第一章　欢迎进入软件创建世界

目录

相关章节

本书适合什么人阅读：见前言
阅读本书的益处：见前言
为什么要写这本书：见前言

大家都知道“Construction”这个词在一般情况下的意思是“建筑”。建筑工人盖房子、建学校、造摩天大楼等时所进行的工作都是建筑。当你小的时候，你用积木进行“建筑工作”。因此“Construction”指的是建造某个东西的过程。这个过程可能包括：计划、设计、检验等方面的某些工作，但是，它主要是指在这其中的创造性工作。

1.1　什么是软件创建

开发计算机软件是一项非常复杂的工作，在过去的十五年中，研究者们指出了这项工作所包括的主要方面，包括：

- 问题定义
- 需求分析
- 实现计划
- 总体设计
- 详细设计
- 创建即实现
- 系统综合
- 单元测试
- 系统测试
- 校正性的维护
- 功能强化

如果你以前从事过一些不太正规的研制工作，你可能以为列出的这个表有些太详细了。而如果你从事过一些正式的项目，你就会认为这个表非常准确。在正规性与随意性之间达到平衡是非常困难的．这会在以后章节中讨论。

如果你是自学编程员或是主要从事非正规研制工作，你很可能还没有意识到这些在生产软件中所需要的工作步骤。在潜意识中，你把这些工作统统称为编程。在非正式项目中，当你在考虑设计软件时，你所想到的主要活动可能就是研究者们所指的“创建”工作。

关于“创建”的直觉概念是非常准确的，但它往往缺乏正确观点。把创建活动放到与其相

关活动的背景中，有助于我们在适当重视其它非创建工作的同时，把主要精力放在正确的任务上。图 1-1 中给出了创建活动在典型软件生存周期循环中的地位和包括的范围。

正如图中所指出的，创建活动主要指编码和调试过程，但也包括详细设计和测试中的某些工作。假如这是本关于软件开发所有方面的书，它应该涉及到开发过程所有方面并给予同等重视。但因为这是一本关于创建技术的手册，所以我们只重点论述创建活动及相关主题。如果把这本书比喻成一只狗，那么它将用鼻子轻擦创建活动，尾巴扫过设计与测试，而同时向其它开发活动汪汪叫。

创建活动有时被称作“实现”，它有时被叫作“编码和调试”，有时也称之为“编程”。“编码”实在不是一个很好的叫法，因为它隐含着把已经设计好的程序机械地翻译成机器语言的过程；创建则无此含义，它指的是在上述过程中的创造性和决策性活动，在本书中，将交替使用“实现”、“编程”和“创建”。

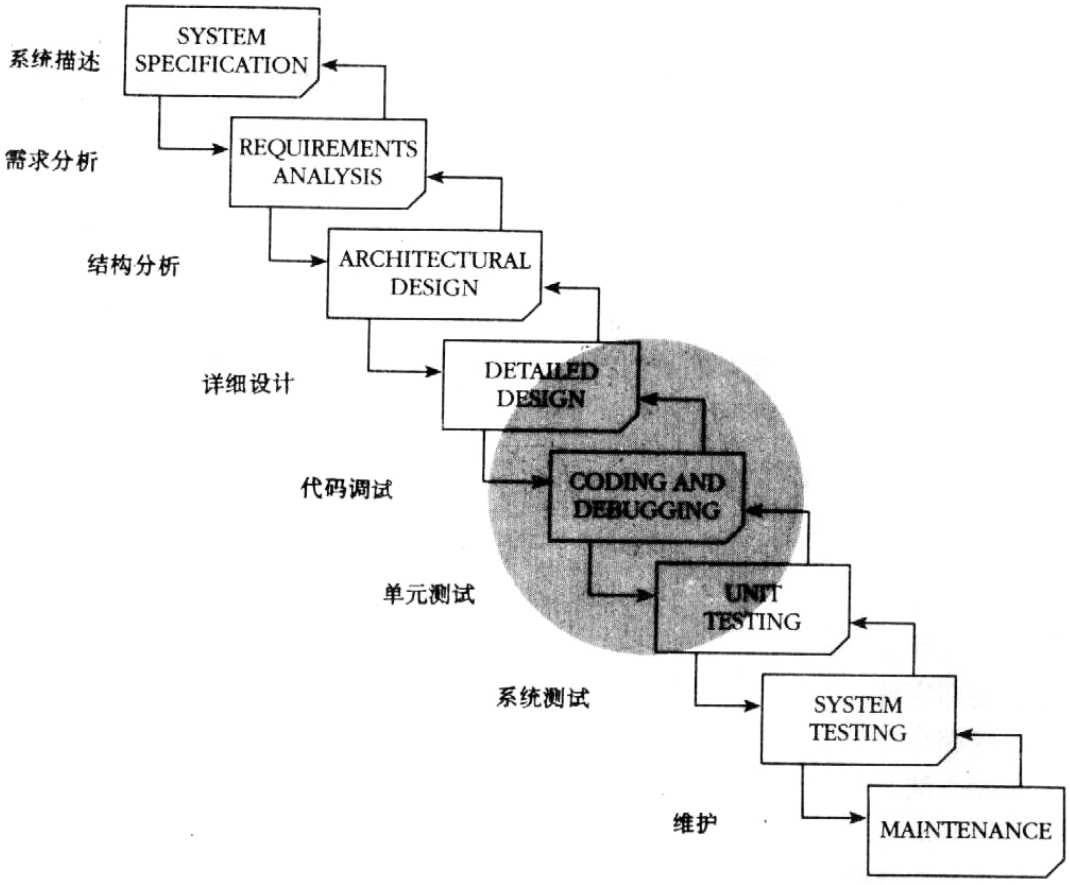


图 1－1 软件生存周期中软件开发过程的平面图

在图 1－1 中，给出了软件开发过程的平面图示，而在图 12 中，则给了它的立体图示。

图 1－1 和图 l－2 是创建活动的总体图示，但是，什么是它的细节呢？下面是创建活动中所包含的一些特定任务。

- 验证基础工作已经完成，可以进行创建工作
- 设计和编写子程序与模块

- 创立数据类型并命名变量
- 选择控制结构并组织语句块
- 找出并修正错误
- 评审其它小组的细节设计和代码，同时接受其它小组评审
- 通过仔细地格式化和征集意见改进编码
- 对分别完成的软件单元进行综合
- 调整编码使其更小、更快

