；那么就失去了抽象的能力，模块所具有的管理复杂度的能力也就削弱或完全丧失了。

（8）查阅常用的设计模式：

设计模式精炼了众多现成的解决方案，可以解决很多软件开发中，最常见的问题。有些软件问题要求全新的解决方案，但是大多数问题都和过去遇到的问题类似，因此可以使用类似的解决方案或者模式加以解决。下表为常见的设计模式：

表 5-1　常见设计模式

模　式	描　述
Abstract Factory（抽象工厂）	通过指定对象组的种类而非单个对象的类型来支持创建一组相关的对象
Adapter（适配器）	把一个类的接口转变成为另一个接口
Bridge（桥接）	把接口和实现分离开来，使它们可以独立地变化
Composite（组合）	创建一个包含其他同类对象的对象，使得客户代码可以与最上层对象交互而无须考虑所有的细节对象
Decrorator（装饰器）	给一个对象动态地添加职责，而无须为了每一种可能的职责配置情况去创建特定的子类（派生类）
Facade（外观）	为没有提供一致接口的代码提供一个一致的接口
Factory Method	做特定基类的派生类的实例化时，除了在 Factory Method 内部之外均无须了解各派生对象的具体类型
Iterator（迭代器）	提供一个服务对象来顺序地访问一组元素中的各个元素
Observer（观察者）	使一组相关对象相互同步，方法是让另一个对象负责：在这组对象中的任何一个发生改变时，由它把这种变化通知给这个组里的所有对象
Singleton（单件）	为有且仅有一个实例的类提供一种全局访问功能
Strategy（策略）	定义一组算法或者行为，使得它们可以动态地相互替换
Template Method（模板方法）	定义一个操作的算法结构，但是把部分实现的细节留给子类（派生类）

与完全定制的设计方案相比，设计模式提供了下列好处：

- 设计模式通过提供现成的抽象，来减少复杂度；
- 设计模式通过把常见解决方案的细节，通过制度化来减少出错；
- 设计模式通过提供多种设计方案，带来启发性的价值；
- 设计模式通过把设计对话，提升到一个更高的层次上，来简化交流。

应用设计模式的一个潜在陷阱，是强迫让代码适用于某个模式。有时候，对代码进行一些微小的更改，以便符合某个广为人知的模式，会使这段代码更容易理解。但是，如果一段代码做出巨大改动，迫使它去符合某个标准设计模式，有时反而会把问题复杂化。

（9）使用启发式方式的原则：

最有效的原则之一，就是不要卡在单一的方法上。如果用 UML 画设计图不可行，那么就直接用英语写；写段简短的测试程序；尝试一种截然不同的方法；用铅笔画出轮廓和草图来指导思维等等。

你无须马上解决整个设计难题。一旦被卡住了，那么请记住回过头来时，有一处地方需要做决策，但眼下你还没有足够的信息来解决这个问题。如果你尝试了一些设计方案，但没有很好的解决问题的时候，更自然的方式，是让那些问题留在未解决的状态，等拥有更多信息后，在去做。

5.4 设计实践

在设计过程中，可以采用如下一些工作步骤，以便获得良好的设计结果：

（1）迭代

设计是一个迭代的过程。当在备选的方案之中，循环并尝试一些不同的做法时，将会同时从高层和低层的不同视角，去审视问题。从高层视角从得到的大范围图景，有助于你把相关的低层细节纳入考虑；从低层视角中获得的细节，也会为你的高层决策奠定基础。这种高低层面之间的互动，被认为是一种良性的原动力，它所创建的结构，要远远稳定于单纯自上而下，或自下而上创建的结构。

（2）分而　之

没有人的头脑能装下一个复杂程序的全部细节，对设计同样适用。将程序分解为不同的关注区域，然后分别处理每一个区域；如果在某个区域里碰上了死胡同，那么就迭代。

（3）自上而下和自下而上

- 自上而下的设计：是指从某个很高的抽象层次开始，定义出基类或其他不那么特殊的设计元素；在开发这一设计的过程中，逐渐增加细节的层次，找出派生类、合作类，以及其他更细节的设计元素。这种设计方式可以看作是一层层分解的过程，在分解过程的不同阶段，需要选择用什么方法，去分解子系统，给出继承关系树，形成对象的组合。持续分解，直到在下一层，直接编码比分解更容易。
- 自下而上的设计：是指设计始于细节，向一般性延伸；这种设计通常是从寻找具体对象开始，最后从细节之中生成对象以及基类。需要考虑的一些步骤：
  - 对系统需要做的事项，有哪些已知条件；
  - 找出具体的对象和职责；
  - 找出通用的对象，把它们按照适当方式组织起来：子系统、包、对象组合，或者继承；看哪种方式合适；
  - 在更上一层继续设计，或者回到最上层，尝试向下设计
- 两者并不矛盾：两种策略最关键的区别在于，自上而下是一种分解策略；而自下而上是一种合成策略；前者从一般性的问题出发，把该问题分解成可控的部分；后者从可控的部分出发，去构造一个通用的方法。

（4）建立试验性原型

有些时候，除非能很好的了解实现细节，否则很难判断一种设计方法是否凑效。例如，在知道它能满足性能要求之前，很难判断某种数据库的组织结构是否适用。此时，建立试验性原型，便能低成本解决这些问题：写出用于回答特定设计问题的、量少且能够随时扔掉的代码。但是使用试验原型，存在以下一些风险：

- 开发人员没有遵循“用最少代码回答提问”的原则。例如，设计问题是“我们选的的数据库框架，能否支撑所需的交易量？”，你不需要为了这一问题，而编写任何产品代码，也不需要去了解数据库的详情；只需要了解能估计出问题范围的最少信息：有多少张表、表中有多少条记录等等，接下来就可以用 Table1、Table2、Column1、Column2 等名字，写出最简单的原型代码，往表里随意填入些数据，然后做你所需要的性能测试；

- 设计的问题不够特殊。例如，设计问题“这样的数据库框架，能否工作？”，并没有为建立原型提供多少指引；而像“这个数据库框架能不能在 X、Y 和 Z 的前提下，支持每秒 1000 次交易？”这样的问题，则能为建立原型，提供更坚实的基础；

- 开发人员不把原型代码当作可抛弃的代码。如果开发人员相信某段代码，将被用在最终产品里，那么他根本不可能写出最少数量的代码来。避免产生这一问题的一种做法，是用与产品代码不同的技术，来开发原型。例如用 Python 来为 C++ 设计做原型。

（5）合作设计

无论组织形式的正式与否，在设计过程中，三个臭皮匠顶得上一个诸葛亮，合作可以以任意方式展开。例如，随便走到一名同事办公桌前，向他征求一些想法；结对编程等等。

（6）做多少设计才足够

对于实施正式编码前的设计工作量和设计文档的正规程度，很难有个确定的定论，下表可以做个参考，

表 5-2　设计文档的正规化以及所需的细节层次

因素	开始构建之前的设计所需的细化程度	文档正规程度
设计或构建团队在应用程序领域有很丰富的经验	低	低
设计或构建团队有很丰富的经验，但是在这个应用程序领域缺乏经验	中	中
设计或构建团队缺乏经验	中到高	低到中
设计或构建团队人员变动适中或者较高	中	—
应用程序是安全攸关的	高	高
应用程序是使命攸关的	中	中到高
项目是小型的	低	低
项目是大型的	中	中
软件预期的生命周期很短（几星期或者几个月）	低	低
软件预期的生命周期很长（几个月或者几年）	中	中

如果在编码前，还没法判断是否应该做更多深入设计，那么宁愿去做更详细的设计。最大的设计失误，来自于误认为自己已经做得很充分，可事后却发现还是做得不够。另一方面，也不要太过于专注，对设计进行文档化，而导致失败；程序化的活动，容易把非程序化的活动驱逐出去，过早地去润色设计方案，就是所描述的例子。

（7）记录设计成果

传统的记录设计成果的方式，是把它写成正式的设计文档；然而，你还可以用很多种方法来记录设计成果，这些方法对于那些小型的、非正式项目，或者只需要轻量级地记录设计成果的项目，效果是很不错的：

- 把设计文档插入到代码里：在代码注释中写明关键的设计决策，通常放在文件或类的开始位置；

- 用 Wiki 来记录设计讨论和决策；

- 写总结邮件；

- 使用数码相机；

- 保留设计挂图；

- 使用 CRC 卡片 (类、职责、合作者)；
- 在适当的细节层，创建 UML 图。

6 可以工作的类

类是由一组数据和子程序构成的集合，这些数据和子程序共同拥有一组内聚的、明确定义的职责。类也可以只是由一组子程序构成的集合，这些子程序提供一组内聚的服务，哪怕其中未涉及共用的数据。成为高效程序员的一个关键，在于当你开发程序任一部分的代码时，都能安全地忽视程序中尽可能多的其余部分。而类就是实现这一目标的首要工具。

6.1 类的基础：抽象数据类型

抽象数据类型（ADT,abstract data type）是指一些数据，以及对这些数据所进行的操作的集合。这些操作，既是像程序的其余部分描述了这些数据是怎么样的，也允许程序的其余部分改变这些数据。一个 ADT 可能是一个图形窗体，以及所有能影响该窗体的操作；也可以是一个文件，以及对这个文件进行的操作等等。

（1）一个 ADT 例子：

假如你正在写一个程序，控制文本的字体，它能用不用的字型、字号等；如果用一个 ADT，就能在相关数据上，捆绑一组操作字体的子程序；相关的数据包括字体名称、字号和文字属性等。这些子程序和数据集合，就是一个 ADT。

如果不使用 ADT，就只能用一种拼凑的方法来操纵字体了。例如，如果要将字体大小改为 12 磅 (point)，即高度为 16 像素 (pixel)，那么就需要这样的代码：

currentFont.sizeOnPixels = PointsToPixels(12)

但不能同时使用 currentFont.sizeInPixels 和 currentFont.sizeInPoints；因为如果你同时使用这两项数据成员，currentFont 就无从判断到底该用哪个。而且，如果你在程序很多地方都需要修改字体大小，那么这类语句就会散步在整个程序中。如果需要把字体设为粗体，或许需要写成这样：

currentFont.attribute = CurrentFont.attribute or 0x02 或

currentFont.bold = True

这些做法都存在一个限制，即要求调用方代码，直接控制数据成员，这无疑限制了 currentFont 的使用。

（2）使用 ADT 的好处

- 可以隐藏实现细节：把信息隐藏起来，能保护程序的其余部分不受影响。
- 改动不会影响到整个程序：如果数据类型改变，只需在一处修改而不会影响到整个程序。
- 让接口能提供更多信息：把所有相似的操作，都集中到一个 ADT 中，就可以基于磅数或像素来定义整个接口，或者把二者明确区分开，从而有助于避免混淆。
- 更容易提高性能：如果想提高操作字体时的性能，就可以重新编写出一些更好的子程序，而不用来回修改整个程序。
- 让程序的正确性更显而易见：验证像 currentFont.attribute = CurrentFont.attribute or 0x02 的语句是否正确，是很枯燥的，可以替换成像 currentFont.SetBoldOn() 这样的语句，验证它是否正确就更容易一些。
- 程序更具自我说明性：可以改进 currentFont.attribute = CurrentFont.attribute or 0x02 这样的语句：将 0x02 换成 BOLD，但无论怎么杨修，其可读性都不如 currentFont.SetBoldOn() 这条语句。
- 无须在程序内到处传递数据：在上面的例子中，必须直接修改 currentFont 的值，或把它传给每一个要操作字体的子程序；如果使用了 ADT，就不用再在程序里到处传递 currentFont 了。
- 可以像在现实世界中那样操作实体，而不用在底层实现上操作它：可以定义一些针对字体的操作，这样程序的绝大部分，就能完全以“真实世界中的字体”这个概念来操作，而不再用数组访问、结构体定义、True 与 False 等这些底层的实现概念了。

为了定义一个 ADT，只需要定义一些用来控制字体的子程序，例如：

currentFont.SetSizeInPoints(sizeInPoints)
currentFont.SetSizeInPixels(sizeInPixels)
currentFont.SetBoldOn()
currentFont.SetBoldOff()

（3）使用 ADT 的一些建议：

- 把常见的底层数据类型构建为 ADT，并使用这些 ADT，而不再使用底层数据类型：堆栈、列表、队列，以及几乎所有常见的底层数据类型，都可以用 ADT 表示；如果堆栈代表的是一组员工，就该把它看作是一些员工，而不是堆栈；如果列表代表一个出场演员名单，就该把它看作是出场演员名单，而不是列表等等。
- 把像文件这样的常用对象当成 ADT。
- 简单的事物也可当作 ADT：例如，一盏灯只有开和关两种操作，也可以放到 ADT 里，这样可以提高代码的自我说明能力，并减少需要到处传递的数据。
- 不要让 ADT 依赖于存储介质：假设有一张保险费率表，它太大了，因此只能保存在磁盘上；你可能编写出 rateFile.Read() 这样的访问器子程序，然而当你把它当作一个“文件”时，就已经暴露了过多的数据信息，一旦对程序进行修改，把这张表存到内存中，而不是磁盘上，把它当作文件的那些代码将不正确，而且产生误导并使人迷惑。因此，请尽量让类和访问器子程序的名字，与存储数据的方式无关，并只提及抽象数据类型本身，例如 rateTable.Read() 或更简单 rates.Read()。

6.2 良好的类接口

创建高质量的类，第一步，可能也是最重要的一步，就是创建好的接口。这也包括了创建一个可以通过接口来展现的合理抽象，并确保细节仍被隐藏在抽象背后。

（1）好的抽象：

抽象是一种以简化的形式，来看待复杂操作的能力。类的接口为隐藏在其后的具体实现，提供了一种抽象。类的接口应能提供一组明显相关的子程序。

假设有一个实现雇员 (Employee) 这一实体的类，其中可能包含雇员的姓名、地址、电话号码等数据，以及一些用来初始化并使用雇员的服务子程序，例如：

C++示例：展现良好抽象的类接口

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

```
    String GetHomePhone() const;
    TaxId GetTaxIdNumber() const;
    JobClassification GetJobClassification() const;
    ...
private:
    ...
};
```

在类的内部，还可能会有支持这些服务的其他子程序和数据，但类的使用者，并不需要了解它们；类接口的抽象能力非常有价值，因为接口中的每个子程序，都在朝这个一致的目标而工作。

一个没有经过良好抽象的类，可能会包含有大量混杂的函数，例如

C++示例：展现不良抽象的类接口

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

其中有很多子程序，有用来操作命令栈的，有用来格式化报表的，有用来打印报表的，还有用来初始化全局数据的。在命令栈、报表和全局数据之间，很难看出什么联系。类的接口不能展现出一种一致的抽象，因此它的内聚性就很弱；应该把这些子程序，重新组织到几个职能更专一的类里去，在这些类的接口中，提供更好的抽象。

如果这些子程序是一个叫做 Program 类的一部分，那么可以这样来修改它，以提供一种一致的抽象：

C++示例：能更好展现抽象的类接口

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

在清理这一接口时，将原有的一些子程序，转移到其他更适合的类里面，而把另一些转为 InitializeUserInterface() 和其他子程序中使用的私有子程序。

这种对类的抽象进行评估的方法，是基于类所具有的 public 子程序所构成的集合，即类的接口。即使类的整体表现出一种良好的抽象，类内部的子程序也未必都能表现出良好的抽象，也同样要把它们设计得可以表现出很好的抽象。

一些创建类的抽象接口的指导建议：

- 类的接口应该展现一致的抽象层次：在设计类的时候，有一种很好的方法，就是把类看作一种用来实现 ADT 的机制；每一个类应该实现一个 ADT，并且仅实现这一个 ADT。如果你返现某个类实现了不止一个 ADT，或者你不能确定究竟它实现了何种 ADT，你就应该把这个类，重新组织为一个或多个定义更加明确的 ADT。

C++示例：混合了不同层次抽象的类接口

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

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

这个类展现了两个 ADT：Employee 和 ListContainer。出现这种混合的抽象，通常是源于程序员使用容器类或其他类库来实现内部逻辑，但却没有把“使用类库”这一事实隐藏起来。下面是隐藏了实现细节的类接口：

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

- 一定要理解类所实现的抽线是什么：一些类非常像，必须非常仔细地理解类的接口，应该捕捉的抽象到底是哪一个。当你不得不在两个相似的抽象之间做出选择时，请确保你的选择是正确的。
- 提供成对的服务：大多数操作都有和其相应的、相等的以及相反的操作。如果有一个操作用来把灯打开，那很有可能也需要另一个操作，来把灯关闭。在设计类的时候，需要检查每一个 public 子程序，决定是否需要另一个与其互补的操作。不要盲目地创建相反操作，但你一定要考虑，看看是否需要它。

- 把不相关的信息转移到其他类中：有时你会发现，某个类中，一般子程序使用着该类的一半数据，而另一半子程序则使用另一半数据。这时你其实已经把两个类混在一起使用了，这就需要把它们拆开。

- 尽可能让接口可编程，而不是表达语义：每个接口都由一个可编程的部分和一个语义部分组成。可编程的部分，由接口中的数据类型和其他属性构成，编译器在检查编译错误时，能强制性地要求它们。而语义部分则由“本接口将会被怎样使用”的假定组成，而这些是无法通过编译器来强制实施的。例如，语义接口中包含的考虑“RoutineA 必须在 RoutineB 之前被调用”。语义接口应通过注释说明，但要尽可能让接口不依赖于这些说明。一个接口中任何无法通过编译器强制实施的部分，就是一个可能被误用的部分。要想办法把语义接口的元素，转换为编程接口的元素，比如用断言或其他的技术。

- 谨防在修改时破坏接口的抽象：在对类进行修改和扩展的过程中，常常会发现额外所需的一些功能；这些功能并不十分适应于原有的类接口，可看上去却很难用另一种方法实现。例如，上面的 Employee 类演变成了下面这个样子：

C++语言示例：在维护时被破坏的类接口

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

前面代码示例中的清晰抽线，现在已经变成了，由一些零散功能组成的大杂烩。在雇员和检查邮政编码、电话号码或职位的子程序之间，并不存在什么逻辑上的关联，那些暴露 SQL 语句查询细节的子程序，所处的抽象层次比 Employee 类也要低得多，它们都破坏了 Employee 类的抽象。

- 不要添加与接口抽象不一致的公用成员：每次你向类的接口中，添加子程序时，问问“这个子程序与现有接口所提供的抽象一致吗”，如果发现不一致，就要换另一种方法来进行修改，以便能保持抽象的完整性。

- 同时考虑抽象性和内聚性：抽象性和内聚性这两个概念之间的关系，非常紧密：一个呈现出很好的抽象的类接口，通常也有很高的内聚性。而具有很强内聚性的类，往往也会呈现为很好的抽象。关注类的接口所表现出来的抽象，比关注类的内聚性更有助于深入地理解类的设计。如果你发现某个类的内聚性很弱，也不知道该怎么改，那就换一种方法，问问你自己这个类是否表现为一致的抽象。

（2）良好的封装

封装是一个比抽象更强的概念。抽象通过提供一个，可以让你忽略实现细节的模型，来管理复杂度；而封装则强制阻止你看到细节，即便你想这么做。这两个概念之所以相关，是因为没有封装时，抽象往往很容易被打破。依据经验，要么就是封装与抽象两者皆有，要么就是两者皆失。除此之外，没有其他可能。

- 尽可能地限制类和成员的可访问性：让可访问性尽可能低，是促成封装的原则之一。当你在犹豫某个子程序的可访问性，应该设为 public、private 还是 protected 时，经验之举应该采用最严格且可行的访问级别。这是一个很好的指导建议，但还有更重要的建议，即考虑“采用哪种方式，最好地保护接口抽象的完整性?”，如果暴露一个子程序不会让抽象变得不一致的话，这么做就很可能是可行的。如果不确定，那么多隐藏，通常比少隐藏要好。

- 不要公开暴露成员数据：暴露成员数据会破坏封装性，从而限制你对这个抽象的控制能力。例如，一个 Point 类如果暴露了下面这些成员的话：

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

它就破坏了封装性，因为调用该类可以自由地使用 Point 类里面的数据，而 Point 类却连这些数据什么时候被改动过都不知道。然而，如果 Point 类暴露的是这些方法的话：

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

那它还是封装完好的。你无法得知底层实现用的是不是 float x、y、z，也不会知道 Point 是不是把这些数据保存为 double 然后再转换成 float，也不可能知道 Point 是不是把它们保存在月亮上，然后再从外层空间中的卫星上把它们找回来。

- 避免把私有的实现细节放入到类的接口中：做到真正的封装后，程序员们是根本看不到任何实现细节的。无论是在字面上还是在喻意，它们都被隐藏了起来。然而，包括 C++ 在内的一些流行编程语言，却从结构上，要求程序员在类的接口中透露实现细节。例如，

C++示例：暴露类内部实现细节

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

```
    ...
};
```

其中，private 中的私有成员变量，暴露了实现细节。将 private 段的声明放在类的头文件中，看上去似乎只是小小地违背了原则，但它实际上是在鼓励程序员们查阅实现细节。在这个例子中，客户代码本意是要使用 Address 类型来表示地址信息，但头文件中却把“地址信息用 String 来保存”，这一实现细节暴露了出来。为解决这一问题，建议将类地接口与类的实现隔离开，并在类的声明中包含一个指针，让该指针指向类的实现，但不能包含任何其他实现细节。

C++示例：隐藏类的实现细节

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

现在你就可以把实现细节，放到 EmployeeImplementation 类里了，这个类只对 Employee 类可见，而对使用 Employee 类的代码来说，则是不可见的。

- 不要对类的使用者做任何假设：类的设计和实现，应该符合类的接口中所隐含的契约。它不应该对接口会被如何使用，或不会被如何使用，做出任何假设；除非在接口中有明确说明。例如下面这样一段注释，就显示出这个类，过多地假定了它的使用者：

  请将x、y和z初始化为1.0，因为如果把它们
  初始化为0.0，DerivedClass就会崩溃。

- 避免使用友元类（friend class）：友元类会破坏封装，因为它让你在同一时刻，需要考虑更多的代码量，从而增加了复杂度。

- 不要因为一个子程序里，仅使用公用子程序，就把它归入到公开接口。

- 让阅读代码比编写代码更方便：为了让编写代码更方便，而降低代码的可读性，是非常不经济的。尤其是在创建类的接口时，即使某个子程序与接口的抽象不是很相配，有时人们也往往把这个子程序加到接口里，从而让正开发的这个类的某处调用代码，能更方便地使用它。然而，这段子程序的添加，正时代码走下坡路的开始。

- 要格外警惕从语义上破坏封装性：比较起来，语义上的封装和语法上的封装，二者的难度相差无几。从语法的角度说，要想避免窥探另一个类的内部实现细节，只要把它内部的子程序和数据，都声明为 private 就可以了，这是相对容易办到的。然而，要想达到语义上的封装，就完全时另一回事了。下面一些类的调用方代码例子，从语义上破坏了其封装性：

  - 不去调用 A 类的 InitializeOperations() 子程序，因为你知道 A 类的 PerformFirstOperation() 子程序会自动调用它。
  - 不在调用 employee.Retrivae(database) 之前调用 database.Connect() 子程序，因为你知道在未建立数据库连接的时候，employee.Retrieve() 会去连接数据库的。
  - 不去调用 A 类的 Terminate() 子程序，因为你知道 A 类的 PerformFinalOperation() 子程序已经调用它了。

  - 即便在 ObjectA 离开作用域之后，你仍然去使用由 ObjectA 创建的、指向 ObjectB 的指针或引用，因为你知道 ObjectA 把 ObjectB 放置在静态存储空间中了，因此 ObjectB 肯定还可以用。
  - 使用 ClassB.MAXIMUM_ELEMENTS 而不用 ClassA.MAXIMUM_ELEMENTS，因为你知道它们两个的值是相等的。

  上面这些例子的问题都在于，它们让调用方代码不是依赖类的公开接口，而是依赖类的私用实现。每当你发现自己是通过查看类的内部实现，来得知如何使用这个类的时候，你就不是在针对接口编程了，而是透过接口针对内部实现编程了。如果你透过接口来编程的话，封装性就被破坏了，而一旦封装性开始遭到破坏，抽象能力就快遭殃了。如果仅仅根据类的接口文档，还是无法得知如何使用一个类的话，正确的做法不是拉出这个类的源代码，从中查看其内部实现，而是应该联系类的作者，告诉他“我不知道该怎么用这个类”。而对于类的作者来说，正确的做法，不是面对面地告诉你答案，而是从代码库中 check out 类地接口文件，修改类地接口文档，再把文件 check in 回去，然后告诉你“看看现在你知不知道该怎么用它了”。

- 留意过于紧密的耦合关系：耦合是指两个类之间关联的紧密程度。通常这种关联越松越好。一些建议：
  - 尽可能地限制类和成员地可访问性。
  - 避免友元类，因为它们之间是紧密耦合地。
  - 在基类中把数据声明为 private，而不是 protected，以降低派生类和基类之间耦合地程度。
  - 避免在类地公开接口中，暴露成员数据。
  - 要对从语义上破坏封装性保持警惕。
  - 察觉“Demeter 法则”。

  耦合性与抽象和封装性，有着非常紧密的联系。紧密的耦合性，总是发生在抽象不严谨，或封装性遭到破坏的时候。如果一个类提供了一套不完整的服务，其他的子程序就可能要去直接读写该类的内部数据。这样一来，就把类给拆开了，把它从一个黑盒子，变成了一个玻璃盒子，从而事实上消除了类的封装性。

6.3 有关设计和实现问题

给类定义合理的接口，对于创建高质量程序，起到了关键作用。然而，类内部的设计和实现也同样重要。这一节就来论述关于包含、继承、成员函数和数据成员、类之间的耦合性、构造函数、值对象与引用对象等问题。

（1）包含：“有一个...”的关系

包含是一个非常简单的概念，它表示一个类含有一个基本数据元素或对象。

- 通过包含实现“有一个”的关系：例如，一名雇员“有一个”姓名、“有一个”电话号码等，通常，你可以让姓名、电话号码成为 Employee 类的数据成员，从而建立这种关系。

- 在万不得已时通过 private 继承来实现“有一个”的关系：在某些情况下，你会发现根本无法，使用将一个对象当作另一个对象的成员的办法，来实现包含关系。一些专家建议，此时可采用 private 继承所要包含的对象的办法。这么做的主要原因，是要让外层的包含类，能够访问内层被包含类的 protected 成员函数与数据成员。然而在实践中，这种做法会在派生类与基类之间，形成一种过于紧密的关系，从而破坏了封装性。而且，这种做法也往往会带来一些设计上的错误，而这些错误是可以用“private 继承”之外的其他方法解决的。

- 警惕有超过约 7 个数据成员的类：研究表明，人们在做其他事情时，能记住的离散项目的个数是 7±2。如果一个类包含有超过约 7 个数据成员，请考虑要不要把它分解为几个更小的类。如果数据成员都是整型或字符串这种简单数据类型，可以按 7±2 的上限考虑；反正，如果数据成员都是复杂对象的话，就应按 7±2 的下限来考虑了。

（2）继承：“是一个...”的关系

继承的概念是说一个类是另一个类的一种特化。继承的目的在于，通过“定义能为两个或更多个派生类提供共有元素的基类”的方式，写出更精简的代码。其中共有元素可以是子程序接口、内部实现、数据成员或数据类型等。继承能把这些共有的元素集中在一个基类中，从而有助于避免在多处出现重复的代码和数据。

当决定使用继承时，必须做如下几项决策：

- 对于每一个成员函数，它应该对派生类可见吗？它应该有默认的实现吗？这一默认的实现，能被覆盖（override）吗？
- 对每一个数据成员而言，包括变量、具名常量、枚举等，它应该对派生类可见吗？

下面详细解释如何考虑这些事项：

- 用 public 继承来实现“是一个...”的关系：当决定通过继承一个现有类的方式，来创建一个新类时，表明这个新的类，是基类的一个更为特殊的版本。基类既对派生类将会做什么设定了预期，也对派生类能怎么运作，提出了限制。如果派生类不准备完全遵守，由基类定义的同一个接口契约，继承就不是正确的实现技术了；请考虑换用包含的方式，或者对继承体系的上层做修改。
- 要么使用继承并进行详细说明，要么就不用它：继承给程序增加了复杂度，因此它是一种危险的技术。如果某个类并未设计为可被继承，就应该把它的成员定义成 non-vertual。
- 遵循 Liskov 替换原则：除非派生类真的“是一个”更特殊基类，否则不应该从基类继承。派生类必须能通过基类的接口而被使用，且使用者无须了解两则之间的差异。换句话说，对于基类中定义的所有子程序，用在它的任何一个派生类中时的含义，都应该是相同的。
- 确保只继承需要继承的部分：派生类可以继承成员函数的接口和/或实现。继承而来的子程序，有三种基本情况：
  - 抽象且可覆盖的子程序，是指派生类只继承了该子程序的接口，但不继承实现；
  - 可覆盖的子程序，是指派生类继承了该子程序的接口及其默认实现，并且可以覆盖该默认实现；
  - 不可覆盖的子程序，是指派生类继承了该子程序的接口及其默认实现，但不能覆盖该默认实现。

  如果你只是想使用一个类的实现，而不是接口，那么就应该采用包含方式，而不该用继承。
- 不要“覆盖”一个不可覆盖的成员函数：C++ 和 Java 都允许成员函数“覆盖”那些不可覆盖的成员函数。如果一个成员函数在基类中是 private 的，其派生类可以创建一个同名的成员函数。对于阅读派生类代码的程序员来说，这个函数是令人困惑的，因为它看上去似乎应该是多态的，但事实上却并非如此，只是同名而已。因此建议派生类中的成员函数，不要与基类中不可覆盖的成员函数重名。
- 把共用的接口、数据及操作，放到继承树中尽可能高的位置：接口、数据和操作在继承体系中的位置越高，派生类使用它们的时候就越容易。多高就算太高了呢？根据抽象性来决定吧，如果你发现把一个子程序移到更高的层次后，会破坏该层对象的抽象性，就该停手了。
- 只有一个实例的类是值得怀疑的：只需要一个实例，这可能表明设计中把对象和类混为一谈了。考虑能否只创建一个新的对象，而不是一个新的类。
- 只有一个派生类的基类也是值得怀疑：不要创建任何并非绝对必要的继承结构。
- 派生后覆盖了某个子程序，但在其中没做任何操作，这种情况也是值得怀疑的：这通常表明基类的设计中有错误。例如，假设你有一个 Cat（猫）类，它有一个 Scratch() 成员函数，可是最终你发现，有些猫的爪尖没了，不能抓了。你可能想从 Cat 类派生一个叫 ScratchlessCat 的猫，然后覆盖 Scratch() 方法，让它什么都不做。但这种做法有几个问题：
  - 它修改了 Cat 类的接口所表达的语义，因此破坏了 Cat 类所代表的抽象；
  - 当你从它进一步派生出其他派生类时，采用这一做法会迅速失控。如果你又发现有只猫没有尾巴怎么办？或者有只猫不捉老鼠呢？再或者有只猫不喝牛奶？最终你会派生出一堆类似 ScratchlessCat、TaillessCat、MicelessCat、MilklessCat 这样的派生类来；
  - 采用这种做法一段时间后，代码会逐渐变得混乱而难以维护，因为基类的接口喝行为几乎无法让人理解其派生类的行为

修正这一问题的位置不是在派生类，而是在最初的 Cat 类中，应该创建一个 Claw 类，并让 Cat 类包含它。问题的根源在于做了所有猫都能抓的假设，因此应该从源头上解决问题，而不是发现问题的地方修改。

- 避免让继承体系过深：大多数人在脑中，同时应付超过 2 到 3 层继承时就有麻烦了，过深的继承层次，会显著导致错误率的增长。过深的继承层次增加了复杂度，请确保你在用继承来避免代码重复，并使复杂度最小。

- 尽量使用多态，避免大量的类型检查：频繁重复出现的 case 语句，有时是在暗示，采用继承可能是种更好的设计选择，尽管并不总是如此。下面就是一段迫切需要采用，更为面向对象方法的典型代码示例：

```
C++示例：多数情况下，应该用多态来替代的case语句
switch ( shape.type ) {
    case Shape_Circle:
        shape.DrawCircle();
        break;
    case Shape_Square:
        shape.DrawSquare();
        break;
    ...
}
```

在这个例子中，对 shape.DrawCircle() 和 shape.DrawSquare() 的调用，应该用一个叫 shape.Draw() 的方法来替代，因为无论形状是圆的，还是方的，都可以调用这个方法来绘制。另外，case 语句有时也用来把种类确实不同的对象或行为分开。下面就是一个在面向对象中，合理采用 case 语句的例子：

```
C++示例：也许不该用多态来替代的case语句
Switch ( ui.Command() ) {
    case Comman_OpenFile:
        OpenFile();
        break;
    case Comman_Print:
        Print();
        break;
    case Command_Save:
        Save();
        break;
    case Command_Exit:
        ShutDown();
        break;
    ...
}
```

此时也可以创建一个基类，并派生一些派生类，再用多态的 DoCommand() 方法，来实现每一种命令；但在像这个例子一样简单的场合中，DoCommand() 意义实在不大，因此采用 case 语句才是更容易理解的方案。

- 让所有数据都是 private，而非 protected：继承会破坏封装，当你从一个对象继承时，你就拥有了能够访问该对象中 protected 子程序和 protected 数据的特权。如果派生类真的需要访问基类的属性，就应该提供 protected 访问器函数（accessor function）。

- 多重继承：在大多数情况下，应该避免使用多重继承。多重继承的用途，主要是定义“混合体”，也就是一些能给对象增加一组属性的简单类。之所以称其为混合体，是因为它们可以把一些属性“混合”到派生类里面。“混合体”可以是形如 Display，Presistant，Serializable 或 Sortable 这样的类。它们几乎总是抽象的，也不打算独立

于其他对象而被单独实例化。混合体需要使用多重继承，但只要所有的混合体之间保持完全独立，它们也不会导致典型的菱形继承问题。通过把一些属性夹在一起，还能使设计方案更容易理解。程序员会更容易理解一个用了 Displayable 和 Persistent 混合体的对象，因为这样只需要实现两个属性即可，而较难理解一个需要实现 11 个更具体的子程序的对象。C++ 同时支持接口和实现的多重继承，但程序员在决定使用多重继承之前，应该仔细考虑其他替代方案，并谨慎地评估它可能对系统复杂度和可理解性产生地影响。

以上许多规则，能帮你远离继承相关地麻烦，所有这些规则背后地潜台词都是在说，继承往往会让你和程序员地首要技术使命（管理复杂度）背道而驰。从控制复杂度的角度说，你应该对继承持有非常歧视的态度。下面来总结一下，何时可以使用继承，何时又该使用包含：

- 如果多个类共享数据而非行为，应该创建这些类可以包含的共用对象；
- 如果多个类共享行为而非数据，应该让它们从共同的基类继承而来，并在基类里定义共用的子程序；
- 如果多个类既共享数据，也共享行为，应该让它们从一个共同的基类继承而来，并在基类里定义共用的数据和子程序；
- 当你想由基类控制接口时，使用继承；当你想自己控制接口时，使用包含。

（3）成员函数和数据成员

下面就有效地实现成员函数和数据成员给出一些指导建议：

- 让类中子程序的数量尽可能少：一份针对 C++ 程序的研究，类里面的子程序的数量越多，出错率也就越高。然而，也发现其他一些竞争因素产生的影响更显著，包括过深的继承体系、在一个类中调用了大量的子程序，以及类之间的强耦合等。请在保持子程序数量最少和其他这些因素之间评估利弊；
- 禁止隐式地产生你不需要的成员函数和运算符：有时你会发现，应该禁止某些成员函数，比如说你想禁止赋值，或不想让某个对象被构造。在这种情况下，可以通过把构造函数、赋值运算符或其他成员函数，或运算符定义为 private，从而禁止调用方代码访问它们；
- 减少类所调用的不同子程序的数量：一份研究发现，类里面的错误数量，与类所调用的子程序的总数是统计相关的。同一研究还发现，类所用到的其他类的数量越高，其出错率也往往会越高。这些概念有时也称为“扇入”；
- 对其他类的子程序的间接调用要尽可能少：直接的关联已经够危险了，而间接的关联，如 account.ContactPerson() .DaytimeContactInfo().PhoneNumber() 往往更加危险。研究人员就此总结了一条”Demeter 法则”，基本上就是说 A 对象可以任意调用它自己的所有子程序。如果 A 对象创建了一个 B 对象，它也可以调用 B 对象的任何公用子程序，但是它应该避免再调用由 B 对象所提供的对象的子程序。就是说 account.ContactPerson() 这一调用是合适的，但 account.ContactPerson().DaytimeContactInfo() 则不合适。
- 一般来说，应尽量减小类和类之间相互合作的范围，尽量让下面这几个数字最小：
  - 所实例化的对象种类；
  - 在被实例化对象上直接调用的不同子程序的数量；
  - 调用由其他对象返回的对象的子程序的数量。

（4）构造函数

接下来给出一些只适用于构造函数的指导建议。针对构造函数的这些建议，对于不同的语言都差不多；但对于析构函数而言，则略有不同。

- 如果可能，应该在所有的构造函数中，初始化所有的数据成员：在所有的构造函数中初始化所有的数据成员，是一个不难做到的防御式编程实践；
- 用 private 构造函数来强制实现单件属性（singleton property）：如果你想定义一个类，并需要强制规定它只能有唯一一个对象实例的话，可以把该类所有的构造函数，都隐藏起来，然后对外提供一个 static 的 GetInstance() 子程序，来访问该类的唯一实例。它的工作方式如下：