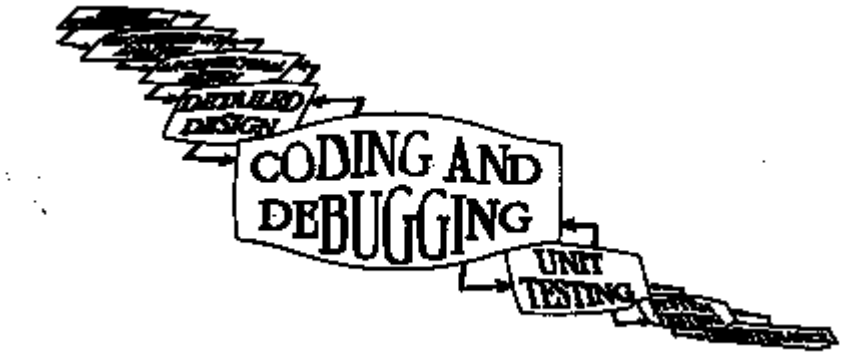


图 1－2 本书主要详细论述详细设计、编码、调试和单元测试（所占比例如图示）

要想更详尽地了解创建活动，请参阅目录中每一章的标题。

创建活动包括如此众多的工作，人们可能会禁不住要问：“哪些是创建活动呢？”。一般认为，非创建活动包括：管理活动、需求分析、软件总体设计、用户交互界面设计、系统测试、维护工作等。这其中每项工作都与创建工作一样，会直接影响到项目的最终成败（那些需要两个人以上合作至少一星期项目的成败）。关于这其中每一项活动都有很不错的论著，在本书每一章后都列出这些书的书名。

1.2　软件创建的重要性

正如我们所知，改进软件质量、提高软件生产率是非常重要的。当今世界许多激动人心的工程计划中，软件都被广泛地应用：太空飞行、航空、医学与生命保障科学、电影特技、金融信息的快速处理、科学研究等，这仅是其中的几个例子。如果读者您也认为软件开发是重要的，那么您就会问，为什么创建活动是重要的？原因如下：

创建活动是开发软件的重要组成部分。随项目规模不同，创建活动在整个开发活动中所占时间为 30%～80%之间，在任何计划中占有如此大时间比例的活动必然会影响计划的成败，这是不言而喻的。

创建活动在软件开发中处于枢纽地位。分析和设计是创建活动的基础工作，对系统进行测试以证实创建活动是正确的则是其后续工作，因而创建活动是软件开发的核心工作。

把主要精力集中于创建活动，可以极大地提高程序员的生产效享。由 Sackman、Erikson 和

Grant 在 1968 年进行的实验表明，每个程序员的效率系数的变化范围为 10～20，这一结果随后又被其它几个实验所证实。最优秀程序员与普通程序员的巨大差异表明，普通程序员提高效率的潜力是非常大的。

创建活动的产品，源代码，往往是软件的唯一精确描述。在许多项目中，程序员可得到的唯一文件便是代码本身。需求说明和设计文档可能会过时，但源代码却总是最新的。因此，源代码必须具有最好的质量。一个软件成功与否的关键，就在于是否不断运用技术来改进源代码。而这些技术恰恰是在创建阶段，才能得以最有效的应用。

创建活动是唯一一项必不可少的工作。理论上一个软件项目要经过精心的需求分析和总体设计，然后再进行创建，接着对其进行彻底的、严格的系统测试。然而，实际工作中的软件项目，往往越过前两个阶段而直接进行创建活动，最后，由于有太多的错误要修改，系统测试又被弃之路旁。但是，不管一个项目的计划多么疏漏而又如何匆忙，创建活动都是必不可少的。无论怎样精简，改进创建活动都是改进软件开发工作的方法。

1.3　小　结

- 创建活动是总体设计和系统测试之间承上启下的工作。
- 创建活动主要包括：详细设计、编码、调试和单元测试。
- 关于创建活动的其它称谓有：实现、编程等。
- 创建活动质量对软件质量有潜在影响。
- 在最后的分析中，对创建活动理解的好坏，决定了一个程序员素质的高低，这将在本书其余部分论述。

第二章 利用隐喻对编程进行更深刻的理解

目录

相关章节

设计中的启发：“设计是一个启发过程”见 7.5 节

计算机科学的语言可能是所有科学领域中最丰富的。想象一下。你走进一间干净整洁、温度严格控制在 68℉的房间，在这里，你将会找到病毒、蠕虫、臭虫、炸弹、崩溃、火焰、扭曲的变形者、特洛伊木马和致命错误，在其它领域中，你会遇到这种情况吗？

这些形象的隐喻描述了特定的软件现象。同样形象的隐喻描述了更为广泛的现象，你可以利用它们来加深你对软件开发的理解。

本书其余部分与本章关于隐喻的论述无关，如果你想了解实质问题可以跳过这一章。但你要想对软件开发有更清楚的理解，请阅读这一章。

2.1 隐喻的重要性

重大发现往往是从类比中产生的。通过把一个你所陌生的事物与你所熟知的事物比较，你会对它有进一步的认识，从而形成你对它的独到的深刻理解，这种隐喻方法被称之为“模型化”。在科学发展史上，充满了利用类比而产生的发现。化学家 Kekle 梦见一条蛇咬住了自己的尾巴，醒来后，他由此联想到苯的结构，提出了苯是环形分子的假说，这一假说在 1966 年被 Barbour 用实验所证实。

分子运动论是在“保龄球”模型上建立起来的。在这里，分子被假想为具有质量并且与保龄球一样相互之间进行完全弹性碰撞的小球，并且在此基础上，又产生了许多有用的模型。

光的波动理论是在与声音类比的基础上产生的。光与声都具有振幅（亮度与音量），频率（颜色与音调）和其它类似性质。这种类比是如此有效，以致于科学家们花费了大量时间来寻找像空气传播声音一样传播光的物质——“以太”，但他们从来也没能找到。有时如此有效的类比这次却导出了错误结果。

通常，模型的力量在于它能提供生动形象的概念而易被人整个接受。并提供特性、联系和附加的疑问，有时模型会提出令人困惑的问题，这时往往是由于模型被误解了，那些建筑“以太”的科学家们，就是因为误解了模型。