Java示例：用私有构造函数来实现Singleton

```
public class MaxId {
    // constructors and destructors
    private MaxId() {
        ...
    }
    ...
    // public routines
    public static MaxId GetInstance() {
        return m_instance;
    }
    ...

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

仅在初始化 static 对象 m_instance 时，才会调用私有构造函数。用这种方法后，当你需要引用 MaxId 单件时，就只需要简单地引用 MaxId.GetInstance() 即可。

- 优先采用 deep copies，除非论证可行，才采用 shallow copies：在设计复杂对象地时候，需要做出一项主要决策，即应为对象实现深拷贝，还是浅拷贝。对象地深层复本，是对象成员数据逐项复制地结果；而其浅层复本，则往往只是指向或引用同一个实际对象，当然，“深”和“浅”的具体含义可以有些出入。实现浅拷贝的动机，一般是为了改善性能。尽管把大型的对象复制出多份副本，从美学上看十分令人不快，但这样做很少会导致显著的性能损失。某几个对象可能会引起性能问题，但众所周知，程序员们很不擅长推测真正招致问题的代码。为了不确定的性能提高，而增加复杂度是不妥的，因此，在面临选择实现深拷贝还是浅拷贝时，一种合理的方式，便是优先实现深拷贝，除非能够论证浅拷贝更好。深层复本在开发和维护方面，都要比浅层复本简单。实现浅拷贝除了要用到两种方法都需要的代码之外，还要增加很多代码用于引用计数、确保安全地复制对象、安全地比较对象，以及安全地删除对象等。而这些代码是很容易出错地，除非你有充分的理由，否则就应该避免它们。

6.4 创建类的原因

下面列出了一些创建类的合理原因：

- 为现实世界中的对象建模：请为程序中需要建模的每一个出现在现实世界中的对象类型，创建一个类；把该对象所需的数据，添加到类里面，然后编写一些服务子程序，来为对象的行为建模；
- 为抽象的对象建模：创建类的另一个合理的原因是要建立抽象对象的模型，所谓的抽象对象，并不是一个现实世界中的具体对象，但它却能为另一些具体的对象提供一种抽象。经典的 Shape 对象，就是一个很好的例子；Circle 和 Square 都是真实存在的，但 Shape 则是对其他具体形状的一种抽象。
- 降低复杂度：创建类的一个最重要的理由，便是降低程序的复杂度；创建一个类来把信息隐藏起来，这样就无须再去考虑它们。当然，当你写到这个类的时候，还是要考虑这些信息的。但类写好后，你就应该能够忘掉这些细节，并能在无须了解其内部工作原理的情况下使用这个类。
- 隔离复杂度：无论复杂度表现为何种形式，复杂的算法、大型数据集、或错综复杂的通讯协议等，都容易引发错误；一旦错误发生，只要它还在类的局部，而未扩散到整个程序中，找到它就会比较容易。修正错误时引发的改动不会影响到其他代码，因为只有一个类需要修改，不会碰到其他代码。如果你找到了一种更好、更简单或更可

靠的算法，而原有的算法已经用类隔离起来的话，就可以很容易地把它替换掉。在开发过程中，这样可以让你更容易地尝试更多设计方案，保留最好地一种方案；

- 隐藏实现细节；
- 限制变动地影响范围：把容易变动的部分隔离开来，这样就能把变动所带来的影响，限制在一个或少数几个类的范围内。把最容易变动的部分，设计成最容易修改的。容易变动的部分有硬件依赖、I/O、复杂数据类型、业务逻辑等；
- 隐藏全局数据：如果需要用到全局数据，就可以把它的实现细节，隐藏到某个类的接口背后。与直接使用全局数据相比，通过访问器子程序来操控全局数据有很多好处。你可以改变数据结构，而无须修改程序本身。你可以监视对这些数据的访问。“使用访问器子程序”的这条纪律，还会促使你思考有关数据，是否应该就是全局的；经常你会豁然开朗地发现，“全局数据”原来只是对象地数据而已；
- 让参数传递更顺畅：如果你需要把一个参数，在多个子程序之间传递，这有可能表明应该把这些子程序重构到一个类里，把这个参数当作对象数据来共享。
- 建立中心控制点：在一个地方来控制一项任务是个好主意。控制可以表现为很多形式：了解一张表中记录的数目是一种形式；对文件、数据库连接、打印机等设备进行的控制又是另一种。用一个类来读写数据库，则是集中控制的又一种形式。如果需要把数据库转换为平坦的文件或者内存数据，有关改动也会影响一个类。
- 让代码更易重用：将代码放入精心分解的一组类中，比将代码全部塞入某个更大的类里面，更容易在其他程序中重用。如果有一部分代码，它们只是在程序里的一个地方调用，只要它可以被理解为一个较大类的一部分，而且这部分代码可能会在其他程序中用到，就可以把它提出来形成一个单独的类。
- 为程序族做计划：如果你预计到某个程序会被修改，你可以把预计要被改动的部分放到单独的类里，同其他部分隔离开来，这是个好主意。之后你就可以只修改这个类，或用新的类来取代它，而不会影响到程序的其余部分了。仔细考虑整个程序族的可能情况，而不是单是考虑单一程序的可能情况，这又是一种用于预先应对各种变化的强有力的启发式方法。
- 把相关操作包装到一起：即便你无法隐藏信息、共享数据或规划灵活性，你仍然可以把相关的操作合理地分组，比如分为三角函数、统计函数、字符串处理子程序、为操作子程序以及图形子程序，等等。类是把相关操作组合在一起地一种方法。除此之外，根据你所使用的编程语言不同，你还可以使用包、命名空间或头文件等方法。
- 实现某种特定的重构：在后面“重构”章节中，所描述的很多特定的重构方法，都会生成新的类，包括把一个类转换为两个、隐藏委托、去掉中间人以及引入扩展类等。

应该避免创建的类：

- 避免创建万能类：要避免创建什么都知道、什么都能干的万能类。如果一个类把工夫都花在用 Get() 方法和 Set() 方法，向其他类索要数据的话，请考虑是否应该把这些功能组织到其他那些类中去，而不要放到万能类里；
- 消除无关紧要的类：如果一个类包含数据，但不包含行为的话，应该问问自己，它真的是一个类吗？同时应该考虑把这个类降级，让它的数据成员成为一个或多个其他类的属性。
- 避免用动词命名的类：只有行为而没有数据的类，往往不是一个真正的类。请考虑把类似 DatabaseInitialization 或 StringBuilder 这样的类变成其他类的一个子程序。

6.5 与具体编程语言相关的问题

下面列出跟类相关的，不同语言之间有着显著差异的一些地方：

- 在继承层次中，被覆盖的构造函数和析构函数的行为；
- 在异常处理时构造函数和析构函数的行为；

- 默认构造函数，即无参数的构造函数，的重要性；
- 析构函数或终结器的调用时机；
- 和覆盖语言内置的运算符相关的知识；
- 当对象被创建和销毁时，或当其被声明时，或者它所在的作用域退出时，处理内存的方式。

7 高质量的子程序

什么是“子程序（routine）”？子程序是为实现一个特定目的而编写的，一个可被调用的方法或过程。例如 C++ 中的函数（function），Java 中的方法（method）等等。对于某些使用方式，C 和 C++ 中的宏也可认为是子程序。那什么又是高质量的子程序呢？这个问题更难回答。也许回答这个问题的最简单的方法，是来看看什么东西不是高质量的子程序。这里举个低质量的子程序例子：

C++示例：低质量的子程序

```
void HandleStuff ( CORP_DATA & inputRec, int crntQtr, EMP_DATA empRec,
    double & estimRevenue, double ytdRevenue, int screenX, int screenY,
    COLOR_TYPE & newColor, COLOR_TYPE & prevColor, StatusType & status,
    int expenseType)
{
int i;
for ( i = 0; i < 100; i++) {
    inputRec.revenue[i] = 0;
    inputRec.expense[i] = corpExpense[ crntQtr ][ i ];
    }
UpdateCorpDatabase( empRec );
estimRevenue = ytdRevenue * 4.0 / (double) crntQtr;
newColor = prevColor;
status = SUCCESS;
if ( expenseType == 1 ) {
    for ( i = 0; i < 12; i++)
        profit[i] = revenue[i] - expense.type1[i];
    }
else if ( expenseType == 2 ) {
        profit[i] = revenue[i] - expense.type2[i];
        }
else if ( expenseType == 3 )
        profit[i] = revenue[i] - expense.type3[i];
        }
```

这个例子中至少有如下一些问题：

- 有个很差劲的名字。HandleStuff() 一点也没有告诉你这个子程序究竟是做什么的；
- 没有文档；
- 布局不好，代码的物理组织形式，几乎没有给出任何关于其逻辑组织的提示。布局的使用过于随意，程序内的不同部分，使用了不同的布局风格；
- 输入变量 inputRec 的值被改变了。如果它是一个输入变量，它的值就不该被修改，而且在 C++ 中它应该定义为 const；如果变量的值就是要被修改的，就不要把它命名为 inputRec；

- 读写了全局变量，它从 corpExpense 中读取数值并将其写入 profit。它应该更直接地与其他子程序通讯，而不是去读写全局变量；
- 没有一个单一地目的，它初始化了一些变量，向数据库写入数据，又做了一些计算，从这些事情之间看不出任何联系。子程序应该有单一而明确地目的；
- 没有注意防范错误数据，如果 crntQtr 等于 0，那么表达式 ytdRevenue*4.0 / (double)crntQtrh 将会导致除零错误；
- 用了若干魔鬼数值：100、4.0、12、2、3 等；
- 未使用其中一些参数：screenX 和 screenY 在程序中都没有被引用过；
- 一个参数传递方式有无：prevColor 被标为引用参数（&），但在这个子程序内却未对其赋值；
- 参数太多：合理地参数个数，其上限大概 7 个左右，而这个子程序有 11 个。这些参数地排列方式也难以理解，估计没人想仔细研究它们、甚至没人想数数有几个参数；
- 参数顺序混乱且未经注释。

7.1 创建子程序地正当有理由

（1）下面列出一些创建子程序地正当理由：

- 降低复杂度：创建子程序地一个最重要地原因，就是为了降低程序地复杂度。可以通过创建子程序来隐藏一些信息，这样就不必再去考虑这些信息了。当然，在你要编写这个子程序地时候，肯定是要考虑它们的。不过一旦程序写好了，就应该忘记这些细节，可以直接调用该子程序而无须了解其内部工作细节。创建子程序还有其他一些原因，如缩小代码规模、改善可维护性、提高正确性等，也都是很不错的，但如果没有子程序的抽象能力，我们的智力将根本无法管理复杂度的程序。当内部循环或条件判断的嵌套层次很深时，就意味着需要从子程序中提取出新的子程序了。把嵌套的部分提取出来，形成一个独立的子程序，可以降低外围子程序的复杂度；
- 引入中间、易懂的抽象：把一段代码放入一个命名恰当的子程序内，是说明这段代码用意最好的方法之一。例如，与读下面这一串语句相比，

```
if ( node <> NULL ) then
    while ( node.next <> NULL ) do
        node = node.next
        leafName = node.name
    end while
else
    leafName = ""
end if
```

读懂下面这条语句就更容易：

```
leafName = GetLeafName( node )
```

- 避免代码重复：如果在两段子程序内编写相似的代码，就意味着代码分解出现了差错；这时，应该把两段子程序中的重复代码放入派生类中；还有另一种办法，你也可以把相同的代码放入新的子程序中，再让其余的代码来调用这个子程序。与代码的重复出现相比，让相同的代码只出现一次可以节约空间。代码改动起来也更方便，因为你只需要在一处修改即可。这时的代码也会更加可靠，因为为了验证代码的正确性，你只需要检查一处代码。同时，这样做也会使改动更加可靠，因为你可以避免需要相同的修改时，却做了一些略有不同的修改；
- 支持子类化：override 简短而规整的子程序，所需新代码的数量，要比覆盖冗长而邋遢的子程序更少。如果你能让可覆盖的子程序保持简单，那你在实现派生类的时候也会减少犯错的几率；

- 隐藏顺序：把处理事件的顺序隐藏起来是一个好主意。比如，如果一个程序通常都是先从用户那里读取数据，然后再从一个文件中读取辅助数据；那么，无论从用户那里读取数据的子程序，还是从一个文件中读取数据的子程序，都不应该依赖另一个子程序是否已执行。

- 隐藏指针操作：指针操作的可读性通常都很差，而且也容易出错。通过把这些操作隔离在子程序内部，你就可以把精力集中于操作的意图本身，而不是指针操作机制的细节。同时，如果此类操作都能在一个位置完成，那么你对代码的正确性就会更有把握。同时，如果此类操作都能在一个位置完成，那么你对代码的正确性就会更有把握。如果你发现了比指针更合适的数据类型，也可以对程序做出修改，而不用担心会破坏了那些原本要使用指针的代码；

- 提高可移植性：可以用子程序来隔离程序中不可移植的部分，从而明确识别和隔离未来的移植工作。不可移植的部分包括编程语言所提供的非标准功能、对硬件的依赖，以及对操作系统的依赖等。

- 简化复杂的布尔判断：为了理解程序的流程，通常并没有必要去研究那些复杂的布尔判断的细节。应该把这些判断放入函数中，以提高代码的可读性，因为这样就把判断的细节放到一边了，同时一个具有描述性的函数名字，可以概括出该判断的目的。把布尔判断的逻辑放入单独的函数中，也强调了它的重要性。

- 改善性能：通过使用子程序，你可以只在一个地方优化代码。把代码集中在一处可以更方便地查出哪些代码地运行效率低下。同时，在一处进行地优化，就能使用到该子程序地所有代码都从中受益。把代码集中在一处之后，想用更高校地算法或更快速高校的语言来重写代码也更容易做了。

- 不用确保所有的子程序都很小：事实上，有些事情写一个大的子程序来完成，会更好一些。

（2）似乎简单而没必要写成子程序的操作

一个很好而又小巧的子程序会有很多用处和优点，其一便是它们能够提高可读性。例如，

伪代码示例：某种计算

```
points = deviceUnits * ( POINTS_PER_INCH / DeviceUnitsPerInch() )
```

它进行的是从设备单位到磅数的转换计算。人们也会看出这十几处代码都在做着同样的事情。但是，它们原本可以更清楚些，所以我创建了一个子程序，并给它起了个好的名字，使这一转换可以只在一个地方进行：

伪代码示例：用函数来完成计算

```
Function DeviceUnitsToPoints ( deviceUnits Integer ): Integer
    DeviceUnitsToPoints = deviceUnits *
        ( POINTS_PER_INCH / DeviceUnitsPerInch() )
End Function
```

在用这个子程序取代了哪些直接嵌入计算的代码之后，程序中的那十几行代码就差不多都成了下面这样：

伪代码示例：调用计算函数

```
points = DeviceUnitsToPoints( deviceUnits )
```

这行代码更具有可读性，甚至已经达到自我注解的地步。这个例子还暗示出把简单操作写成函数的另一个原因：简单的操作常常会变成复杂操作。在某些情况下，当某个设备激活时，DeviceUnitPerInch() 会返回 0，这意味着必须考虑到除零的情况，为此需要再多写 3 行代码：

伪代码示例：维护代码时扩展了的函数

```
Function DeviceUnitsToPoints( deviceUnits: Integer ) Integer;
    if ( DeviceUnitsPerInch() <> 0)
        DeviceUnitsToPoints = deviceUnits *
            ( POINTS_PER_INCH / DeviceUnitsPerInch() )
    else
        DeviceUnitsToPoints = 0
```

```
    end if
End Function
```

如果还是在程序中的十几处地方出现原来那样的代码，那么这一测试也就要重复十几次，这样就需要增加总共几十行代码。用一个简单的子程序，就把那几十行代码减到了 3 行。

7.2 在子程序层上设计

对子程序而言，内聚性是指子程序中各种操作之间联系的紧密程度。像 Cosine() 这样的函数就是极端内聚的，因为整个程序只完成一项功能；而 CosineAndTan() 这个函数的内聚性相对较弱，因为它完成了多余一项的操作。我们的目标是让每一个子程序，只把一件事做好，不再做任何其他事情。这样做的好处是得到更高的可靠性。高内聚的子程序比低内聚的子程序，错误更少。关于内聚性的讨论一般会涉及到内聚性的几个层次，理解一些概念要比记住一些特定的术语更重要。下面这些概念可以帮助思考如何让子程序尽可能地内聚：

- 功能的内聚性：是最强也是最好的一种内聚性，也就是说让一个子程序仅执行一项操作。例如 sin()，GetCustomerName()、EraseFile()、CalculateLoanPayment() 以及 AgeFromBirthdate() 这样的子程序，都是高度内聚的。当然，以这种方式来评估内聚性，前提是子程序所执行的操作与其名字相符，如果它还做了其他的操作，那么它就不够内聚，同时其命名也有问题。

除此之外，还有一些种类的内聚性，人们却通常认为是不够理想的：

- 顺序上的内聚性：是指在子程内包含有需要按特定顺序执行的操作，这些步骤需要共享数 ，而且只有在全部执行完毕后才完成了一项完整的功能；例如，假设某个子程序需要按照给定出生日期来计算出员工的年龄和退休时间。如果子程序先计算员工的年龄，在根据他的年龄来计算退休时间，那么它就具有顺序的内聚性。

- 通信上的内聚性：是指一个子程序中的不同操作，使用了同样的数据，但不存在其他任何联系。例如某个子程序先根据传给它的汇总数据，打印一份汇总报表，然后再把这些汇总数据重新初始化，那么这个子程序就具有通信上的内聚性，因为这两项操作只是因为使用了相同的数据才彼此产生联系。要改善这个子程序的内聚性，应该让重新初始化汇总数据的操作，尽可能靠近创建汇总数据的地方，而不是放在打印报表的子程序里。应该把这些子程序进一步拆分成几个独立的子程序：一个负责打印报表，一个负责在靠近创建或修改数据的代码的地方，重新初始化数据。然后在原本调用那个具有通信内聚的子程序的更高层的子程序中，调用这两个子程序。

- 临时的内聚性：是指含有一些因为需要同时执行，才放到一起的操作的子程序。典型的例子有：Startup()、CompleteNewEmployee()、Shutdown() 等。这些程序员认为临时的内聚性是不可取的，因为它们有时与不良的编程实践相关，比如说在 Startup() 子程序里塞进一大堆互不相关的代码等。为避免这个问题，可以把临时性的子程序，看做是一系列事件的组织者。前面提到的 Startup() 子程序可能需要读取配置文件、初始化临时文件、设置内存管理器，再显示启动画面。要想使它最有效，应该让原来那个具有临时内聚性的子程序，去调用其他的子程序，由这些子程序来完成特定的操作，而不是由它直接执行所有的操作。这个例子提出这样一个问题，即如何选择一个能够在恰当的抽象层次上，描述子程序的名字。你可能决定把一个子程序命名为 ReadConfigFileInitScratchFileEtc()，它可以暗示该子程序只有巧合的内聚性；而如果你把它命名为 Startup()，那么很明显，这个子程序就只具有一个功能，且具有功能上的内聚性。

下面是一些不可取的内聚性：

- 过程上的内聚性：是指一个子程序的操作是按特定的顺序进行的。例如，依次获取员工的姓名、住址和电话号码的子程序；这些操作执行的顺序之所以重要，只是因为它和用户按屏幕提示而输入数据的顺序相一致。另一个子程序用来获取员工的其他数据。这段程序也具有过程上的内聚性，因为它把一组操作赋以特定的顺序，而这些操作并不需要为了除此之外的任何原因而彼此关联。为了得到更好的内聚性，可以把不同的操作纳入各自的子程序中。让调用方的子程序具有单一而完整的功能：GetEmployee() 就比 GetFirstPartOfEmployeeData() 更为可取。你可能还需要修改用来读取其余数据的子程序。为了让所有的程序都具有功能上的内聚性，对两个或更多的原有子程序进行修改是很常见的。

- 逻辑上的内聚性：是指若干操作被放入同一个子程序中，通过传入的控制标志选择执行其中的一项操作。之所以称之为逻辑上的内聚性，是因为子程序的控制流或所谓“逻辑”，是将这些操作放到一起的唯一原因，它们都被抱在一个很大的 if 语句或 case 语句中，而不是因为各项操作之间有任何逻辑。认为是逻辑上的内聚性的标志属性，就是各项操作之间没有关联，因此，似乎更应称其为“缺乏逻辑的内聚性”。例如，一个名为 InputAll() 的子程序，它根据传入的控制标志来决定是输入客户姓名、员工考勤卡信息，还是库存数据。类似的例子还有 ComputeAll()、EditAll()、PrintAll() 和 SaveAll()。这种子程序的主要问题，是你不该通过传入控制标志，来控制另一个子程序的处理方式。相比之下，让三个子程序分别完成不同的操作，要比用一个“根据传入的控制标志选择执行三项不同的操作之一”的子程序，清晰得多。如果这些操作中，含有一些相同代码或共用了数据，那么应该把那些代码移入一个低层子程序中，这些子程序也应该包裹在一个类中。如果子程序里的代码仅由一系列的 if 语句或 case 语句，以及调用其他子程序的语句组成，那么创建这样一个具有逻辑上的内聚性的子程序，通常也是可以的。在这种情况下，如果子程序唯一的功能，就是发布各种命令，其自身并不做任何处理，这通常也是一个不错的设计。这类子程序的技术术语便是“事件处理器”。事件处理器通常用在各种交互性的环境中，例如像 Apple Macintosh、Microsoft Windows 及其他一些 GUI 环境。

- 巧合的内聚性：是指子程序中的各个操作之间，没有任何可以看到的关联。它也可称为“无内聚”或“混乱的内聚性”。本章开头的 C++ 子程序就具有巧合的内聚性。很难从巧合的内聚性转变为任何一类更好的内聚性，通常你需要深入地重新设计和重新实现。

7.3 好的子程序名字

好的子程序名字，能清晰地描述子程序所做地一切。下面是有效地给子程序命名地一些指导原则：

- 描述子程序所做的所有事情：子程序的名字应当描述其所有的输出结果，以及副作用。如果一个子程序的作用，是计算报表总额并打开一个输出文件，那么把它命名为 ComputeReportTotals() 就还不算完整。ComputeReportTotalsAndOpenOutputFile() 是很完整的，但是又太长且显得有点傻。如果你写的是有一些副作用的子程序，那就会起很多又长又笨的名字。解决的办法，不是使用某个描述较弱的子程序名，而应该换一种方式编写程序，直截了当地解决问题而不产生副作用。

- 避免使用无意义的、模糊或表述不清的动词：有些动词的含义非常灵活，可以延伸到涵盖几乎任何含义。像 HandleCalculation()、PerformServices()、OutputUser()、ProcessInput() 和 DealWithOutput() 这样的子程序名字，根本不能说明子程序是做什么的。最多就是告诉你这些子程序所做的事情与计算、服务、用户、输入、输出有关。当然，当动词“handle”用作“事件处理”这一特定的技术含义时是个例外。

- 不要仅通过数字来形成不同的子程序名字：程序员们有时会用数字来区分类似于 OutputUser、OutputUser1 和 OutputUser2 这样的子程序。这些名字后面的数字无法显示子程序所代表的抽象有何不同，因此这些子程序的命名也都很糟糕。

- 根据需要确定子程序名字的长度：研究表明，变量名的最佳长度是 9 到 15 个字符。子程序通常比变量更为复杂，因此，好的子程序名字通常会更长一些。另一方面，子程序名字通常是跟在对象名字之后，这实际上为其免费提供了一部分名字。总的来说，给子程序命名的重点是尽可能含义清晰，也就是说，子程序名的长短要视该名字是否清晰易懂而定。

- 给函数命名时，要对返回值有所描述：函数有返回值，因此，函数的命名应该针对其返回值进行。比如说 cos()、customerId.Next()、printer.IsReady() 和 pen.CurrentColor() 都是不错的函数名，它们精确地表述了该函数将要返回地结果。

- 给子程序起名时使用语气强烈的动词加宾语的形式：一个具有功能内聚性的子程序，通常是针对一个对象执行一种操作。子程序的名字应该能反应所做的事情，而一个针对某对象执行的操作，就需要一个动词 + 宾语形式的名字。如 PrintDocument()、CalcMothlyRevenues()、CheckOrderInfo() 和 RepaginateDocument() 等，都是很不错的函数名。在面向对象语言中，不用在函数名中，加入对象的名字（宾语），因为对象本身已经包含在调用语句中了。你会用 document.Print()、orderInfo.Check() 和 mothlyRevenues.Calc() 等语句调用子程序。

而诸如 document.PrintDocument() 这样的语句则显得太臃肿，并且当它们在派生类中被调用时，也容易产生误解。如果 Check（支票）类从 Document 类继承而来的，那么 check.Print() 就是很显然表示打印一张支票，而 check.PrintDocument() 看上去像是要打印支票簿或是信用卡的对账单，而不像是打印支票本身。

- 准确使用对仗词：命名时遵守对仗词的命名规则，有助于保持一致性，从而也提高可读性。像 first/last 这样的对仗词组，就很容易理解；而像 FileOpen() 和 _lclose() 这样的组合则不对称，容易使人迷惑。

- 为常用操作确立命名规则：在某些系统里，区分不同类别的操作非常重要，而命名规则往往是指示这种区别的最简单，也是最可靠的方法。例如在一个项目里，每个对象都被分配了一个唯一标识符，如果忽视了返回这种对象标识的子程序命名规则，以至于有了下面这些子程序名字：

```
employee.id.Get()
dependent.GetId()
supervisor()
candidate.id()
```

其中，Employee 类提供了其 id 对象，而该对象又进而提供了 Get() 方法；Dependent 类提供了 GetId() 方法；Supervisor 类把 id 作为它的默认返回值；Candidate 类则认为 id 对象的默认返回值是 id，因此暴露了 id 对象。到了项目中期，已经没有人能记住哪个对象应该用哪些子程序，但此时已经编写了太多的代码，已经无法返回再重新统一命名规则了。这样一来，项目组中每个人都不得不花费精力，去记住每个对象上采用的获取 id 的语法细节。而这些问题完全可以通过建立获取 id 的命名规则而避免。

7.4 子程序可以写多长

再面向对象的程序中，一大部分程序，都是访问器子程序，它们都非常短小；在任何时候，复杂的算法总会导致更长的子程序，在这种情况下，可以允许子程序的长度，有序地增长到 100 到 200 行（不算源代码中的注释行和空行）。数十年的研究证据表明，这么长的子程序，也和短小的子程序，一样不易出错。与其对子程序的长度强加限制，还不如让下面这些因素：如子程序的内聚性、嵌套的层次、变量的数量、决策点的数量、解释子程序用意所需的注释数量，以及其他一些跟复杂度相关的考虑事项等；来决定子程序的长度。也就是说，如果要编写一段超多 200 行代码的子程序，那你就要小心了。对于超过 200 行代码的子程序来说，美誉哪项研究发现，它能降低成本或降低出错率，而且在超过 200 行后，你迟早会在可读性方面遇到问题。

7.5 如何使用子程序参数

子程序之间的接口是程序中最易出错的部分之一，研究表明，程序中有 39% 的错误都是属于内部接口错误，也就是子程序间互相通信时所发生的错误。以下是一些指导原则：

- 按照输入-修改-输出的顺序排列参数：这种排列方法暗含了子程序的内部操作所发生的顺序，先是输入数据，然后修改数据，最后输出结果。

- 考虑自己创建 in 和 out 关键字：对于有些不支持 in 和 out 关键字，可以通过预处理指令来创建 in 和 out 关键字：

```
C++示例：定义你自己的IN和OUT关键字
#define IN
#define OUT
void InvertMatrix(
    IN Matrix originalMatrix,
    OUT Matrix *resultMatrix
);
...
```

```
void ChangeSentenceCase(
    IN StringCase desiredCase,
    IN OUT Sentence *sentenceToEdit
);
...
void PrintPageNumber(
    IN int PageNumber,
    OUT StatusType &status
);
```

在这里，IN 和 OUT 这两个宏关键字，只是起说明性的作用。如果你想让被调用的子程序修改某一个参数的值，那么还是得通过指针或引用参数来传递该参数。请在应用这一技术之前，请考虑它得以下两种显著弊端。自行定义得 IN 和 OUT 关键字，扩展了 C++ 语言，从而在某种程度上，让多数阅读这一代码的人，感到生疏。如果你以这种方式扩展所用的语言，请确保能持续一直地使用该方法，最好是在整个项目地范围内。第二个弊端在于，编译器并不会强制 IN 和 OUT 关键字的使用，也就是说，你可能把某个参数标记为 IN，但仍在子程序中修改了该参数的值，阅读代码的人，可能会误认为有关代码是正确的，然而事实却并非如此。使用 C++ 中的 const 关键字来定义输入参数，通常更为适宜。

- 如果几个子程序都用了类似的一些参数，应该让这些参数的排列顺序保持一致：子程序的参数顺序，可以产生记忆效应，不一致的顺序，会让参数难以记忆。
- 使用所有的参数：既然往子程序传递了一个参数，就一定要用到这个参数。如果你不用它，就把它从子程序的接口中删去。未被用到的参数会增加出错率。
- 把状态或出错变量放在最后：按照习惯做法，状态变量和那些用于指示发生错误的变量，应放在参数的最后。它们只是附属于程序的主要功能，而且它们是仅处于输出的参数，因此这是一种很有道理的规则。
- 不把子程序的参数用作工作变量：把传入子程序的参数，用作工作变量是很危险的。应该使用局部变量。比如下面这段 Java 程序中，inputVal 这个参数就被不恰当地用于存储计算地中间结果：

Java示例：不恰当地使用输入参数

```
int Sample( int inputVal ) {
    inputVal = inputVal * CurrentMultiplier ( inputVal );
    inputVal = inputVal * CurrentAdder ( inputVal );
    ...
    return inuptVal;
}
```

在这段代码中，inputVal 这个名字很容易引起误解，因为当执行到最后一行代码时，inputVal 包含的已经不是最初的输入值了，它的值是用输入值计算出的结果，因此这个参数名起得不对。如果日后你又要修改这段程序，要在其他地方使用原有得输入值，你可能会想当然地以为 inputVal 是含有原始输入值的参数并使用它，而事实上并非如此。好的处理方式是明确地引入一些工作变量，从而避免当前或日后地麻烦：

Java示例：正确地使用输入参数

```
int Sample( int inuptVal ) {
    int WorkingVal = inputVal;
    workingVal = workingVal * CurrentMultiplier( workingVal );
    workingVal = workingVal + CurrentAdder( workingVal );
    ...
    return workingVal;
}
```

引入新变量 workingVal，就澄清了 inputVal 的角色，同时也消除了在错误的时间误用 inputVal 的可能。注意不要将变量命名为 workingVal 或 inputVal，该示例只是用来展示不把子程序的参数用作工作变量。

- 在接口中对参数的假定加以说明：如果你假定了传递给子程序的参数，具有某种特征，那就要对这种假定加以说明。在子程序内部和调用子程序的地方，同时对所做的假定进行说明是值得的。不要等到把子程序写完后，在回过头去写注释，你是不会记住所有这些假定的。一种比注释还好的方法，是在代码中使用断言。应该对哪些接口参数的假定进行说明呢？
  - 参数是仅用于输入的、要被修改的、还是仅用于输入的；
  - 表示数量的参数的单位（英寸、英尺、米等）；
  - 如果没有用枚举类型的话，应说明状态代码和错误值的含义；
  - 所能接受的数值的范围；
  - 不该出现的特定数值。

- 把子程序的参数个数限制在大约 7 个以内：对于人的理解力来说，7 是一个神奇的数字；心理学研究发现，通常人类很难同时记住超过 7 个单位的信息。这一发现已经用于各种领域之中，因此，假定人不能同时记住超过约 7 个的子程序参数，也是合适的。如果你使用的是一种支持结构化数据的现代编程语言，就可以传递一个含有 13 个成员的合成数据类型，并将它看作一个大数据块；如果你使用的是一种更为原始的编程语言，那你可能就需要分别传递全部 13 个成员。如果你发现自己一直需要传递很多参数，这就说明子程序之间的耦合度太过紧密了。应该重新设计这个或这组子程序，降低其间的耦合度。如果你向很多不同的子程序传递相同的数据，就请把这些子程序组合成一个类，并把那些经常使用的数据用作类的内部数据。

- 考虑对参数采用某种表示输入、修改、输出的命名规则：如果你觉得把输入、修改、输出参数区分开很重要，那么就建立一种命名规则，来对它们进行区分。你可以给这些参数名字加上 i_、m_、o_ 前缀。

- 为子程序传递用以维持其接口抽象的变量或对象：关于如何把对象的成员传给子程序这一问题，存在着两种互不相让的观点。比如说你有一个对象，它通过 10 个访问器子程序暴露其中的数据，被调用的子程序，只需要其中的 3 项数据就能进行操作。持第一种观点的人们认为，只应传递子程序所需的 3 项特定数据即可。他们的论据是，这样做可以最大限度地减少子程序之间地关联，从而降低其耦合度，使它们更容易读，更便于重用等等。他们强调说，把整个对象传递给子程序就破坏了封装的原则，因为这样就是潜在地把所有 10 个访问器子程序都暴露给被调用地那个子程序了。持第二种观点地人们，则认为应该传递整个对象。他们认为，如果在不修改子程序接口地情况下，让被调用子程序能够灵活使用对象地其余成员，就可以保持接口更稳定。他们争辩说，只传递 3 项特定的数据，破坏了封装性，因为这样做就是把特定的数据项暴露给被调用的那个子程序了。我认为这两种规则都过于简单，并没有击中问题的要害：子程序的接口要表达何种抽象？如果要表达的抽象是子程序期望 3 项特定的数据，但这 3 项数据只是碰巧由同一个对象所提供的，那就应该单独传这 3 项数据；然而，如果子程序接口要表达的抽象，是想一直拥有某个特定对象，且该子程序要对这一对象执行这样那样的操作，如果单独传递 3 项特定的数据，那就是破坏了接口的抽象。如果你采用传递整个对象的做法，并发现自己是先创建对象，并被调用子程序所需的 3 项数据填入该对象，在调用过子程序后，又从对象中取出 3 项数据的值，那就是一个证据，说明你应该只传递那 3 项数据，而不是整个对象。一般说来，如果在调用子程序之前出现进行装配的代码，或者在调用子程序后，出现拆卸的代码，都是子程序设计不佳的表现。如果你发现自己经常需要修改子程序的参数表，而每次修改的参数，都是来自于同一对象，那就说明你应该传递整个对象，而不是个别数据了。

- 使用具名参数：当你有超乎平均数量的同样类型的参数时，就可能发生参数放错位置，且编译器却检测不到的情况，这时使用具名参数就格外有用了。在很多场合下，显示地把参数对应起来，可能会矫枉过正，但在需要高安全性或高可靠性地情形下，花额外地功夫，把参数按照设想的方式对应起来，是十分值得的。

- 确保实际参数与形式参数相配：一个常见的错误是在调用子程序时，使用了类型错误的变量；例如，在本该使用浮点类型的地方用了整型。如果是仅用于输入的参数，这种情况很少会带来问题；编译器在把参数传递给子程序之前，通常会将实际类型转换成形式类型。如果有问题的话，编译器通常会给出警告。但在某些情况下，特别是

当所用的参数既用于输入也用于输出时，如果传错了参数类型，你就会遇到麻烦了。请养成好的习惯，总要检查参数表中参数的类型，同时留意编译器给出的关于参数类型不匹配的警告。

7.6 使用函数时要特别考虑的问题

现代的编程语言，如 C++、Java、VB 等，都同时支持函数和过程，函数都是指有返回值的子程序；过程是指没有返回值的子程序。在 C++ 中，通常把所有子程序都成为“函数”；然而，那些返回值类型为 void 的函数在语义上其实就是过程。函数与过程的区别更多的是语义的区别，而不是语法的区别，你还是要以语义为准。如果一个子程序的主要用途就是返回由其名字所指明的返回值，那么就应该使用函数，否则就应该使用过程。使用函数时，总存在返回不正确的返回值的风险。当函数内有多条可能的执行路径，而其中一条执行路径没有设置返回值时，这一错误就出现了。为了减少这一风险，请按照下面给出的建议来做：

- 检查所有可能的返回路径：在编写函数时，请在脑海里执行每一条执行路径，确保所有可能的情况下，该函数都会返回值。在函数开头用一个默认值来初始化返回值，是个很好的做法，这种方法能够在未正确地设置返回值时，提供一张保险网。
- 不要返回指向局部数据的引用或指针：一旦子程序执行结束，其局部数据就都出了作用域，那么指向局部数据的引用或指针也随之失效。如果一个对象需要返回有关其内部数据的信息，那就应该把这些信息保存为类的数据成员。然后，它还应该提供可以返回这些数据成员的访问器子程序，而不是返回对局部数据的引用或者指针。

7.7 宏子程序和内联子程序

用预处理器的宏语言编写子程序，还需要一些特别的考虑。下面的这些规则和示例适用于在 C++ 中使用预处理器的情形。

- 把宏表达式整个包含在括号内：由于宏和其参数会被最终展开到代码中，因此请多加小心，确保代码是按照你所预期的方式被展开的。下面这个宏中，包含了一个常见的错误：

```
C++示例：一个不能正确展开的宏
#define Cube( a ) a*a*a
```

如果传给这个宏的 a 不是不可分割的值，那它就不能正确地进行这一乘法计算了。比如说你写的这个表达式是 Cube(x+1)，那么它会展开成 x+1*x+1*x+1，而由于乘法运算符地优先级高于加法运算符，这显然不是你所预期的结果。这个宏的下面这种写法要好一些，但是也不完美：

```
C++示例：仍不能正确展开的宏
#define Cube( a ) (a)*(a)*(a)
```

如果存在使用 Cube() 的表达式里，含有比乘法运算符优先级更高的运算符，那么 (a)*(a)*(a) 也会再次失效。为了防止这种情况的发生，你应该给整个表达式加上括号：

```
C++示例：可以正确展开的宏
#define Cube( a ) ((a)*(a)*(a))
```

- 把含有多条语句的宏用大括号括起来：一个宏可以含有多条语句，如果你把它当作一条语句使用，就会出错。例如：

```
C++示例：一个无法正确工作的含有多条语句的宏
#define LookupEntry( key, index) \
    index = (key - 10) / 5; \
    index = min( index, MAX_INDEX ); \
    index = max( index, MIN_INDEX );
...
```

```
for( entryCount = 0; entryCount < numEntries; entryCount++ )
    LookupEntry( entryCount, tableIndex[ entryCount ] );
```

这个宏之所以会带来麻烦，是因为它和常规函数的执行方式是不同的，按照例中所示的形式，在 for 循环语句中，只有宏的第一部分代码被执行：index = (key - 10) / 5；要避免这一问题，请把宏用大括号括起来：

```
C++语言示例：可以正确工作的含有多条语句的宏
#define LookupEntry( key, index) {\
    index = (key - 10) / 5; \
    index = min( index, MAX_INDEX ); \
    index = max( index, MIN_INDEX );
}
```

通常认为，用宏来替代函数调用的做法具有风险，而且不易理解，这是一种很糟糕的编程实践；因此，除非必要，否则还是应该避免使用这种技术。

- 用给子程序命名的方法，来给展开后代码形同子程序的宏命名，以便在需要时可以用子程序来替换宏。

像 C++ 这样的现代编程语言，都提供了大量可以取代宏的方案：

- const 可以用于定义常量；
- inline 可以用于定义可被编译为内嵌的代码的函数；
- template 可以用于以类型安全的方式，定义各种标准操作，如 min、max 等；
- enum 可以用于定义枚举类型；
- typedef 可以用于定义简单的类型替换。

C++ 支持 inline 关键字，inline 子程序允许程序员，在编写代码时，把代码当成子程序，但编译器在编译期间，通常会把每一处调用 inline 子程序的地方，都转换成插入内嵌的代码，因为避免了子程序调用的开销，因此 inline 机制可以产生非常高效的代码：

- 节制使用 inline 子程序：inline 子程序违反了封装原则，因为 C++ 要求程序员把 inline 子程序的实现代码写在头文件里，从而也就把这些实现细节暴露给了所有使用该头文件的程序员。inline 子程序要求在调用子程序的每个地方，都生成该子程序的全部代码，这样无论 inline 子程序是长是短，都会增加整体代码的长度，这也会带来其自身的问题。

8 防御式编程

防御式编程并不是说让你在编程时，持“防备批评或攻击”的态度，“它就是这么工作！”这一概念来自防御式驾驶。在防御式驾驶中，要建立这样一种思维，那就是你永远也不能确定另一位司机将要做什么。这样才能确保在其他人做出危险动作时，你也不会受到伤害。你要承担起保护自己的责任，哪怕是其他司机犯的错误。防御式编程的主要思想是：子程序应该不因传入错误数据而被破坏，哪怕是由其他子程序产生的错误。更一般地说，其核心想法是要承认程序都会有问题，都需要被修改，聪明的程序员应该根据这一点来编程。

8.1 保护程序免遭非法输入数据的破坏

通常有三种方法来处理进来垃圾的情况：

- 检查所有来源于外部的数据的值：当从文件、用户、网络或其他外部接口中获取数据时，应检查所获得的数据值，以确保它在允许的范围内。

- 检查子程序所有输入参数的值：数据来自于其他子程序而非外部接口。
- 决定如何处理错误的输入数据：后面介绍。

防御式编程的最佳方式，就是在一开始不要再代码中引入错误。使用迭代式设计、编码前先写伪代码、写代码前先写测试用例、低层设计检查等活动，都有助于防止引入错误。因此，要在防御式编程之前，优先运用这些技术。

8.2 断言

断言是指在开发期间使用的、让程序在运行时进行自检的代码，通常是一个子程序或宏。断言为真，则表明程序运行正常；断言你为假，则意味着它已经在代码中发现了意料之外的错误。一个断言通常含有两个参数：一个描述假设为真时的情况的布尔表达式，和一个断言为假时需要显示的信息。下面是假定变量 denominator 的值应为非零值时 Java 断言的写法：

Java示例：断言

```
assert denominator !=0 : "denominator is unexpectedly equal to 0.";
```

这个断言声明 denominator 不会等于 0。其中第一个参数，denominator !=0，是个布尔表达式，其结果为 true 或 false。第二个参数是当第一个参数为 false 时，即断言为假时，要打印的消息。

断言可以检查如下这类假定：

- 输入参数或输出参数的取值处于预期的范围内；
- 子程序开始或结束执行时，文件或流处于打开或关闭的状态；
- 子程序开始或结束执行时，文件或流的读写位置处于开头或结尾处；
- 文件或流已用只读、只写或可读可写方式打开；
- 仅用于输入的变量的值没有被子程序所修改；
- 指针非空；
- 传入子程序的数组或其他容器至少能容纳 X 个数据元素；
- 表已初始化，存储着真实的数值；
- 子程序开始或结束执行时，某个容器是空的或满的；
- 一个经过高度优化的复杂子程序的运算结果，和相对缓慢但代码清晰的子程序，的运算结果相对一致。

正常情况下，你并不希望用户看到产品代码中的断言信息；断言主要用于开发和维护阶段。通常，断言只是在开发阶段编译到目标代码中，而在生成产品代码时并不编译进去。在开发阶段，断言可以帮助查清相互矛盾的假定、预料之外的情况，以及传给子程序的错误数据等。在生成产品代码时，可以不把断言编译进目标代码里去，以免降低系统的性能。

（1）建立自己的断言机制

如果你用的语言，不直接支持断言语句，自己写也是很容易的。C++ 中标准的 assert 宏并不支持文本信息。下面的例子给出了一个使用 C++ 宏改进的 ASSERT 实现：

C++示例：一个实现断言的宏

```
#define ASSERT( condition, message ) {      \
    if( !(condition) ) {                    \
        LogError( "Assertion failed: ",     \
            #condition, message );          \
        exit( EXIT_FAILURE );               \
    }
}
```

（2）断言的指导建议

- 用错误处理代码来处理预期会发生的状况，用断言来处理绝对不应该发生的状况：断言用来检查永远不该发生的情况，而错误处理代码，是用来检查不太可能经常发生的非正常情况，这些情况是能在写代码时，就预料到的，且在产品代码中也要处理这些情况。错误处理通常用来检查有害的输入数据，而断言时用来检查代码中的 bug。

- 避免把需要执行的代码放到断言中：如果把代码写在断言里，那么当你关闭断言功能时，编译器很可能把这些代码排除在外了。例如，

```
Visual Basic示例：一种危险的断言使用方法
Debug.Assert( Performance() ) ' Couldn't perform action
```

这段代码的问题在于，如果未编译断言语句，那么其中用于执行操作的代码也就不会被编译。应该把需要执行的语句提取出来，并把其运算结果赋给状态变量，再对这些状态变量进行判断。例如，

```
Visual Basic示例：一种危险的断言使用方法
actionPerformed = PerformAction()
Debug.Assert( actionPerformed ) ' Couldn't perform action
```

- 用断言来注解并验证前条件和后条件：前条件和后条件是一种“契约式设计”的程序设计和开发方法的一部分。前条件是子程序或类的调用方代码，在调用子程序或实例化对象之前，要确保为真的属性。前条件是调用方代码对其所调用的代码要承担的义务。后条件是子程序或类，在执行结束后，要确保为真的属性。后置条件是子程序或类对调用方代码所承担的责任。下面例子，使用了断言来说明 Velocity 子程序的前条件和后条件：

```
Visual Basic示例：使用断言来记述前条件和后条件
Private Function Velocity ( _
        ByVal latitude As Single, _
        ByVal longitude As Single, _
        ByVal elevation As Single, _
    ) As Single

    ' Preconditions
    Debug.Assert ( -90 <= latitude And latitude <=90 )
    Debug.Assert ( 0 <= longitude And longitude < 360 )
    Debug.Assert ( -500 <= elevation And elevation <= 75000 )
    ...