正如你所预料的，有些模型比其它的要好。好的模型要简单、与其它模型关联密切、能解释

大部分实验事实和观测现象。

比如一个悬在铁链上来回晃动的大石头。在 Galileo 之前，Aristotelian 看到它时想到的是重物必然要从高处落下来停在低处，他认为石头是在克服阻力下落，而当 Galileo 看到同一现象时，他认为自己看到了一个单摆，他认为石头是在不断地重复同一运动。

这两个模型所提供的信息是截然不同的。Aristotelian 认为石头是在下落，因而他关心的是石头的重量、升起的高度及停下所需的时间。而 Galileo 从单摆模型出发，他关心的是石头的重量、铁链的半径、石头的角位移及石头每摆一次所需要的时间。Galileo 之所以能发现单摆定律，就是因为他的模型与 Aristotelian 不同，从而导致他们提出了不同的问题。

隐喻对加深软件理解所做出的贡献，与它对其它领域所做出的贡献一样大。1973 年，在图灵奖颁奖演说中，Charles Bachman 叙述了从地心说向日心说转移的过程。Ptolemy 的地心说统治了近 1400 年。直到 1543 年，Copernicus 提出了日心说，这一思想模型的转变导致了一系列新星的发现，把月亮定义为卫星而不是行星，也改变了人类对自身在宇宙中地位的理解。

Bachman 把天文学中从地心说向日心说的转变，与 70 年代前期在计算机编程中的变化作了个对比。在当时，数据处理正从以计算机为中心向以数据库为中心进行转变。Bachman 指出，在旧的处理模式中，数据被当成是一个连续流过计算机的卡片流（以计算机为中心）；而在新的模式中，数据好比是一个水池，而计算机则偶尔涉足其中（以数据库为中心）。

今天，很难想象谁会认为太阳绕着地球转；也同样难以想象推会把数据当成流过计算机的卡片流。在这两个例子中，旧的理论一旦被抛弃，很难想象有谁会再把它捡起来。具有讽刺意味的是，旧理论的相信者认为新理论荒唐可笑，就像我们今天看旧理论一样。

当日心说出现之后，地心说便成了那些相信它的天文学家的阻碍。同样，计算机中心模式也已经成了那些相信它的计算机科学家的阻碍，因为我们现在已经有了数据库中心模式。

如果一旦看了新的模型，我们便说：“哦，当然正确的模型更有用，其余的都是错误的”，那只会降低模型的作用。因为这太偏激了。科学史并不是由一系列从“错误”模型到“正确”模型开关组成的，而是逐渐由“坏的”模型变为“较好”的模型，从包含面较窄到包含面较宽，从覆盖领域较少到覆盖领域较多。

事实上，很多被较好模型替代的旧模型仍然在发挥作用。例如，工程师们仍然在用牛顿力学进行工程计算，虽然它已经被相对论力学所取代。

软件科学是一门比其它学科年轻得多的学科，还很不成熟，远未形成一套标准的模型。所以，现在拥有的是大量相互矛盾的模型。这其中有些很好，有些则很差。因此，对这些模型理解得好坏，便决定了你对软件开发理解的好坏。

2.2 如何使用软件隐喻

软件隐喻更像是一束搜索灯光，而不是一张地图，它并不会告诉你到哪里去寻找答案；它只给你以启发，教你如何寻找答案，而不是像数学算法一样硬性规定出到哪里找出答案。

一个公式是一套完整建立的、进行某一些任务的规则。它的结果是可以预测的、确定的，并不取决于运气。公式会告诉你直接从 A 点走到 B 点，中间不准绕路，不准随意顺便访问 C、D、E 或 F 点，也不准停下来闻一下玫瑰花香或者喝杯咖啡什么的，一切必须按规定来。

启发是一种帮助你寻求答案的技术。它的结果往往和运气有关，因为它只告诉你如何去

找，而并未告诉你应该找到些什么。它不会告诉你怎样直接从点 A 到点 B．甚至很可能它根本就不知道点 A 和点 B 在哪里。事实上，可以认为启发是一个穿着小丑儿外套的公式。它往往不可预测，更富有趣味，不会保证一定会发生或不会发生什么。

比如，开车去某人家的公式是这样的：沿 167 号公路向南到 Sumner，从 Bonney 湖出口向山上开 2.4 英里，借助加油站的灯光向左拐，在第一个右转弯处向右转，再拐入通向褐色房子的公路，寻找的门牌号是北大街 714 号。

以下则是一个如何找到我们房屋的启发：找到我们寄给你的最后一封信，开车到回信地址所说的小镇，到了镇上后随便问哪个人我们住哪儿，别担心，镇上的人都认识我们。如果你谁也遇不到的话，就打电话找我们。

公式和启发之间的区别是微妙的，这两个例子或许会说明一些问题。从本书的角度来看，它们之间的主要区别是：它们与答案之间的直接程度。公式给予直接指令；而启发则告诉你该怎样找到这些指令，或者至少告诉你到哪里寻找它们。

如果有一套指令告诉你该如何解决程序中的问题，这当然会使编程变得很容易，而且结果也可以预测了。但是编程科学目前还没有那样发达，也许永远也不会。编程中最富于挑战性的问题便是将问题概念化，编程中许多错误往往都是概念性错误，因为每个程序在概念上都是独特的，所以创立一套可以指导每一个问题的规则是非常困难，甚至是不可能的。这样，从总体上知道该如何解决问题，便几乎和知道某一特定问题的答案一样重要了。

你是怎样使用软件隐喻的呢？应该用它来帮助你获得关于编程过程的内在理解，利用它们来帮助你考虑编程活动，想象解决问题的更好办法。你不要一看到某一行代码就说这与这一章所使用的某个隐喻相矛盾。随着时间推移，在编程过程当中使用隐喻的程序员肯定比不使用这一方法的人编写代码更快更好。

2.3 通常的软件隐喻

随着软件的发展，隐喻越来越多，已经到了使人迷惑的地步，Fred Brooks 说写软件就像耕种、猎狼或者在一个沥青矿坑中淹死一只恐龙。Paul Heekel 说这就像电影《白雪公主与七个小矮人》。David Gries 说这是科学，Donald Knuth 则说这是门艺术，Watts HamPhrey 则说这是一个过程，Peter Freeman 说这是个系统，Harlan Mills 认为这就像解数学题、做外科手术、或者是宰一条狗，Mark Spinrad 和 Curt Abraham 说这更像是开发西部、在冰水中洗澡或者围着营火吃豆子。

2.3.1 软件书写：写代码（Writing Code）

开发软件最原始的隐喻出自“写代码”一词。这个写的隐喻说明开发一个程序就像随便写封信，你准备好纸、笔和墨水，坐下从头写到尾就算完成了。这不需要任何正式计划，你只是把你要说的都写出来。

许多想法都源于写隐喻。Jon Beitle 说，你应该准备好一杯白兰地，一支上等雪茄，与你喜欢的猎狗一同坐在火边，像一个优秀小说家一样享受一次“自由编程”。Brian 和 Kernighan 把写隐喻风格的书称为《风格要素》(《The Elements of Style》)之后，把他们编程风格的书称作《编程风格要素》(《The Elements of Programming Style》)，程序员们则经常谈论程序的“可读性”。

在一些小问题中，写代码隐喻可以充分描述它们。但是对于其余的问题，它就力不从心了，它不可能全面彻底地描述软件开发过程。写往往是一种个人活动，而软件开发往往需要许多人分担各种不同的责任。当你写完一封信时，你把它装进信封并把它寄出去后，你就再也不能改变它的内容了，无论从哪个角度说，这项工作都已经完成了。软件的内容是很容易改变的却很难彻底完成。几乎有50%的软件开发工作量是在软件最初发行之后才进行的（Lientz和Swanson，1980）。编写软件，主要工作量集中在初始阶段。在软件创建中，把精力集中于初始阶段往往不如在初始工作完成后，再集中精力进行代码的重新调整工作。简而言之，写隐喻往往把软件工作表示成是一项过于简单而刻板的工作。

不幸的是，写隐喻已经通过我们这个星球上最流行的软件书——Fred Brooks 的《The Mythical Man Month》而变得永存了。Brooks 说，“扔掉一个计划，又有什么呢？”这使得我们联想到一大堆被扔进废纸篓的手稿。当你写封家常信问候你叔叔时，准备扔掉一封信是可能的，这也可能是 Brooks1975 年写那本书时，当时软件工程的水平。

但是，到了九十年代，再把写隐喻解释为准备扔掉一封信时，恐怕是不合时宜的。现在，开发一个主要系统的投资已经相当于建一幢十层办公楼或造一艘远洋客轮的费用了。我们应该在第一次调试时就完成它，或者在它们成本最低时试几次运气，其它几个隐喻较好地解决了说明达到这一目的的方法问题。

2.3.2 软件播种：生成系统（Growing a System）

与刻板的写隐喻相反，一些软件开发者认为你应该把创建软件当作播种或培植庄稼。你设计一小部分，编码一小部分，测试一小部分，然后在某个时候把它加到系统上，通过小步走，你减小了每次可能遇到的错误。

有时，一项先进的技术可能是通过拙劣的隐喻来表达的。在这种情况下，应努力保留这项技术并换一个隐喻来表达它。在这里增量技术是先进的，但是种庄稼的比喻则是十分拙劣的。

一次干一点儿的想法可能和植物生长有某种类似之处，但是耕种类比实在太牵强，而且也令人感到陌生，因而也就很快被后面的隐喻所取代了。很难把耕种隐喻推广到每次做一点儿这一简单想法之外。如果你来用耕种隐喻，你就会发现自己在谈论给系统计划施肥，减少详细设计，通过有效地田间管理提高编码产量，最后收获编码。你也会谈论进行轮作，用种小麦代替大麦，让土地休息一年以提高土壤中的养分。

软件种植隐喻的弱点是你对于软件开发失去了直接控制。你在春天播种代码，最后在秋天收获一大堆代码。

2.3.3 软件珍珠培植法：系统积累（System Accretion）

有时候，人们在谈论种植软件而事实上他们指的是软件积累。这两个隐喻是密切联系的，但是软件积累更深刻一些。“积累”这个词，含有通过外加或吸收，缓慢生长的意思，就像河蚌逐渐分泌酸钙形成珍珠一样。在地质学上，水中悬浮物逐渐沉积形成陆地的过程也与此相似。

这并不是说你要从水中悬浮物里沉积出代码来；这只意味着你应该学会每次向你的系统中加一点儿东西。另外一个与积累密切相联的词是增量。增量设计、构造、测试是软件开发的最强有力工具之一。“增量”一词在设计者心目中还远未达到“结构化”或“面向对象设计”等的地位，所以迄今为止也没有一本关于这方面的论述，这实在是令人遗憾的，因为这种书中所收集的技术将具有极大的潜力。

在增量开发中，你首先设计系统可以运行的最简单版本。它甚至可以不接受实际数据输入，或者对数据进行处理。它也可以不产生输出，只需要成为一个坚实的骨架结构，以便能承受将要在它之上发展的真实系统。它可以调用任何一个实现预定功能而设立的伪子程序。就像河蚌刚开始产生珍珠的核——一粒沙子。

当你搭好骨架后，逐渐地往上添加肌肉和皮肤。你把每一个伪子程序变成真正的子程序。

此时你不必再假设产生结果了，你可以随意访问一个代码来产生结果。也不必使其假设接收输入，你可以用同样的方法让它接收输入。你每次加入一点儿代码直到你最终完成它。

这种方法的发展是令人印象非常深刻的。Fred Brooks，在 1975 年时还认为：“应做好建造一个扔掉一个的准备”，在 1987 年时，却说在过去的岁月里，还没有一样东西像增量概念这样如此深刻地改变了他自己的实践或效率。

增量隐喻的力量在于：作为一个隐喻，它并没有过分作出许诺，它不像耕种隐喻那样容易被错误延伸。河蚌育珍珠的联想对理解增量发展法或积累法有很大帮助。

2.3.4 软件创建：建造软件（building software）

“建造”一词的想象比“写”或者“种植’软件的想象更为贴切，它与“增量”软件的想法是基本一致的。建造隐喻暗示了许多诸如计划、准备、执行等工作阶段。如果你仔细研究这个隐喻，你还会发现它还暗示着其它许多东西。

建造一个四英尺高的塔需要一双稳健的手、一个平台和十个完好的啤酒罐。而建造一个四百英尺高的塔却决不仅仅是需要一千个啤酒罐就够了，它还需要一种完全不同的计划和创建方法。