    ' Postconditions
    Debug.Assert ( 0 <= returnVelocity And returnVelocity <= 600 )

    ' return value
    Velocity = returnVelocity
End Function
```

如果变量 latitude、longtitude 和 elevation 都是来源于系统外部，那么就应该用错误处理代码来检查和处理非法的数值，而不是使用断言。而如果变量的值源于可信的系统内部，并且这段程序是基于这些值不会超过合法范围的假定而设计，使用断言则是非常合适的。

- 对于高健壮性的代码，应该先使用断言再处理错误：对于每种可能出错的条件，通常子程序要么使用断言，要么使用错误处理代码来进行处理，但是不会同时使用二者。然而，现实世界中的程序和项目通常都很混乱，仅依赖断言是不够的，而应该先使用断言再处理错误。下面例子如何把这一规则应用到 Velocity，

Visual Basic示例：使用断言来说明前条件和后条件

```
Private Function Velocity (_
    ByRef latitude As Single, _
    ByRef longitude As Single, _
    ByRef elevation As Single, _
) As Single

' Precondition
Debug.Assert ( -90 <= latitude And latitude <= 90 )
Debug.Assert ( 0 <= longitude And longitude <360 )
Debug.Assert ( -500 <= elevation And elevation <= 75000 )
...
' Santitize input data.Values should be within the ranges asserted above,
' but if a value is not within its valid range, it will be changed to the
' closed legal value
If ( latitude < -90 ) Then
    latitude =-90
ElseIf ( latitude > 90 ) Then
    latitude = 90
End If
If ( longitude < 0) Then
    longitude = 0
ElseIf ( longitude > 360 ) Then
...
```

8.3 错误处理技术

一些可用的错误处理技术：

- 返回中立值：有时，处理错误数据的最佳做法，就是继续执行操作，并简单返回一个没有危害的数值。比如，数值计算可以返回 0，字符串操作可以返回空字符，指针操作可以返回空指针等等。
- 换用下一个正确的数据：在处理数据流的时候，有时只需要返回下一个正确的数据即可。
- 返回与前次相同的数据；
- 换用最接近的合法值：例如温度计校准到 0 到 100，如果有次读书小于 0，可返回 0 代替；
- 把警告信息记录到日志文件中：在检测到错误数据的时候，可以选择在 log 文件中记录一条警告信息，然后继续执行；
- 返回一个错误码：可以决定只让系统的某些部分处理错误，其他部分则不在本地处理错误，而只是简单地报告说有错误发生，并信任调用链上游的某个子程序会处理该错误。通知系统其余部分已经发生错误可以采用下列方法之一：
  - 设置一个状态状态变量的值
  - 用状态值作为函数的返回值
  - 用语言内建的异常机制抛出一个异常

在这种情况下，与确定特定的错误报告机制相比，更为重要的是要决定系统里的哪部分应该直接处理错误，哪些部分只是报告发生的错误。

- 调用错误处理子程序或对象：把错误处理都集中在一个全局的错误处理子程序中，这种方法的优点在于能把错误处理的职责，集中到一起，从而让调试工作更为简单。而代价则是整个程序都要知道这个集中点，并与之紧密耦合。
- 当错误发生时显示出错消息：这种方法把错误处理的开销减到最少，然而它也可能会让用户界面中出现的信息，散布到整个应用程序中。
- 用最妥当的方式在局部处理错误：一些设计方案要求在局部解决所有遇到的错误，而具体使用何种错误处理方法，则留给设计和实现会遇到错误的这部分系统的程序员来决定。
- 关闭程序：有些系统一旦检测到错误发生，就会关闭，这一方法适用于人身安全攸关的应用程序。

（1）健壮性与错误性

正确性意味着永不返回不正确的结果，哪怕不反悔结果，也比返回不正确的结果好；然而，健壮性则意味着要不断尝试采取某些措施，以保证软件可以继续地运转下去，哪怕有时做出一些不够准确的结果。人身安全有关的软件，往往更倾向于正确性而非健壮性。不返回结果也比返回错误的结果要好。放射性治疗仪就是体现这一原则的好例子。消费类应用软件，往往更注重健壮性而非正确性。通常只要返回一些结果就比软件停止运行要强。

（2）高层次设计对错误处理方式的影响

对于错误处理，应该在整个程序里采用一致的方式处理非法的参数。对错误进行处理的方式，会直接关系到软件能否满足在正确性、健壮性和其他非功能性指标方面的要求。确定一种通用的处理错误参数的方法，是架构层次的设计决策，需要在那里的某个层次上解决。一旦确定了某种方法，就要确保始终如一地贯彻这一方法。这些指导建议对于系统函数和你自己写的函数都是成立的。除非你已确定了一套不对系统调用进行错误检查的架构性指导建议，否则潜在每个系统调用后检查错误代码。一旦检测到错误，就记下错误代号和它的描述信息。

8.4 异常

异常是把代码中的错误或异常事件，传递给调用方代码的一种特殊手段。如果在一个子程序中遇到了预料之外的情况，但不知道该如何处理的话，就可以抛出一个异常。对出错的前因后果不甚了解的代码，可以把对控制权转交给系统中其他能更好地解释错误并采取措施的部分。异常和继承有一点是相同的，用得好，可以降低复杂度；用得不好，只会让代码变得几乎无法理解。下面是一些建议：

- 用异常通知程序的其他部分，发生了不可忽略的错误：异常机制的优越之处，在于它能提供一种无法被忽略的错误通知机制。
- 只在真正例外的情况下才抛出异常；
- 不能用异常来推卸责任：如果某种错误情况可以在局部处理，那就应该在局部处理它。
- 避免在构造函数和析构函数中抛出异常，除非你在同一地方把它们捕获：在 C++ 里，只有对象已完全构造之后，才可能调用析构函数，也就是说，如果在构造函数的代码中抛出异常，就不会调用析构函数，从而造成潜在的资源泄露。
- 在恰当的抽象层次抛出异常：当你决定把一个异常传给调用方时，请确保异常的抽象层次，与子程序接口的抽象层次是一致的。这个例子说明了应该避免什么的做法，

```
Java反例：抛出抽象层次不一致的异常类
class Employee {
    ...
    public TaxId GetTaxId() throws EOFException {
        ...
    }
    ...
}
```

GetTaxId() 把更低层的 EOFException（文件结束，end of file）异常返回给了它的调用方。它本身并不拥有这一异常，但却通过把更低层的异常，传递给其调用方，暴露了自身的一些实现细节。这就使得子程序的调用方代码不是与 Employee 类的代码耦合，而是比 Employee 类层次更低的抛出 EOFException 异常的代码耦合起来了。这样既破坏了封装性，也减低了代码的可管理性。

- 在异常消息中加入关于导致异常发生的全部信息：每个异常都是发生在代码抛出异常时，所遇到的特殊情况下。这一信息对于读取异常消息的人们来说，是很有价值的，因此要确保该消息中含有为理解异常抛出原因所需的信息。如果异常是因为一个数值的下标错误而抛出的，就应在异常消息中包含数组的上界、下界以及非法的下标值等信息。

- 避免使用空的 catch 语句：有时你可能会试图敷衍一个不知该如何处理的异常，比如，

```
Java示例：忽略异常的错误做法
try {
    ...
    // 很多代码
    ...
} catch ( AnException exception ) {}
```

这种做法就意味着，要么是 try 里的代码不对，因为它无故抛出了一个异常；要么是 catch 里的代码不对，因为它没能处理一个有效的异常。确定一下错误产生的根源，然后修改 try 或 catch 二者其一的代码。偶尔你也可能会遇到某个较低层次上的异常，它确实无法表现为调用方抽象层次上的异常。如果确实如此，至少需要写清楚为什么采用 catch 语句是可行的。你也可以用注释或向日志文件中记录信息来对这一情况进行“文档化”，例如

```
Java示例：忽略异常的错误做法
try {
    ...
    // 很多代码
    ...
} catch ( AnException exception ) {
    LogError( "Unexpected exception" );
}
```

- 了解所用函数库可能抛出的异常：如果你所用的编程语言不要求子程序或类定义它可能抛出的异常，那你一定要了解所用的函数库，都会抛出哪些异常。

- 考虑创建一个集中的异常报告机制：有种方法可以确保异常处理的一致性，即创建一个集中的异常报告机制。这个集中报告机制能够为一些与异常有关的信息，提供一个集中的存储，如所发生的异常种类、每个异常该如何处理以及

- 把项目中对异常的使用标准化：为了保存异常处理尽可能便于管理，可以用以下几种途径，将异常的使用标准化：

  - 如果你在使用一种像 C++ 这样的语言，其中允许抛出多种多样的对象、数据及指针的话，那么就应该为到底可以抛出哪些种类的异常，建立一个标准。为了与其他语言相兼容，可以考虑只抛出从 std::exception 基类派生出的对象。
  - 考虑创建项目的特定异常类，它可用做项目中所有可能抛出的异常的基类。
  - 规定在何种场合允许代码使用 throw-catch 语句，在局部对错误进行处理。
  - 规定在何种场合允许代码抛出不在局部进行处理的异常。
  - 确定是否要使用集中的异常报告机制。
  - 规定是否允许在构造函数和析构函数中使用异常。

- 考虑异常的替换方案。

最后，请考虑你的程序是否真的需要处理异常。有人指出，应对程序运行时发生的严重错误，最佳做法有时就是释放所有已获得的资源并终止程序执行，而让用户去重新用正确的输入数据再次运行程序。

8.5 隔离程序，使之包容由错误造成的损害

隔栏是一种容损策略，就像建筑物里的防火墙一样。以防御式编程为目的而进行隔离的一种方法，是把某些接口选定为“安全”区域的边界。对穿越安全区域边界的数据，进行合法性校验，并当数据非法时，做出敏锐的反应。下图展示了这一概念，同样地可以在类的层次采用这种方法。类的公用方法可以假设数据是不安全的，它们要负责检查数

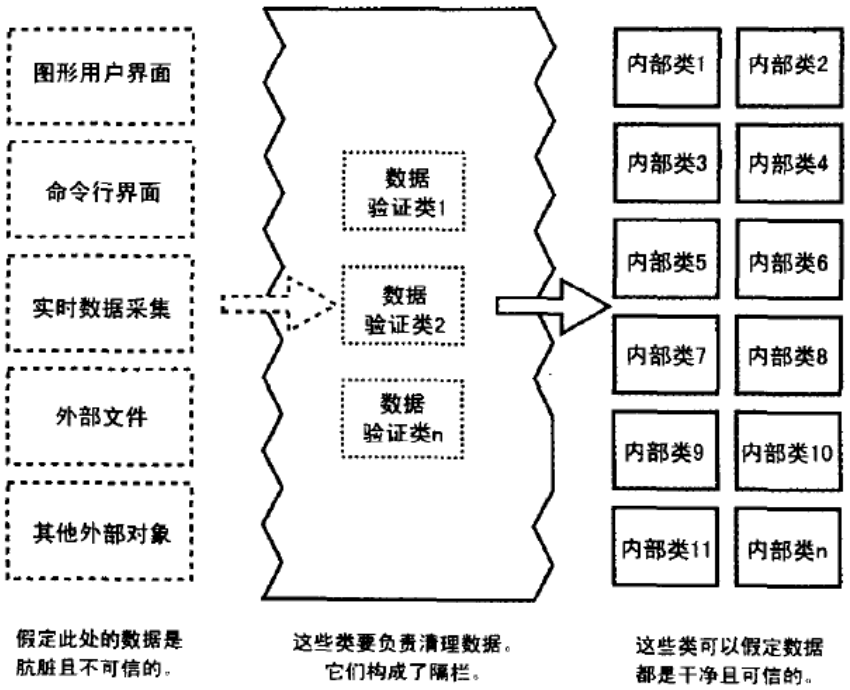


图8-2 让软件的某些部分处理“不干净的”数据，而让另一些部分处理“干净的”数据，即可让大部分代码无须再担负检查错误数据的职责

据并进行清理。一旦类的公用方法接受了数据，那么类的私用方法就可以假定数据都是安全的了。

- 在输入数据时将其转换为恰当的类型：输入的数据通常都是字符串或数字的形式。这些数据有时要被映射为“是”或“否”这样的布尔类型，有时要被映射为像 Color_Red、Color_Green 和 Color_Blue 这样的枚举类型。在程序中长时间传递类型不明的数据，会增加程序的复杂度和崩溃的可能性，比如有人在需要输入颜色枚举值的地方，输入了“是”，因此，应该在输入数据后立即将其转换到恰当的类型。

隔栏的使用使断言和错误处理，有了清晰的区分。隔栏外部的程序应使用错误处理技术，在那里对数据做的任何假定都是不安全的。而隔栏内部的程序里，就应使用断言技术，因为传进来的数据应该已通过隔栏时，被清理过了。如果隔栏内部的某个子程序检测到了错误的数据，那么这应该时程序里的错误，而不是数据里的错误。

8.6 辅助调试的代码

防御式编程的另一重要方面是使用辅助调试代码，辅助调试代码非常强大，可以帮助快速地检测错误。

（1）不要自动地把产品版地限制，强加于开发版本之上

应该在开发期间，牺牲一些速度和对资源的使用，来换取一些可以让开发更顺畅的内置工具。

（2）尽早地引入辅助调试的代码

通常，除非被某个错误反复地纠缠，否则你是不愿意花精力，去编写一些调试辅助地代码地；然而，如果你一遇到问题，就马上编写或使用前一个项目中，用过的某个调试助手的话，它就会自始至终在整个项目中帮助你。

（3）采用进攻式编程

进攻式编程，是指以这样的方式处理异常情况：在开发阶段，让它显现出来，而在产品代码运行时，让它能够自我恢复。假设有一段 case 语句，期望用它处理 5 类事件。在开发期间，应该让针对默认情况的 case 分支（即 default case 子句）显示警告信息；然而，在最终的产品代码里，针对默认情况的处理，则应更稳妥一些，例如可以在错误日志文件中，记录该消息。下面列出一些可以让你进行进攻式编程的方法：

- 确保断言语句使程序终止运行。
- 完全填充分配到的所有内存，这样可以让你检测到内存分配错误。
- 完全填充已分配到的所有文件或流，这样可以让你排查出文件格式错误。
- 确保每一个 case 语句中的 default 分支，或 else 分支，都能产生严重错误，比如说让程序终止，或者至少让这些错误不会被忽视。
- 在删除一个对象前，把它填满垃圾数据。
- 让程序把它的错误日志文件，用电子邮件发给你，这样就能了解到在已发布的软件中，还发生了哪些错误，如果这对你所开发的软件适用的话。

（4）计划移除调试辅助的代码

事先做好计划，避免调试用的代码，和程序代码纠缠不清。下面使一些可以采用的方法：

- 使用类似 ant 和 make 这样的版本控制工具和 make 工具：版本控制工具可以从同一套源码，编译出不同版本的程序。在开发模式下，你可以让 make 工具把所有的调试代码都包含进来一起编译。而在产品模式下，又可以让 make 工具把那些你不希望包含在商用版本中的调试代码排除在外。
- 使用内置的预处理器：可以用编译器开关来包含或排除调试用的代码。你既可以直接使用预处理器，也可以写一个与预处理器指令同时使用的宏。例如

C++示例：直接使用预处理器来控制调试用的代码

```
#define DEBUG
...
#if defined( DEBUG )
// debugging code
...
#endif
```

如果你不喜欢让 #if defined() 这样的语句，散布在代码里的各处，那么可以写一个预处理器宏，来完成同样的任务。例如：

C++示例：使用预处理器宏，来控制调试用的代码

```
#define DEBUG
#if  defined( DEBUG )
#define DebugCode( code_fragment ) { code_fragment }
#else
#define DebugCode( code_fragment )
#endif
...
DebugCode(
    statement 1;
    statement 2;
    ...
```

```
        statement n;
    );
    ...
```

- 编写你自己的预处理器：如果某种语言没有包含一个预处理器，可以很容易自己写一个，用于包含或排除调试代码。
- 使用调试存根：很多情况下，可以调用一段子程序进行调试检查。在开发阶段，该子程序可能要执行若干操作之后，才把控制权交还给其调用方代码。而在产品代码里，可以用一个存根子程序来替换这个复杂的子程序，而这段 stub 子程序要么立即把控制权交还调用方，要么使执行几项快速的操作就返回。这种方法金辉带来很小的性能损耗，并且比自己编写预处理器要快一些。把开发版本和产品版本的 stub 子程序都保留起来，以便将来可以随时在两者之间来回切换。例如，可以先写一个检查出入的指针是否有效的子程序：

C++示例：一段使用调试stub的子程序

```
void DoSomething(
    SOME_TYPE *pointer;
    ...
    ) {

    // check parameters passed in
    CheckPointer( pointer )
    ...
}
```

在开发阶段，CheckPointer() 子程序会对传入的指针，进行全面检查。这一检测可能相当耗时，但一定要非常有效，比如这样：

C++示例：在开发阶段检查指针的子程序

```
void CheckPointer( void *pointer ) {
    // 执行第1项检查：可能是检查它不为NULL
    // 执行第2项检查：可能是检查它的地址是合法的
    // 执行第3项检查：可能是检查它所指向的数据完好无损
    ...
    // 执行第n项检查：...
}
```

当代码准备妥当，即将要编译为产品时，你可能不希望这项指针检查，影响性能。这是你就可以用下面子程序，来代替前面的那段代码：

C++示例：在产品代码中检查指针的子程序

```
void CheckPointer( void *pointer ) {
    // no code; just return to caller
}
```

8.7 确定在产品代码中该保留多少防御式代码

下面是一些指导建议，帮助决定哪些防御式编程工具可以留在产品代码里，而哪些应该排除在外：

- 保留那些检查重要错误的代码：你需要确定程序的哪些部分，可以承担未检测出错而造成的后果，而哪些部分不能承担；

- 去掉检查细微错误的代码：如果一个错误带来的影响确实微乎其微的话，可以把检查它的代码去掉。
- 去掉可以导致程序硬件崩溃的代码：如果程序里存在可能导致数据丢失的调试代码，一定要把它们从最终软件产品中去掉。
- 保留可以让程序稳妥地崩溃地代码：如果程序里有能够检测出潜在严重错误地调试代码，那么应该保留那些能让程序稳妥地崩溃地代码。
- 为你地技术支持人员记录错误信息：如果开发时在代码里大量使用了断言来中止程序执行，那么在发布产品时，可以考虑把断言子程序改为向日志文件中记录信息，而不是彻底去掉这些代码。
- 确认留在代码中的错误信息是友好的：如果你在程序中留下了内部错误消息，请确认这些消息的用语对用户而言是友好的。

8.8 对防御式编程采取防御的姿态

过度的防御式编程也会引起问题。如果你在每一个能想到的地方，用每一种能想到的方法检查从参数传入的数据，那么你的程序将会变得臃肿而缓慢。更糟糕的是，防御式编程引入的额外代码，增加了软件的复杂度。防御式编程引入的代码也许并不会有缺陷，和其他代码一样，你同样能轻而易举地，在防御式编程添加地代码中，找到错误，尤其是当你随手编写这些代码时，更是如此。因此，要考虑好什么地方需要进行防御，然后因地制宜地调整，进行防御式编程地优先级。

9 伪代码编程过程

9.1 创建类和子程序地步骤概述

创建一个类可以有多种不同的方式，但一般而言，都是一个迭代过程：先对一个类做总体设计，列出这个类内部的特定子程序，创建这些子程序，然后从整体上复查这个类的构建结果。如下图所示：

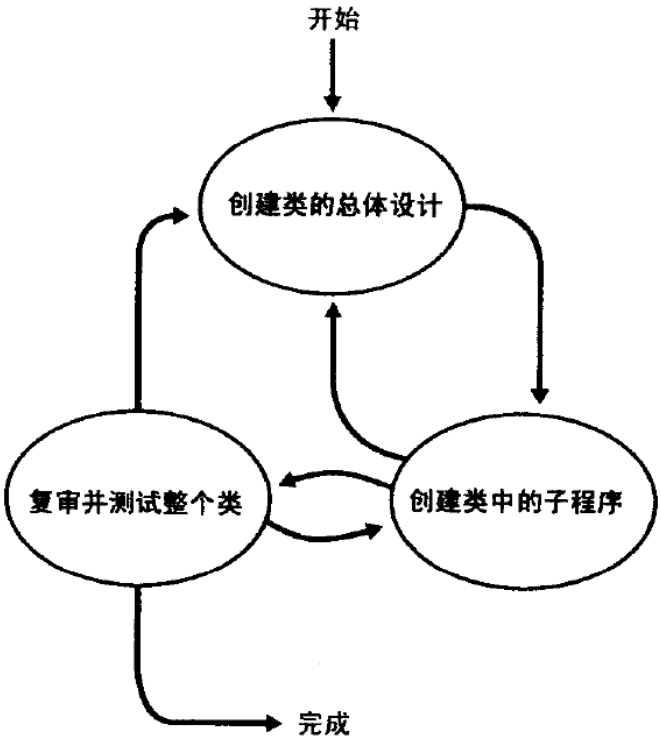


图 9-1 一个类的创建过程可以千变万化，但基本上会以本图所示的顺序发生

（1）创建一个类的步骤

- 创建类的总体设计：设计一个类的过程中，包含一些特有的设计任务：定义类的特定职责，定义类所要隐藏的秘密，以及精确地定义类的接口所代表的抽象概念，决定这个类是否要从其他类派生而来，以及是否允许其他类再从它派生，指出这个类中关键的公用方法，标识并设计出类所需用到的重要数据成员。上述这些设计任务可能需要反复迭代，直到能直截了当地设计出子程序为止。
- 创建类中地子程序：在标识出类的主要子程序后，还要创建这些子程序。在编写各个程序时，通常还会引出更多的或重要、或次要的子程序，创建这些新加入的子程序的过程，往往还会反过来波及类的总体设计。
- 复审并测试整个类：通常情况下，子程序在创建的同时，也经过了测试。在整个类可以工作之后，应该再对其整体进行复查和测试，以便发现那些在子程序的独立测试层次上无法测出的问题。

（2）创建子程序的步骤

在创建子程序的过程中，涉及到的主活动：设计子程序、检查设计、编写子程序地代码、检查代码，通常会以下图所示的顺序进行，

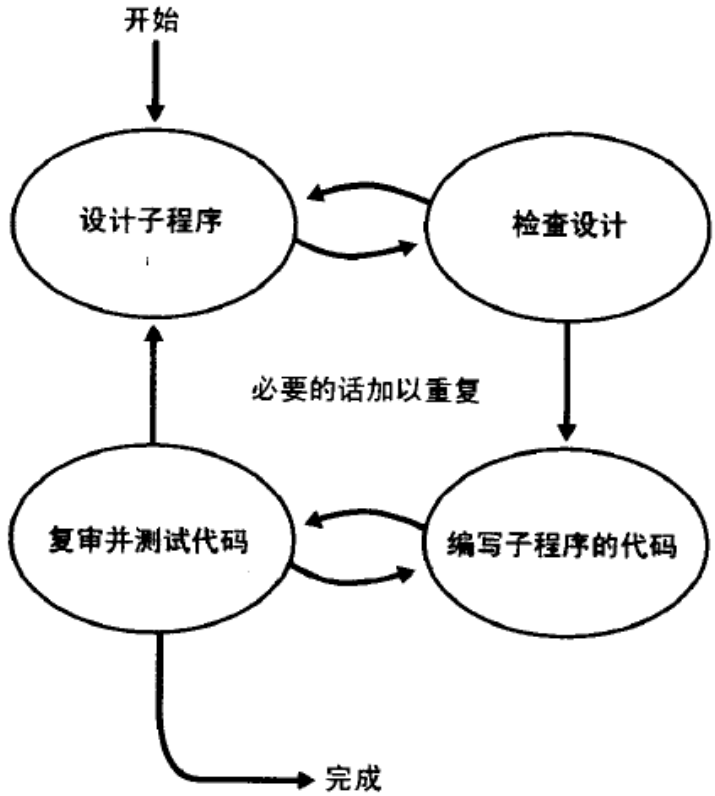


图 9-2　这些是创建一个子程序所需经历的主要活动，常是以图示的顺序进行

9.2　伪代码

伪代码是指某种用来描述算法、子程序、类或完整程序的工作逻辑、非形式的、类似于英语的记法。伪代码编程过程，则是一种通过书写伪代码而更高效地创建程序代码地专门方法。下面是一些有效使用伪代码地指导原则，

- 用类似英语的语句来精确描述特定的操作。
- 避免使用目标编程语言中的语法元素。
- 在意图的层面上编写伪代码。用伪代码去描述解决问题的方法的意图，而不是去写如何在目标语言中实现这个方法。
- 在一个足够低的层次上编写伪代码，以便可以近乎自动地从它生成代码。如果伪代码的层次太高，就会掩盖代码中的问题细节。应该不断地精细化伪代码，加入越来越多的细节，直到看起来已经很容易直接写出代码为止。

下面通过例子展示不好的伪代码和好的伪代码：

```
一段不好的伪代码示例
increment resource number by 1
allocate a dlg struct using malloc
if malloc() returns NULL then return 1
invoke OSrsrc_init to initialize a resource for the operating system
*hRsrcPtr = resource number
return 0
```

差的原因：包含许多目标语言编码的细节，例如 *hRsrcPtr 和 malloc() 是 C 语言的特色；这段伪代码太关注于如何编写代码，而没有突出设计意图；另外它还深入到了编码的细节，即这个子程序是返回 1 还是 0。

```
一段好的伪代码示例
Keep track of current number of resource in use
  If another resource is available
    Allocate a dialog box structure
    If a dialog box structure could be allocated
        Note that one more resource is in use
        Initialize the resource
        Store the resource number at the location provided by the caller
    Endif
  Endif
Return true if a new resource was created; else return false
```

使用这种风格的伪代码，可以得到下面这些好处：

- 伪代码使得评审更容易。你无须检查源代码就可以评审细节设计。
- 伪代码支持反复迭代精细化的思想。从一个高层设计开始，把这一设计精细化为伪代码，然后再把伪代码精细化为源代码。
- 伪代码使得变更更加容易。短短几行伪代码，要比整页的代码跟容易修改。
- 伪代码能使给代码作注释的工作量减到最少。。在伪代码编程过程中，伪代码中的语句，将会变为代码中的注释。
- 伪代码比其他形式的设计文档更容易维护。使用其他方法时，设计和代码是分离的，当其中之一变动的时候，两者就不再一致。而使用伪代码编程过程时，伪代码中的语句将会转变为代码中的注释。因此只要维护代码间的这些注释，那么这些伪代码所形成的设计文档就仍然时准确的。

9.3 通过伪代码编程过程创建子程序

假如你要写一个子程序，它能根据错误码输出错误信息，称它为 ReportErrorMessage()。下面是 ReportErrorMessage() 程序的一个非形式的规格说明：ReportErrorMessage() 接收一个代表错误码的输入参数，输入与该错误码相对应的错误信息，它应该能够处理无效的错误码。如果程序是以交互式界面运行的，那么 ReportErrorMessage() 需要向用户显示错误信息；如果程序是以命令行方式运行的，那么 ReportErrorMessage() 应把错误信息记录在一个消息文件里。在输出错误信息之后，ReportErrorMessage() 应返回一个状态值，以表明其操作时成功还是失败。

（1）设计子程序

- 检查先决条件：在动手去做子程序本身的任何工作之前，应该先查看一下该子程序要做的工作是不是已经定义好了，是不是能够与整体设计相匹配。另外要结合项目的需求，检查这个程序是否时真正必需的，至少是间接需要的。

- 定义子程序要解决的问题：陈述该子程序将要解决的问题，叙述要足够详细，以便能去创建这个子程序。如果高层设计已经足够详细，那么这项工作可能已经完成了。在这个高层的设计里，至少应该详细说明下列信息：
  - 这一子程序将要隐藏的信息。
  - 传给这个子程序的各项输入。
  - 从该子程序得到的输出。
  - 在调用程序之前确保有关的前条件成立，如输入数据的取值位于特定的范围之内、有关的流程已经初始化、文件已经打开或关闭、缓冲区已经填满或清空等。

  下面看看在 ReportErrorMessage() 示例中是如何考虑这些问题的：
  - 该子程序隐藏了两项事实：错误信息的文本和当前的处理方式，交互式界面或命令行。
  - 对于这个子程序，没有任何可保证的前条件。
  - 给该子程序的输入数据是一个错误码。
  - 存在两种输出：首先是错误信息，其次是 ReportErrorMessage() 返回给调用方程序的状态值。
  - 该子程序保证状态值为 Success 或 Failure。
- 为子程序命名：一般来说，子程序应该有一个清晰、无歧义的名字。如果你在给程序起个好名字的时候犯难，通常就表明这个子程序的目标还没明确。
- 决定如何测试子程序：在编写一个子程序的时候，要想一想怎么才能测试它。
- 在标准库中搜寻可用的功能：要想提高代码的质量和生产率，一个重要的途径就是重用好的代码。
- 考虑错误处理：考虑在子程序中所有可能出错的环节。
- 考虑效率问题：根据所确定的资源及速度的目标来设计子程序。如果速度看上去更为重要，那么就牺牲一部分资源来换取速度，反之亦然。在每个子程序上为效率问题卖力通常是白费功夫，最主要的优化，还是在于完善高层设计，而不是完善每个子程序。
- 研究算法和数据类型：在决定从头开始编写一段复杂的代码之前，查一下书法书里有什么可用的内容。
- 编写伪代码：在代码编辑工具或集成开发环境里写伪代码就可以了，因为很快就要用这些伪代码作为编程语言写的实际编码的基础。从最一般情况写起，向着更具体的细节展开工作。子程序最常见的部分是一段头部注释，用于描述这段程序应该做什么，所以首先简要地用一句话来写下该子程序的目的。一般而言，如果很难总结出一个子程序的角色，你可能就应该考虑是否什么环节出问题了。下面的例子是描述一个子程序的伪代码示例

```
一个子程序的伪代码示例
This routine outputs an error message based on an error code
supplied by the calling routine. The way it outputs the message
depends on the current processing state, which it retrievess
on its own. It returns a value indicating success or failure.

set the default statues to "fail"
look up the message based on the error code
if the error code is valid
    if doing interactive processing, display the error message
    interactively and declare success
    if doing command line processing, log the error message to the
    command line and declare success
if the error code isn't valid, notify the user that an internal
```

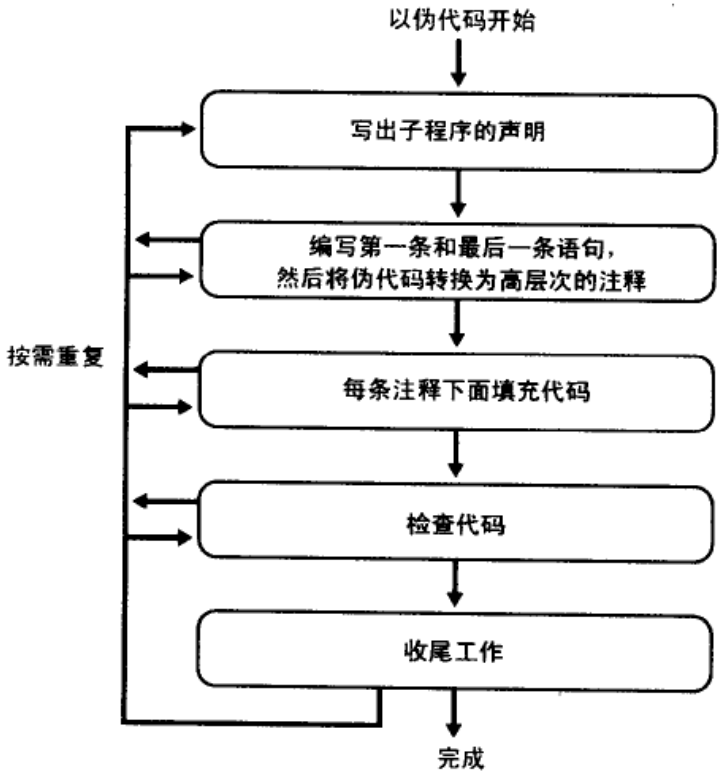


图 9-3　在构建程序的时候，你将实施所有这些步骤，但不一定要按照任何特定的顺序

```
error has been detected.
return status information
```

- 考虑数据：如果对数据的操作是某个子程序的重点，那么值得在考虑子程序的逻辑之前，首先考虑主要的数据部分。把关键的数据类型定义好，对于设计一个子程序的逻辑很有用。
- 检查伪代码：请确认你能很容易、很自然地理解这个子程序做些什么，以及它是怎样做地。、
- 在伪代码中试验一些想法，留下最好的想法（迭代）：在你开始编写代码之前，应尽可能用伪代码去尝试更多的想法。一旦你真正开始编码，你和你所写下的代码就会有感情，从而更难以抛弃不好的设计再重头来过了。

（2）编写子程序的代码
构建子程的步骤如下，

- 写出子程序的声明：首先写出子程序的接口声明，例如 C++ 中的函数声明。把原有的头部注释变为编程语言中的注释。把它保留在你写的伪代码的上方。

C++示例：向伪代码添加子程序接口声明和头部注释

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

Status ReportErrorMessage(
    ErrorCode errorToReport
)
set the default statues to "fail"
```

```
look up the message based on the error code
if the error code is valid
    if doing interactive processing, display the error message
    interactively and declare success
    if doing command line processing, log the error message to the
    command line and declare success
if the error code isn't valid, notify the user that an internal
error has been detected.
return status information
```