如果你想建一个简单的建筑物，比如说一个狗舍，你买来了木板和钉子，到下午的时候，你已经给你的爱犬造好了一幢新房子，假设你忘了修一个门，不过这没关系，你可以补救一下或推倒一节重新开始。你所浪费的不过是一个下午的时间罢了。这与小型软件的发展失败非常类似。如果你有 25 行代码设计错了。那你重新再来一遍好了，你不会因此浪费许多的。

然而如果你是在造一幢房子，那修建的过程就要复杂些了，而拙劣设计的后果也严重得多。首先，你必须决定造一幢什么样的房子，这就像软件开发中的问题定义。然后，你与建筑师必须搞出一个你们都同意的总体方案，这和软件的总体设计是一样的。接着，你又画出细节蓝图并找来一位承包商，这相当于软件中的详细设计。下面的工作是选好房址、打地基、建造起房屋的框架、建好墙壁并加上屋顶、用千斤锤检查墙壁是否垂直，这同软件创建基本差不多。当房屋的绝大部分工作已经完成时，你请来园艺师和装修师，以便使你的房间和空地得到最好的利用，这可以与软件优化相类似。在整个过程中，会有各种监督人员来检查房址、地基、框架、供电系统和其它东西，这也可以与软件开发中的评审和鉴定相类似。

较大的规模和复杂性往往意味着可以产生较大的成果。在修房子的时候，材料可能比较贵，但更大的花费是劳动力。拆掉一面墙并把它移到六英尺之外是很昂贵的，但并不是因为你浪费了许多钉子，而是因为你需要付出劳动。你应该尽可能精心设计，以避免那些本可避免的错误，以降低成本。在开发软件过程中，材料更便宜，然而劳动力成本却更高。改变一个报告的格式，可能与移走一幢房子里的墙壁一样昂贵，因为二者成本的主要部分都是劳动力。

这两个活动之间还有什么类似之处呢？在建房子中，你不会去建造那些你可以现成买来的东西，比如洗衣机、烘干机，电冰箱、吸尘器等，除非你是个机械迷。同时，你也会去购买已经做好的地毯、门、窗和浴室用品，而不是自己动手建。如果你正在建造一个软件，你也会这样做。你会推广使用高级语言的特点，而不是去编写操作系统一级的代码。你也会利用已经存在的显示控制和数据库处理系统，利用已经通过的子程序。如果样样都自己动手是很不明智的。

如果你想修建一幢陈设一流的别墅，情况就不同了，你可能定做全套家具，因为希望洗碗机、冰箱等与你的协调一致，同时你还会定做别具风格的门和窗户。这种定做化的方式与一流软件开发也是非常类似的。为了这一目的，你可能创建精度更高、速度更快的科学公式。你也会设计自己的显示控制、数据库处理系统和自己的子程序，以使整个软件给人以一气呵成，天衣无缝的感觉。

当然这两种建造方法也要付出代价，工作的每一步都要依据事先制定好的计划进行。如果软件开发工作的顺序有误，那么这个软件将是难以编码、难以测试和难以调试的。这可能会使整个计划延误甚至失败，因为每个人从事的工作都非常复杂，把它们综合到一起后会使人无所适从。

如果你在盖办公楼时工作做得不好，那么在楼内办公的人便可能面临危险。同样，如果你在创建医药、航空电子、空中交通管制、加工控制等软件时工作做得不好，后果也可能是灾难性的。危及别人生命是劣质软件的最可怕后果，但并不是它的唯一危害。如果公司的股东们因为你编写了错误软件而赔钱，那也是令人遗憾的。无论如何，无辜的人们没有义务为你的工作失误而付出代价。

对于软件作修改与建造建筑物也有类似之处。如果你要移走的那面墙壁还要支撑其它东西而不仅仅是隔开两个房间，那么你要付出的成本将会更高。同样，对软件做结构性的修改也将比增加或减少外设特征付出更高昂的代价。

最后，建筑类比对于超大型软件也是同样适用的。一幢超大型建筑物存在错误的后果将是灾难性的，整个工程可能不得不返工。建筑师们在制定和审查计划时是非常仔细的，他们往往留出安全裕度，多用 10%的材料来加强结构总比一幢大楼坍塌要好得多，同时还必须仔细注意工时计划，在修建帝国大厦时，每辆卡车的每次卸货时间都留出了十五分钟的裕度。因为如果有一辆卡车不能在指定时间到达指定的位置，整个计划就有可能被延误。

同样，对于超大型软件来说，计划工作需要比一般的大型软件在更高的层次上进行。1977 年，CapersJones 估计说，对于一个拥有 750，000 行代码的系统来说，可能需要多达 600 页的功能定义文件。对于一个人来说，不要说理解这种规模全部的设计，就是读完它也是非常困难的。安全系数对于这种项目是必须的，制定该系统的工时计划尤为重要。当我们在建造与帝国大厦同等经济规模的软件时，我们也需要同等严密的计划。而我们现在才刚刚开始考虑这种规模项目的计划技术。

这两者之间的相似还可以推广到其它方面，这就是为什么建筑物创建隐喻是如此强有力的原因。许多常用的软件词汇来源于建筑学，如：软件体系结构、搭结构架、构造、分割代码、插入子程序等等。

2.3.5 实用软件技术：智能工具箱（The Intellectual Toolbox）

在过去的十几年中，优秀的软件开发人员们积累了几十条关于开发软件的技术和技巧，有

些像咒语般灵验，这些技术不是规则，它们是分析工具。一个优秀的工匠知道用什么样的工具干哪一样工作，而且知道该如何使用它们。程序员也是如此，关于编程你理解得越深入，你的工具箱里的工具也就越多，何时何地该如何运用它们的知识也就越多。

把方法和技巧当作工具是很有益处的，因为这样可以使我们对其有一个正确的态度。不要把最新的“面向对象设计技术”当作上帝赐予的法宝，它不过是一件在某些场合下有用，而在某些场合下又无用的技术。如果你拥有的唯一工具就是一把锤子，那么你就会把整个世界都当作一个钉子。好在没有人会花 500 美元一天的费用来雇佣一个仅告诉你去买一把可以解决一切问题的锤子的研究小组，也没有人建议你丢掉你的改锥、手钻和电烙铁。

在软件开发中，常常会有人告诉你用一种方法来代替另外一种方法。这实在不幸，如果你仅仅采用一种方法，那你就会把整个世界都当成那个工具的作用对象。你会失去用更适合的方法解决问题的机会。工具箱隐喻有助于我们保留一切方法、技巧、技术等，并在适当的时候使用它们。

2.3.6 复合隐喻（Combing Metaphors）

因为隐喻更像是一种启发，而不是公式，所以，它们并不是互相排斥的。你可以同时使用增量隐喻和建筑隐喻。如果你愿意的话，你也可以采用“写”隐喻，或者把写隐喻与耕种隐喻一起使用。只要能激发你的思想，你尽可以采用一切你认为合适的隐喻。

使用隐喻是一项模糊的事情。你不得不把它们外推到可以从中受到启发的外延中。如果你把它过分外推或者推广到了错误方向，它很可能使你误入歧途。就像是再好的工具也有可能被误用一样，你也可能错误使用隐喻。但是，它们的作用将无可置疑地使其成为你的智能工具箱中的一件有力工具。

2.4 小 结

隐喻仅仅是启发，而不是公式，因此，它们更倾向于比较随便，无拘无束。

- 隐喻通过把软件开发与你所熟知的事情联系在一起，从而使你对其有更深刻的理解。
- 一些隐喻要好于其它隐喻。
- 把软件创建与建造建筑物类比，表明开发软件前要精心准备，并表明了大规模项目与小规模项目之间的差别。
- 认为软件开发实践是智能工具箱中的工具进一步表明，每个程序员都有许多自己的工具，没有任何一种工具是万能的。为每件工作选择合适的工具，是成为一个优秀程序员的首要素质之一。

第三章　软件创建的先决条件

目录

相关章节

- 不同规模程序的不同条件：见第 21 章
- 管理创建：见第 22 章
- 设计：见第 7 章

在开始修造一幢房屋之前，建筑工人会评审蓝图，确认所有用料已经备齐，并检查房子的地基。建筑工人为修建摩天大楼和修建狗舍所做的准备工作是截然不同的。但不管是什么样的项目，准备工作总是和需要相适应的，并且应在工程正式开始前做完。

本章主要论述在软件创建之前所要做的准备工作，对于建筑业来说，项目的成败往往在开工前就已经决定了。如果基础打得不好，或者项目计划进行得不充分，你所能做的最多也就是防止计划失败，根本谈不上做好。如果你想做一件精美的首饰，那么就得用钻石作原料。如果你用的是砖头，那你所能得到的最好结果不过是块漂亮的砖头而已。

虽然本章讲的是软件创建基础工作，但并没有直接论述创建工作。如果你觉得不耐烦，或是你对软件工程生存期循环已经很熟悉了，那么请跳过本章而直接进入下一章。

3.1　先决条件重要性

优秀程序员的一个突出特点是他们采用高质量的过程来创建软件。这种过程在计划的开始、中间和末尾都强调高质量。

如果你只在一个计划即将结束时强调质量，那你注重的只是测试。当某些人一谈起软件质量时，他们首先想到的便是测试。然而，事实上测试只是全部质量控制策略的一部分。而且并不是最重要的部分。测试既不能消除在正确方向上的错误工作，也不能消除在错误方向上的正确工作的错误，这种错误必须在测试开始之前就清除掉，甚至在创建工作开始之前就要努力清除掉它们。

如果你在一个计划的中间强调质量，那么你强调的是创建活动，这一活动是本书论述的中心。

如果在一个计划的开始强调质量，这意味着你计划并要求设计一种高质量的产品。假设你在过程开始时要求设计的是一种菲亚特汽车，你尽可以用你所喜欢的各种手段测试它，但是无论你怎样测试，它也决不会变成一辆罗尔斯——罗伊斯牌汽车。或许你所得到的是一辆最好的菲亚特汽车，但如果你想要的是罗尔斯——罗伊斯车，你就不得不从计划开始时就提出要求。在软件开发中，当你进行诸如问题定义、规定解决办法等等计划工作时，你所进行的就是这样的工作。

由于创建工作处在一个计划的中间，所以，当你开始创建工作时，早期的工作已经奠定了项目成败的基础。在创建工作中，至少你应该知道自己的处境如何，当你发现失败的乌云从地平线上升起时，赶快返回第一阶段。本章其余部分主要讲述准备工作已经作好了。

3.1.1　造成准备不足的原因

你也许会认为所有的职业程序员都懂得准备工作的重要性，并且在开始正式工作之前确认所有的先决条件都已得到满足。不幸的是，事实并非如此。

一些程序员并不作准备工作，因为他们抵制不了立刻开始进行编码工作的渴望。如果你就是这种程序员，那我对你有两条忠告。第一，阅读一下下一部分工作的内容提示，或许你会从中发现一些你没想到的问题。第二，要注意自己的问题。只要创建过几个大的程序，你就会明白强调准备工作的必要性。不要忘记自己的经验教训。

程序员不重视准备工作的另一个原因是管理人员往往不理解那些在创建先决条件上花费时间的程序员。 Ed Yourdon 和 Tom DeMarco 等人强调准备工作已经有十五年了。在这期间，他们不时地敲响警钟，或许有一天，管理人员们最终会明白软件开发不仅仅是编写代码。

八十年代后期，我曾经在一项军用项目的某一部门中工作。当项目进行到需求分析阶段时，负责这个计划的一位将军前来视察。我们告诉了他目前所处的阶段，并主要谈论了文件编写工作，而这位将军却坚持要看一下代码，我们告诉他目前还没有代码，而他却走进一间正有一百多人工作的房间，转了一圈，企图找到谁在编码。由于未能如愿以偿，他变得有些气急败坏，这位身材高大的将军指着自己身边的工程师喊道：“他在干什么？他一定是在写代码。”事实上，这位软件工程师正在进行文档格式编排的工作，由于这位将军想得到代码，认为那看起来像代码并且想让工程师编码，所以我们不得不骗他说这位工程师写的确实是代码。

这可以称为 WISCA 或 WIMP 现象，即：为什么 Sam 没有正在写代码？或 Mary 为什么没正在编程？

如果你正在从事的项目经理像那个将军一样，命令你立刻开始编码，说声“是，长官”是很容易的。但这是一个坏的反应，你应该还有几个替代办法。

第一，你应该平静地拒绝按照错误顺序工作。如果你与老板的关系很正常的话，那么这太好了。

第二，你可以假装正在编码而事实上没有。把一个旧的程序清单放到桌角上，然后埋头从事你的要求和构想文件编写工作，不管你的老板同不同意。这样你可以把工作做得更快更好。从你老板的观点来看，这个忽视是一个福音。