- 把伪代码转变为高层次的注释：接下来，把第一条和最后一条语句写出来，在 C++ 中也就是“{”和”}”。然后把伪代码转变为注释。

```
C++示例：在伪代码首尾写出第一条和最后一条语句
/* This routine outputs an error message based on an error code
supplied by the calling routine. The way it outputs the message
depends on the current processing state, which it retrievess
on its own. It returns a value indicating success or failure.
*/

Status ReportErrorMessage(
    ErrorCode errorToReport
) {
    //set the default statues to "fail"
    //look up the message based on the error code
    //if the error code is valid
        //if doing interactive processing, display the error message
        //interactively and declare success
        //if doing command line processing, log the error message to the
        //command line and declare success
    //if the error code isn't valid, notify the user that an internal
    //error has been detected.
    //return status information
}
```

- 在每条注释下面填充代码：每一段注释产生出一行或多行代码。以这些注释为基础，每一代码块都形成了一套完整的思想。这些注释仍然保留下来，从一个更高的层次上对代码做出说明。所有的变量都是在靠近第一次使用的地方来进行声明和定义。每段注释通常应该展开为 2 至 10 行代码。

```
C++示例：在伪代码首尾写出第一条和最后一条语句
/* This routine outputs an error message based on an error code
supplied by the calling routine. The way it outputs the message
depends on the current processing state, which it retrievess
on its own. It returns a value indicating success or failure.
*/

Status ReportErrorMessage(
    ErrorCode errorToReport
```

```
) {
    //set the default statues to "fail"
    Status errorMessageStatus = Status_Failure;

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

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

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

        //if doing command line processing, log the error message to the
        //command line and declare success
        else if ( errorProcessingMethod ==
            ProcessingMethod_CommanLine ) {
                CommandLine messageLog;
                if ( messageLog.Status() == CommondLineStatus_OK ) {
                    messageLog.AddToMessageQueue( errorMessage.Text() );
                    messageLog.FlushMessageQueue();
                    errorMessageStatus = Status_Success;
                } else {
                    // can't do anything because the routine is already
                    // error processing
                }
            } else {
                // can't do anything because the routine is already
                // error processing
            }
    }
    //if the error code isn't valid, notify the user that an internal
    //error has been detected.

    else {
        DisplayInteractiveMessage(
            "Internal Error: Invalid error code in ReportErrorMessage()"
        );
    }
```

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

- 检查代码是否需要进一步分解：有时候，你会发现几行伪代码展开后，形成大量的代码。在这种情况下，应该考虑如下两种方法中的一种：
  - 把这段注释下的代码重构成一个新的子程序。
  - 递归地应用伪代码编程过程。把原来的那一行伪代码分解为更多行的伪代码，然后再在新写出的伪代码下面填入代码。

（3）检查代码

- 在脑海中检查程序中的错误：在脑海中执行每一条代码路径，确保检查到了所有可能的执行路径、端点和所有异常条件。底线：只是写出一个可以工作的子程序是不够的，如果你不知道它为什么可以工作，那就去研究它，讨论它，用其他的设计方案做试验，直到你弄明白为止。
- 编译子程序：本书的一个目的就是告诉你，怎样脱离那种先东拼西凑，然后通过运行来看看代码是否工作的怪圈，不要挣扎于“拼凑、编译、修改”的开发工作中。下面一些指导建议，可以最大限度地发挥编译子程序所产生的功效：
  - 把编译器的警告级别调到最高，让编译器来检测错误，可以很容易地查出大量细微的错误。
  - 使用验证工具。可以通过使用类似 lint 这样的工具，对 C 语言这类语言的编译器所作的检查结果，进行补充检查。
  - 消除产生错误消息和警告的所有根源。通常，大量的告警信息暗示着代码的质量欠佳，你需要尽量理解所得到的每一个告警。通过重写代码来解决潜在问题并消除告警信息，是更保险也更省力的。
- 在调试器中逐行执行代码。
- 测试代码。使用你在开发该子程序期间，计划写的或已写成的测试用例，来测试代码。
- 消除程序中的错误。一旦检测到错误，就一定要把它除掉。如果你发现一段程序的毛病不是一般的多，那请、从头再来吧。修修补补通常表明你还未能全面地理解程序，这样也必将不时地产生错误。对于一个毛病百出的程序而言，设计一个全新的方案是值得的。

（4）收尾工作

可以实行若干扫尾步骤来确保子程序的质量合乎标准：

- 检查子程序的接口。确认所有的输入、输出数据都参与了计算，并且所有的参数也都用到了。
- 检查整体的设计质量。确认下列事项：这个子程序只干一件事情，并且把这件事情做得很好；子程序之间是松散耦合的；子程序采用了防御式设计。
- 检查子程序中的变量。检查是否存在不准确的变量名称、未被用到的对象、未经声明的变量，以及未经正确初始化的对象等。
- 检查子程序的语句和逻辑。检查是否存在 off-by-one 这样的错误、死循环、错误的嵌套以及资源泄露。
- 检查子程序的布局。确认你正确地使用了空白来明确子程序、表达式及参数列表的逻辑结构。
- 检查子程序的文档。确认哪些由伪代码转化而来的注释，仍然是准确无误的。检查有关算法的描述、接口假定的说明、那些并非显而易见的依赖性，以及难以理解的编码行为的解释等。

- 除去冗余的注释。

（5）根据需要重复上述步骤

如果这个程序的质量不佳，那就回到伪代码那一步去。高质量的编程是一个迭代的过程，所以不要犹豫，把构建的工作再做一遍。

10 使用变量的一般事项

本章中用“变量”一词同时指代对象和内置数据类型，如整数和数组等。“数据类型”一词通常是内置数据类型，而“数据”一词则可能代表对象，也可能代表内置数据类型。

10.1 数据认知

创建有效数据的第一步是了解所要创建数据的种类。积累大量的数据类型，对于程序员来说是至关重要的。

10.2 轻松掌握变量定义

养成一个良好的变量定义习惯，会为你再整个项目周期内，省去很多时间和麻烦。有些语言支持隐式变量声明。例如 Microsoft Visual Basic 中使用一个未申明变量的时候，编译器会自动为你声明该变量。隐式变量声明对于任何一种语言来说，都是最具危险性的特性之一。例如，当你绞尽脑汁想要明白变量 acctNo 的值为什么不正确，最终却发现是不慎将 acctNo 写成 acctNum，而又将 acctNum 重新初始化为 0。要求显示声明数据的编程语言，实际上是在提醒你要更加仔细地使用这些数据，而这一点也是它们的主要优势之一。如果使用的编程语言支持隐式声明，下面是一些建议：

- 关闭隐式声明。
- 声明全部变量。
- 遵循某种命名规则。
- 检查变量名。

10.3 变量初始化原则

不合理地初始化数据是产生编程错误地常见根源之一。不恰当地变量初始化所导致的一系列问题，都源于变量的默认初始化值，与你的预期不同。以下行为都会产生此类问题：

- 从未对变量赋值。它的值只是程序启动时，变量所处内存区域的值。
- 变量值已经过期。变量在某个地方曾经被赋值，但该值已经不再有效。
- 变量的一部分被赋值，而另一部分没有。你可能初始化了一个对象的部分成员，而不是全部成员。也可能忘记事先分配内存，就去初始化一个未经初始化的指针所指的“变量”。这就意味着你是随机选取了一块内存，然后对其赋值。这块内存可能存放的是数据，也可能存放的是代码，甚至可能指向操作系统内部。指针操作错误可能产生很奇怪的现象，并且每次都不相同，这也导致了调试指针错误，比调试其他错误更困难。

下面是一些避免产生初始化错误的建议：

- 在声明变量的时候初始化：在声明变量的同时对其初始化，是一种非常方便的防御式编程方法，是一种很好的用于防范初始化错误的保险策略。
- 在靠近变量第一次使用的位置初始化它：这是就近原则的一个例子，即把相关的操作放在一起。这一原则也适用于让注释靠近它所描述的代码，让控制循环的代码靠近循环本身，以及把语句写成直线的代码等各个领域。

- 理想情况下，在靠近第一次使用变量的位置声明和定义该变量：声明指定了变量的类型，定义为变量指定特定的取值。在理想情况下，每个变量都应该在声明的同时被定义。
- 在可能的情况下使用 final 或 const：可以防止该变量在初始化后，再被赋值。
- 特别注意计数器和累加器：i、j、k、sum 和 total 等变量常用作计数器或累加器，在下次使用这些变量之前，忘记重置其值，也是一种常见的错误。
- 在类的构造函数里初始化该类的数据成员：类的数据也应该在其构造函数中初始化，如果在构造函数里分配了内存，那么就应该在析构函数中释放这些内存。
- 检查是否需要重新初始化：如果的确需要重新初始化，那么要确保初始化语句位于那些重复执行的代码内部。
- 一次性初始化具名常量，用可执行代码来初始化变量：如果你想用变量来模拟具名常量，那么在程序开始处对常量做一次初始化即可。你可以用一个 Startup() 子程序去初始化它们。
- 使用编译器设置来自动化初始化所有变量：如果你用的编译器支持自动化初始化所有变量的选项，那么请把它打开。这是一种靠编译器完成初始化工作的简单方式。然而，跨平台移植，则会带来问题。因此要确保记下你所使用的编译器设置。
- 利用编译器的告警信息：很多编译器会在你使用了未经初始化的变量的时候发出警告。
- 检查输入参数的合法性：在你把输入数值赋给任何对象之前，要确保这些数值是合理的。
- 使用内存访问检查工具来检查错误的指针。
- 在程序开始时初始化工作内存：把工作内存初始化为一个已知数值，将会有助于发现初始化错误。可以采用下面的任意一种方法，
  - 可以用某种在程序运行前预先填充内存的工具，来把程序的工作内存填充为一个可以预料的值。
  - 如果使用内存填充工具，那么可以偶尔改变一下用来填充的内存的值。有时，这么“晃动”一下程序，也许可以发现一些在在背景环境保持不变的情况下，无法察觉出来的错误。
  - 可以让程序在启动时初始化工作内存。前面所述的使用在程序运行时，预先填充内存的工具的目的，是要暴露缺陷，而这种方法的目的，则是隐藏缺陷。通过每次把工作内存赋以同样的值，就能保证程序不会因内存初始值的随机性而受到影响。

10.4 作用域

作用域或可见性指的是变量在程序内的可见和可引用的范围。下面是一些使用作用域的规则：

（1）使变量引用局部化

那些介于同一变量多个引用点之间的代码，成为“攻击窗口”。可能会由新代码加到这种窗口中，不当地修改了这个变量，或者阅读代码地人，可能会忘记该变量应有地值。一般而言，把对一个变量的引用局部化，即把引用点尽可能集中在一起，总是一种很好的做法。衡量一个变量的不同引用点的靠近程度的一种方法，是计算该变量的“跨度（span）”。例如，

Java示例：变量跨度

```
a = 0;
b = 0;
c = 0;
b = a + 1;
b = b / c;
```

其中 a 的跨度为 2，b 的跨度为 1 和 0，平均跨度为 0.5，c 的跨度为 1。如果这些引用点之间的距离非常远，那你就要迫使阅读者的目光在程序里跳来跳去。因此，把变量的引用点集中起来的主要好处，是提高程序的可读性。

（2）尽可能缩短变量的存活时间

与变量跨度相关的一个概念是“存活时间”，即一个变量存在期间，所跨越的语句总数。变量的存活时间，开始于引用它的第一天语句，结束于引用它的最后一条语句。与跨度类似，应使得对象的存活时间尽可能短，保持短的存活时间的主要好处，也是减小攻击窗口。这样，在你真正想要修改一个变量的那些位置之间的区域，该变量被错误或无意修改的可能性就降低了。短的变量存活时间同样减少了初始化错误的可能。在修改程序的时候，常会把直线型代码（顺序代码）修改为循环，这样就容易忘记远离循环位置的那些初始化代码。通过把初始化代码和循环代码放在一起，就减少了由于修改语句，而导致初始化错误的可能性。变量存活时间还会使代码更具有可读性。阅读者在同一时间内，需要考虑的代码行数越少，也就越容易理解代码。最后，当需要把一个大的子程序拆分为多个小的子程序时，短的变量存活时间也是有价值的。如果你用跨度和生存时间的概念来考虑全局变量，就会发现全局变量的跨度和生存时间都很长，这也是避免使用全局变量的好理由之一。

（3）减小作用域的一般原则

- 在循环开始之前，再去初始化该循环里使用的变量，而不是在该循环所属的子程序的开始处，初始化这些变量。
- 直到变量即将被使用时，再为其赋值。
- 把相关语句放到一起。

C++示例：使用两套变量，使人困惑的做法

```
void SummarizeData(...) {
    ...
    GetOldData( oldData, &numOldData );
    GetNewData( newData, &numNewData );
    totalOldData = Sum( oldData, numOldData );
    totalNewData = Sum ( newData, numNewData );
    PrintOldDataSummary( oldData, totalOldData, numOldData );
    PrintNewDataSummary( newData, totalNewData, numNewData );
    SaveOldDataSummary( totalOldData, numOldData );
    SaveNewDataSummary( totalNewData, numNewData );
    ...
}
```

C++示例：使用两套变量，更容易理解的做法

```
void SummarizeData(...) {
    GetOldData( oldData, &numOldData );
    totalOldData = Sum( oldData, numOldData );
    PrintOldDataSummary( oldData, totalOldData, numOldData );
    SaveOldDataSummary( totalOldData, numOldData );
    ...
    GetNewData( newData, &numNewData );
    totalNewData = Sum ( newData, numNewData );
    PrintNewDataSummary( newData, totalNewData, numNewData );
    SaveNewDataSummary( totalNewData, numNewData );
    ...
}
```

- 把相关语句组提取成单独子程序。在其他相同的情况下，一个更短的子程序中的变量，通常比更长的子程序中的变量，有更小的跨度和存活时间。

- 开始时采用最严的可见性，然后根据需要，扩展变量的作用域。当对变量的作用域犹豫不决时，应该倾向于选择该变量所能具有的最小的作用域：首先将变量局限于某个特定的循环，然后局限于某个子程序，其次成为类的 private 变量，protected 变量，再其次对 package 可见，最后不得已的情况下，在把它作为全局变量。

10.5 持续性

“持续性”是对一项数据的生命周期的另一种描述。持续性具有多种形态：

- 特定代码段或子程序的生命周期。例如在 C++ 或 Java 中的 for 循环里，声明的变量。
- 只要你允许，就会持续下去。例如 C++ 里用 new 创建的变量，会一直持续到 delete 它。
- 程序的生命期。大多数语言的全局变量，都属于这一类，C++ 中的 static 变量也是如此。
- 永远持续。这一类变量可能包括你存储在数据库中、能够在程序的多处执行之间存留的数据。

与持久性相关的主要问题，是变量实际生命周期比想象的要短，而且难以预料。如果你试图在一个变量正常的生命周期结束之后，访问它的数据，那么它的数值还会保持吗？有的时候变量中保持的数值已经改变了，你通过收到错误提示获知这一点。而有时，计算机会把旧的数值留在变量里，使你误认为自己用对了变量。为了避免上述问题，可以采取以下措施：

- 在程序中加入调试代码或断言，来检查那些关键变量的合理取值。如果变量取值变得不合理，就发出警告信息，通知你去寻找是否有不正确的初始化。
- 准备抛弃变量时，给它们赋上“不合理的数值”，例如，可以在删除一个指针后，把它的值设为 null。
- 编写代码时，要假设数据并没有持续性。例如，如果某个变量在你退出某个子程序的时候，具有特定的值，那么当你当你下一次进入该子程序的时候，就不要假定该变量还有同样的数值。
- 养成在使用所有数据之前，声明和初始化的习惯。如果你发现某项数据的使用位置与初始化位置相去甚远，那么就要小心了。

10.6 绑定时间

对程序维护和更改，有很深远影响的一个话题，就是“绑定事件”：把变量和它的值绑定在一起的时间。采用越晚的绑定时间会越有利。

Java示例：在编写代码时绑定其值的变量

```
titleBar.color = 0xFF; // 0xFF is hex value for color blue
```

由于 0xFF 是硬编码在程序里的数值，在编写代码时，它就会绑定到 title.color 变量上。这种硬编码技术通常总是很糟糕的，因为一旦要修改这个 0xFF，那么这个新值就无法同代码中其他那些必须和它一样的 0XFF 值保持一致了。

Java示例：在编译时绑定其值的变量

```
private static final int COLOR_BLUE = 0xFF;
private static final int TITLE_BAR_COLOR = COLOR_BLUE;
...
titleBar.color = TITLE_BAR_COLOR;
```

TITLE_BAR_COLOR 是一个具名常量，编译器会在编译的时候，把它替换为一个数值。如果你用的语言支持这种特性，那么这种方法几乎总要好于硬编码。由于 TITLE_BAR_COLOR 比 0xFF 更能反映出所代表的信息，因此增加了可读性。它也使得修改标题颜色变得更容易，因为一处改动就能对所有位置生效。同时也不会影响运行期的性能。

Java示例：在运行时绑定其值的变量

```
titleBar.color = ReadTitleBarColor();
```

与硬编码相比，上述代码更具可读性和灵活性。无须通过修改程序来改变 titleBar.color，只需简单修改 ReadTitleBarColor() 子程序要读取的数据源内容即可。这种方法常用于允许用户自定义应用程序环境的交互式应用程序。

一般而言，绑定时间越早，灵活性就会越差，但复杂度也会降低。就前两种方案而言，使用具名常量，要在很多方面好于使用魔鬼数字 (magic number)。

10.7 数据类型和控制结构之间的关系

有三种类型的数据和相应控制结构之间的关系：

- 序列数据翻译为程序中的顺序语句。
- 选择型数据翻译为程序中的 if 和 case 语句。
- 迭代型数据翻译为程序中的 for、repeat、while 等循环结构。

10.8 为变量指定单一用途

- 每个变量只用于单一用途。下面例子显示了一个用于两种用途的临时变量：

C++示例：同一变量用于两种用途，糟糕的实践

```
// Compute roots of a quadratic equation
// This code assume that (b*b-4*a*c) is positive.
temp = Sqrt( b*b-4*a*c );
root[0] = ( -b + temp ) / ( 2 * a );
root[1] = ( -b - temp ) / ( 2 * a );
...
// swap the roots
temp = root[0];
root[0] = root[1];
root[1] = temp;
```

C++示例：同一变量用于两种用途，糟糕的实践

```
// Compute roots of a quadratic equation
// This code assume that (b*b-4*a*c) is positive.
discriminant = Sqrt( b*b-4*a*c );
root[0] = ( -b + discriminant ) / ( 2 * a );
root[1] = ( -b - discriminant ) / ( 2 * a );
...
// swap the roots
oldRoot = root[0];
root[0] = root[1];
root[1] = oldRoot;
```

- 避免让代码具有隐含含义。把同一变量用于多个用途的另一种方式，是当变量代表不同事物时，让其具有不同的取值集合。例如：变量 pageCount 的取值可能表示已打印纸张的数量，除非它等于-1，在这种情况下，表明有错误发生。
- 确保使用了所有已声明的变量。与同一变量多种用途相反的，是声明了变量，却不使用。

11 变量名的力量

11.1 选择好变量名的注意事项

（1）最重要的命名注意事项

为变量命名时，最重要的考虑事项是，该名字要完全、准确地描述出该变量所代表的事物。获得好名字的一种实用技巧，就是用文字表达变量所代表的是什么。通常，对变量的描述，就是最佳的变量名。这种名字很容易阅读，因为其中并不包含晦涩的缩写，同时也没有歧义。因为它是对该事物的完整描述，因此不会和其他事物混淆。另外，由于这一名字所表达的概念相似，因此也很容易记忆。

(2) 以问题为导向

一个好记的名字，反映的通常都是问题，而不是解决方案。一般而言，如果一个名字反映了计算的某些方面，而不是问题本身，那么它反映的就是“how”，而非“what”了。请避免取这样的名字，而应该在名字中反映出问题本身。例如，一条员工数据记录，可以称作 inputRec 或 employeeData；inputRec 是一个反映输入、记录这些计算概念的计算机术语；employData 则指问题领域，与计算的世界无关，因此应该用 employeeData。

(3) 最适当的名字长度

太短的名字无法传达足够的信息;太长的名字很难写,同时也会使程序的视觉结构变得模糊不清。研究发现,当变量名的平均长度在 10 到 16 个字符时,调试程序所需花费的力气是最小的。例如 numberOfPeopleOnTheUsOlympicTeam 太长、ntm 太短、numTeamMembers 正好。

(4) 变量名对作用域的影响

短的变量名，有时也是可以的。例如 i，这一长度本身就对该变量做出了一些说明，也就是说，该变量代表的是一个临时的数据，它的作用域非常有限。较长的名字适用于很少用到的变量或全局变量，而较短的名字则适用于局部变量或循环变量。不过短的变量名常常会带来一些麻烦，因此，作为一项防御式编程策略，应避免使用短的变量名。

（5）变量名中的计算值限定词

很多程序都有表示计算结果的变量：总额、平均值、最大值等等。如果用类似于 Total、Sum、Average 这样的限定词来修改这个名字，那么请记住把限定词加到名字的最后。这种方法具有很多优点。首先，变量名中最重要的那部分，即为这一变量赋予主要含义的那部分应当位于最前面，这样，这一部分就可以显得最为突出，并会被首先阅读到。其次，采纳了这一规则，你将避免由于同时在程序中使用 totalRevenue 和 revenueTotal 而产生歧义。总之，一致性可以提高可读性，简化维护工作。把计算的量放在名字最后也有例外，那就是 Num 限定词的位置已经约定俗成。Num 放在变量名的开始位置代表一个总数：numCustomers 表示员工总数。Num 放在变量名的结束位置代表一个下标：customerNum 表示当前员工的序号。通过 numCustomers 最后代表复数的 s 也能够看出这两种应用之间的叙别。然而，由于这样使用 Num 常常会带来麻烦，因此可能最好的办法是避开这些问题，用 Count 或者 Total 来代表员工的总数，用 Index 来指代某个特定的员工。这样，customerCount 就代表员工的总数，customerIndex 代表某个特定的员工。

（6）变量名中的常用对仗词

对仗词的使用要准确。通过应用命名规则来提高对仗词使用的一致性，从而提高其可读性。比如像 begin/end 这样的一组用词非常容易理解和记忆。

11.2 为特定类型的数据命名

（1）循环下标命名

i、j 和 k 这些名字都是约定俗成的：

Java示例：简单的循环变量命名

```
for( i = firtstItem; i < lastItem; i++){
    data[i] = 0;
}
```

如果一个变量要在循环之外使用，那么就应该为它取一个比 i、j 或 k 更有意义的名字。如果循环不是只有几行，那么读者会很容易忘记 i 本来具有的含义，因此你最好给循环下标换一个更有意义的名字。由于代码会经常修改、扩充，或

者复制到其他程序中去，因此，很多有经验的程序员索性不使用类似于 i 这样的名字。

Java示例：嵌套循环中的好循环变量名

```
for( teamIndex = 0; teamIndex < teamCount; teamIndex++ ){
    for( eventIndex = 0; eventIndex < eventCount[teamIndex]; eventIndex++){
        score[teamIndex][eventIndex] = 0;
    }
}
```

（2）状态变量命名

为状态变量取一个比 flag 更好的名字：状态变量的名字中，不应该含有 flag，因为你丝毫看不出该状态变量是做什么的。为了清楚起见，状态变量应该用枚举类型、具名常量，或用作用具名常量的全局变量来对其赋值，而且其值应该与上面这些量做比较。

C++示例：含义模糊的标记

```
if (flag) ...
if (statusFlag & 0x0F) ...
if (printFlag == 16) ...
if (computeFlag == 0) ...

flag = 0x1;
statusFlag = 0x80;
printFlag = 16;
computeFlag = 0;
```

C++示例：更好地使用状态变量

```
if (dataReady) ...
if (characterType & PRINTABLE_CHAR) ...
if (reportType == ReportType_Annual) ...
if (recalNeeded == false) ...

dataReady = true;
characterType = CONTROL_CHARACTER;
reportType = ReportType_Annual;
recalcNeeded = false;
```

下面例子展示了如何使用具名常量和枚举类型来组织例子中地数值：

在C++中声明状态变量

```
\\ value for CharacterType
const int LETTER = 0x01;
const int DIGIT = 0x12;
const int PUNCTUATION = 0x04;
const int LINE_DRAN = 0x08;
const int PRINTABLE = (LETTER | DIGIT | PUNCTUATION | LINE_DRAW);
cont int CONTROL_CHARACTER = 0x80;

\\ values for ReportType
enum ReportType {
    ReportType_Daily,
```

```
    ReportType_Monthly,
    ReportType_Quarterly,
    ReportType_Annual,
    ReportType_All
};
```

(3) 临时变量命名

临时性地保存一些值常常是很有必要的，但是无论从哪种角度看，程序中的大多数变量都是临时性的。把其中几个称为临时的，可能表明你还没弄清它们的实际用途，例如：

C++示例：不提供信息的临时变量

```
\\ Compute roots of a quadratic equation.
\\ This assumes that (b^2-4*a*c) is positive.
temp = sqrt(b^2-4*a*c);
root[0] = (-b + temp) / (2*a);
root[1] = (-b - temp) / (2*a);
```

其中 temp 没有反映该变量的功能，下面例子展示了一种更好的做法：

C++示例：用真正的变量替代临时变量

```
\\ This assume that (b^2-4*a*c) is positive.
discriminant = sqrt(b^2-4*a*c);
root[0] = (-b + discriminant) / (2*a);
root[1] = (-b - discriminant) / (2*a);
```

(4) 布尔变量命名

为布尔变量命名时要遵循的几条原则：

- 谨记典型的布尔变量名：下面是一些格外有用的布尔变量名，
  - done：用 done 表示某件事情已经完成。这一变量可用于表示循环结束或一些其他的操作已完成。在事情完成之前把 done 设为 false，在事情完成之后设为 true。
  - error：用 error 表示错误发生。在错误发生前设为 false，错误发生时设为 true。
  - found：表明某个值已经找到。false->true
  - success 或 OK：表明一项操作是否成功。操作失败时为 false，成功为 true。
- 给布尔变量赋予隐含“真/假”含义的名字：像 done 和 success 这样的名字是很不错的布尔变量名，因为其状态要么是 true，要么是 false；另一方面，像 status 和 sourceFile 这样的名字，就是很糟的布尔变量名，因为它们没有明确的 true 或 false 状态。
- 使用肯定的布尔变量名：否定的名字如 notFound、notDone 以及 notSuccessful 等较难阅读，特别是它们被求反：if not notFound。

(5) 为枚举类型命名

在使用枚举类型的时候，可以通过使用前缀，如 Color_ 来明确表示该类型的成员都同属于一个组。例如：

Visual Basic示例：为枚举类型采用前缀命名约定

```
Public Enum Color
    Color_Red
    Color_Green
    Color_Blue
End Enum
```

在有些编程语言里，枚举类型的处理和类很像，枚举成员也总是被冠以枚举名字前缀，例如 Color.Color_Red，那么重复上述前缀的意义就不大了，可以简化为 Color.Red。

(6) 为常量命名

为具名常量命名时，应该根据该常量所表示的含义，而不是该常量所具有的数值，为该抽象事物命名。例如，FIVE 就是个很糟的常量命，CYCLES_NEEDED 是个不错的名字。

11.3 命名规则的力量

规则的存在为你的代码增加了结构，减少了你需要考虑的事情。命名规则可以带来以下好处：

- 可以集中关注代码更重要的特征。
- 有助于在项目之间传递知识。
- 有助于在新项目中，更快速地学习代码。
- 有助于减少名字增生。在没有命名规则的情况下，会很容易给同一个对象起两个不同的名字。例如，pointTotal->totalPoints。
- 弥补编程语言的不足之处。可以用规则来仿效具名常量和枚举类型。规则可以根据局部数据、类数据以及全局数据的不同，而有所差别，并且可以包含编译器不直接提供的类型信息。
- 强调相关变量之间的关系。如果编程语言不支持对象，可以用命名规则来予以补充。例如，address、phone 以及 name 这样的名字，并不能表明这些变量是否相关；但 employeeAddress、employeePhone 和 employeeName 就会毫无疑问地表明这些变量时彼此相关的。

11.4 非正式命名规则

大多数项目采用的都是类似于本节所讲的相对非正式的命名规则。

（1）与语言无关的命名规则的指导原则

- 区分变量名和子程序名字，例如用变量名小驼峰，子程序大驼峰。
- 区分类和对象。方案 1：通过大写字母开头区分类型和变量，Widget widget；方案 2：通过全部大写区分类型和变量，WIDGET widget；方案 3：通过给类型加“t_”，t_Widget Widget；方案 4：通过给变量加“a”前缀区分，Widget aWidget；方案 5：通过对变量采用更明确的名字区分，Widget employeeWidget。
- 标识全局变量。在所有的全局变量名之前加上 g_ 前缀。
- 标识成员变量。可以用 m_ 前缀来标识类的成员变量，以表明它是成员数据。
- 标识类型说明。为类型建立命名规则有两个好处：首先它能够明确表明一个名字是类型名，其次能够避免类型名与变量名冲突。增加前缀或后缀是不错的方法。
- 标识具名常量。你需要对具名常量加以标识，以便明确在为一个变量赋值时，你用的是另一个变量的值，还是一个具名常量。给常量命名的方法之一是给常量增加 c_ 前缀。在 C++ 和 Java 里的规则是全部大写，以及如果可能，用下划线来分隔单词。
- 标识枚举类型的元素。标准方法如下：全部大写，或为类型名增加 e_ 或 E_ 前缀，同时为该类型的成员名增加基于特定类型的前缀，如 Color_ 或 Planet_。
- 在不能保证输入参数只读的语言里标识只读参数。有时输入参数会被意外修改。在 C++ 这样的语言里，你必须明确表明是否希望把一个修改后的值返回给调用方子程，分别用 *、& 和 const 指明。
- 格式化命名以提高可读性。有两种常用方法可以用来提高可读性，那就是用大小写和分隔符来分隔单词。尽量不要混用上述方法，那样会使代码难以阅读。

(2) 与语言相关的命名规则的指导原则

应该遵循你所用语言的命名规则。对于大多数语言，你都可以找到描述其风格原则的参考书。C++ 命名规则：

- i 和 j 是整数下标。
- p 是指针。
- 常量、typedef 和预处理宏全部大写 (ALL_CAPS)。
- 类和其他类型的名字混合大小写 (MixedUpperAndLowerCase())。
- 变量名和函数名中的第一个单词小写，后续每个单词的首字母大写，例如 variableOrRoutineName。
- 不把下划线用作名字中的分隔符，除非用于全部大写的名字以及特定的前缀中，如用于标识全局变量的前缀。

(3) 混合语言编程的注意事项

在混合语言环境中编程，可以对命名规则做出优化，以提高整体的一致性和可读性，即使这意味着优化后的规则会与其中某种语言所用的规则相冲突。

(4) 命名规则示例变量名包含了以下三类信息：

- 变量的内容：它代表什么
- 数据的种类：具名常量、简单变量、用户自定义类型或类
- 变量的作用域：私用的、类的、包的或全局作用域

表 11-3 C++和 Java 的命名规则示例

实　体	描　述
ClassName	类名混合使用大小写，首字母大写
TypeName	类型定义，包括枚举类型和 typedef，混合使用大小写，首字母大写
EnumeratedTypes	除遵循上述规则之外，枚举类型总以复数形式表示
localVariable	局部变量混合使用大小写，首字母小写。其名字应该与底层数据类型无关，而且应该反映该变量所代表的事物
routineParameter	子程序参数的格式与局部变量相同
RoutineName()	子程序名混合使用大小写（第 7.3 节已经讨论过什么是好的子程序名）
m_ClassVariable	对类的多个子程序可见（且只对该类可见）的成员变量名用 m_前缀
g_GlobalVariable	全局变量名用 g_前缀
CONSTANT	具名常量全部大写
MACRO	宏全部大写
Base_EnumeratedType	枚举类型名用能够反映其基础类型的、单数形式的前缀——例如，Color_Red，Color_Blue

11.5 标准前缀

对具有通用含义的前缀标准化，为数据命名提供一种简洁、一致并且可读性好的方法。标准化的前缀由两部分组成：用户自定义类型（UDT）的缩写和语义前缀。

(1) 用户自定义类型缩写

UDT 缩写可以标识被命名对象或变量的数据类型。UDT 缩写可以被用于表示像窗体、屏幕区域以及字体一类的实体。UDT 缩写通常不会表示任何由编程语言所提供的预置数据类型。UDT 用很短的编码描述，这些编码是为特定的程序创建的，并且经过标准化，以在该程序内使用。这些编码有助于用户理解其所代表的实体，如用 wn 代表窗体，wnMain；scr 代表屏幕区域，scrUserWorkspace。

(2) 语义前缀

语义前缀比 UDT 更进一步，它描述了变量或对象是如何使用的。语义前缀域 UDT 不同，后者会根据项目的不同而不同，而前者在某种程度上，对于不用的项目均是标准的。语义前缀可以全用小写，也可以混合使用大小写，还可以根据需要与 UDT 和其他的语义前缀结合使用。例如，文档中的第一段应该命名为 pa，以表明它是个段落，还要加上 first 以强调它是第一个段落：即 firstPa。一组段落的下标，可以命名为 iPa；cPa 是相应的计数值，段落的总数量；firstPaActiveDocument 和 lastPaActiveDocument 表示当前活动文档中的第一个和最后一个段落。

(3) 标准前缀的优点

- 标准前缀能够更为精确地描述一些含义比较模糊的名字。min、first、last 和 max 之间的严格区分就显得格外右用。
- 标准化的前缀是名字变得更加紧凑。例如，用 cpa 而不是 totalParagraphs 表示段落总数；可以用 ipa 表示一个段落数组的下标，而不是用 indexParagraphs 或者 paragraphsIndex。
- 在编译器不能检查所用的抽象数据类型时，标准前缀能帮助你准确地对类型做出判断：paReformat = docReformat 很可能不对，因为 pa 和 doc 是不同的 UDT。

标准前缀的主要缺陷是程序员在使用前缀的同时，忽略给变量起有意义的名字。如果 ipa 已经能够非常明确地表示一个段落数组地小标，那么程序员就不会主动地去想类似于 ipaActiveDocument 这样有意义地名字。为了提高可读性，应该停下来为数组下标，起一个具有描述性地名字。

11.6 创建具备可读性的短名字

可以通过消除冗余的单词、使用简短的同义词，以及使用诸多缩写策略中的任意一种，来创建更好的短命名。熟悉多种缩写技巧会很有用，因为没有那种方法能够适用所有的情况。

(1) 缩写的一般指导原则

下面几项用于创建缩写的指导原则，其中一些原则彼此冲突，所以不要试图同时应用所有的原则。

- 使用标准的缩写；
- 去掉所有非前置元音，例如 computer 变为 cmptr，screen 变为 scrn，apple 变为 appl 等等；
- 去掉虚词 and，or，the 等；
- 适用每一个单词的第一个或前几个字母；
- 统一地在每个单词地第一、第二或第三个字母后截断；
- 保留每个单词的第一个和最后一个字母；
- 使用名字中的每一个重要单词，最多不超过三个；
- 去除无用的后缀，ing，ed 等；
- 保留每个音节中最引人注意的发音；

- 确保不要改变变量的含义；
- 反复使用上述技术，直到你把每个变量的名字长度，缩减到了 8 到 20 个字符，或者达到你预期目标。

(2) 有关缩写的评论
下面是一些用来避免犯错的规则。

- 不要从每个单词中删除一个字符的方式来缩写。对于大多数删除一个字母的做法，你很难回忆起自己是不是删了一个字符。所以，要么删除不止一个字符，要么就把单词拼写完整。
- 缩写要一致。应该一直使用相同的缩写。例如，要么全用 Num，要么全用 No，不要两个都用。与之类似，不要在一些名字里缩写某个单词，而在其他名字里不缩写。例如，不要在有些地方使用完整的单词 Number，同时在其他地方使用 Num 缩写。
- 创建你能读出来的名字。例如用 xPos 而不用 XPstn。
- 避免使用容易看错或者读错的字符组合。例如为了表示 B 的结尾，ENDB 要比 BEND 更好。如果你使用了一种好的分隔技术，那么就不需要这一条原则，例如 BEnd 或 b_end。
- 使用词典来解决命名冲突。创建简短名字会带来一个麻烦就是命名冲突：缩写后名字相同。避免命名冲突的一种简单做法，是使用含义相同的不同单词，这样一来，有一部词典就显得很方便了。
- 在代码里用缩写对照表解释极短的名字的含义。增加一张缩写对照表，来为用户提示更多的变量含义。把该表格作为注释加到一段代码的开始。
- 在一份项目级的“标准缩写”文档中，说明所有的缩写。代码中的缩写，会带来两种常见风险：代码的读者可能不理解这些缩写、其他程序员可能会用多个缩写来代表相同的词，从而产生不必要的混乱。为了同时解决着两个潜在的问题，可以创建一份“标准缩写”文档，来记录项目中用到的全部编码缩写。这份文档既可以是文字处理程序的文档，也可以是电子表格文档。在很大的项目里，它还可以是一个数据库。这份文档应签入 check in 到版本控制系统里，当任何人与任意时间在代码里创建了一种新的缩写时，把它签出 check out 来修改。文档中的词条应该按照完整单词排序，而不是按照缩写排序。
- 名字对于代码读者的意义，要比对作者更重要。去读一读你自己写的，并且至少有六个月没看过的代码，注意哪些名字是你需要花功夫才能理解其含义的。应下决心改变导致这种混乱的做法。

11.7 应该避免的名字

下面就哪些变量名应该避免给出指导原则。

- 避免使用令人误解的名字或缩写。要确保名字的含义是明确的。
- 避免使用具有相似含义的名字。如果你能交换两个变量的名字，而不会妨碍对程序的理解，那么你就需要为这两个变量重新命名了。例如，recordNum 和 numRecords。
- 避免使用具有不同含义，却有相似名字的变量。如果你有两个名字相似，但含义不同的变量，那么试着给其中之一重新命名，或者修改你的缩写。例如，clientRecs 和 clinetReps，修改为 clientRecords 和 clientReports。
- 避免使用发音相近的名字。例如 warp 和 rap。当你试图和别人讨论代码的时候，同音异义就会产生麻烦。
- 避免在名字中使用数字。如果名字中的数字真的非常重要，就用数组来代替一组单个的变量。要避免使用 file1 和 file2 这样的命名。
- 避免在名字中拼错单词。弄清楚单词实际应该怎么拼写是很难的。
- 避免使用英语中常常拼错的单词。absense, acummulate, acsend 等很多单词经常会拼错。
- 不要仅靠大小写来区分变量名。