第三，你可以用技术项目的开发方式来教育一下老板。这是一个好办法因为这可以增加这

个世界上开明老板的数量。在下一部分，我们将给出更多在创建活动前做好准备工作的理由。

最后，你可以另找一份工作。优秀的程序员是非常短缺的。可以找到更好的工作，干吗非要呆在一个很不开明的程序店里，徒损生命呢？

3.1.2　在进行创建工作之前必须做准备工作的论据

假设你已经登上了问题定义的山峰，与负责需求分析的人并肩走了一英里，在结构设计之泉中，洗净了你沾满灰尘的衣服，并且沐浴在已经作好准备的纯洁之水中。那么你就会知道在实现一个系统之前，你应该清楚需要一个系统干什么和需要怎样去干。

作为一个工程技术人员，教育你周围的人，让他们懂得技术项目的开发过程，也是你工作的一部分。本书的这一部分可以帮你对付那些还不懂得技术项目开发过程的老板和管理人员。它是关于进行构造设计和问题定义设计权利的延伸论据。在你进行编码、测试和调试之前，学会这些论据，并且和你的老板推心置腹地谈谈技术项目的开发过程。

求助于逻辑推理

进行有效程序设计的关键之一就是认识到准备工作是非常重要的。在进行一项大的项目之前，事先做好计划是明智的。项目越大，需要的计划工作量也越大，从管理人员的角度来看，计划是指确定一个项目所需要的时间、人力、物力和财力。从技术人员的观点来看，计划是指弄清楚你想要干什么，以免做出错误的工作而徒耗精力与钱财。有时候你自己并不十分清楚自己想要的到底是什么？起码刚开始是这样。这时，就会比清楚知道用户需求的人要付出更多努力，但是，这总比做出一件错误的东西，然后把它扔掉，再从头开始的成本要低得多。

建造一个系统之前，弄清楚怎样开始和如何建造它也是非常重要的，你当然不希望在完全没有必要的情况下，浪费时间与钱财去钻死胡同而白白增加成本。

求助于类比

创建一个软件系统与其它需要耗费人力与财力的工程是一样的。如果你要造一幢房子，在开始砌第一块砖之前，你必须事先画好建筑图与蓝图。在你开始浇铸水泥之前，你必须让人评审你的蓝图并获得通过，在软件开发中事先做计划也与此类似。

在你把圣诞树立起来后，你才会开始装饰它，在没有修好烟囱之前你也不会点燃炉火的，同样，也没有人会打算在油箱空空的情况下踏上旅程，在软件开发中，你也必须按照正确的顺序来进行。

程序员处于软件开发食物链的最后一环。结构设计吃掉需求分析；详细设计者以结构设计者为食，而他自己又成为编码者的食物。

比较软件食物链和真正的食物链，我们会发现如下事实，在一个正常的生态系统中，海鸥以沙丁鱼为食，沙丁鱼吃鲜鱼，鲜鱼吃水虱，其结果会形成一个正常的食物链。在编程工作中，如果软件食物链的每一级都可以吃到健康的食物，其结果是由一群快乐的程序员写出的正确代码。

在一个被污染了的环境中，水虱在受到核沾染的水中游泳，鲫鱼体内积聚了滴滴涕，而沙丁鱼生活的水域又遭受了石油污染，那么，不幸的海鸥由于处在食物链的最后一环，因此，它吃的不仅仅是沙丁鱼体内的石油，还有鲜鱼体内的滴滴涕和水虱体内的核废料。在程序设计中，

如果需求定义遭受了污染，那么这又会影响结构设计，而这将最终影响创建活动。这将导致程序员们脾气暴躁而营养不良，同时生产出遭受严重污染而充满缺陷的软件。

求助于数据

过去十五年的研究证明，一次完成是最好的选择，不必要的修改是非常昂贵的。

TKW 的数据表明，在项目的初期阶段进行设计更改，比如在需求定义和结构设计阶段进行更改，与在项目的后期，即创建和维护阶段进行更改相比较，其成本要低 50 到 100 倍（Boehm 和 Pappecio，1988）。

对 IBM 的研究也表明了同样结果。在设计开始阶段，如详细设计、编码或单元测试阶段就消除错误，其成本要比在后期即系统测试和功能强化阶段低 10 到 100 倍（Fagan，1976）。

通常的准则是，一旦引人错误，就尽早发现和消除它。错误在软件食物链中存留的时间越长，它的危害也就传播得越远。因为需求分析是我们做的第一项工作，因此这时引入的错误在系统中存留时间最长，危害最大。在软件开发初期引入的错误往往比后来引入的错误传播的面更广，这也使得早期错误会极大地提高成本。

由 Robert Dunn 总结的表 3-1，给出了由于错误引入和发现时间不同，而产生修复它们所要耗费的相对成本差异。

表 3-1

错误引入时间			
错误发现时间	需求分析	细节设计	编码
需求分析	1	—	—
细节设计	2	1	—
波动测试	5	2	1
结构测试	15	5	2
功能测试	25	10	5

表 3-1 的数据表明，在需求分析阶段引入的错误，如果马上发现并消除所耗费的成本是 1000 美元的话，那么如果到了功能测试阶段才发现和消除，耗费的成本则会高达 25000 美元。这说明我们应该尽早地发现并消除错误。

如果你的老板不相信这些数据，那你可以告诉他，立刻开始编码的程序员往往要比那些先作计划、而后才编码的程序员花费更长的时间，由 NASA 计算机科学公司和马里兰大学联合建立的软件工程实验室的研究表明，过分地使用计算机（进行编辑、编译、链接、测试等）往往与低生产率紧密相联。而在计算机旁花费较少时间的程序员，往往更快地完成工作。这是由于频繁使用计算机的程序员在进行编码和测试之前，花在计划和设计上的时间较少。