- 避免使用多种自然语言。在多语言的项目中，对于全部代码，如类名、变量名等，要强制使用一种自然语言。
- 避免使用标准类型、变量和子程序的名字。
- 不要使用与变量含义完全无关的名字。
- 避免在名字中包含容易混淆的字符。例如 tt15 和 ttlS。

12 基本数据类型

12.1 数字使用一般原则

- 避免使用魔鬼数字 (magic number)。魔鬼数字是在程序中出现的、没有经过解释的数值文字量，如 100 或 47523。如果你编程用的语言支持具名常量，可以用它来代替魔鬼数字。如果你无法使用具名常量，在可行的情况下，应该使用全局变量。避免魔鬼数字会带来以下三点好处：
  - 修改会变得更可靠。如果使用了具名常量，不会在修改时漏掉多个 100 中的某一个。
  - 修改会变得更容易。
  - 代码变得更可读。
- 如果需要，可以使用硬编码的 0 和 1。数值 0 和 1 用于增量、减量和从数组的第一个元素开始循环。一条很好的经验法则，是程序主体中仅能出现的数字就是 0 和 1。
- 预防除零错误。每次使用除法符号的时候，都要考虑表达式的分母是否可能为 0。如果这种可能性存在，就应该写代码防止除零错误的发生。
- 使类型转换变得明显。例如，y = x + static_cast<float>(i)
- 避免混合类型的比较。在编译器设法弄清了应该用什么类型去进行比较之后，它会把其中一种类型，转换为另一种，执行一些四舍五入运算之后，才得出结果。请自己动手进行类型转换，这样编译器就能比较两个相同类型的数字了，你也会确切地知道它比较的是什么。
- 注意编译器的警告。当你在同一表达式中，使用了多种类型的数值，很多现代的编译器都会通知你。杰出的程序员会修改他们的代码来消除所有的编译器告警。

12.2 整数

在用整数的时候，要记住下面的注意事项。

- 检查整数除法。当你使用整数的时候，7/10 不等于 0.7，它总是等于 0。
- 检查整数溢出。在做整数乘法或加法的时候，要留心可能的最大整数。避免整数溢出的最简单办法，是考虑清楚算术表达式中的每个项，设想每项可能达到的最大值。另外还要考虑程序在未来的扩展，如果 m 的取值永远不会超过 5000，那很好；但如果你预计 m 的取值会在几年时间内稳定增长，那么就要把这种情况考虑进来。
- 检查中间结果溢出。可以用处理整数溢出的相同办法，来处理中间结果溢出，换用一种更长的整数型或浮点类型。

12.3 浮点数

下面是一些在使用浮点数时，应该遵循的指导原则

- 避免数量级相差巨大的数之间的加减运算。32 位浮点变量，1000000.00+0.1 可能会得到 1000000.00，因为 32 位不能给你足够的有效位数，包容 1000000 和 0.1 之间的数值区间。如果你必须把一系列差异如此巨大的数相加，那么就先对这些数排序，然后从最小值开始把它们加起来。同样，如果你需要对无穷数列进行求和，那么就从最小的值开始。从本质上来说，是要做逆向的求和运算。这样做并不能消除舍入问题，但是能使这一问题的影响，减少到最低限度。

- 避免等量判断。用两种不同方法求同一数值，结果不一定总得到同一个值。例如，10 个 0.1 相加，很少会等于 1.0。因此，应该找一种代替对浮点数字，执行等量判断的方案。一种有效的方法，是先确定可接受的精确范围，然后用布尔函数判断数值是否足够接近。通常应该写一个 Equals() 函数，如果数值足够接近，就返回 true，否则就返回 false。

- 处理舍入误差问题。由于舍入误差的错误，与由于数字之间数量级相差太大，而导致的错误并无二致。问题相同，解决的技术也相同。除此之外，下面列出一些专门用于解决舍入问题的常见方案。

  – 换用一种精确度更高的变量类型。

  – 换用二进制编码的十进制变量。

  – 把浮点变量变成整型变量。

12.4 字符和字符串

本节给出一些使用字符串的技巧。其中第一条适用于所有的语言。

- 避免使用神秘字符和神秘字符串。神秘字符是指程序中随处可见的字面形式表示的字符，例如“A”、“0x1B”，神秘字符串是指字面形式表示的字符串，例如“Gigamatic Accounting Program”。如果你用的编程语言支持具名常量，则用具名常量来加以取代，否则就用全局变量。

- 避免 off-by-one 错误。由于子字符串的下标索引方式，几乎与数组相同，因此要避免因为读写操作，超过了字符串末尾，而导致的 off-by-one（偏差一）错误。

- 了解你的语言和开发环境是如何支持 Unicode 的。

- 在程序生命周期中，尽早决定国际化/本地化策略。与国际化与本地化相关的事项，都是很重要的问题。关键的考虑事项包括：决定是否把所有字符串保存在外部资源里，是否为每一种语言创建单独的版本，或者在运行时确定特定的界面语言。

- 如果你知道只需要支持一种文字的语言，请考虑使用 ISO 8859 字符集。对于只需要支持单一文字（例如英语）、无须支持多语言或者某种表意语言（例如汉语）的应用程序，可以使用 ISO 8895 扩展 ASC11 类型标准来很好地替代 Unicode。

- 如果需要支持多种语言，请使用 Unicode。与 ISO 8895 或其他标准相比，Unicode 对国际字符集提供了更为全面地支持。

- 采用某种一致的字符串类型转换策略。

12.5 布尔变量

要把逻辑变量或者布尔变量用错是非常困难的，而更仔细地运用它，会让你地程序变得更清晰。

- 用布尔变量对程序加以文档说明。不同于仅仅判断一个布尔表达式，你可以把这种表达式的结果，赋给一个变量，从而使这一判断的含义变得明显。例如，

Java示例：目的不明确的布尔判断

```
if ((elementIndex < 0) || (MAX_ELEMENTS < elementIndex)) ||
    (elementIndex == lastElementIndex)
    ) {
        ...
}
```

Java示例：目明确的布尔判断

```
finished = ((elementIndex < 0) || (MAX_ELEMENTS < elementIndex));
repeatedEntry = (elementIndex == lastElementIndex);
if(finished || repeatedEntry) {
    ...
}
```

- 用布尔变量来简化复杂的判断。常有这样的情况，在需要编写一段复杂的判断时，你要尝试好几次才能成功。在你事后想要修改这一判断的时候，首先弄清楚这段判断在做什么，就已经很困难了。逻辑变量可以简化这种判断。
- 如果需要的话，创建你自己的布尔类型。有些语言，比如 C++、Java 和 Visual Basic，含有预定义的布尔类型。其他语言，比如 C，却没有，这时候你可以定义自己的布尔类型。

12.6 枚举类型

枚举类型是一种允许用英语，来描述某一类对象中，每一个成员的数据类型。通常用在你知道变量的所有可能取值，并且希望把它们用单词表达出来的时候。例如，

Visual Basic示例：枚举类型

```
Public Enum Color
    Color_Red
    Color_Green
    Color_Blue
End Enum
```

下面给出一些如何使用枚举类型的指导原则。

- 用枚举类型来提高可读性。例如 if chosenColor = 1 改为 if chosenColor = Color_Red。每当你看到字面形式数组的时候，就应该问问自己，把它换成枚举类型，是不是更合理。枚举类型特别适用于定义子程序参数。

C++示例：函数调用，用枚举会更好

```
int result = RetrievePayrollData(data, true, false, false, true);
```

C++示例：函数调用，使用枚举提高可读性

```
int result = RetrievePayrollData(
    data,
    EmploymentStatus_CurrentEmployee,
    PayrollType_Salaried,
    SavingsPlan_NoDeduction,
    MedicalCoverage_IncludeDependents
);
```

- 用枚举类型提高可靠性。对于少数语言而言（尤其是 Ada），枚举类型会使编译器执行比整数和常量更为彻底的类型检查。

- 用枚举类型来简化修改。枚举类型使得你的代码更容易修改。
- 将枚举类型作为布尔变量的替换方案。布尔变量往往无法充分表达它所需要表达的含义。例如，假设你有一个子程序，在成功地完成任务之后返回 true，否则范围 false。后来你可能发现事实上有两种 false。第一种表示任务失败了，并且其影响只局部于子程序自身；第二种表示任务失败了，而且产生了一个致命错误，需要把它传播到程序的其余部分。在这种情况下，一个包含 Status_Success、Status_Warning 和 Status_FatalError 值的枚举类型，就比一个包含 true 和 false 的布尔类型更有用。如果成功和失败的具体类型有所增加，对其进行扩展以区分这些情况，也是非常容易的。
- 检查非法数值。在 if 或 case 语句中，测试枚举类型时，务必记得检查非法值。
- 定义出枚举的第一项和最后一项，以便用于循环边界。把枚举的第一个和最后一个元素，
  例如定义为 Country_First，Country_Last，以便你更方便地写出能遍历所有枚举元素的循环来。你可以用明确的数值，来定义该枚举类型。例如，

```
Visual Basic示例：设置枚举类型数据第一项和最后一项
Public Enum Country
    Country_First = 0
    Country_China = 0
    Country_England = 1
    Country_France = 2
    Country_Germany = 3
    Country_Last = 3
End Enum
```

- 把枚举类型的第一个元素留作非法值。很多编译器会把枚举类型中的第一个元素赋值为 0。把映射到 0 的那个元素，声明为无效，会有助于捕捉那些没有合理初始化的变量，因为这些变量值更有可能为 0，而不是其他的非法值。

```
Visual Basic示例：将枚举中第一个元素声明为无效值
Public Enum Country
    Country_InvalidFirst = 0
    Country_First = 1
    Country_China = 1
    Country_England = 2
    Country_France = 3
    Country_Germany = 4
    Country_Last = 4
End Enum
```

- 明确定义项目代码编写标准中，第一个和最后一个元素的使用规则，并且在使用时保持一致。
- 警惕给枚举元素明确赋值而带来的失误。有些语言允许对枚举里面的各项元素，明确地赋值。例如，

```
C++示例：对枚举元素直接赋值
enum Color {
    Color_InvalidFirst = 0,
    Color_First = 1,
    Color_Red = 1,
    Color_Green = 2,
    Color_Blue = 4,
```

```
    Color_Black = 8,
    Color_Last = 8
};
```

在这个例子中，如果你把一个循环的下标，声明为 Color 类型，并且尝试去遍历所有的 Color，那么你遍历 1，2，4，8 这些合法数值的同时，也会遍历 3，5，6，7 这些非法数值。

12.7 具名常量

使用具名常量，是一种将程序“参数化”的方法：把程序中可能变化的一个方面写为一个参数，当需要对其修改时，只改动一处就可以了，而不必在程序中到处带动。

- 在数据声明中使用具名常量。在需要定义所用数据的大小的数据声明和其他语句里，使用具名常量可以提高程序的可读性和可维护性。
- 避免使用文字量，即使是“安全”的。在下面的循环里，你认为 12 代表什么含义？

```
Visual Basic示例：含义模糊的代码
For i = 1 To 12
    profit(i) = revenue(i) - expense(i)
Next

Visual Basic示例：含义清晰的代码
For month = 1 To NUM_MONTHS_YEAR
    profit(month) = revenue(month) - expense(month)
Next
```

- 用具有适当作用域的变量或类来模拟具名常量。如果你的语言不支持具名常量，可以自行创建一套解决方案。
- 统一地使用具名常量。如果需要表示的是同一个实体，在一处使用具名常量，而在另一处使用数字符号，是非常危险的。

12.8 数组

数组是最简单和最常用的结构化数据类型。一个数组中含有一组类型完全相同，并且可以用数组下标来直接访问的条目。下面就如何使用数组给出一些建议。

- 确认所有的数组下标，都没有超出数组的边界。最常见的问题就是，程序试图用超出数组边界的下标，去访问数组元素。
- 考虑用容器来取代数组，或者将数组作为顺序化结构来处理。建议使用集合、栈和队列等，按顺序存取元素的数据结构，来取代数组。
- 检查数组的边界点。你可以通过检查数组的边界点，来捕获很多错误。问问自己，代码有没有正确地访问数组的第一个元素？还是错误地去访问了第一个元素之前，或之后的那个元素？而最后一个元素呢？代码会导致 off-by-one 的错误吗？代码有没有正确地访问数组中间的元素？
- 如果数组是多维的，确认下标的使用顺序是正确的。很容易把 Array[j][i] 写成 Array[i][j]。与其使用 i 和 j 这类不明不白的东西，不如去考虑更有意义的名字。
- 提防下标串话。在使用嵌套循环时，很容易把 Array[i] 写成 Array[j]。调换循环下标称为“下标串话”。请检查这种问题。更好的做法是使用比 i 和 j 更有意义的下标名。
- 在 C 中结合 ARRAY_LENGTH() 宏来使用数组。通过定义类似下面的宏，可以更加灵活地使用数组。

C示例：定义ARRAY_LENGTH()宏

```
#define ARRAY_LENGTH(x) (sizeof(x)/sizeof(x[0]))
```

12.9 创建自己地类型（类型别名）

假如你正在写一个程序，把 x、y、z 坐标系中的坐标值，转化为纬度、经度和海拔高度，可以使用 C 或 C++ 中的 typedef 语句，或者其他语言中的相关语句，来为坐标创建一个新的特殊类型。

C++示例：创建一个数据类型

```
typedef float Coordinate; // for coordinate variables
```

该类型定义声明了一个新的类型，Coordinate，其功能与 float 类型完全相同。在使用这一新类型时，就像使用 float 等预定义类型一样，用它来声明变量。

C++示例：使用前面创建的数据类型

```
Routine1(...) {
    Coordinate latitude; // latitude in degrees
    Coordinate longitude; // longitude in degrees
    Coordinate elevation; // elevation in meters from earth center
    ...
}
...

Routine2(...) {
    Coordinate x; // x coordinate in meters
    Coordinate y; // y coordinate in meters
    Coordinate z; // z coordinate in meters
    ...
}
```

假设程序发生了变化，需要用双精度变量来表示坐标。由于你已经专门为坐标数据定义了一种类型，因此唯一需要修改的就是类型的定义。

C++示例：改变后的类型定义

```
typedef double Coordinate; // for coordinate variables
```

创建自己类型的原因：

- 易于修改。
- 避免过多的信息分发。采用硬编码，而非集中在一处管理数据的方式，会导致数据类型的细节，散布于程序内部。
- 增加可靠性。
- 弥补语言的不足。如果你的语言不具有你所需要的预定义类型，可以自己创建它。例如，C 没有布尔或逻辑类型，可以定义“typedef int Boolean”来弥补。

创建自定义数据类型的指导原则

- 给所创建的类型取功能导向的名字。避免使用那些代表了类型底层计算机数据类的类型名，例如 BigInteger 或 LongString；应该用能代表该新类型所表现的现实世界问题的类型名，例如坐标、年龄、货币等。创建自定义类型的最大优点，就在于它提供了介于你的程序和实现语言之间的一层绝缘层。引用了底层编程语言类型的类型名，就是在该绝缘层上戳了一个洞。它不会比使用一种预定义类型给你带来更多好处。另一方面，以现实问题为导向的名字，也使自定义类型容易修改。

- 避免使用预定义类型。像 Coordinate x 这样的声明，要比 float x 这样的声明，告诉你更多关于 x 的信息，请尽可能多地使用自己创建的类型。
- 不要重定义一个预定义的类型。如果你的语言有一个预定义的类型 Integer，那么就不要创建名为 Integer 的自定义类型，容易产生混淆。
- 定义替代类型以便于移植。与不要重定义一个预定义类型的建议相反，你可能需要为标准类型定义替代类型，以便让变量在不同的硬件平台上，正确地代表相同的实体。例如，你可以定义一个 INT32 类型，用它来代替 int。最初，这样的两个类型之间唯一的区别，就是它们名字的大小写不同，但是当你把程序移植到一个新的硬件平台上时，你就可以重新定义大写的那个类型版本，以便它们能够与原始硬件的数据类型匹配。一定不要定义容易被错认为是预定义类型的类型，最好把自定义类型和语言所提供的类型，明显区分开来。
- 考虑创建一个类，而不是使用 typedef。简单的 typedef 对隐藏变量的底层类型信息是大有帮助的。然而，在一些情况下，你可能需要定义类所能获得的那些额外的灵活度和控制力。

13 不常见的数据类型

13.1 结构体

结构体是指使用其他数据类型组建的数据。下面列出了一些使用结构体的理由。

- 用结构体来明确数据关系。结构体把相关联的一组数据项，聚集在一起。有时了解一个程序，最困难的部分，就在于理清哪些数据之间相互有联系。

```
Visual Basic示例：令人误解的、无组织的一堆变量
name = inputName
address = inputAddress
phone = inputPhone
title = inputTitle
department = inputDepartment
bonus = inputBonus
```

结构体的引入，使得这些关系变得更加清晰：

```
Visual Basic示例：提供更多信息的结构化变量
employee.name = inputName
employee.address = inputAddress
employee.phone = inputPhone

supervisor.title = inputTitle
supervisor.department = inputDepartment
supervisor.bonus = inputBonus
```

在使用了结构化变量的代码里，很明显可以看出有一些数据与雇员有关，其他的数据与主管有关。

- 用结构体简化对数据块的操作。你可以把相关的元素组织到结构体里，然后对该结构体执行操作。对结构体执行操作，要比对各元素执行同样的操作，容易得多。这样做也会更可靠，并且只需要更少得代码。

```
Visual Basic示例：复制一组数据项，笨拙的做法
newName = oldName
newAddress = oldAddress
newPhone = oldPhone
```

如果你想增加一条新的雇员信息，例如 gender，你就不得不找到各块赋值语句的位置，一一添加赋值语句 newGender = oldGender。如果需要交换两个雇员的数据，就会更加复杂了，

```
Visual Basic示例：交换两组数据，困难的做法
' Swap new and old employee data
previousOldName = oldName
previousAddress = oldAddress
previousPhone = oldPhone

oldName = newName
oldAddress = newAddress
oldPhone = newPhone

newName = previousOldName
newAddress = previousAddress
newPhone = previousPhone
```

解决该问题的一种更简便的方法，是申明一个 Structure 变量，

```
Visual Basic示例：声明Structure
Structure Employee
    name As String
    address As String
    phone As String
End Structure
Dim newEmployee As Employee
Dim oldEmployee As Employee
Dim previousOldEmployee As Employee

交换两组数据
previousOldEmployee = oldEmployee
oldEmployee = newEmployee
newEmployee = previousOldEmployee
```

- 用结构体来简化参数列表。可以利用结构体变量，简化子程序的参数列表。

```
Visual Basic示例：笨拙的子程序调用，未使用结构体
HardWayRoutine(name, address, phone, ssn, gender, salary)

优雅的子程序调用，使用结构体
EasyWayRoutine(employee)
```

你可以把这种技术运用到极致，即把程序中所有的变量，都放置在一个巨大的内容丰富的变量里，然后到处传递它。但是，除非逻辑上必要，细致的程序员会避免把数据困扎在一起。此外，他们还会避免把一个，只需要其中一两个字段的结构体，作为参数传递；相反，他们只会传递那些必需的特定字段。这是信息隐藏原则的一个方面：有些信息藏在子程序里面，而有些信息是对子程序隐藏的。信息应该按照有必要了解（need-to-know）的原则进行传递。

- 用结构体来减少维护。由于你在使用结构体的时候，是把相关的数据组织在一起，因此对结构体的修改，只会导致对程序做很小的改动。特别是对那些逻辑与结构体变化没有关联的代码来说，这一点尤为正确。由于变化容易

带来错误，因此变化越少，错误也就越少。如果 Employee 结构体中含有一个 title 字段，但你又想删除它，那么不需要对任何参数列表，或者用到了整个结构体的赋值语句做出修改。当然，你必须要修改那些专用于处理雇员 title 的代码，但是从概念上来说，这直接关系到删除 title 字段，因而不容易被忽略。

13.2 指针

对指针的运用，具有其固有的复杂性，正确使用指针，要求你对所有编译器的内存管理机制，有很好的理解。很多常见的安全问题，特别是缓冲区溢出，其产生都可以追溯到错误地运用指针上去。

(1) 用来理解指针的范例

从概念上看，每一个指针都包含两部分：内存中的某处位置，以及如何解释该位置中的内容。

- 内存中的位置。内存中的一个位置，就是一个地址，常用十六进制形式表示。32 位处理器中的一个地址，用一个 32 位的值表示，例如 0x0001EA40。指针本身只包含这个地址。为了使用该指针所指向的数据，就必须访问该地址，解释该位置的内存内容。如果去查看该地址的内存，可以发现它只是一组二进制位，必须经过解释，才能使它变得有意义。

- 如何解释指针所指的内容。如何解释内存中某个位置的内容，是由指针的基类型决定的。如果某个指针指向整数，这就意味着编译器，会把该指针所指向内存位置的数据，解释为一个整数。当然，你可以让一个整数指针、一个字符串指针和一个浮点数指针，都指向同一个内存位置。但是至多只有一个指针，能正确解释该位置的内容。在理解指针的时候，应该记住，内存并不包含任何与之相关联的内在的解释。只有通过使用一个特定类型的指针，一个特殊位置的比特，才能解释为有意义的数据。如图所示，每一种情况里，指针指向的都是以十六进制数组 0x0A 开始的位置；0A 之后使用的字节数量，却决于这篇内存的解释。内存内容如何使用，也要取决于内存的解释方式。同样的原始内存空间，可以解释为一个字符串、一个整数、一个浮点数，或者任何其他事物，一切都却决于指向该内存的指针基类型。

(2) 使用指针的一般技巧

通常，指针错误都产生于，指针指向了它不应该指向的位置。当你通过一个坏了的指针变量赋值时，会把数据写入本不该写值的内存区域，这称为“内存破坏”。有时内存破坏会导致可怕、严重的系统崩溃；有时他会篡改程序其他部分的计算结果；有时它会致使你的程序，不可预知地跳过某些子程序；而有时候它又什么事情都没做。在最后一种情况下，这种指针错误就像一颗嘀嗒作响地定时炸弹，等着在你把程序演示给最重要客户的前 5 分钟引爆。指针错误的症状，常常与引起指针错误的原因无关。因此，更正指针错误的大部分工作量，便是找出它的位置。

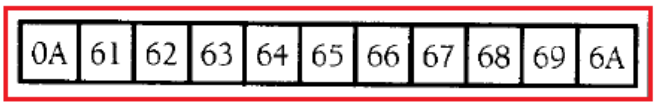
正确地使用指针，要求程序员采用一种双向策略。第一，要首先避免造成指针错误。指针错误很难发现，因此采取一些预防性措施是值得地；其次，在编写代码之后，尽快地检测出指针错误来。指针错误地症状飘忽不定，采取一些额外措施来使得这些症状可以被预测，是非常值得的。下面说明如何才能实现这些目标。

- 把指针操作限制在子程序或类里面。假如你在程序中，多次使用了一个链表。每次使用它的时候，不要通过手动操作指针，去遍历该链表，应该编写一组诸如 NextLink()、PreviousLink()、InsertLink() 和 DeleteLink()，这样的访问器子程序，来完成操作更好些。减少访问指针代码位置的数量，你也就减少了犯下遍布程序各处、永远也找不完的粗心过错的可能性。这样一来，这些代码能相对独立于数据的实现细节，因此也增大了在其他程序内，重用这些代码的可能性。为指针分配编写子程序，是另一种集中控制数据的方法。

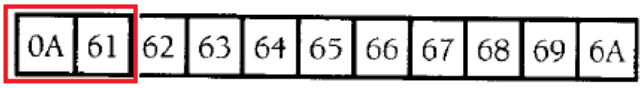
- 同时申明和定义指针。在靠近变量申明的位置，为该变量赋初值，通常是一项好的编程实践；在使用指针时，应用这条原则会更有价值。应该避免下面这种做法：

C++示例：糟糕的指针初始化

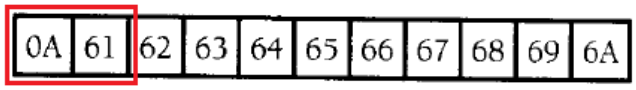
```
Employee *employeePtr;
// lots of code
...
employeePtr = new Employee;
```



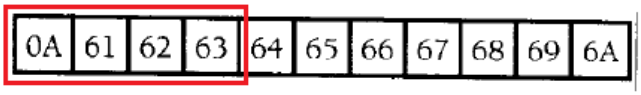
解释方式： 用于进一步举例的原始内存空间（用 16 进制表示）
意义： 没有与之关联的指针变量，没有任何意义



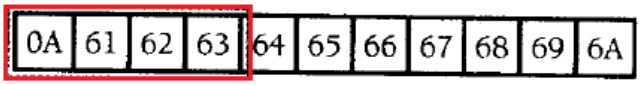
解释方式： String[10]（以 Visual Basic 的格式表示，第一个字节存储长度）
意义： abcdefghij



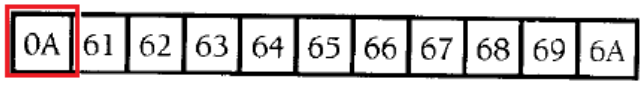
解释方式： 双字节的整数
意义： 24842



解释方式： 四字节的浮点数
意义： 4.17595656202980E+0021



解释方式： 四字节的整数
意义： 1667391754



解释方式： 字符
解释： 换行符（ASCII 码 16 进制的 0A 或者 10 进制的 10）

图 13-1 各数据类型所用的内存量用双线框表示

即使这段代码一开始能正确工作，修改时也容易出错，可能有人会试着在指针声明和初始化之间的位置，使用 employeePtr。下面是一种更为安全的做法：

C++示例： 良好的指针初始化

```
// lots of code
...
Employee *employeePtr = new Employee;
```

- 在与指针分配相同的作用域中删除指针。要保持指针分配和释放操作的对称性。如果你需要在一个单一作用域内使用指针，那么就应该在此作用域范围内用 new 分配指针，用 delete 释放指针。如果你在一个子程序内分配了一个指针，那么就在同一个子程序里释放它。如果你在一个对象的构造函数里面分配了一个指针，那么就在该对象的析构函数里释放它。如果一个子程序分配了内存，却指望调用它的代码来释放该内存，这样造成的不一致的处理方式，很容易出错。

- 在使用指针之前检查指针。在程序的关键之处，使用一个指针之前，要确保它所指向的内存位置是合理的。例如，如果你希望的内存位置介于 StartData 和 EndData 之间，那么你就该对一个指向 StartData 之前或者 EndData 之后的指针产生疑问。你还必须要确定，在你的环境下，StartData 和 EndData 的取值。如果你是用访问器子程

序来操作指针，而不是直接操作它们的话，那么这一检查工作就可以自动进行。

- 先检查指针所引用的变量再使用它。有时你应该对指针所指向的数据，执行合理性检查。例如，如果你认为指针指向的，是一个介于 0 和 1000 之间的整数，那么你就应该对大于 1000 的数值产生怀疑。同样，如果你是通过访问器子程序来使用指针，那么这一项工作就可以自动完成了。

- 用狗牌字段来检测损毁的内存。“标记字段（tag field）”或者“狗牌（dog tag）”是指你加入结构体内的，一个仅仅用于检测错误的字段。在分配一个变量的时候，把一个应该保持不变的数值，放在它的标记字段里。当你使用该结构的时候，特别是当你释放其内存的时候，检测这个标记字段的取值如果这个标记字段的取值与预期不相符，那么这一数据就被破坏了。

- 增加明显的冗余。还有一种可以代替标记字段的方案，就是某些特定字段重复两次。如果位于冗余字段中的数据不匹配，你就可以确定数据已经破坏了。如果你直接操作指针，这么做会带来很高的成本。然而，如果你把指针操作限制在子程序里，那么就只需要少数几处重复的代码。

- 用额外的指针变量，来提高代码清晰度。一定不要节约使用指针变量。这个要点来自别处，那就是不要把同一个变量，用于多种用途。这一点对指针变量来说尤其正确。在没弄清楚为什么要反反复复使用 genericLink 变量，也没弄清楚 pointer->next->last->next 指向什么之前，很难弄清别人在用链表做些什么。

C++示例：传统的插入节点的代码

```
void InsertLink(
    Node *currentNode,
    Node *insertNode
    ) {
    // insert "insertNode" after "currentNode"
    insertNode->next = currentNode->next;
    insertNode->previous = currentNode;
    if(currentNode->next != NULL) {
        currentNode->next->previous = insertNode;
    }
    currentNode->next = insertNode;
}
```

这是往链表里插入一个节点的传统代码，晦涩难懂。插入一个新节点，涉及到三个对象：当前节点，此时位于当前节点后面的节点，以及将要被插入到二者之间的节点。上述代码片段只明确地承认了两个对象：insertNode 和 currentNode。它要求你弄清楚并记住 currentNode->next 也被包含在内了。更好的做法是把三个对象都识别出来，

C++示例：更具可读性的节点插入代码

```
void InsertLink(
Node *startNode,
Node *newMiddleNode
) {
    // insert "newMiddleNode" between "startNode" and "followingNode"
    Node *followingNode = startNode->next;
    newMiddleNode->next = followingNode;
    newMiddleNode->previous = startNode;
    if(followingNode != NULL){
        followingNode->previous = newMiddleNode;
    }
```

```
        startNode->next = newMiddleNode;
    }
```

虽然这段代码中多了一行代码，但是由于没有了之前的 currentNode->next->previous，因此它读起来更容易些。

- 简化复杂的指针表达式。复杂的指针表达式是很难读懂的，如果在你的代码里面，包含了一个复杂的表达式，那么就把它赋给一个命名良好的变量，以明确该操作的用意。

```
C++示例：难以理解的指针表达式
for(rateIndex=0; rateIndex<numRate; rateIndex++){
    netRate[rateIndex] = baseRate[rateIndex] *
                         rates->discounts->factors->net;
}

C++示例：简化一个复杂的指针表达式
quantityDiscount = rates->discounts->factors->net;
for(rateIndex=0; rateIndex<numRates; rateIndex++){
    netRate[rateIndex] = baseRate[rateIndex] * quantityDiscount;
}
```

经过简化后，不但提高了可读性，而且还可能因为简化了循环内的指针操作，而改善了性能。

- 画一个图。用代码来解释指针可能会让读者感到困惑，画一个图通常会有所帮助。例如，有关链表插入问题，

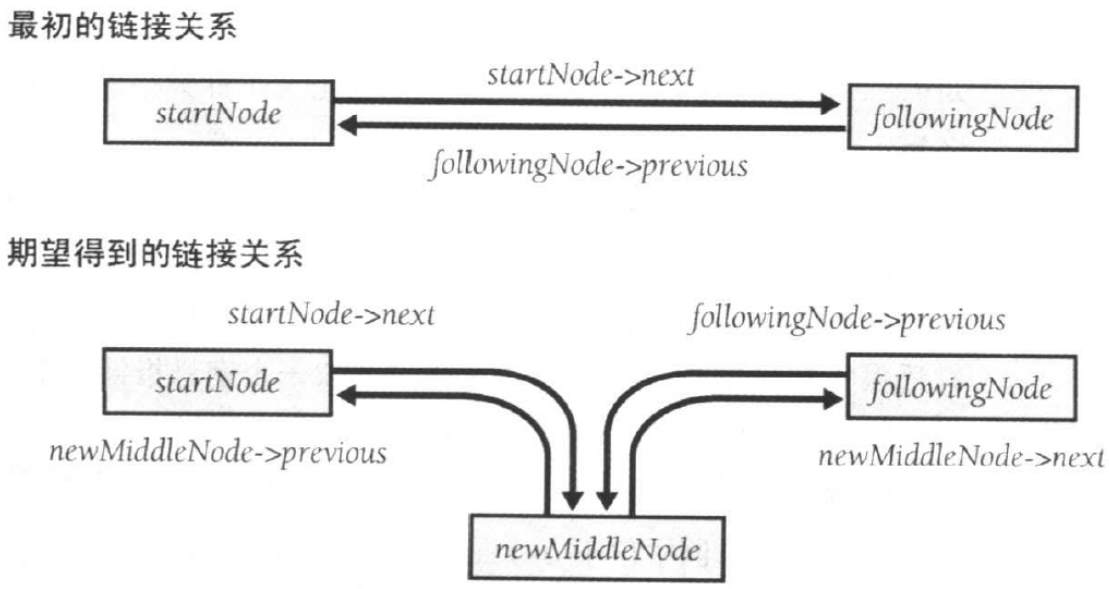


图 13-2　能帮助我们考虑指针链接步骤的示例图

- 按照正确的顺序删除链表中的指针。在使用动态分配链表时，经常遇到的一个问题是，如果先释放了链表中的第一个指针，就会致使表中的下一个指针无法访问。为了避免这一问题，在释放指针之前，要确保已经有指向链表中，下一个元素的指针。

- 分配一篇保留的内存后备区域。如果在你的程序中使用了动态内存，就需要避免发生，程序忽然用尽了内存、把你的用户和用户的数据，丢在 RAM 空间里的尴尬场景。使你的程序对这类错误，留出缓冲地带的一种方法，是预先分配一片内存后备区域。设法确定程序为了“保存所做的工作，执行清理并体面地退出”，需要用多少内存。在程序初始化阶段，把这部分内存分配出来作为后备，然后就可以不再管它。一旦你真的用光了内存，就释放保留下来的这片后备区，执行清理工作，然后退出。

- 粉碎垃圾数据。指针错误是很难调试的，因为你无法确定指针所指向的内存，何时变成非法的。有时指针已经释放了很长一段时间，相应内存的内容看起来还像是有效的。而在另一些时候，这些内存内容马上就会改变。在 C 语言中，在释放内存区域之前，用垃圾数据来覆盖这些内存区域，可以让与使用已释放的指针有关的错误，表现方式更一致。和其他很多操作相似，如果你使用了访问器子程序，那么就可以让这一功能自动执行。在 C++ 里面，你可以在每次删除指针的时候，使用类似下面的代码：

```
C++示例：强制让释放的内存包含垃圾数据
memset(pointer, GARBAGE_DATA, MemoryBlockSize(pointer));
delete pointer;
```

- 在删除或释放指针之后，把它们设为空值。一种常见的指针错误，是“悬空指针（dangling pointer）”，即使用一个已经被 delete 或者 free 的指针。指针错误难于检测的原因之一，就是这类错误有时并不产生任何征兆。尽管在删除指针后，再将其设为 null，并不能阻止你去读取一个悬空指针指向的数据，但这可以保证，当你向一个悬空指针写入数据时，会产生错误。这一错误可能引发一场巨大的灾难，以便发现问题。

```
C++示例：再删除指针之后将其设为NULL
memset(pointer, GARBAGE_DATA, MemoryBlockSize(pointer));
delete pointer;
pointer = NULL;
```

- 在删除变量之前检查非法指针。破坏程序的最好办法之一，就是在已经删除或者释放了一个指针之后，再 delete() 或 free() 它。不幸的是，很少有语言能够检测出这类问题。如果把已经释放的指针设置为空，就可以在使用或试图再次删除指针之前，对其进行检测。如果不把已释放的指针设为空值，就不能拥有这项选择。这就提出了对前面删除指针的代码的另一扩充：

```
C++示例：在删除指针之前，断言其不为NULL
ASSERT(pointer != NULL, "Attempting to delete null pointer.");
memset(pointer, GARBAGE_DATA, MemoryBlockSize(pointer));
delete pointer;
pointer = NULL;
```

- 跟踪指针分配情况。维护一份你已经分配的指针列表，这样就能让你在释放一个指针之前，检查它是不是位于列表里面。下面是一个例子，说明怎样修改普通指针删除代码，以增加这项功能：

```
C++示例：检查是否已经分配了某个指针
ASSERT(pointer != NULL, "Attempting to delete null pointer.");
if(IsPointerInList(pointer)){
    memset(pointer, GARBAGE_DATA, MemoryBlockSize(pointer));
    RemovePointerFromList(pointer);
    delete pointer;
    pointer = NULL;
} else {
    ASSERT(FALSE, "Attempting to delete unallocated pointer.");
}
```

- 编写覆盖子程序，集中实现避免指针问题的策略。从上面例子可看出，每次新建或删除指针的时候，都需要多写很多额外的代码。在本节所描述的技术里，有些是相互排斥或者前后重复的，而你却不希望在同一个代码库上面，应用多种相互冲突的策略。比如说，如果你维护了一份自己的合法指针列表，那么就不需要设置并检查狗牌值。通过对常见的指针操作，编写覆盖子程序，可以减少编程的工作量，并且降低错误几率。在 C++ 中，你可以使用下面两个子程序。

- SAFE_NEW 这个子程序调用 new 来分配指针，把这一新的指针，加入已分配指针列表中，然后将这一新分配的指针，返回给调用方子程序。它还可以在该子程序内，检查 new 操作的返回值，是否为空或是否抛出异常（是否发生“内存不足”错误），从而简化了程序其他部分的错误处理。
- SAFE_DELETE 这个子程序检查传递给它的指针，是否在已分配指针的列表里。如果它在列表里，就把该指针所指向的内存设置为垃圾数值，把该指针从列表中移除，再调用 C++ 的 delete 运算符释放该指针，并且把该指针设为空值。如果该指针不在别表里，那么 SAFE_DELETE 将显示一条诊断信息，并且终止程序运行。

这里我们用宏来实现 SAFE_DELETE 子程序，

C++示例：在删除指针的代码外加一层包裹

```
#define SAFE_DELETE(pointer){\
    ASSERT(pointer != NULL, "Attempting to delete null pointer.");\
    if(IsPointerInList(pointer)){\
        memset(pointer, GARBAGE_DATA, MemoryBlockSize(pointer));\
        RemovePointerFromList(pointer);\
        delete pointer;\
        pointer = NULL;\
    } else {\
        ASSERT(FALSE, "Attempting to delete unallocated pointer.");\
    }\
}
```

在 C++ 里，这一子程序会删除单个的指针，但是你还需要一个，实现相似功能的 SAFE_DELETE_ARRAY 子程序，来删除指向数组的指针。

通过把指针操作集中到这两个子程序中，你还可以使用 SAFE_NEW 和 SAFE_DELETE 在调试模式和产品模式下的表现有所不同。例如，当 SAFE_DELETE 在开发阶段检测到，试图释放空指针的操作时，它可能会终止程序的执行，但是在产品阶段，它可能只简单记录下这个错误，然后继续执行。你可以很容易把这种机制，应用在 C 中的 calloc 和 free，以及其他使用指针的语言中。

- 采用非指针的技术。指针比较难理解，容易用错，也容易导致依赖于具体机器的不可移植代码。如果你能想到任何替代指针的方案，而它又能工作得很合理，那么就应该去用它，避免这些令人头疼的问题。

(3)C++ 指针

C++ 指针引入了一些特定的使用指针和引用的方法。下面介绍一些适用于在 C++ 中使用指针的指导原则。

- 理解指针和引用之间的区别。在 C++ 中，指针 (*) 和引用 (&) 都能够间接地引用对象。对新手而言，这唯一的区别，似乎只是字面上 object->field 和 object.field 的不同。但事实上，最重要的区别，引用必须总是引用一个对象，而指针则可以指向空值，还有，引用所指向的对象，在该引用初始化之后，不能改变。

- 把指针用于“按引用传递”参数，把 const 引用用于“按值传递”参数。C++ 向子程序传递参数的默认方式，是传递值而不是传递引用。当你以传递值得方式，向一个子程序传递一个对象的时候，C++ 创建了该对象的一份拷贝，当该对象传递回调用方子程序的时候，又创建了一份拷贝。对于占用内存很大的对象而言，这种复制可能耗费大量时间和其他资源。因此，当你向一个子程序传递对象的时候，通常会希望避免复制该对象，这就意味着你希望按照引用来传递它，而不是按值来传递。然而，有时你可能希望具有“传值”的语义，也就是说，不能修改传入的对象，却用“传引用”的方式实现，即传递对象本身，而非它的拷贝。在 C++ 中，上述事项的解决方案，是使用指针来实现按引用传递，同时，这一术语可能听上去挺怪，用“const 引用”来实现按值传递。例如，

C++示例：参数传递方式：by reference和by value

```
void SomeRoutine(
```

```
        const LARGE_OBJECT &nonmodifiableObject,
        LARGE_OBJECT *modifiableObject
    );
```

这种方法还有额外的好处，那就是在被调用的程序里，对可修改的和不可修改的对象，做了语法上的区分。在可以修改的对象里，成员引用使用 object->member 的表示法；而不可修改的对象，成员引用使用 object.member 表示法。

- 使用 auto_ptr。如果你还没养成使用 auto_ptr 的习惯，那就努力吧！通过在离开作用域时自动释放内存，auto_ptr 能避免很多与常规指针相关的内存泄露问题。（现在 C++ 已经基本抛弃了 auto_ptr，改为推荐使用 shared_ptr。）
- 灵活运用智能指针。智能指针是指常规指针或“dumb”指针的一种替代品。它用起来与常规指针十分相像，但是针对资源管理、拷贝操作、对象构造和对象析构提供了更多控制。

13.3 全局数据

大多数有经验的程序员已经得出结论：使用全局数据的风险，比使用局部数据大。极富经验的程序员还认为，通过一些子程序来访问数据很有帮助。

(1) 与全局数据有关的常见问题

模块化、信息隐藏，并结合使用设计良好的类，可能还算不上是绝对真理，但是它们能极大地提升大型程序的可理解性和可维护性。一旦明白了这一点，你就会努力去写出与全局变量和外界联系尽可能少的子程序和类来。人们指出了使用全局数据的许多问题，实际上这些问题都可以归结到下面集中情况。

- 无意间修改了全局数据。你可能会无意间在某处修改了一个全局变量的值，然后错误地认为它在其他的位置，还是保持不变的。
- 与全局数据有关的奇异的和令人激动的别名问题。别名指的是两个或更多不同名字，说的是同一个变量。当一个全局变量被传递给一个子程序，然后该子程序将它既用作全局变量，又用作参数使用的情况下，就会出现这种情况。
- 与全局数据有关的代码重入问题。可以由一个以上的线程访问的代码，正变得越来越常见。多线程代码造成了这样一种可能性，那就是全局数据将不但在多个子程序之间共享，而且也将在同一个程序的不同拷贝之间共享。在这种环境下，你必须确保即使一个程序的多个拷贝同时运行，全局数据也会保持其意义。
- 全局数据阻碍代码重用。要把一个程序里的代码，应用于另一个程序，你必须能够把它从第一个程序里取出，然后插入到另一个程序里。如果你想重用的类读或写了全局数据，那么你就无法简单地把它插入到新的程序里，你将不得不修改新的程序或旧类，以便让它们相容。
- 与全局数据有关的非确定的初始化顺序事宜。如果在初始化一个文件中的全局变量时，使用了另一个不同文件中，初始化的全局变量，那么除非你用明确的手段，来确保这两个变量，能按照正确的顺序初始化，否则请不要对第二个变量的取值下任何赌注。
- 全局数据破坏了模块化和智力上的可管理性。创建超过几百行代码的程序，核心便是管理复杂度。你能够在智力上管理一个大型程序的唯一方法，就是把它拆分成几部分，从而可以在同一时间只考虑一部分。模块化就是你手中可以使用的，把程序拆分成几部分的强大工具。全局数据使得你的模块化能力大打折扣。如果你用了全局数据，你无法在同一时间只关注一个子程序。你不得不关注一个子程序，以及使用了同样全局数据的其他所有子程序。尽管全局数据并没有完全破坏程序的模块化，但是却削弱了它，而这已经是很充分的理由，要求你去寻找问题的更好解决方案了。

(2) 使用全局数据的理由

- 保存全局数值。有时候你会有一些在概念上，用于整个程序的数据。这可能是一个用于表示程序状态的变量，例如，交互式模式或命令行模式、正常模式或错误恢复模式等的模式标识。

- 模拟具名常量。像 Python 语言不支持具名常量，可以用全局变量代替它们。例如，你可以用取值分别为 1 和 0 和全局变量 TRUE 和 FALSE 来代替字面量 1 和 0。一旦采用了这种方法，那么日后再修改代码就会更容易了，而且这样的代码会更方便阅读。
- 模拟枚举类型。可以在 Python 等不直接支持枚举类型的语言里，用全局变量来模拟枚举类型。
- 简化对极其常用的数据的使用。
- 消除流浪数据。有时候你把数据传递给一个子程序或类，仅仅是因为想把它传递给另一个子程序或类。当调用链中间的子程序并不使用这一对象的时候，这一对象就被称为“流浪数据”。使用全局变量可以消除流浪数据。

(3) 只有万不得已时才使用全局数据
在你选择使用全局数据之前，请考虑下面这些替换方案。

- 首先把每一个变量设置为局部的，仅当需要时才把变量设置为全局的。开始的时候，先把所有的变量设置为单一子程序内部的局部变量。如果你发现还需要在其它位置用到它们，那么再把它们转变为类里的 private 或 protected 变量。如果你最终发现必须要把它们转变为全局变量，那么就转变它们。不过请先确定除此之外，别无选择。如果你一开始就把变量设置为全局的，那么你将永远不可能把它转变为局部的；反之，如果你开始时把变量设置为局部的，那么你可能永远也不需要把它转变为全局的。
- 区分全局变量和类变量。有些变量由于要被整个程序访问，因此时真正的全局变量。其他只在一组特定的子程序里，被频繁使用的，实际是类变量。在频繁使用某个类变量的子程序组里，你可以采用任何希望的方式来访问它。如果类外部的子程序需要使用该变量，那么就用访问器子程序来提供对该变量的访问。不要直接访问类变量，好像它们是全局变量一样，即便你的编程语言允许你这么做。这一建议等价于模块化。
- 使用访问器子程序。创建访问器子程序，是避免产生与全局数据相关问题的主要方法。

(4) 用访问器子程序来取代全局数据
你用全局数据能做的任何事情，都可以用访问器子程序做得更好。使用访问器子程序是实现抽象数据类型，和信息隐藏的一种核心方法。即使你不希望使用装备齐全的抽象数据类型，你仍然可以用访问器子程序来控制你的数据，并保护你免受变化的困扰。

- 访问子程序的优势。
  - 你获得了对数据的集中控制。如果你日后发现了一种，更合适的实现该结构的方法，那么你无须到处修改引用该数据的代码；所需做的修改不会波及整个程序，它被限制在访问器子程序内部。
  - 你可以确保对变量的所有引用都得到了保护。如果你用 stack.array[stack.top] = newElement 这样的语句，向栈中压入元素，你会很容易就忘记检测栈溢出，从而犯下严重的错误。如果你使用了访问器子程序，例如 PushStack(newElement)，你就可以把栈溢出检测写到 PushStack() 子程序里。这一检测会在每次调用该子程序的时候自动执行。
  - 你可以自动获得信息隐藏的普遍益处。你可以修改一个访问器子程序的内部代码，而无须涉及程序的其余部分。
  - 访问器子程序可以很容易地转变为抽象数据类型。访问器子程序地一项优点是，你可以创建一个很难用全局数据，来直接创建地抽象层。例如，与其写 if lineCount > MAX_LINES，访问器子程序让你能采用 if PageFull()。这样一种小修改说明了这个 if lineCount 检测地用意，代码也实现了所表示的用途。这是对可读性的一点小小改进，但是如果能坚持重视这些细节，就能写出同那些东拼西凑到一起的代码，迥然不同的精致程序了。
- 如何使用访问器子程序。下面是有关访问器子程序的理论和实践总结：把数据隐藏到类里面。用 static 关键字，或者它的等价物来声明该数据，以确保只存在该数据的单一实例。写出让你可以查看并修改该数据的子程序来。要求类外部的代码，使用该访问器子程序来访问数据，而不是直接操作它。例如，如果你有一个全局状态变量

g_globalStatus，用于描述这个程序整体状态，你可以创建两个访问器子程序：globalStatus.Get() 和 globalStatus.Set()，它们所执行的操作和名字所描述的一样。这些子程序访问了隐藏在类内部的，一个取代 g_globalStatus 的变量。程序的其余部分可以借助 globalStatus.Get() 和 globalStatus.Set()，获得原有全局变量所能提供的所有好处。如果你的语言不支持类，你仍然可以创建访问器子程序，来操纵全局数据，但必须制定严格的代码编写标准，限制对全局数据的使用，以代替编程语言内置的约束。下面是在你的语言没有内置对类的支持的情况下，使用访问器子程序来隐藏全局变量的一些详细的指导原则。

- 要求所有的代码通过访问器子程序来存取。一个好习惯是要求所有的全局数据都冠以 g_ 前缀，并且除了该变量的访问器子程序以外，所有的代码都不可以访问具有 g_ 前缀的变量，其他全部代码都通过访问器子程序来存取该数据。
- 不要把你所有的全局数据都仍在一处。如果把所有的全局数据都堆到一起，然后为它编写一些访问器子程序，你可以消灭所有与全局数据有关的问题，但这也使代码丧失了信息隐藏，和抽象数据类型所带来的好处。既然已经在编写访问器子程序，就请花些时间考虑每一个全局数据属于哪个类，然后把该数据和它的访问器子程序，以及其他的数据和子程序打包放入那个类里面。
- 用锁来控制对全局变量的访问。锁定要求在使用或则更新一个全局变量之前，该变量必须被 check out，在用完这一变量之后，再把它 check in 回去。在使用期间，如果程序的其余部分尝试要将它 check out，那么锁定子程序就会显示一条错误消息，或者触发一个断言。
- 在你的访问器子程序里构建一个抽象层。要在问题域这一层次上构建访问器子程序，而不是在细节实现层次上。这种方法会为你的代码带来更好的可读性，同时防止在代码编写过程中，不小心修改到实现细节。
- 使对一项数据的所有访问，都发生在同一个抽象层上。如果你用一个访问器子程序对一个结构体执行了某种操作，那么在对此结构体执行任何其他操作时，你同样也应该使用一个访问器子程序。如果你用某个访问器子程序读取该结构体，那么就用另一个访问器子程序写入该结构体。如果你调用 InitStack() 来初始化栈，就应该调用 PushStack() 来往栈上压值，这样就为该数据创建了一个一致的视角。但如果通过 value=array[stack.top] 来从栈中弹出数据，你所创建的对该数据的操作，就不一致。这种不一致性会使得其他人很难理解该代码。应该创建一个 PopStack() 子程序来代替 value=array[stack.top]。

(5) 如何降低使用全局数据的风险

在许多情况下，全局数据事实上就是没有设计好，或实现好的，类中的数据。在少数情况下，一些数据的确需要作为全局数据，但是可以使用访问器子程序对其进行封装，从而最大限度地减少发生问题的可能性。在剩余的极少数情况下，你真的需要使用全局数据。下面是一些降低使用全局数据风险的措施。

- 创建一种命名规则来突出全局变量。在对全局变量进行操作时，为全局变量命以更醒目的名字，可以让你少犯错。
- 为全部的全局变量创建一份注释良好的清单。
- 不要用全局变量来存放中间结果。如果你需要为一个全局变量计算新值，那么应该在计算结束后，再把最终结果赋给该全局变量，而不要用它来保存计算的中间结果。
- 不要把所有的数据，都放在一个大对象中并到处传递，以说明你没有使用全局变量。把所有一切都放到一个大对象里，可能会满足不使用全局变量的要求，但是这样做，纯粹是一种负担，它无法真正带来封装所能带来的好处。

14 组织直线型代码

尽管组织直线型代码是一个相对简单的任务，但代码结构上的一些微妙之处，还是会对代码的质量、正确性、可读性和可维护性带来影响。

14.1 必须有明确顺序的语句

最容易组织的连续语句，是那些顺序相关的语句，例如，

Java示例：有前后依赖关系的语句

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

在下面代码中，依赖关系就不那么明显了：

Java示例：不太明显的前后依赖关系语句

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

在这个例子中，对季度收入的计算，要求假定月收入已经计算出来了。熟悉会计学的人，可能会告诉你，必须先计算季度收入，然后才能计算年收入。这就是一种依赖关系，但是通过阅读代码是看不出来的。下面代码中的顺序依赖关系也不明显，

Visual Basic示例：隐藏了语句的前后依赖关系

```
ComputeMarkertingExpense
ComputeSalesExpense
ComputeTravelExpense
ComputePersonnelExpense
DisplayExpenseSummary
```

假定 ComputeMarketingExpense() 会初始化类的成员变量，以便其他所有子程序都能把它们的数据放进去；在这种情况下，它需要在其他子程序被调用之前，被调用。然后进通过阅读代码不知道这一点。如果语句之间存在依赖关系，并且这些关系要求你把语句按照一定的顺序加以排列，那么请设法使得这些依赖关系变得明显。下面是一些用于组织语句的简单原则。

- 设法组织代码，使依赖关系变得非常明显。上面的例子中，应该另外写一个子程序，如 InitializeExpenseData()，来初始化成员变量。这个子程序的名字，清楚地表明了，程序员应该在运行其他的开支计算子程序之前，调用它。
- 使子程序名能突显依赖关系。上面的例子中，ComputeMarketingExpense() 的命名时错误的，因为它做的不仅仅是计算 Marketing 费用；它还初始化了成员数据。如果你反对再写一个子程序，来初始化数据，那么至少要给 ComputeMarketingExpense() 一个能够反映它所执行的全部功能的名字，例如，ComputeMarketingExpenseAndInitializeMemberData()。你也许会说这个名字太糟糕了，因为它太长，但是它却描述了这个子程序做了什么，实际上是这个子程序本身太糟糕了。
- 利用子程序参数明确显示依赖关系。例如，

  Visual Basic示例：暗示顺序依赖关系的数据和子程序调用

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

  也可以用数据来表明执行顺序并不重要，例如

  Visual Basic示例：表示没有顺序依赖关系的数据

  ```
  ComputeMarketingExpense(marketingData)
  ComputeSalesExpense(salesData)
  ComputeTravalExpense(travelData)
  ```

```
ComputePersonnelExpense( personnelData )
DisplayExpenseSummary( marketingData , salesData , travelData , personnelData )
```

- 用注释对不清晰的依赖关系进行说明。首先要尽力写没有顺序依赖关系的代码；其次尽力写依赖关系明显的代码；如果你还担心某一项依赖关系不够清楚，那么就用文档说明它。对不清晰的依赖关系进行说明，是描述编程意图的一个方面，它对编写出可维护、以修改的代码来说是至关重要的。

- 用断言或错误处理代码，来检查依赖关系。如果代码非常重要，你可以用状态变量，以及错误处理代码或断言，来对关键的顺序依赖关系做出说明。例如，在类的构造函数里面，你可以把一个名为 isExpenseDataInitialized 的类成员变量，初始化为 false；然后在 InitializeExpenseData() 中，把 isExpenseDataInitialized 设置为 true。每一个依赖于已初始化的 expenseData 的函数，就可以在对 expenseData 做出其他操作之前，检查 isExpenseDataInitialized 有没有被设为 true。

14.2 顺序无关的语句

你也许见过这种情形，即代码中若干语句或语句块的先后顺序上看，完全没有关系。一条语句并不依赖于或者在逻辑上承接另一条语句。但是顺序的确对可读性、性能和可维护性有影响，而且当缺少执行顺序依赖关系的时候，你可以用第二标准来判断语句，或者代码块的顺序。这其中的指导原则，就是就近原则：把相关的操作放在一起。

(1) 使代码易于自上而下地阅读

作为一条普遍性原则，要让程序易于自上而下阅读，而不是让读者地目光跳来跳去。专家们认为自上而下的顺序，对提高可读性最有帮助。简单地让控制流在运行时，自上而下地运行还不够。如果有人在阅读你代码的时候，不得不搜索整个应用程序，以便找到所需的信息，那么就应该重新组织你的代码了。例如，

C++示例：跳来跳去的糟糕代码

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

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

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

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

假设你希望知道 marketingData 是怎么计算出来的，那就必须从最后一行开始，跟踪所有对 marketingData 的引用，直至回到第一行。下面是组织得更好的代码，

C++示例：组织良好的顺序代码，能从头到尾阅读

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

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

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

这段代码在很多方面都好。把对每一个对象的引用都放在一起；把它们“局部化”了。对象“存活”的代码行数非常少。然而也许最重要的是，这段代码现在的样子，说明它可以拆分为分别结算 marketing、sales 和 travel 数据的子程序。

(2) 把相关的语句组织在一起

一些语句之所以相关，是因为它们处理了相同的数据、执行了相似的任务，或者具有某种执行顺序上的依赖关系。检查相关的语句是不是组织得很好起来得一种简便方法是，打印出你得子程序代码，然后把相关的语句画上框。如果这些语句排列得很好，其中的方框是不会彼此交叠的。

15 条件语句

15.1 if 语句

根据所用语言的不同，你可能使用几种 if 语句中的任何一种。其中最简单的是简单 if 或 if-then 语句。if-then-else 稍微复杂一点，而连续一组 if-then-else 所构成的语句串是最为复杂的。

(1) 简单 if-then 语句

在写 if 语句的时候，请遵循下述指导原则。

- 首先写正常代码路径，再处理不常见情况。在编写代码时，要使得正常情况的执行路径，在代码中是清晰的。确认那些不常见的情况，不会遮掩正常的执行路径。这对可读性和代码性能来说都很重要。
- 确保对于等量的分支是正确的。请不要用“>”代替”>=”，或用“<”代替“<=”，这类似于在访问数组，或计算循环下标的时候，犯下 off-by-one 错误。在循环里，要仔细考虑端点，已避免犯 off-by-one 错误。在条件语句里，也要仔细考虑条件是否同实际情况相符，避免犯同样的错误。
- 把正常情况的处理放在 if 后面，而不要放在 else 后面。把你认为正常出现的情况，放在前面处理。这符合把决策的结果代码，放在尽可能靠近决策位置的一般原则。
- 让 if 子句后面跟随一个有意义的语句。有时候你会看到下例中的这种代码，其中 if 子句是空的：

```
Java示例：空的if子句
if(SomeTest)
    ;
else {
    // do something
    ...
}
```

哪怕仅仅为了少写那个额外的空语句行和 else 代码行，大多数有经验的程序员也都会避免这么编写代码。可以修改为，

```
Java示例：空if子句转换后的代码
if(!SomeTest){
```

```
    // do something
    ...
}
```

- 考虑 else 子句。如果你认为自己只需要一个简单的 if 语句，那么请考虑你是否真的不需要一个 if-then-else 语句。当你有一个不包含 else 部分的 if 测试的时候，除非其原因显而易见，否则请用注释来解释为什么在这里 else 子句是没有必要的。
- 测试 else 子句的正确性。在测试代码的时候，你可能会认为只有主句子，即 if 子句，需要测试；然而，如果有可能测试 else 子句的话，也一定要测试它。
- 检查 if 和 else 子句是不是弄反了。

(2)if-then-else 语句串

在不支持或只部分支持 case 语句的语言里，你会发现自己常常要写 if-then-else 检测串。在写这种语句串的时候，请注意下述指导原则。

- 利用布尔函数调用，简化复杂的检测。
- 把最常见的情况放在最前面。可以让阅读代码的人读最少的代码，就能找出正常情况的处理代码。同时，由于把在执行最常见情况代码之前，所需的其他判断减到最少，代码效率也得到了提高。
- 确保所有的情况都考虑到了。写一个放在最后的 else 子句，用出错消息或者断言，来捕捉那些你不考虑的情况。
- 如果你的语言支持，请把 if-then-else 语句串，替换成其他结构。

15.2 case 语句

下面是如何有效使用 case 语句，给出一些指导原则。

(1) 为 case 选择最有效的排列顺序

- 按字母顺序或按数字顺序排列各种情况。如果所有情况的重要性都相同，那么就把它们按 A-B-C 顺序加以排列，以便提高可读性。
- 把正常的情况放在前面。如果有一个正常的情况，和多个异常情况，那么就把那个正常的情况放在最前面。用注释来说明它是正常的情况，而其他的属于非正常情况。
- 按执行效率排列 case 子句。把最经常执行的情况放在最前面，最不常执行的放在后面，这样做有两个方面的好处。首先，阅读程序的人，可以很容易找到最常见的情况。另外，对于检索情况列表，找出某个具体情况的读者很可能会对最常见的情况感兴趣，而把常见的情况，放在代码的上部，会加速这种检索。

(2) 使用 case 语句的诀窍

- 简化每种情况对应的操作。简短的情况处理代码，会使 case 语句的结构更加清晰。如果某种情况执行的操作非常复杂，那么就写一个子程序，并且该情况对应的 case 子句中调用它，而不要把代码本身放进这一 case 子句里。
- 不要为了使用 case 语句，而刻意制造一个变量。case 语句应该用于处理简单的、容易分类的数据。如果你的数据并不简单，那么就使用 if-then-else 语句串。为使用 case 而刻意构造出来的变量，很容易把人搞糊涂，你应该避免使用这种变量。

Java示例：刻意制造一个虚假的case变量，糟糕的实践

```
action = userComman[0];
switch(action){
    case 'c':
```

```
            Copy();
            break;
        case 'd':
            DeleteCharacter();
            break;
        case 'f':
            Format();
            break;
        case 'h':
            Help();
            break;
        ...
        default:
            HandleUserInputError(ErrorType.InvalidUserCommand);
    }
```

这里控制 case 语句的变量是 action。在本例中，action 是通过截取 userCommand 字符串，一个用户输入的字符串，的第一个字符创建的。如果用户输入 copy，则会正确调用 Copy() 子程序；但是如果用户输入 clambake，则会与预期不一致，并且 default 子句也会失效。与其刻意制造一个本不适用于 case 的假冒变量，不如使用一个 if-then-else-if 检测串，来检查整个字符串。

- 把 default 子句只用于检查真正的默认情况。
- 利用 default 子句来检测错误。如果一条 case 语句中的默认子句，既没有用来做其他的处理，按照正常执行顺序也不太可能会发生，那么就向里面加入一条诊断消息。如果把默认子句用于错误检测之外的其他目的，那就意味着每一种情况的选择都是正确的。请检查以确认每一个可能进入 case 语句的值都是合法的。如果你发现了一些不合法的值，那么就重写这些语句，让默认子句去执行错误检测。
- 在 C++ 和 Java 里，避免代码执行越过一条 case 子句的末尾。类似于 C 的语言，不会自动地跳出每一种情况的执行。相反，你必须明确地为每一 case 子句写结束（break 语句）。如果你不这么做，程序就会越过其末尾并继续执行下一 case 子句的代码。
- 在 C++ 里，在 case 末尾明确无误地标明，需要穿越执行的程序流程。如果要你故意让代码越过某一 case 子句的末尾，那么就在相应的位置给出明确的注释，解释为什么要这样编写代码。

16 控制循环

16.1 选择循环的种类

(1) 什么时候使用 while 循环

如果你预先并不知道要迭代多少次，那么就使用 while 循环。执行每通过这种循环一次，while 只做一次循环终止的检测，而且有关 while 循环的最主要事项，就是决定在循环开始处，还是结尾处做检测。

- 检测位于循环的开始。对于在开始处进行检测的循环，在 C++ 等大多数语言里，你可以使用 while 循环。在其他语言里，你也可以模拟 while 循环。
- 检测位于循环的结尾。你也许偶尔会遇到这种情况：需要一个灵活的循环，但是该循环至少需要执行一次。在这种情况下，你可以用一个在结尾处，做条件检测的 while 循环。在 C++ 里，你可以用 do-while。

(2) 什么时候用 loop-with-exit（带退出）的循环

带退出的循环，就是终止条件出现在循环中间，而不是开始或末尾的循环。

- 正常的带退出循环。一个带退出循环通常由循环头、循环体和循环尾组成。例如，

```
Visual Basic示例：带退出循环的一般结构
Do
    ...
    If(some exit condition) Then Exit Do
    ...
Loop
```

在使用这种循环的时候，请把下面这些细节考虑进去：

- 把所有的退出条件放在一起。如果把它们写得到处都是，会使得某些终止条件，在调试、修改或测试得时候被忽略。
- 用注释来阐明操作意图。如果你在一个不直接支持带退出循环的语言里，使用直接退出法，那么就应该用注释，把你做的事情解释清楚。
- 带退出的循环，也是单入单出的结构话控制结构，也是一种首选的循环控制。事实证明，它比其他种类的循环，都要容易理解。带退出循环结构，要比其他循环结构更接近于，人类思考迭代型控制的方式。

(3) 何时使用 for 循环

如果你需要执行次数固定的循环，那么 for 循环就是一个很好的选择。for 循环的关键之处在于，你在循环头处把它写好后就可以忘掉它了，无需再循环的内部做任何事情去控制它。如果存在一个必须使执行从循环中跳出的条件，那么就应该改用 while 循环。类是地，不要在 for 循环里通过直接修改下标值得方式，迫使它终止，在这种情况下，应该改用 while 循环。for 循环就是为了简单的用途，更复杂的循环最好用 while 循环去处理。

16.2 循环控制

循环会出什么错误呢？任何一种答案，都可以归结到下面所说的问题之一：忽略或错误地对循环执行初始化、忽略了对累加变量，或其他与循环有关的变量执行初始化、不正确的嵌套、不正确的循环终止、忽略或错误地增加了循环变量地值、以及用不正确地循环下标，访问数组等等。你可以采用两种方法，来阻止这些错误的发生。首先，减少能影响该循环各种因素的数量。其次，把循环内部，当作一个子程序看待，把控制尽可能地放到循环体外。把循环体执行地条件表述清楚，不要让读者看了循环体以后，才明白循环的控制。应该把循环看作是一个黑盒子：外围程序只知道它的控制条件，却不知道它的内容。

(1) 进入循环

在进入循环的时候，使用下述知道原则。

- 只从一个位置进入循环。
- 把初始化代码仅紧放在循环里面。就近原则主张把相关的语句放在一起，如果相关的语句分散在子程序的各处，那么在修改子程序的时候就容易忽略它们，导致不正确的修改。
- 用 while(true) 表示无限循环。普遍认为 while(true) 是 C++、Java 等表示无限循环的标准写法。
- 在适当的情况下多使用 for 循环。for 循环把循环控制代码集中在一处，从而有助于写成可读性强的循环。在 for 循环中，将所有相关的代码全部写在循环的顶部，因此修改起来更加容易。
- 在 while 循环更适用的时候，不要使用 for 循环。对 C++ 中 for 循环结构的一种很常见的陋习，是很随便地用 while 循环的内容，来填充 for 循环的循环头。

```
C++示例：胡乱把while循环体，填充到for循环头
// read all the records from a file
for(inputFile.MoveToStart(),recordCount=0;!inputFile.EndofFile();
    recordCount++){
```

```
        inputFile.GetRecord();
    }
```

应该把 for 循环的位置保留给循环控制语句：那些用于初始化循环和终止循环，或者用于使循环趋向于终止的语句。在上面的例子中，位于循环体内的 inputFile.GetRecord() 语句将循环推向终止，但是有关 recordCount 的语句却没有这种作用；它们属于内务语句，并不控制循环的进度。把 recordCount 语句放在循环头内，但却把 inputFile.GetRecord() 语句置于其外，会产生误导：让人错误地认为是 recordCount 在控制这个循环。如果你希望在这种情况下使用 for 循环，而不是 while 循环，那么就应该只把循环控制语句放在循环头里，把其他的都放在外面，例如

C++示例：合乎逻辑但不常规的for循环头

```
recordCount=0;
for (inputFile.MoveToStart();
    !inputFile.EndOfFile();
    inputFile.GetRecord()){
        recordCount++;
    }
```

在这个例子里，循环头部得内容，全部与控制循环有关。其中 inputFile.MoveToStart() 语句对循环执行初始化，!inputFile.EndOfFile() 语句检测循环是否已经终止，inputFile.GetRecord() 语句把循环推向终止。对 recordCount 由影响得那些语句，并不直接把循环推向终止，因此很适宜放在循环头部之外。用 while 循环做这件工作可能更合适些，但是至少这段代码对循环头得使用是符合逻辑得。例如

C++示例：while循环得适当用法

```
// read all the records from a file
inputFile.MoveToStart();
recordCount=0;
while(!inputFile.EndOfFile()){
    inputFile.GetRecord();
    recordCount++;
}
```

(2) 处理好循环体

- 用“{”和“}”把循环中的语句括起来。任何时候都要在代码中使用括号，它们不会增加任何运行时所需的时间或存储空间，只会增加可读性，并且防止修改代码时出错。它们是一种很好的预防性编程实践。
- 避免空循环。

C++示例：空循环

```
while((inputChar=dataFile.GetChar())!=CharType_Eof){
    ;
}
```

C++示例：将空循环改为有循环体的循环

```
do{
    inputChar = dataFile.GetChar();
} while (inputChar!=CharType_Eof)
```

- 把循环内务操作，要么放在循环的开始，要么放在循环的末尾。循环内务操作是指像 i=i+1 或 j++ 这样的表达式，它们的主要目的不是完成循环工作，而是控制循环。一般而言，在循环之前开始的那些变量，也就是需要在循环内务部分里处理的变量。
- 一个循环只做一件事。循环应该和子程序一样，每个循环只做一件事，并且把它做好。如果用两个循环会导致效率低下，而使用一个循环很合适；那么还是先把代码写成两个循环，并注明可以把它们合并起来以提高效率，然后等到测试数据显示，程序的这一部分性能低下时，再去合并它们。

(3) 退出循环

- 设法确认循环能够终止。这是基本要求。在脑海里模拟执行这个循环，直到你可以确认无论在任何情况下，循环都能终止。要考虑到正常的情况、端点，以及每一种异常情况。
- 使循环终止条件看起来很明显。关键在于把控制都放在同一个地方。
- 不要为了终止循环，而胡乱改动 for 循环的下标。事实上，所有好的程序员都会避免这么做，这是业余爱好者的标志。
- 避免出现依赖于循环下标最终取值的代码。在循环终止后，使用循环下标值，是很不好的。
- 考虑使用安全计数器。安全计数器是一个变量，你在每次循环之后，都递增它，以便判断该循环的执行次数是不是过多。如果程序中发生的错误，将带来灾难性的后果，那么就可以用安全计数器，来确保所有的循环都终止了。
- 提前退出循环。break 语句使循环通过正常的方式退出，程序会从循环后面的第一条语句开始执行下去。最为循环控制的辅助语句，continue 和 break 非常相似；然而，continue 不会让程序从循环退出，而是让程序跳过循环体的余下部分，从该循环的一下次迭代的开始位置继续执行。
  - 考虑在 while 循环中使用 break 语句，而不是布尔标记。在有些情况下，通过往 while 循环中，加入布尔标志来实现退出循环，可能使循环变得很难理解。用 break 来取代一系列的 if 检测，有时候你可以从循环中移除一些缩进层次，从而简化循环控制。把多个 break 条件放到一些独立的语句里，并且让它们靠近产生 break 的代码，就能减少嵌套，并且让循环更容易阅读。
  - 小心那些有很多 break 散布其中的循环。一个循环包含很多的 break，有可能意味着程序员对该循环的结构，或对循环中的角色，缺乏清晰的认识。在大量使用 break 的场合中，用一些的循环，而非一个含有多个出口的循环，可能会使表达更为清晰。
  - 在循环开始处用 continue 进行判断。一种使用 continue 的好方法，是在循环开始处，做完条件判断后，让代码跳过剩下的循环体，继续执行。这样使用 continue，可以避免用一个，需要让整个循环，都缩进的 if 判断；反之，如果 continue 出现在循环中部或末尾，那么就应该改用 if。
  - 如果语言支持，请使用带标号 break 结构。
  - 使用 break 或 continue 时要小心谨慎。使用 break 消除了，把循环看作黑盒子的可能性。使用 break 使阅读代码的人，必须去读循环体，才能理解循环是如何控制的。这使循环变得更难以理解。

(4) 检查端点

对一个简单循环，通常需要注意三种情况：开始的情况，任意选择的中间情况，以及最终的情况。在你创建循环的时候，应在脑海里运行这三种循环情况，以确认该循环不会出现任何 off-by-one 错误。如果有一些特殊情况，是与第一次或最后一次的情况都不同，那么也要检查它们。如果循环中包含了复杂的计算，那么就拿出计算器，手动检查这些计算是否正确。是否愿意执行这种检查，是高效程序员和低效程序员之间的一项关键差别。高效的程序员，既会在脑海里进行模拟，也会手动地执行运算，因为他们知道这些手段有助于找出错误来。低效的程序员，会随意地做一些试验，直到他们找到了一种看上去能工作的组合。如果某个循环，没有按照想象的那样工作，低效的程序员可能会把 < 改成 <=。如果还不行，他们就会把循环下标加 1 或减 1。这样做最终可能会碰出正确的组合来，也可能把原有的错误，改成了另一个更微妙的错误。即使这样随意的开发过程，能够产生出一个正确的程序，这些程序员也不明白为

什么这个程序是正确的。你可以从头脑模拟和手工运算中，获益多多。这种智力训练带来的好处是：在最初的编码阶段少犯错误，在调试阶段更快地找出错误，以及从整体上更好地理解应用程序。它意味着：你能够真正理解你的代码，是如何工作，而不是瞎猜。

(5) 使用循环变量

- 用整数或枚举类型，表示数组和循环的边界。通常来说，循环计数器应该是整数，浮点数递增有时会有问题。例如 1.0+26742897.0，可能会是 26742897.0，而不是 26742898.0
- 在嵌套循环中，使用有意义的变量名，来提高其可读性。数组的下标和循环的下标，常用同一个变量。有意义的数组下标名字，既能表明循环的用途，也能表明所访问的那部分数组的用途。
- 用有意义的名字，来避免循环下标串话。习惯性地使用 i、j 和 k，可能会导致下标串话。一般说，如果某个循环体内，有多余两三行地代码，或者它可能会增长，或者它位于一组嵌套的循环里，那么就应该避免使用 i、j 和 k。
- 把循环下标变量的作用域，限制在本循环内。

(6) 循环应该有多长

循环的长度，可以用代码行数，或者嵌套层次来衡量。

- 循环要尽可能地短，以便能够一目了然。如果你常常在显示器上看循环，而你的显示器能够显示 50 行代码，那么就应该把循环的长度，限制在 50 行以内。
- 把嵌套限制在 3 层以内。研究表明，当嵌套超出 3 层以后，程序员对循环的理解能力，会极大地降低。
- 把长循环的内容移到子程序里。如果循环设计好，通常可以把循环体内的代码，移到一个或几个子程序里，并在循环体内加以调用。
- 要让长循环格外清晰。长度会增加复杂度。如果你写的是短循环，那么就可以使用 break、continue、多个出口、复杂的终止条件等，有风险的控制结构。如果你写的循环比较长，并且担心给阅读者带来不便，那么就给它写一个单一出口，并且要保持退出条件清晰。

16.3 由内而外，轻松创建循环

如果你在编写复杂循环的时候遇上麻烦，可以使用一种简单的技巧来让它从一开始就正确。下面就是一般的处理过程。从一种情况开始，用字面量（literal）来编写；然后缩进它，在外面加上一个循环，然后用循环下标，或计算表达式，替换那些字面量。如果需要，在它的外面再套上一个循环，然后再替换掉一些字面量。根据你的需要，持续这一过程。等你做完以后，再加上所有需要的初始化。由于你是从简单的情况开始，并且有内向外生成代码的，因此你可以把这一过程看作是由内而外的编码。

假设你正在为一家保险公司开发程序。其中的寿险费率，要根据人员年龄和性别的不同，而变化。你的职责是开发一个，能够计算一组人员的，人寿保险费用总额的子程序。需要写一个能够从列表中，取出每个人的费率，并能进行累加的循环。你应该像下面这样做。

首先，再注释里写下循环体需要执行的操作步骤。在你还没开始考虑语法、循环下标、数组下标等细节之前，把需要做的事情先写下来，会比在进入细节之后写，更容易一些。

```
-- get rate from table
-- add rate to total
```

然后，在还没编写整个循环之前，尽可能多地把循环体内的注释，转化成代码。在本例中，就是为每个人提取费率，并且把它们累加起来。使用的数据要明确、具体，而不要抽象。

```
rate = table[]
totalRate = totalRate + rate
```

例子中，假定 table 是保存了费率数据的数组。开始的时候，你无须关心数组的下标。Rate 变量用于存储，从费率表中取出的个人费率数据。与之相似，totalRate 变量用于存储总费用。

```
rate = table[census.Age][census.Gender]
totalRate = totalRate + rate
```

该数组是通过年龄和性别来访问的，因此用 census.Age 和 census.Gender 来做它的下标。在这个例子中，假定 census 是一个结构体，其中存储着各个人的信息。接下来的一部是给现有语句外面，加上一层循环。因为该循环的目的，是要计算一个组中每个人的费率，所以它的下标应该是人。

```
For person=firstPerson to lastPerson
    rate = table[census.Age][census.Gender]
    totalRate = totalRate + rate
End For
```

你在这里要做的，就是在现有代码的外面，加上一个 for 循环，然后缩进现有的代码，并把它们放在一个 begin-end 里面。最后，检查并确认依赖 person 循环下标的那些变量，都已经被推广了。在本例中，census 变量随 person 变化而变化，所以要适当地推广它。

```
For person = firstPerson to lastPerson
    rate = table[census[person].Age][census[person].Gender]
    totalRate = totalRate + rate
End For
```

最后，写出必要地初始化代码。在本例中，totalRate 变量是需要初始化的。

```
totalRate = 0
For person = firstPerson to lastPerson
    rate = table[census[person].Age][census[person].Gender]
    totalRate = totalRate + rate
End For
```

如果你必须在 person 循环之外，再加上一层循环，那么方法还是一样的。你不必严格地遵循上述步骤，这里的要点在于，从具体事件入手，在同一时间只考虑一件事，以及从简单的部分开始创建循环。在开发通用、更复杂循环的过程中，你迈的步子要小，并且每一步的目的要更容易理解。这样一来，你就可以减少在同一时间，需要关注的代码量，从而减少出错的可能。

16.4 循环和数组的关系

在许多情况中，循环就是用来操纵数组的，而且循环计数器和数组下标一一对应。例如，

Java示例：数组乘法

```
for(int row = 0; row<maxRows; row++){
    for (int column=0; column<maxCols; column++){
        product[row][column] = a[row][colum] * b[row][column];
    }
}
```

17 不常见的控制结构

17.1 子程序中的多处返回

多数语言都提供了某种半途退出子程序的方法。程序可以通过 return 和 exit 这类控制结构，在任何需要的时候，退出子程序。它导致子程序按照正常的退出途径终止，并把控制权转交给调用方子程序。下面给出一些使用 return 语

句的指导原则。

- 如果能增强可读性，那么就使用 return。在某些子程序里，一旦知道了答案，你会希望马上返回到调用方子程序。下面就是一个好例子，它演示了一种需要从子程序里的多个位置，返回的合理情况：

```
C++示例：子程序中的多处返回（好做法）
Comparison Compare(int value1, int value2){
    if(value1 < value2){
        return Comparison_LessThan;
    } else if (value1 > value2){
        return Comparison_GreaterThan;
    }
    return Comparison_Equal;
}
```

- 用防卫子句（早返回或早退出）来简化复杂的错误处理。如果代码中，必须要执行正常操作之前，做大量的错误条件检测，就很可能导致代码的缩进层次过深，并且遮蔽正常情况的执行路径，如下所示：

```
Visual Basic示例：遮盖了正常的执行路径
If file.validName() Then
    If file.Open() Then
        If encryptionKey.valid() Then
            If file.Decrypt(encryptionKey) Then
                ' lots of code
                ...
            End If
        End If
    End If
End If
```

从审美的角度来说，把子程序的主体，缩在 4 条 if 语句里，很难看，尤其是当最里层 if 语句的代码，非常多的时候。在这种情况下，如果先检查错误情况，用这些代码来为正常的执行路径情路，那么代码的布局，有时可能变得更清楚。例如，

```
Visual Basic示例：用防卫子句澄清正常路径
' set up, bailing out if errors are found
If Not file.validName() Then Exit Sub
If Not file.Open() Then Exit Sub
If Not encryptionKey.valid() Then Exit Sub
If Not file.Decrpy(encryptionKey) Then Exit Sub
' lots of code
...
```

上述代码很简单，看来采用这种技术实现的解决方案很整洁。但是产品代码通常要求，在发现错误的时候，做大量的内务或者清理操作。下面是一个更符合实际的例子：

```
Visual Basic示例：更实际地利用防卫子句，澄清正常路径地代码
' set up, bailing out if errors are found
If Not file.validName() Then
    errorStatus = FileError_InvalidFileName
    Exit Sub
```

```
End If

If Not file.Open() Then
    errorStatus = FileError_CantOpenFile
    Exit Sub
End If

If Not encryptionKey.valid() Then
    errorStatus = FileError_InvalidEncryptionKey
    Exit Sub
End If

If Not file.Decrypt(encryptionKey) Then
    errorStatus = FileError_CantDecryptFile
    Exit Sub
End If

' lots of code
...
```

对于产品规模代码，这种 Exit Sub 方法，将在处理正常情况之前，加入相当数量的代码。不过，这种 Exit Sub 方法的确避免了，第一个例子里的那种深层嵌套。如果把第一个例子中的代码也扩展开来，展示其中对 errorStatus 变量的赋值，那么相对而言，Exit Sub 在集中相关语句方面，做得更好。最后，Exit Sub 方法的可读性和可维护性，也会更好，而这是一片非常大的空白区域，所做不到的。

- 减少每个子程序中 return 的数量。如果在读子程序的后部时，你没有意识到，它从前面某个地方返回的可能性，想理解这个子程序就很困难。由此可见，使用 return 要十分审慎，只当它们能增强可读性的时候，才去使用。

17.2 递归

在递归里，一个子程序自己负责解决某个问题的一小部分，它还把问题分解成很多的小块，然后调用自己来分别解决每一小块。当问题的小部分很容易解决，而问题的大部分，也很容易分解成众多的小部分时，常常会用到递归。递归并不常用，但如果使用得谨慎，还是可以得到非常优雅得解。例如，其中得排序算法，就很好地使用了递归：

Java示例：使用递归地排序算法

```
void QuickSort(int firstIndex, int lastIndex, String[] names){
    if(lastIndex > firstIndex){
        int midPoint = Partition(firstIndex, lastIndex, names);
        QuickSort(firstIndex, midPoint-1, names);
        QuickSort(midPoint+1, lastIndex, names)
    }
}
```

在本例中，该排序算法把数组分成两部分，然后再调用自身，来分别为数组地每一部分排序。当它调用自身的时候，所使用的数组太小，而无须排序时，比如 lastIndex <= firstIndex，就会停止对自身的调用。对于某一小范围内的问题，使用递归会带来简单、优雅的解。再稍大一些范围里，使用递归会带来简单、优雅，但是难懂的解。对于大多数问题，它所带来的解，将会是及其复杂的，在那些情况下，使用简单的迭代，通常会比较容易理解。因此要有选择地使用递归。

(1) 递归的例子

假设你有一个表示迷宫的数据类型。从本质上来说，迷宫就是 i 一个网格，在网格的每一个点，你都有可能向上、下、左、右四个方向移动。你通常可以往不止一个方向移动。那么，你如何才能写出一个，能够走出如图所示的迷宫程序呢？如果用递归，答案就会显而易见。你从入口处开始，然后尝试所有可能的路径，直到找到走出去的路来。当你第

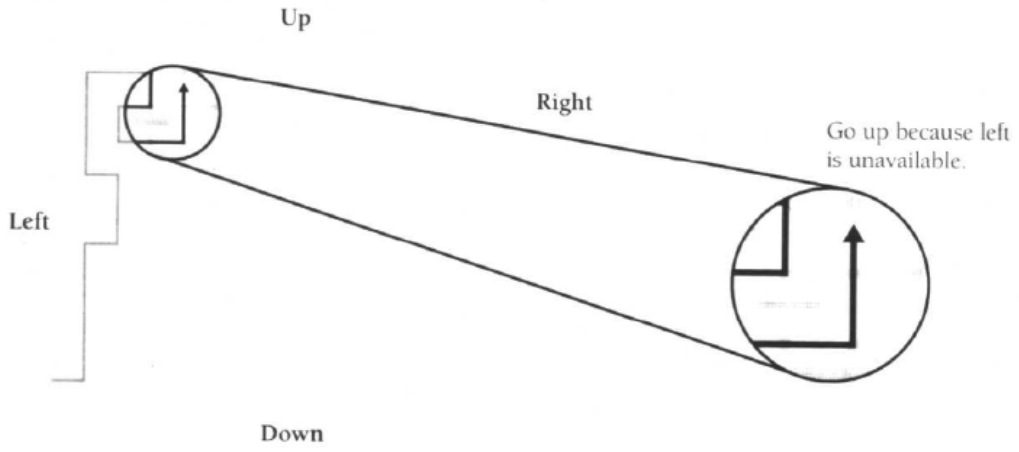


图17-1　递归式是对付复杂事物的很有价值的工具——在用于对付适当问题的时候

一次走到某一个点上的时候，你试着向左走。如果不能向左走，那就试着向上走或者向下走，如果这样还不行，那就试着向右走。你不用担心迷路，因为你经过某个点的时候，都会在那里留下标记，所以同一个点不会走两次。相应的递归代码如下：

C++示例：用递归穿越迷宫

```
bool FindPathThroughMaze(Maze maze, Point position){
    // if the position has already been tried, don not try it again
    if(AlreadyTried(maze, position)){
        return false;
    }

    // if this position is the exit, declare success
    if(ThisIsTheExit(maze, position)){
        return true;
    }

    // remember that this position has been tried
    RememberPosition(maze, position);

    // check the paths to the left, up, down, and to the right;
    // if any path is successful, stop looking
    if(MoveLeft(maze, position, &newPosition)){
        if(FindPathThroughmaze(maze, newPosition)){
            return true;
        }
    }

    if(MoveUp(maze, position, &newPosition)){
        if(FindPathThroughmaze(maze, newPosition)){
            return true;
        }
```

```
    }

    if(MoveDown(maze, position, &newPosition)){
        if(FindPathThroughmaze(maze, newPosition)){
            return true;
        }
    }

    if(MoveRight(maze, position, &newPosition)){
        if(FindPathThroughmaze(maze, newPosition)){
            return true;
        }
    }
    return false;
}
```

(2) 使用递归的技巧

- 确认递归能够停止。检查子程序，以确认其中含有一条非递归的路径。通常这意味着，该子程序中，含有一项判断，无须进一步递归，就能停下来。在迷宫例子中，AlreadyTried() 和 ThisIsTheExit() 两项判断，保证递归能够停止。
- 使用安全计数器，防止出现无穷递归。如果你在一种，不允许使用上述简单测试的环境中，使用递归，那么就用安全计数器来防止产生无穷递归。该安全计数器必须是一个，不随每次子程序调用，而重新创建的变量。可以用一个类成员变量，或者把该安全计数器，作为参数加以传递。例如，

```
Visual Basic示例：用安全计数器避免无穷递归。
Public Sub RecursiveProc(ByRef safetyCounter As Integer)
    If (safetyCounter > SAFETY_LIMIT) Then
        Exit Sub
    End If
    safetyCounter = safetyCounter + 1;
    ...
    RecursiveProc(safetyCounter)
End Sub
```

  如果你不希望把安全计数器，作为明确的参数传递，那么你可以使用 C++ 中的成员变量，或者其他语言中的等价物。

- 把递归限制在一个子程序内。循环递归（A 调用 B，B 调用 C，C 调用 A）非常危险，因为它很难检查。如果你有循环递归，那么通常你可以重新设计这些子程序，以便把递归限制在一个单一的子程序内。如果你做不到这一点，并且仍然认为原来的递归是最好的解决方案，那么作为一种保险的递归策略，就请使用安全计数器把。
- 留下栈空间。用了递归以后，你将无法保证你的程序会使用多少栈空间，也很难预测程序在运行期间，会表现怎样。不过，你可以按照下述步骤，来控制程序在运行期间的表现。
  - 首先，如果使用了安全计数器，那么在给它设置上限时，需要考虑的事项之一，就是 ini 愿意给该递归子程序分配多少栈空间。要把它的上限设置得足够低，以防止栈溢出。
  - 其次，应注意观察递归函数中，局部变量得分配情况，特别要留意那些内存消耗大得对象。换句话说，要用 new 在堆（heap）上创建对象，而不要让编译器，在栈上面自动创建对象。

- 不要用递归，去计算阶乘或斐波纳契数列。除了速度缓慢，并且无法预测运行期间的内存使用状况外，用递归写出的子程序，要比用循环写出的子程序，更难理解。

17.3 goto

在实际中，杜绝使用 goto，跳过该章节！

18 表驱动法

表驱动法是一种编程模式，从表里查找信息，而不是使用逻辑语句（if 或 case）。事实上，凡是能通过逻辑语句来选择的事物，都可以通过查表来选择。对简单的情况而言，使用逻辑语句更为容易和直白；但随着逻辑链越来越复杂，查表法也就愈发显得更具有吸引力。

18.1 表驱动法使用总则

在使用表驱动法的时候，必须要解决两个问题。首先，你必须回答，怎样从表中查询条目的问题。你可以用一些数据，来直接访问表。例如，如果你希望把数据按月进行分类，那么创建一个月份表，是非常直接了当的。你可以用一个下标，从 1 到 12 的数组实现它。另一些数据可能很难直接用于查表。例如，假如你希望按照社会安全号码，做数据分类，那么除非你可以承受，在表里面，存放 999-99-9999 条记录，否则就不能用社会安全号码直接查表。你会被迫采用一种更为复杂的方法。下面是从表里查询记录的方法列表：

- 直接访问
- 索引访问
- 阶梯访问

使用表驱动法需要解决的第二个问题，是你应该在表里存什么内容。有时候，表查询出来的结果是数据；如果你遇到的是这种情况，那么就可以把这些数据保存在表里面。在另一些情况下，表查询出来的结果是动作（action）。在这种情况下，你可以保存一个描述该动作的代码，或者在有些语言里，你可以保存对实现该动作的子程序的引用。

18.2 直接访问表

之所以说是“直接访问”的，是因为你无须绕很多复杂的圈子，就能在表里面找到你想要的信息。下面通过一个月中天数的示例，说明直接访问表。假设你需要计算每个月中的天使（不考虑闰年）。笨拙的做法，就是写一个复杂的 if 语句：

```
Visua Basic示例：确定各月天数的笨拙做法
If (month = 1) Then
    days = 31
ElseIf (month = 2) Then
    days = 28
ElseIf (month = 3) Then
    days = 31
ElseIf (month = 4) Then
    days = 30
ElseIf (month = 5) Then
    days = 31
ElseIf (month = 6) Then
    days = 30
ElseIf (month = 7) Then
```

```
        days = 31
    ElseIf (month = 8) Then
        days = 31
    ElseIf (month = 9) Then
        days = 30
    ElseIf (month = 10) Then
        days = 31
    ElseIf (month = 11) Then
        days = 30
    ElseIf (month = 12) Then
        days = 31
    End If
```

实现同样功能的一种更简单、更容易修改的方法，是把这些数据存到一张表里面。在 Visual Basic 里面，你需要首先创建出这张表：

```
Visual Basic示例：确定各月天数的优雅做法
' Initialize Table of "Days Per Month" Data
Dim daysPerMonth() As Integer = _
    {31,28,31,30,31,30,31,31,30,31,30,31}
```

现在，你无须再写那条长的 if 语句，只需要一条简单的数组访问语句，就可以得出每个月中的天数了：

```
Visual Basic示例：确定各月天数的优雅做法
days = daysPerMonth(month - 1)
```

如果你想在查表的版本中，把闰年考虑进去，那么代码仍然会很简单，假设 LeapYearIndex() 的取值要么为 0，要么为 1：

```
Visual Basic示例：确定各月天数的优雅做法
days = daysPerMonth(month - 1, LeapYearIndex())
```

如果把闰年也考虑进来，那么 if 语句将会变得更为复杂了。计算每月的天数，是一个很合适用直接访问描述的例子，因为你可以用 month 变量去表里面查询记录。一般来说，你可以用原本控制着很多 if 语句的数据，去直接访问表。

18.3 索引访问表

当你使用索引的时候，先用一个基本类型的数据，从一张索引表中，查出一个键值，然后再用这一键值，查出你感兴趣的主数据。假设你经营着一家商店，有大约 100 种商品。再假设每种商品，都有一个 4 位数字的物品编号，其范围是 0000 到 9999。在这种情况下，如果你想用这个编号作为键值，直接查询一张描述商品信息的表，那么就要生成一个，具有 10000 条记录的索引数组（从 0 到 9999）。该数组中，除了与你商品中的货物的标志，相对应的 100 条记录以外，其余记录都是空的。如图所示，这些记录指向了一个物品描述表，而该表所含的记录数量，要远远小于 10000。索引访问技术有两个主要优点：首先，如果主查询表中的每一条记录都很大，那么创建一个，浪费了很多空间的索引数组，所用的空间，就要比创建一个，浪费了很多空间的主查询表，所用的空间小得多。举例来说，如果主表中的每条记录，需要占用 100 字节，而索引表中的每条记录，需要占用 2 字节。假设主表中有 100 条记录，而用来访问它的数据，有 10000 种可能取值。这样一来，你面临的就是，在 10000 条索引记录，和 10000 条主数据成员记录之间，做出选择。如果你用的是一套索引，那么用掉的总存储量是 30000 字节。如果你放弃索引结构，而把空间耗费在主表里面，那么用掉的总存储量，就会是 1000000 字节。第二项优点是，即使你用了索引以后，没有节省内存空间，操作位于索引中的记录，有时也要比操作位于主表中的记录，更方便更廉价。比如说，如果有一张含有员工姓名、雇佣日期和薪水的表，你可以生成一个索引，来按照员工姓名访问该表，生成另一个索引表，按照雇佣时间来访问该表，以及生成第三个索引，按照薪水来访问该表。索引访问技术的最后一个优点，就是表查询技术，在可维护性上，所具有的普遍优点。

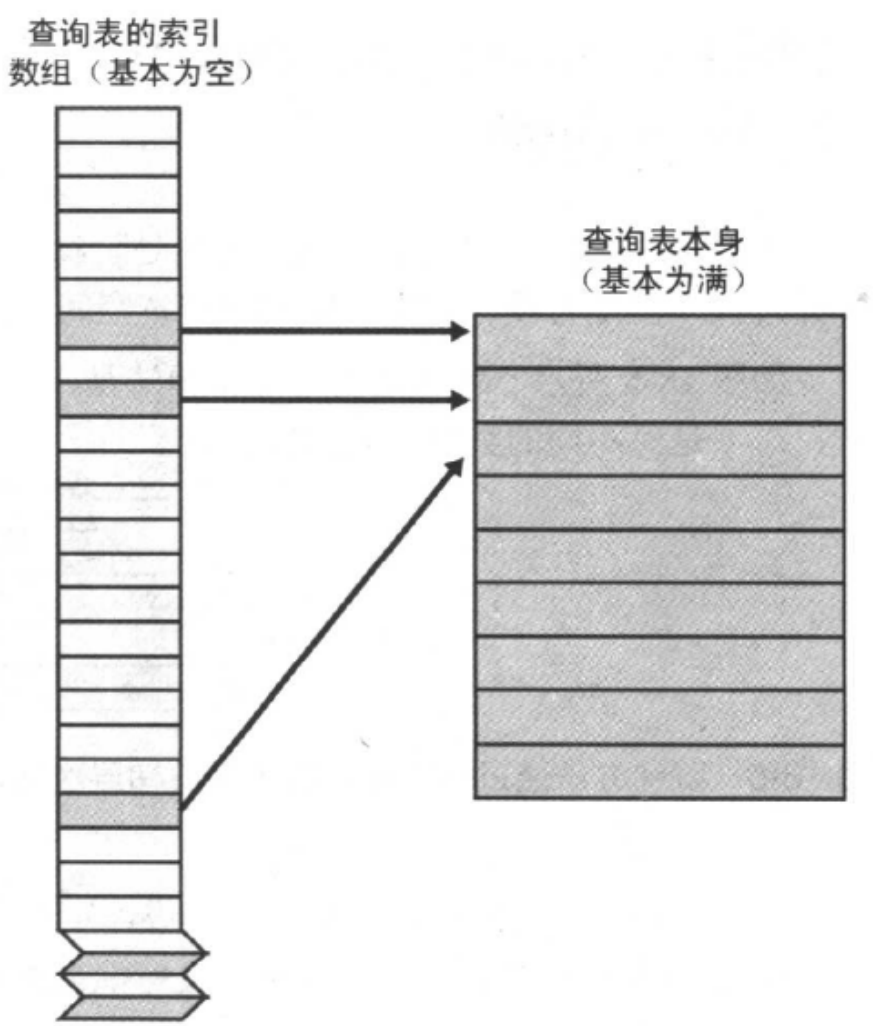


图 18-4　索引表不是直接访问，而是经过居间的索引去访问

编写到表里面的数据，比嵌入代码中的数据，更容易维护。为了使这种灵活性最大化，可以把借助索引访问数据的代码，提取成单独的子程序，然后在希望通过物品编号获得键值的时候，调用该子程序。当需要修改表的时候，你可以考虑更换这种索引访问技术，或者换用另一种表查询的技术。如果你不把索引访问代码，随便写到应用程序中各个地方，那么这种访问技术更改起来是非常容易的。

18.4　阶梯访问表

这种访问方法，不像索引结构那样直接，但它比索引访问方法节省空间。如图所示，阶梯结构的基本想法，是表中的记录，对于不同的数据范围有效，而不是对不同的数据点有效。举例来说，如果你正在开发一个等级评定的应用程

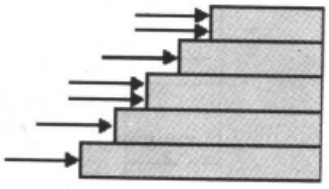


图 18-5　阶梯方法通过确定每项命中的阶梯层次确定其归类，它命中的“台阶”确定其类属

序，其中“B”记录所对应的范围是 75% 到 90%。下面是你某一天可能会编写到的等级区间：

```
>= 90.0%      A
```

<90.0%　　B
<75.0%　　C
<65.0%　　D
<50.0%　　F

这种划分范围，用在表查询中，是很糟糕的，因为你不能用简单的数据转换函数，来把表键值，转换位 A 至 F 字母所代表的等级。用索引也不合适，因为这里用的是浮点数。你可能想到把浮点数转换成整数，从而使应用索引技术变成可能。但是，为了演示起见，这个例子还会继续使用浮点数。
为了使用阶梯方法，你要把每一区间的上限，写入一张表里，然后写一个循环，按照各区间的上限来检查分数。当分数第一次超过某个区间的上限时，你就知道相应的等级了。在使用阶梯方法的时候，你必须要谨慎地处理范围地端点。下面就是根据这个例子写的代码：

```
Visual Basic示例：阶梯表查询
' set up data for grading table
Dim rangeLimit() As Double = {50.0, 65.0, 75.0, 90.0, 100.0}
Dim grage() As String = {"F", "D", "C", "B", "A"}
maxGradeLevel = grade.Length - 1
...

' assign a grade to a student based on the student's score
gradeLevel = 0
studentGrade = "A"
While((studentGrade="A") and (gradeLevel<maxGradeLevel))
    If(studentScore<rangeLimit(gradeLevel)) Then
        studentGrade = grade(gradeLevel)
    End If
    gradeLevel = gradeLevel + 1
Wend
```

尽管这个例子很简单，但却可以很容易把它推广到，处理多个学生、多种等级，以及等级发生变化的情况。与其他表驱动法相比，这种方法的优点在于，它很适合处理那些无规则的数据。下面是你在使用阶梯技术的时候，需要注意的一些细节。

- 留心端点。确认你已经考虑到每一个阶梯区间的上界。进行阶梯查询，以找出那些位于上界之外的项目来，然后把剩下的那些项目，归入最上一级范围之内。这样做有时要求为最高一级区间的最高点，假拟出一个值。注意不要把 < 误用为 <=。确认循环能够在找出最高一级的区间之后，恰当地终止，同时确保正确地处理了区间的边界。
- 考虑用二分法查找取代顺序查找。
- 考虑用索引访问来取代阶梯技术。阶梯方法中的查找操作，可能会比较耗时，如果执行速度很重要，你也许会愿意用索引访问法，来取代阶梯法，即以牺牲存储空间来换取速度。
- 把阶梯表查询操作，提取成单独的子程序。在你创建了一个转换函数，能把像 StudentGrade 这样的数值，转换成表的键值时，请把它提取成单独的子程序。

19　一般控制问题

19.1　布尔表达式

除了最简单的、要求语句按顺序执行的控制结构之外，所有的控制结构，都依赖于布尔表达式的求值。
(1) 用 true 和 false 做布尔判断

在布尔表达式中，应该用标识符 true 和 false，而不要用 0 和 1 等数值。大多数现代编程语言，都提供了布尔数据类型，并且为真和假提供了预定义的标识符。下面是一些如何定义，布尔判断中的 true 和 false 的技巧。

- 隐式地比较布尔值与 true 和 false。把表达式当作布尔表达式，可以写出更清晰地判断语句。例如，写成

```
while(not done) ...
while(a>b) ...
```

而不要写成

```
while(done == false) ...
while((a>b)=true)...
```

通过使用隐式比较，能减少阅读代码时，必须记住的项数，这样写出的表达式读起来，也更像英语中的对话。

(2) 简化复杂的表达式
你可以采取多种方法，来简化复杂的表达式：

- 拆分复杂的判断，并引入新的布尔变量。与其写一个庞大的、具有很多项的复杂判断，还不如把中间结果赋给变量，让你可以执行一个更简单的判断。

- 把复杂的表达式，做成布尔函数。如果某项判断需要重复做，或者会搅乱对程序主要流程的理解，那么可以把该判断的代码，提取成一个函数，然后判断该函数的返回值。新函数名为程序引入了一个抽象，可以清晰地在代码中，说明该逻辑判断的目的。这样做比用注释好，因为人们更关心程序代码，可能不去读注释。而且这种描述更不容易过时。

- 用决策表代替复杂的条件。有时候有一个很复杂的判断，其中涉及到多个变量。这时用一个决策表代替 if 或者 case 语句，来执行判断可能非常有帮助。决策表查询操作写起来很容易，只有几行代码，也不会用到复杂难懂的控制结构。降低了复杂度，也就降低了出错的可能性。如果你用的数据变了，那么只需要修改决策表即可，而无须改动代码；你只需要更新数据结构的内容就可以了。

(3) 编写肯定形式的布尔表达式
你可以采取一系列措施，来避免把复杂否定形式的逻辑表达式，引入到程序之中：

- 在 if 语句中，把判断条件从否定形式，转换为肯定形式，并且互换 if 和 else 子句中的代码。作为另外一种方案，可以给变量换一个名字，以表达判断真值的反义。例如，可以把 statusOK 替换成 errorDetected，这样当 statusOK 为假的时候，errorDetected 就会为真了。

- 用狄摩根定理简化否定的布尔判断。狄摩根定理揭示了一个表达式，和另一个含义相同，但却以双重否定形式表达的表达式之间的逻辑关系。例如，

```
Java示例：否定型判断
if(!displayOK || !printerOK) ...

Java示例：应用狄摩根定理之后的判断
if(!(displayOK && printerOK)) ...
```

(4) 用括号使布尔表达式更清晰
如果你有一个复杂的布尔表达式，那么与其依赖于所用语言的求值顺序，不如用括号更清晰地表达你的意图。

- 用一种简单的计数器技巧来使括号对称。如果你不知道所用的括号用得是不是配对，那么下面的简单计数技巧会很有用。开始的时候说“0”，然后从左到右扫描该表达式。当遇到一个左括号的时候说“1”，并且每次遇到一个左括号的时候，就把这一数字加 1. 每次遇到一个右括号的时候，把这个数字减 1. 如果到表达式最后的时候，所得结果为 0，那么你用的括号就是配对的。

```
Java示例：配对的括号
if(((a<b)==(c==d))&&!done)...
   |||    |  |     ||         |
0  123    2  3     21         0
```

- 把布尔表达式全放在括号里面。括号用起来很方便，而且能改善可读性。在实践中，把布尔表达式整个放在括号里，是一种很好的习惯。

(5) 理解布尔表达式时如何求值的

C++ 采用短路求值：如果 and 的第一个操作数为假，那么就跳过第二个操作数，因为整个与表达式的取值反正一定为假了。换句话说，在 C++ 中，表达式

if(SomethingFalse && SomeCondition) ...

中唯一被求值的部分是 SomethingFalse。一旦确定了 SomethingFalse 的取值为假，求值过程就马上结束。对于 or 运算符来说，求值也有相似的短路行为。在 C++ 和 Java 中，表达式

if(SomethingTrue || SomeCondition) ...

中唯一被求值的部分是 SomethingTrue。一旦确定了 SomethingTrue 的取值为真，求值过程就马上结束，因为只要某一部分为真，整个 or 表达式就为真。

(6) 按照数轴的顺序编写数值表达式

应该很好地组织数值判断，使其顺序与数轴上的点，排列顺序相符。一般来说，应该把数值判断组织好，使你能有像下面这样的比较方式：

```
MIN_ELEMENTS <= i and i <= MAX_ELEMENTS
i < MIN_ELEMENTS or MAX_ELEMENTS < i
```

这里的关键点在于，要从左到右、从小到大地排列元素。在第一行中，MIN_ELEMENTS 和 MAX_ELEMENTS 是两个端点，所以把它们放在两边。变量 i 应该位于这两点之间，所以把它写在中间。在第二个示例中，你是像判断 i 是否位于范围之外，因此把 i 写在判断外围两端地位置，而把 MIN_ELEMENTS 和 MAX_ELEMENTS 写在里面。

(7) 与 0 比较的指导原则

编程语言把 0 用作很多目的。它是一个数值，是字符串中的零终止符，是空指针的取值，是枚举的第一个元素的取值，是逻辑表达式中的 false。既然它有如此多的用途，因此你写的代码中，就应该彰显 0 的特定用法。

- 隐式地比较逻辑变量。如前所述，编写下面这样地逻辑表达式是很合适的：while(!done)...
- 把数和 0 相比较。尽管与逻辑表达式比较适合采用隐式写法，在与数值表达式比较时，却应该采用显式写法。对数值而言，应该写成 while(balance != 0)，而不是写成 while(balance)。
- 在 C 中显式地比较字符和零终止符 ('\0')。
- 把指针与 NULL 相比较。对于指针，就应该写成 while(bufferPtr != NULL)，而不要写成 while(bufferPtr)。

(8) 布尔表达式的常见问题

布尔表达式中，还含有少量与特定语言相关的缺陷：

- 在 C 家族语言中，应该把常量放在比较的左端。C 家族语言在布尔表达式上，存在一些特殊的问题。如果你因为误把 == 写成了 = 而遇到麻烦，那么可以考虑采用把常量和字面量，置于表达式左侧的编程方法，例如：if(MIN_ELEMETNS = i)。对于这个表达式，编译器会提示 = 出了错，因为给常量赋任何值，都是非法的。与之相反，在下面的表达式中，编译器只会给出一个警告，而且仅当你把编译器的警告完全打开时才会有：if(i = MIN_ELEMENTS)。这项建议于按照数轴顺序排列的建议相冲突，本书作者偏向于使用数轴排序法，让编译器来告诉有没有无意写出的赋值语句。
- 在 C++ 中，可以考虑创建预处理宏，来替换 &&，|| 和 ==（不得已才这么做）。如果你遇见过这类问题，就可以为布尔 and 运算，和 or 运算，创建出一些 #define 宏，然后用 AND 和 OR 来取代 && 和 ||。与之相似，把

== 错用成 = 也是一个常犯的错误。如果你常为这个问题所困扰，就可以创建一个类似于 EQUALS 的宏，来表示逻辑等于 ==。很多有经验的程序认为，这种方法可以帮助那些，无法彻底掌握编程语言细节的程序员，提高其代码的可读性；但对于那些已经很熟练掌握编程语言的程序员来说，代码的可读性却降低了。另外，大多数编译器会对看起来用错了的赋值，和按位运算符给出警告。把编译器警告的全部都打开，通常要比创建非标准的宏更好。

19.2 复合语句

“复合语句”或“语句块”指的是一组语句，该组语句被视为一条单一的语句，用于控制程序流。在 C++ 中，可以通过在一组语句的外面，括上“{”和“}”来创建复合语句。下面就如何有效地使用复合语句，给出一些指导原则。

- 把括号对一起写出。先写块地开始和结束部分，然后再填充中间部分。

```
先这么写：
for(i=0; i< maxLines; i++)
再这么写：
for(i=0; i< maxLines; i++) {}
最后这么写：
for(i=0; i< maxLines; i++){
    // whatever goes in here ...
}
```

  这种方法适用于所有地块结构，包括 C++ 和 Java 中的 if、for 和 while，以及 Visual Basic 中的 If-Then-Else、For-Next 和 While-Wend 组合。

- 用括号来把条件表示清楚。要想读懂条件语句，就要先弄清楚哪条语句是跟在 if 判断后面的。在 if 判断后面只写一条语句，可能看上去很美观，但是一经修改，这样的语句就会演变成复杂的块，而在这时用单一的语句就很容易引发错误。请用块来清楚地表达你地用意，无论块内的代码行数是 1 还是 20。

19.3 空语句

在 C++ 中可以写空语句，即一条仅含有分号的语句。下面就是如何在 C++ 中处理空语句的指导原则。

- 小心使用空语句。空语句并不多见，因此应该突出这种用法。方法之一就是让空语句中的分号自占一行，并且加以缩进，就像对待其他的语句一样。另外，你也可以用一组空的括号，来强调该空语句。

- 为空语句创建一个 DoNothing() 预处理宏或者内联函数。这条语句什么也不做，但却能毫无争议地表明“这里不希望做任何事情”的用意。

```
C++示例：用DoNothing()来强调空语句
#define DoNothing()
...
while(recordArray.Read(index++) != recordArray.EmptyRecord()){
    DoNothing();
}
```

  除了在空的 while 和 for 循环中使用 DoNothing()，你也可以在 switch 语句中的无意义选项中使用它；假如 DoNothing() 表明你已经考虑了这种 case，并且的确不需要对该 case 做什么操作。如果你的语言不支持预处理宏或者内联函数，那么你也可以写一个 DoNothing() 子程序，让它简单地把控制权，立即交还调用方子程序。

- 考虑如果换用一个非空的循环体，是否会让代码更清晰。空循环体代码的产生，多数是为了利用循环控制代码的副作用。在大多数情况下，如果把这种副作用明显表示出来，那么代码也将会更加易懂。

C++示例：更加清晰地重写后的代码，采用非空循环体

```
RecordType record = recordArray.Read(index);
index++;
while(record != recordArray.EmptyRecord()){
    record = recordArray.Read(index);
    index++;
}
```

这种方法引入了一个额外地循环控制变量，并要写更多地代码，但是它强调了直接了当的编程方法，而不是自作聪明地利用控制循环代码地副作用。这样的强调，对于产品代码是很可取的。

19.4 驯服危险的深层嵌套

过分深层的缩进，或者嵌套，已经困扰了计算机界达 25 年之久，并且至今仍然是产生混乱代码的罪魁祸首之一。研究表明，很少有人能够理解超过 3 层的 if 嵌套，很多研究人员建议，避免使用超过 3 到 4 层的嵌套。避免深层嵌套并不难。如果你写出了深层的嵌套，那么可以重写涉及 if 和 else 子句中执行的判断，或者把代码重构为更简单的子程序。下面给出一些用于避免深层嵌套的方法：

- 通过重复检测条件中的某一部分，来简化嵌套的 if 语句。如果嵌套层次变得太深，你可以通过重复检测，其中的一些条件，来减少嵌套的层次。

C++示例：糟糕的深层嵌套代码

```
if(inputStatus == InputStatus_Success){
    // lots of code
    ...
    if(printerRoutine != NULL){
        // lots of code
        ...
        if(SetupPage()){
            // lots of code
            ...
            if(AllocMem(&printData)){
                // lots of code
                ...
            }
        }
    }
}
```

C++示例：利用重复测试的非嵌套代码

```
if(inuptStatus == InputStatus_Success){
    // lots of code
    ...
    if(printerRoutine != NULL){
        // lots of code
        ...
    }
}
```

```
if((inputStatus == InputStatus_Success) && (printerRoutine != NULL) &&
    SetupPage()){
        // lots of code
        ...
        if(AllocMem(&printData)){
            // lots of code
            ...
        }
    }
```

这个例子很贴近实际，因为它表明你不能无偿地减少嵌套层次；作为减少嵌套层次的代价，你必须要容忍使用一个复杂的判断。不过，把嵌套层次从 4 层缩减到了 2 层，是很大的改进，所以值得考虑。

- 用 break 块来简化嵌套 if。上面描述的一种替代方案，是定义一段可以作为语句块来执行的代码。如果在语句块的中间某些条件没有满足，那么就让执行直接跳到块的末尾。

C++示例：使用break块

```
do{
    // begin break block
    if(inputStatus != InputStatus_Success){
        break; //break out of block
    }
    // lots of code
    ...

    if(printerRountine == NULL){
        break; //break out of block
    }
    // lots of code
    ...
}
```

这种技巧很不常见，所以只有在你的整个团队都很熟悉这种技巧，并且已经把它纳入了团队可接受的编码实践以后，才能使用。

- 把嵌套 if 转换成一组 if-then-else 语句。如果持批评的眼光，来看待 if 语句嵌套，那么也许会发现，可以重写组织它的结构，即用 if-then-else 语句串，来取代嵌套的 if 语句。

Java示例：判断逻辑组织得很差，判断中有许多冗余

```
if(10<quantity){
    if(100<quantity){
        if(1000<quantity){
            discount = 0.10;
        } else {
            discount = 0.05;
        }
    } else {
        discount = 0.025;
    }
} else {
```

```
    discount = 0.0;
}
```

Java示例：将嵌套得if语句转换为一组if-then-else语句

```
if(1000<quantity){
    discount = 0.10;
} else if(100<quantity){
    discount = 0.05;
} else if(10<quantity){
    discount = 0.025;
} else {
    discount = 0;
}
```

- 把嵌套 if 转换成 case 语句。你可以用 case 语句重写一些判断，特别是那些含有整数得判断，而不是去用一长串 if 和 else。
- 把深层嵌套得代码，抽取出来放进单独的子程序。如果深层嵌套出现在循环里，你通常可以通过把循环体，提取成子程序来加以改善。当嵌套是由于条件和迭代二者共同产生的时候，这么做将特别有效。把 if-then-else 分支保留在主循环中，以便显示决策的分支，然后把分支中的语句，提取成单独的子程序。
- 使用一种更面向对象的方法。在面向对象的环境中，简化代码的一种简单方法，是创建一个抽象的基类，然后从它派生出其他的子类来。
- 重写设计深层嵌套的代码。一些专家认为，在面向对象的程序设计里，出现 case 语句，就说明代码没有做好分解，因此实际上极少有必要使用 case 语句。更一般的说法是，复杂的代码表明，你还没有充分理解你的程序，所以无法简化它。深层嵌套是一个警告，它说明你要么应该拆分出一个子程序，要么应该重新设计那部分代码。当然，这并不意味着，你一定要修改这个子程序，但如果不修改的话，你应该能提出一个好的理由来。

19.5 编程基础：结构化编程

结构化编程的核心思想很简单，那就是一个应用程序，应该只采用一些单入单出的控制结构。单入单出的控制结构，指的就是一个代码块，它只能从一个位置开始执行，并且只能结束于一个位置。除此之外，再无其他入口或出口。结构化编程和结构化的、自上而下的设计，不完全一样。目前只适用于具体编码层。一个结构化的程序，将按照一种有序的，且有规则地方式执行，不会做不可预知地随便跳转。你可以按自上而下地方式阅读它，而它执行起来也大体是遵循这个顺序的。使用规则性不强的方法，所生成的源代码，很难有意义，且形象地反映出程序是如何在机器上执行地。可读性差意味着不容易理解，最终导致应用程序的低质量。

下面是结构化编程的三个组成部分

- 顺序。顺序指一组按照先后顺序执行的语句。典型的顺序型语句，包括赋值和调用子程序。

Java示例：顺序

```
// a sequence of assignment statements
a = "1";
b = "2";
c = "3";

// a sequence of calls to routine
System.out.println(a);
System.out.println(b);
```

```
System.out.println(c);
```

- 选择。选择是一种有选择地执行语句的控制结构。if-then-else 语句就是一个常见的例子。那么执行 if-then 子句，要么执行 else 子句，两者不会同时执行。即“选择”其中的某一条子句加以执行。选择控制的另一个例子是 case 语句。在每一实例中，都只有一种情况 (case) 的语句选定执行。
- 迭代。迭代时一种使一组语句多次执行的控制结构。迭代常常称为“循环”。例如 C++ 中的 while 和 for。

结构化编程的中心论点是，任何一种控制流，都可以由顺序、选择和迭代，这三种结构生成。该书作者的观点是，对于三种标准的结构化编程结构之外的，任何控制结构的使用，例如 break、continue、return，都要持一种批判的态度。

19.6 控制结构与复杂度

控制结构之所以受到如此多的关注，就是因为它们对程序整体复杂度的影响非常大。控制结构用得不好就会增加复杂度；反正则能降低复杂度。与控制流有关的复杂度非常重要，因为它与不可靠的代码，和频繁出现的错误，息息相关。

(1) 如何度量复杂度

在衡量一些复杂度的方法中，最著名的可能就是 Tom McCabe 方法。该方法通过计算子程序中“决策点”的数量，来衡量复杂度，下面是计算决策点的方法：

- 从 1 开始，一直往下通过程序；
- 一旦遇到以下关键字，或者其同类的词，就加 1：if、while、repeat、for、and、or。
- 给 case 语句中的每一种情况都加 1

下面举一个例子：

```
if(((status = Success) and done) or
    (not done and (numLines >= maxLines))) then ...
```

在这段代码中，从 1 开始算起，遇到 if 得 2，and 得 3，or 得 4，and 得 5。加起来，这段代码里总共包含了 5 个决策点。

(2) 如何处理复杂度得度量结果

计算出决策点的数量后，你就可以用得到的数值，分析你写的子程序的复杂度了：

- 0-5 子程序可能还不错
- 6-10 得想办法简化子程序了
- 10+ 把子程序得某一部分，拆分成另一个子程序，并调用它

把子程序的一部分，提取成另一个子程序，不会降低整个程序的复杂度，只是把决策点移到其他地方。但是这样做，可以降低你在同一时间，必须关注的复杂度水平。由于重点是要降低，你需要在头脑中，同时考虑的项目数量，所以降低一个给定子程序的复杂度，是有价值的。

10 个决策点的上限并不是绝对的。应该把决策点的数量，当作一个警示，该警示说明，某个子程序可能需要重新设计了。不要死守这个规则。一条情况很多的 case 语句，可能会包含超过 10 个的元素。如果硬拆开它，可能就是很愚蠢的，这取决于该 case 语句的用途。

20 软件质量概述

20.1 软件质量的特性

软件同时拥有外在的和内在的质量特性。外在的特性，指的是该产品的用户所能感受到的部分，包括下列内容：

- 正确性。指系统规范、设计和实现方面的，错误的稀少程度。
- 可用性。指用户学习和使用一个系统的容易程度。
- 效率。指软件是否尽可能少地占用系统资源，包括内存和执行时间。
- 可靠性。指在指定的必须条件下，一个系统完成所需功能的能力：应该有很长的平均无故障时间。
- 完整性。指系统阻止对程序或者数据，进行未经验证或者不正确访问的能力。这里的完整性，除了包括限制未经授权用户的访问外，还包括确保数据能够正确访问。
- 适应性。指为特定的应用或环境设计的系统，在不做修改的情况下，能够在其他应用或者环境中使用的范围。
- 精确性。指对一个已经开发出的系统，输出结果的误差程度，尤其输出的是数量值的时候。精确性和正确性的不同在于，前者是用来判断系统，完成工作的优劣程度，而后者则是判断系统是否被正确开发出来。
- 健壮性。指的是系统在接收无效输入，或者处于压力环境时，继续正常运行的能力。

质量的外在特性，是用户关系的唯一软件特性。用户只会关心软件是否容易使用，而不会关心是否容易修改。他们关心软件是否能正确运行，而不会关心里面的代码是否可读，或者是否有良好的结构。而程序员除了关心软件质量的外在特性之外，还要关心它的内在特性。本书的核心是代码，所以它更关注软件内在的质量特性：

- 可维护性。指是否能够很容易对系统进行修改，改变或增加功能，提高性能，以及修改缺陷。
- 灵活性。指假如一个系统是为特定用途，或环境而设计的，那么当该系统，被用于其他目的，或环境的时候，需要对系统做修改的程度。
- 可移植性。指为了在原来设计的特定环境之外运行，对系统所进行修改的难易程度。
- 可重用性。指系统的某些部分，可被应用到其他系统中的程度，以及此项工作的难易程度。
- 可读性。指阅读并理解系统代码的难易程度，尤其是在细节语句的层次上。
- 可测试性。指的是你可以进行何种程度的单元测试，或者系统测试，以及在何种程度上，验证系统是否符合需求。
- 可理解性。指在系统组织和细节语句的层次上，理解整个系统的难易程度。

20.2 改善软件质量的技术

软件质量保证，是一个需要预先计划的、系统性的活动，其目标就是为了确保系统，具备人们所期望的特性。下面是软件质量中的某些要素。

- 软件质量目标。改善软件质量的一种强有力的方法，就是根据前面章节所提到的，各种外在特性和内在特性，明确定义出软件质量的目标。如果没有一个明确的目标，那么程序员去极力增强的特性，就可能同你所强调的特性有别。
- 明确定义质量保证工作。在保证质量的工作中，一个最常见的问题，是质量被认为是次要目标。将质量保证工作明确下来，可以清楚地表明这件事地优先程度，如此一来，程序员就会据此做出响应。
- 测试策略。执行测试可以为产品地可靠性，进行详细的评估。质量保证的一部分，就是制定出一套，于产品需求、架构以及设计相关联的测试策略。许多项目开发商，把测试作为质量评估和质量改善的首要方法。
- 软件工程指南。在开发过程中，指南应当控制软件的技术特性，它应当贯彻到所有的开发活动中去，包括问题定义、需求分析、架构设计、构建，以及系统测试。
- 非正式技术复查。许多软件开发人员，会在正式复查之前，自行检查自己的工作。非正式复查包括对设计或代码的桌面检查，或者和若干同事一起，将代码走查一遍。

- 正式技术复查。管理一个软件工程过程的工作之一，就是要在低成本的环节里，抓出问题。要实现这一目标，开发人员周期性使用“质量门”，测试或复查，以检验某一阶段的产品，是否已经具备了，进入下一阶段前，所要求的质量。质量门通常用于从需求分析到架构，从架构到构建，以及从构建到系统测试之间的转换过程。所谓“门”，可能是一次检查，也可能是一次同事互查，或一次客户复查，或者一次独立审查。

- 外部审查。外部审查是一种，用于确定一个已开发项目，或产品的状态的，特殊技术复查方法。一个审查小组，由开发组织以外的人员构成，并且向委托人汇报审查结果，这个委托人通常是经理。

(1) 开发过程

相对于没有质量保证活动的开发流程，具备该活动的开发流程，能产生出更好的软件。

- 对变更进行控制的过程。实现软件质量目标的拦路虎之一，就是失控的变更。需求变更的失控，可能使设计和编码工作前功尽弃；设计变更的失控，则会造成代码与需求背离，或代码自相矛盾，或是程序员为达到变更后的设计要求，不得不耗费比推进项目更多的时间来修改代码。代码变更的失控，则可能造成内部冲突，程序员无法确定哪些代码，已经过完全复查和测试，而哪些没有。变来变去的影响，就是质量的不稳定和恶化，因此，有效地管理变更，是实现高质量地一个关键。

- 结果的量化。量化结果能告诉你，计划成功与否，并且允许你用可控的方式，来调整你的计划，去看你能如何改善它。

- 制作原型。制作原型是指开发出系统中，关键功能的实际模型。对一个开发者来说，开发出一部分用户界面的原型，可以判断系统的可用性，开发出关键算法的原型，可以确定功能的执行时间，开发出典型数据集的原型，能知道程序的内存需求。构建原型能产生更完善的设计，更贴近用户的需求，以及更好的可维护性。

(2) 设置目标

明确设置质量目标，是开发高质量软件的，一个简单而清晰的步骤，但它常常被忽视。人们会做要求他们去做的事情。程序员有很高的成就激励：他们会向明确的目标进发，但必须有人告诉他们，目标是什么。不同目标之间是有冲突的，并且软件通常都不可能在所有方面都做得很好。

20.3 不同质量保障技术的相对效能

各种质量保证方法的效能并不相同。

(1) 缺陷检测率

某些方法在检测缺陷方面，比其他方法更有效，而且不同的方法能找出不同类型的缺陷。测定所找到的缺陷，占该项目当时所有存在缺陷的百分比，是评估各种缺陷检测方法的一种途径。研究表明，单独使用任何一种方法，其典型检出率都没有超过 75%，并且平均来说，这一数值在 40%。最常用的缺陷方法：单元测试和集成测试，它们的一般检测率，仅仅在 30% 到 35% 之间。这些数据强烈提醒我们，如果项目的开发者，要向更高的缺陷检测了发起冲击，他们需要综合运用各种技术。

(2) 找出缺陷的成本

某些缺陷检测方法的成本，比其他方法要高。最经济的方法，应当是找出缺陷的成本最低，而在其他方面，同别的方法并无二致。后一个条件很重要，因为查找单个缺陷的成本，受很多因素的影响。大部分研究发现，检查比测试的成本更小。

(3) 修正缺陷的成本

一个缺陷存在的时间越长，消除它的代价就越高；因此能够尽早发现错误的监测方法，可以降低修正缺陷的成本。有的方法如代码检查，可以一举确定问题的现象和愿意；而另一些方法如测试，则只能发现问题表象，而要找到并从根本上修正缺陷，还需要额外的工作。因此一步到位的方法，明显比两步的方法更划算。一个有效的软件质量项目的底线，必须包括在开发的所有阶段，联合使用多种技术；通过下面这些方法，可以获得高于平均水平的软件质量：

- 对所有的需求、架构以及系统关键部分的设计，进行正式检查

- 建模或者创建原型

- 代码阅读或检查
- 执行测试

20.4 什么时候进行质量保证工作

错误越早引入到软件中，问题就会越复杂，修正这个错误的代价也更高，因为错误会牵涉到系统的更多部分。需求中的一个缺陷，会孕育出设计上的一个或多个缺陷，而这些设计错误，又会繁殖出更多的代码缺陷。需求中的一个错误，会导致多余的架构设计或错误的架构决策。多余的架构设计，又会导致多余的代码、测试用例和文档，一个需求上的错误，可能产生最终不得不被抛弃的架构、代码以及测试用例。就如同在浇注地基之前，应当先在建筑图纸上，把问题解决。
此外，相对于编码阶段的错误，需求或架构上的错误，往往会产生更为广泛的影响。单个架构错误，可以影响多个类，以及几十个子程序，而单个构造错误，则不会超过一个子程序或类。
缺陷可能在任何阶段渗透到软件中，因此，你需要在早期阶段，就开始强调质量保证工作，并且将其贯彻到项目的余下部分。在开工之时，这一工作就应当添加到项目计划中，在项目中作为技术脉络的一部分，并且应该作为项目的结束点，当整个工作结束的时候，查验产品的质量。

(4) 软件质量的普遍原理

软件质量的普遍原理，就是改善质量以降低开发成本。提高生产效率和改善质量的最佳途径，就是减少花在这种代码返工上的时间，无论返工的代码是由需求、设计改变，还是调试引起的。

21 协同构建

21.1 协同开发实践概要

协同构建包括结对编程、正式检查、非正式技术复查、文档阅读，以及其他让开发人员，共同承担创建代码，以及其他工作产品责任的技术。各种协同构建技术之间，尽管存在一些差异，但它们都基于一个相同的思想，那就是工作中，开发人员总会对某些错误点视而不见，而其他人不会有相同的盲点，所以开发人员让其他人，来检查自己的工作是很有好处的。

(1) 协同构建是其他质量保证技术的补充

协同构建的首要目的，就是改善软件的质量。另外，它还可以减少软件中的缺陷数量，进而缩短开发周期，从而降低开发成本。各种不同的研究表明，协同开发不但在捕获错误方面，比测试的效能更高，所能发现的错误类型，也不同于测试。协同开发的另一个作用，就是让人们意识到，它们的工作会被复查，这样他们会小心谨慎地检查自己的工作。因此，即使测试工作完成得很有效率，作为完整的质量计划的一部分，复查或其他类型的协作同样很有必要。

(2) 协同构建有利于传授公司文化以及编程专业知识

软件标准可以写下来，并发布出去，但是如果没有人讨论它们，也不鼓励使用这些标准，那么就不会有人按照这些标准做事情。复查是一个很重要的机制，它可以让程序员，得到关于他们自己代码的反馈。代码、标准以及让代码符合标准的理由等，都是复查讨论中的好主题。复查为刚出道的编程人员和资深程序员，提供了一个技术交流平台，因此是培养新人，以提高其代码质量的好机会。

(3) 集体所有权适用于所有形式的协同构建

在集体所有权下，所有的代码都属于团队，而不是某一个人，并且团队中的所有成员，都可以对其进行访问和修改。这会带来一些很有价值的好处。

- 众多双眼睛的检查，以及众多程序员的协力编写，可以使代码的质量变得更好。
- 某个人离开项目所造成的影响更小了，因为每一段代码，都有很多人熟悉它。
- 总体上缺陷修正周期变短了，因为几个程序员中的任何一个有空，就能随时被指派去修正缺陷。

(4) 在构建前后都应保持协作

21.2 结对编程

(1) 成功运用结对编程的关键

- 用编码规范来支持结对编程。如果两个人整天把时间浪费在争论代码风格的问题上，那么结对编程就不可能发挥它的威力。应该尝试对风格进行标准化。
- 不要让结对编程编程旁观。不掌握键盘的那个人，应该主动参与到编程中，他应该分析代码，提前思考接下来的代码，应该做些什么，对设计进行评估，并对如何测试代码，做出计划。
- 不要强迫在简单的问题上使用结对编程。
- 有规律地对人员和分配地工作任务进行轮换。结对编程地好处在于，能够让不同的人，熟悉系统的不同部分。有规律地进行轮换，有助于知识地互相传播。
- 鼓励双方跟上对方的步伐。要是其中一个人，相对走得太快的话，那就会大大限制了其结对搭档的作用。
- 确认两个人都能看到显示器。
- 不要强迫程序员与自己关系紧张的人组对。
- 避免新手组合。
- 指定一个组长。即使你的整个队伍，希望所有工作都通过结对编程的方法来做，你还是需要指定一个人来协调工作的分配，对结果负责，以及负责与项目外其他人的联系。

(2) 结对编程的好处

- 与单独开发相比，结对能够使人们在压力之下，保持更好的状态。
- 能够改善代码质量。代码的可读性和可理解性，都倾向于上升至，团队中最优秀程序员的水平。
- 它能缩短进度时间表。结对往往能更快地编写代码，代码的错误也更少。这样一来，项目组在项目后期，花费在修正缺陷的时间会更少。
- 它还具有协同构建的其他常见好处，包括传播公司文化，指导初级程序员，以及培养集体归属感。

21.3 正式检查

正式检查是一种特殊的复查，种种迹象表明，它在侦测缺陷方面，特别有效，并且相对测试来说更加经济合理。虽然任何复查都涉及了阅读设计或代码，但是详查还是在几个关键问题上，与普通复查有所区别。

- 详查表关注的是复查者过去所遇到的问题。
- 详查专注于缺陷的检测，而非修正。
- 复查人员要为详查会议做好预先准备，并且带来一份他们所发现的已知问题列表。
- 参与者都被赋予了明确的角色。
- 详查的主持人不是被检查产品的作者。
- 详查的主持人应该已经接收过主持详查会议方面的培训。
- 只有在与会者都做好充分准备之后，才会召开详查会议。
- 每次详查所收集的数据，都会被应用到以后的详查当中，以便对详查进行改进。
- 高层管理人员不参加详查会议，除非你们正在详查一个项目的计划，或者其他管理方面的资料。但技术负责人可能参加。

21.4 其他类型的协同开发实践

- 走查
- 阅读
- 演示

22 开发者测试

软件可以通过许多的方法进行测试：

- 单元测试。将一个程序员或一个开发团队所编写的，一个完整的类、子程序或小程序，从完整的系统中隔离出来进行测试。
- 组件测试。将一个类、包、小程序或其他程序元素，从一个更加完整的系统中，隔离出来进行测试，这些被测试代码涉及到多个程序员或多个团队。
- 集成测试。对两个或更多的类、包、组件或子系统，进行的联合测试，这些组件由多个程序员，或者开发团队所创建。这种测试通常在有了两个可以进行测试的类的时候，就应该尽快开始，并且一直持续到整个系统开发完成。
- 回归测试。重复执行以前的测试用例，以便在原先通过了测试集合的软件中，查找缺陷。
- 系统测试。在最终的配置下运行整个软件。以便测试安全、性能、资源消耗、时序方面的问题，以及其他无法在低级集成上测试的问题。

测试通常分为两大类：黑盒测试和白盒测试。黑盒测试指的是测试者无法了解测试对象内部工作机制的测试。白盒测试指测试者清楚待测试对象，内部工作机制的测试。

22.1 开发者测试在软件质量中的角色

测试与其他开发活动，有很多不同之处。

- 测试的目标与其他开发活动背道而驰，测试的目的是找出错误。一个成功的测试，应该弄垮软件，而其他开发活动的目标，是避免程序错误和软件的崩溃。
- 测试永远不可能彻底证明程序中没有错误。
- 测试本身并不能改善软件质量。测试的结果是软件质量的一个指示器，但是结果本身并不改善软件质量。
- 测试时要求你假设会在代码里找到错误。假如你找不到，那么很可能就真的找不到，也有可能仅仅是因为，你建立了一个自我实现的预言。

一个关键的问题是，在一个典型的项目里，开发者测试应该占多少时间？根据项目大小和复杂程度的不同，开发者测试应该占整个项目时间的 8% 到 25%。第二个问题是，怎样利用开发者测试的结果？最直接的，可以用这个结果，评估正在开发的产品的可靠性。即使你根本不修正测试所发现的错误，测试结果也可以描述该软件的可靠性。测试结果的另一用途是，它可以用于指导对软件的修正，并且通常也是如此。最后，测试发现缺陷的记录，有助于你归纳出程序中，最常见错误的类型。你可以用这一信息，去选择适当的培训课程、指引今后的技术复查活动，设计未来的测试用例。

在构建期间，通常你会写一个子程序或者类，现在头脑中检查它，然后对它进行复查或测试。无论你的集成测试或系统测试策略如何，在将一个部分同其他部分组合之前，你都需要对它进行彻底的单元测试。假如你正在写几个子程序，那么你应该一个一个地对它们进行测试。独立进行子程序地测试，不是一件容易的事，但是单独地调试它们，比集成之后再进行测试，要简单得多。如果将几个没有经过测试的子程序放到一起，如果发现了一个错误，那么这几个子程序都有嫌疑。假如每次只将一个子程序，加入到此前经过测试的子程序集合中，那么一旦发现了新的错误，你就会知道这是新子程序，或者其接口所引发的问题，调试工作就轻松多了。

22.2 开发者测试的推荐方法

采用系统化的开发者测试方法，能最大限度提高你发现各种错误的能力，同时让你的花费也最少。请确保下面所有要点你都能做到：

- 对每一项相关的需求进行测试，以确保需求都已经被实现。在需求阶段就计划好，这一部分的测试用例，或者至少尽早开始，最好在你开始编写待测试的单元之前。
- 对每一个相关的设计关注点进行测试，以确保设计已经被实现。在设计阶段就计划好，这一部分的测试用例，或者尽早开始，在你开始编写待测试子程序或类的具体代码之前。
- 用基础测试来扩充，针对需求和设计的详细测试用例。增加数据流测试，然后补充其他所需的测试用例，以便对代码进行彻底的考验。至少，你应该测试到每一行代码。
- 使用一个检查表，其中记录着你在本项目中，迄今为止所犯的，以及在过去的项目中，所犯的错误类型。

(1) 测试先行还是测试后行

首先写测试用例，可以将从引入缺陷，到发现并排除缺陷之间的时间，缩减至最短。这正是首先写测试用例的诸多原因之一。

- 在开始写代码之前，先写测试用例，并不比之后再写，花更多功夫，只是调整了以下测试用例编写活动的工作顺序而已。
- 假如你首先编写测试用例，那么你将可以更早发现缺陷，同时也更容易修正它们。
- 首先编写测试用例，将迫使你在开始写代码之前，至少思考以下需求和设计，而这往往会催生更高质量的代码。
- 先编写测试用例，能更早地把需求上地问题暴露出来，因为对于一个糟糕的需求来说，要写出测试用例，是一件困难的事情。
- 如果你保持了最初编写的测试用例（这是你应该做的），那么先进行测试并非唯一选择，你仍然可以最后进行测试。

本书作者认为，测试先行的编程，是过去十年中，形成的最有用的软件开发实践之一，同时也是一个非常好的通用方法。

(2) 开发者测试的局限性

应该注意到开发者测试的下述局限性：

- 开发者测试倾向于“干净测试”。开发人员往往去做一些，检验代码能否工作的测试（干净测试），而不是做所有可能让代码失效的测试（肮脏测试）。
- 开发者测试对覆盖率有过于乐观的估计。平均而言，程序员坚信他们的测试覆盖率达到了 95%，但通常，最佳情况下，这一数字只能达到大约 80%。
- 开发者测试往往会忽略一些，更复杂的测试覆盖率类型。大多数开发人员看到的测试覆盖率，应该称为“100% 的语句覆盖率”，而这远远不够；更好的覆盖率标准，是所谓的“100% 分支覆盖率”，也就是对每一个判断语句，都至少测试一个真值，和一个假值。

22.3 测试技巧锦囊

为什么用“通过测试”，来证明程序的正确性，是不可能的呢？如果要用测试来证明一个程序的正确性，你需要对程序的每一种可能的输入值，以及它们之间的所有可以想象的组合进行测试。即使是一个简单的程序，这样庞大的任务都会让人望而却步。

(1) 不完全的测试

由于进行完全测试实际上是不可能的，因此测试的窍门，就在于选择那些最有可能找到错误的测试用例。你需要集中注意力，挑选那些能告诉你不同答案的测试用例，而不选出一堆总是告诉你相同答案的测试用例。当你规划测试的时候，要去除那些不会告诉你任何新情况的测试用例，也就是说，如果测试的某个数据没有产生错误，那么新的类似数据，可能也不会产生错误。

(2) 结构化的基础测试

结构化的基础测试是一个相当简单的概念，其思想是，你需要去测试程序中的每一条语句至少一次。如果语句是一个逻辑语句，例如 if 或 while，那么你就需要根据 if 或 while 中表达式的复杂程度，来修改测试，以确保这个语句完全通过了测试。要确保你已经覆盖了所有的基础情况，最简单的方法，就是算一算有多少条通过程序的路径，然后据此开发出能通过程序里，每条路径的最少数量的测试用例。

你可能已经听说过“代码覆盖”测试或“逻辑覆盖”测试，这是测试穿过程序里的所有路径的两种方法。由于它们覆盖了所有的路径，因此，它们和结构化的基础测试很相似，但是它们并不蕴含着，以最小数量的测试用例，覆盖所有路径的思想。如果使用代码覆盖测试或逻辑覆盖测试，在覆盖相同逻辑的情况下，你需要创建的测试用例，远多于结构化的基础测试。

所需基础测试用例的最少数量，可以用下面的简单方法计算：

- 对通过子程序的直路，开始的时候记 1；
- 遇到下面的每个关键字，或者其等价物，加 1：if、while、repeat、for、and 以及 or；
- 遇到每一个 case 语句就加 1，如果 case 语句没有缺省情况，则再加 1.

(3) 数据流测试

数据流测试基于如下概念：数据使用的出错几率，至少不亚于控制流。数据的状态可以是下列几种状态中的一种：

- 已定义。数据已经初始化，但是还没使用。
- 已使用。数据已经用于计算，或作为某子程序调用的一个参数，或者用于其他用途。
- 已销毁。数据曾经定义过，但是现在已经通过某种途径，取消了对它的定义。
- 已进入。控制流已进入一个子程序，但是还没有使用该变量。例如，一个在子程序中使用的变量，在子程序开始处进行初始化。
- 已退出。在对变量产生影响之后，控制流立即退出子程序。例如，在子程序的结尾处，把返回值赋给一个状态变量。

在开始测试之前，首先要检查以下，是否出现了反常的数据状态顺序。在做过检查之后，编写数据流测试用例的关键，是要对所有可能的“已定义-已使用”路径进行测试。开发测试用例的一个好方法，是首先进行结构化的基础测试，即使它没有测试所有“已定义-已使用”的数据流形式，但至少也完成了其中一部分。然后你再添加完整的“已定义-已使用”数据流测试所需的用例。

(4) 等价类划分

一个好的测试用例，应该覆盖可输入数据中的很大一部分。如果两个用例，能揭示的错误完全相同，那么只要一个就够了。“等价类划分”的概念，是这一想法的严格表达形式，应用它有助于减少所需用例的数量。

(5) 猜测错误

猜差错误是指，猜测程序会在哪里出错的基础上，建立测试用例，尽管这意味着猜测中会有一些牵强附会的成分。猜测可以基于直觉或过去的经验。

(6) 边界值分析

运用边界值条件进行测试，最丰硕的战果之一，就是 off-by-one 错误。这种错误即当你想用 num 的时候，写成了 num-1；当你想用“>”的时候，写成了“>=”，这些都是最常见的失误。边界值分析的思想，就是写一些测试用例，来测试边界值条件。

(7) 几类坏数据

除了假设错误会在边界条件上出现之外，你可以猜测并测试几种类型的坏数据。典型的坏数据测试用例包括：

- 数据太少或没有数据
- 太多的数据
- 错误的数据情况（无效数据）
- 长度错误的数据
- 未初始化的数据

(8) 几类好数据

当你试图在程序中寻找错误的时候，这样一个事实很可能被忽略：正常的情况也可能暗含错误。通常来说，基础测试一节所提到的正常情形，所描述的就是一种好数据。下面是其他几种值得测试的好数据，

- 正常的情形：期望的值
- 最小的正常局面
- 最大的正常局面
- 与旧数据的兼容性

(9) 采用容易手工检查的测试用例

例如在写一个有关正常薪水的测试用例时候，用 20000 比 90783.82 要好。

22.4 典型错误

(1) 哪些类包含最多的错误

软件缺陷在代码里，并不是均匀分布的。绝大多数错误，往往与少数几个具有严重缺陷的子程序有关。下面是错误和代码之间的普遍关系：

- 80% 的错误，存在于项目 20% 的类或子程序中。
- 50% 的错误，被发现存在于项目 5% 的类当中。

如果你认为这些关系无关紧要，很可能是因为你对下面几个结论一无所知。

- 项目中 20% 的子程序，占用了 80% 的开发成本。虽然这并不是说，成本最高的 20% 的代码，就是有最多缺陷的 20% 的代码，但这很有启发性。
- 无论高缺陷率子程序，在成本中所占的具体比例如何，这些子程序的成本，的确是异常高昂的。
- 子程序开发成本昂贵，带来的影响也显而易见。提高质量就能缩短开发周期，同时降低开发成本。
- 避免维护惹人厌烦的子程序，同样具有明显的重要意义。维护工作应该围绕，如何确定容易出问题的子程序，如何把这些部分推倒重来，重新设计并编写代码。

(2) 错误的分类

很多研究者都尝试着对错误进行分类，并对每种错误的出现范围做出判定。

- 大多数错误的影响范围是相当有限的。一项研究发现，85% 的错误，可以在修改不超过一个子程序的范围内，得以修正。
- 许多错误发生在构建的范畴之外。三种最为常见的错误源头：缺乏应用领域知识、频繁变动且相互矛盾的需求，以及沟通和协调的失效。
- 大多数的构建错误，是编程人员的失误造成的。
- 拼写错误是一个常见的问题根源。

- 研究程序员所犯错误原因时，错误理解设计这条，会经常出现。
- 大多数错误都很容易修正。大约 85% 的错误，可以在几个小时内修正；大约 15% 的错误，可以在几小时到几天之内修正；只有大约 1% 的错误，需要花更长时间。
- 总结所在组织中，对付错误的经验。不同组织的人，会有完全不同的错误处理经验。因此，很难将其他组织所获得的经验，应用到你所在的组织中。

(3) 不完善的构建过程，引发错误所占的比例
构建总会出现大量的错误。

- 在小型项目里面，构建中的缺陷，占了所有错误的大多数。
- 无论项目规模如何，构建缺陷至少占了总缺陷的 35%。
- 修正构建错误的代价，虽然比修正需求和设计的错误相对低廉，但从绝对值来看，仍然是高昂的。

(4) 你希望发现多少错误
预期发现错误的数量，会根据你所使用开发过程的质量而变化。下面列出可能的范围：

- 业界的经验是，在已发行的软件中，平均 1000 行代码，发现 1-25 个错误。
- 微软应用程序部门的经验是，内部测试程序大约每 1000 行代码，有 10 至 20 个缺陷，而对于已发布产品，则大约是每 1000 行代码 0.5 个缺陷。要达到这一水平，需要结合运用前述的代码阅读技术，以及独立测试技术。
- Harlan Mills 所倡导的“净室开发”的技术，可以获得低至每 1000 行代码 3 个缺陷（内部测试阶段），以及每 1000 行代码 0.1 个缺陷（产品发布阶段）的错误率。
- Watts Humphrey 报告称，使用“团队软件开发过程”的开发小组，可以达到大约每 1000 行代码 0.06 个缺陷的水平。

开发高质量的软件，比开发低质量软件，然后修正的成本要低廉。

(5) 测试本身的错误

测试用例可能包含，与被测代码同样多，甚至更多的错误。原因很简单：测试用例往往是临时抱佛脚的成果，没有经过仔细的设计和构建。这些测试用例通常被认为只测试一次，并且开发它们的人，是抱着用后即弃的心态来开发的。你可以通过下列几项工作，来减少测试用例当中的错误量。

- 检查你的工作。要以开发代码般的谨慎态度，来开发测试用例，这种谨慎当然包括对测试进行双重检查。在调试器当中单步跟踪测试代码，要一行一行的来，就像对待产品级代码那样。对测试数据进行走查和详细检查，也是适当的做法。
- 开发软件的时候，就要计划好测试用例。在需求阶段或刚刚接收该程序时，就应该开始对测试做出有效的计划。这有助于将基于错误假定的测试用例，扼杀在摇篮中。
- 保留你的测试用例。花点时间来管理测试用例，把它们保存起来，这些东西在回归测试，或者开发下一个版本的时候，还用得上。
- 将单元测试纳入测试框架。首先写单元测试中使用的代码，在每完成一次单元测试后，记得将它们集成到一个系统级测试框架中去。有了这样的集成测试框架，就可以减少丢弃测试用例的可能性。

22.5 测试支持工具

(1) 为测试各个类，构造脚手架

“脚手架”是个建筑术语。建筑工人如果要对建筑的某个部分进行施工，就必须搭建脚手架，除此之外别无他法。在软件中搭建脚手架只有一个目的，那就是更方便地测试代码。有一种脚手架是所谓地哑类，待测试的类可以使用它。

这样的类也被称为“模仿对象”或“桩对象”。对于低层的子程序，也可以用类似的方法，那就是“桩函数”。在制作假对象或桩函数的时候，你可以依据所需的真实性，来决定它们与现实的近似程度。另一种脚手架，是调用待测试的真实函数的伪造函数。这种脚手架称为“驱动函数”，有时也称为“测试夹具”。最后一种脚手架是所谓的哑文件，即真实文件的一个小尺寸版本，它的构成和全尺寸文件一模一样。一个小的哑文件有一些好处：首先，因为它尺寸小，你可以对它的内容一清二楚，并且可以毫不犹豫地断定，这个文件本身没有错误。其次，因为它是为了测试，而被特别制作出来地，你可以设计它的内容，使任何使用文件的错误，都能暴露出来。

(2)Diff 工具

一种检查输出数据的简单方法是，将程序的输出，重定向到一个文件当中，把预计输出也存放到一个文件中，然后用一个文件比较工具，对二者进行比较。如果两个输出不一致，你就已经发现了一个回归错误。

(3) 测试数据生成器

- 正确设计的随机数据生成器，可以产生让你意想不到的、不寻常的测试数据组合。
- 比起手工构造测试数据，随机数据生成器可以更加彻底地对程序进行测试。
- 可以在很长时间中，进一步精炼随机生成地测试用例，以强化所生成地输入的真实性。这样就可以集中测试，用户最可能使用到的范围，从而最大限度地增强，程序在这一输入范围内的可靠性。
- 在测试期间，模块化设计就显现出它的优势来了。
- 在你修改了被测试的代码之后，你还可以重用测试驱动程序。

(4) 覆盖率监视器

覆盖率监视器就是用来，跟踪哪些代码已经测试过了，而哪些代码还没有。它在系统化测试的时候尤其有用，因为它会告诉你，某一组测试用例是否能够彻底地对代码进行测试。如果你在运行了一组完整测试用例之后，覆盖率监视器还显示，某些代码没有执行过，你就知道还需要进行更多的测试。

(5) 数据记录器/日志记录器

详细完整的日志记录，可以为诊断错误提供帮助，还可以在产品发布之后，为客户提供有效的服务。你可以编写一个自己的数据记录工具，把关键事件记录到某个文件里，其中要记录的是发生错误前的系统状态，以及发生错误的确切条件等详细信息。你可以把这一功能，编译进开发版本中，而在发布版本中去掉。另一个方案是，如果你能使日志实现自动裁剪记录长度，并妥善考虑记录存放的位置，以及错误信息的内容，那么将这项功能纳入发布版本，也未尝不可。

(6) 符号调试器

符号调试器可以作为走查和详查代码的技术辅助工具。调试器可以一行行地对代码进行单步调试，跟踪变量的值，并能完全按照计算机的方式来演绎代码的执行情况。在调试器中，对某段代码进行单步调试，并观察其运行情况是很有价值的。

(7) 系统干扰器

另一类测试支持工具，是用来对系统进行干扰的。这类测试支持工具有如下多种功能：

- 内存填充。你可能想要确认程序中的所有变量，都已经初始化了。有些工具可以在你运行程序之前，将任意数值填充到内存中，这样没有初始化的变量，就不会正好是 0。
- 内存抖动。在多任务系统里面，有些工具可以在你的程序运行时，重新组织内存，使用这种工具可以让你确信，所有代码都只依赖，存放在相对位置的数据，而非某些绝对位置。
- 选择性内存失败。一个内存驱动程序，可以模拟内存容量不足的情况。程序在这种情况下，有可能遇到各种内存问题，包括内存耗尽、内存请求失败，若干次请求成功之后，遭遇失败，或是若干次请求失败后，才能成功等。
- 内存访问检查（边界检查）。边界检查监视着各种指针操作，确保所有指针都能工作正常。这种工具在寻找未初始化的，或者悬空的指针方面非常有用。

(8) 错误数据库

存放以往错误的数据库，是另一种强大的测试工具，这样一个数据库既是管理工具，又是技术工具。它让你能检查重复出现的错误，及时获取已纠正错误和已发现错误之比率，以及跟踪错误的处理状态和严重级别。

22.6 改善测试过程

改善测试过程的步骤，同改善其他任何过程的步骤类似。你必须清楚地知道，这一过程是干什么的，这样你才能对其略微调整，然后看看这样改变会产生什么效果。

(1) 有计划的测试

有效测试的关键之一，就是在待测项目开始之初，就拟定测试计划。就重要性而言，测试应当与设计和编码平起平坐，这就要求项目为测试分配时间，重视测试，并保障这一过程的质量。测试计划也是使测试过程可重复的一个要素，如果你无法重复它，那么就不可能改善它。

(2) 重新测试（回归测试）

假设你已经对某产品进行了彻底检查，而且没有发现任何错误。在此之后，该产品的某个部分被修改，你想确定修改后的产品，仍然能通过此前的所有测试，也就是说，这次的修改没有给产品，引入新的错误。为确保软件没有倒退，或者没有“回归”，而设计的测试，被称为“回归测试”。

除非在每次修改后，重新对程序进行系统和的测试，否则要开发出一个高质量的软件产品，几乎是痴人说梦。而如果每次修改后，你都使用了不同的测试用例，那么你将无法保证本次修改，没有给程序引入任何新的错误。因此回归测试，每次都应该使用相同的测试用例。有时候，随着产品的不断成熟，你会添加新的测试用例，但仍然应当保留旧的测试用例。

(3) 自动化测试

管理回归测试唯一可行的方法，就是将其变成一个自动化的过程。在一遍遍执行相同的测试，并观察到相同的测试结果后，人们常常会变得麻木，对所出现的错误视而不见。这直接违背了回归测试的目的。自动化测试有如下好处。

- 自动化测试发生错误的几率，比手动测试要小。
- 一旦你把一个测试自动化了，那么你只需稍下功夫，就很容易在项目的剩余部分，继续实施自动化。
- 如果测试是自动进行的，那么就可以频繁地运行，看看新 check in 的代码是否破坏了原有的程序。
- 自动化测试可以提高，问题刚产生就被发现的可能性，这可能显著减少分析和修正错误所需的工作量。
- 由于自动化测试能够提升，快速发现修改所引入错误的几率，因此它为大规模代码修改，提供了一张安全网。
- 自动化测试在那些新的、不稳定的技术环境当中，特别有价值，因为它提早稀释了环境改变，对系统的影响，而非事后补救。

进行自动化测试，所需要的工具，应该提供脚手架、生成输入、捕获输出，以及比较实际输出与预期输出等功能。、

22.7 保留测试记录

除了使测试过程有重复之外，你还需要对整个项目进行量化评估，以确定所做的修改，使使程序质量有所提高，还是降低。除了保留项目级的测试记录外，你或许还会发现，坚持编写个人的测试记录，也很有用。这份列表除了记录，你最常犯的错误之外，还可以记录编写代码、测试代码，以及修改代码所花费的时间。

23 调试

24 重构

25 代码调整策略

26 代码调整技术

27 程序规模对构建的影响

28 管理构建

29 集成

30 编程工具

31 布局与风格

32 自说明代码

33 个人性格

34 软件工艺

35 更多信息

References

[1] Meyer CD (2000) Matrix Analysis and Applied Linear Algebra. Philadelphia, PA: SIAM.

[2] Agostino Martinelli. Closed-form solution of visual-inertial structure from motion. International Journal of Computer Vision, Springer Verlag, 2013. hal-00905881