老板的意愿测试

当你认为老板已经理解了在开始创建工作之前进行准备工作的重要性，那么情进行下面的测验以证实这一点。

下面这些说法哪些是正确的？

- 我们最好马上就开始编码因为我们将会有许多测试工作要作。

- 我们没有安排许多时间进行测试，因为我们不会发现很多错误。
- 我们已经在计划和设计上花费了这么多精力，我想我们的编码和测试时不会有什么大问题了。

以上这些都是正确的。

在本章的其余部分我们将论述如何确定先决条件是否已经得到满足。

3.2　问题定义先决条件

在进行创建工作之前你要满足的第一个先决条件，便是必须弄清楚你想要解决的问题是什么。由于本书的中心内容是创建活动，因此我们不打算在这里论述如何进行问题定义。我们只想告诉读者如何确认问题定义是否完成，这个定义的质量如何，是否足以作为创建活动的基础。

问题定义只描述要解决的问题是什么，根本不涉及解决方法。它应该是一个简短的说明，听起来像一个问题。比如“我们无法跟上指令系统”听起来像一个问题，也是一个好的问题定义。而“我们需要优化数据入口系统以便跟上指令系统”则是一个糟糕的问题定义，它听起来不像是个问题而更像是个解决方案。

问题定义的工作是在需求分析之前进行，后者是对问题的更为详尽的分析。

问题定义应该从用户的观点出发，使用用户的语言进行定义。一般来说，它不应该使用计算机技术术语进行定义。因为最好的解决办法可能并不是一个计算机程序。比如说，你需要一份关于年度利润的报告，而你已经拥有了一套能产生季度利润的计算机报表系统，如果你的思路仅仅局限于计算机，那你可能会让人再写一个产生年度利润报告的程序加到这个系统中。为达到这个目的，你不得不雇用一个程序员编写并调试出一段相应的程序。可是，要是你的思路开阔一些的话，让你的秘书用计算器把四个季度的利润加到一起，问题不就解决了吗？

当然，如果问题是关于计算机本身时，就是个例外了。比如，计算机的编译速度太慢或者编程工具的问题太多，那我们只能用技术术语来说明问题了。问题定义错误的后果是你可能浪费许多时间精力去解决了一个错误问题。这种惩罚往往是双重的，因为真正的问题并没有解决。

3.3　需求分析先决条件

需求详细描述了一个软件系统需要解决的问题，这是找到问题答案的第一步。这项活动也被称作“需求分析”、“需求定义”等。

3.3.1　为什么要有正式的需求

明确的需求是很重要的，因为：

明确的需求可以保证是由用户而不是程序员决定系统的功能。如果需求是很清楚的，那么用户可以对其进行评定，并确认自己是否同意。如果需求不很清楚，那么程序员在编程过程中就不得不自己决定系统功能，明确的需求防止对用户需求进行猜测。

明确的需求也可以避免引起争议。在开始编程之前，系统的范围已经明确确定了。如果在编程过程中，两个程序员对系统干什么有争议，那么只要查阅一下写好的需求分析，问题就解

决了。

注意需求定义，也可以使得在开发工作开始之后，对系统作的改动最小、如果你在编码时发现某几行有误，那么改掉这几行就是了。而如果在编码阶段发现需求有误，那么你很可能不得不改变所有的代码以适应新的需求。

一些设计不得不被丢掉，是因为按它们要求写好的代码不具备兼容性。新设计可能要花费很长的时间，被一同扔掉的还有受到要求变更影响的代码和测试用例，即使未受影响的代码部分也不得不进行重新测试，以确认其他地方的变动没有引入新的错误。

IBM、GTE、TRW 的数据表明．修正在总体结构阶段发现的需求错误，将比当时就发现并修正的成本要高出 5 倍，如果是在编码阶段，要高出 10 倍，在单元或系统测试阶段，高 20 倍，在验收测试阶段，高 50 倍，而在维护阶段，竟要比原来高出多达 100 倍！在较小规模的计划中，在维护阶段修正错误的放大因子可能是 20 而不是 100，因为这时管理费用较低。但无论如何没有人愿意从自己的收益中拿出这笔钱来。

充分进行需求分析是一个项目成功的关键，很可能比使用有效的创建技术还重要。关于如何进行需求分析有许多好的论著。因此，我们不打算在随后的几部分中探讨如何进行需求分析。我们只想告诉你如何确定需求分析已经完成，如何最充分地利用需求分析。

3.3.2　稳定需求的神话

稳定的需求可以说是软件开发的法宝。有了稳定的需求，软件开发工作可能从结构设计到详细设计到编码，都平稳、顺利的进行。这简直是造就了软件开发的天堂。你可以预测开支，不必担心最终会冒出一个让你多花 100 倍钱的错误来。

用户一旦接受了写好的需求文件，便再也不会提出更改需求，这简直是太好了。然而事实上，在实际项目中，用户在代码写出来之前，往往并不能确切可靠地描述出他想要的到底是什么，这倒并不是说用户是一种低级生物。正如随着工作的进行，你对其理解越来越深刻一样，用户对自己想要的东西，也是随着项目的进行而越来越清楚的，这也是要求变动的主要原因。一个从不变更要求的计划，事实上是一个对用户的要求不予理睬的计划。

典型的变动有多少呢？根据 IBM 的调查，对于一个典型的有一百万字的需求分析，大约25%的内容在开发过程中要进行变动。

或许你认为卡迪拉克小汽车是空前绝后的，帝国大厦将万古永存，如果真是这样的话，那你就相信你的项目要求永远不会更好了。如果不是这样，那么或许我们可以采取一些措施，使得由于要求变更所造成的冲击最小。

3.3.3　在创建阶段如何对付需求变化

以下是在创建阶段，为应付需求变化而应该采取的措施。

用本部分后面的检查表来评估你的需求分析质量

如果你的需求分析不是很好，那么，停止继续工作，重新返回到需求分析阶段。当然，这样会使人觉得你已经落后了。但是，如果你在开车从芝加哥到洛杉矶的途中，发现自己到了纽约市郊，那么停下车来看一下地图是浪费时间吗？当然不是。因此，如果你发现方向不对，赶紧停下来检查你的方向。

让每个人都知道由于变化需求所付出的代价

雇员们往往由于自己有了新的设计想法而激动不已。在这种兴奋驱使之下，他们往往会热血沸腾，得意忘形。什么讨论要求的会议，什么签约仪式、什么要求文件，统统都会被他们扔在一边。对付这种人最简单办法就是对他说：“喂，先生，你的想法不错，但是由于它不在要求文件之中，我想先做一个变动后的进度和成本估计，然后我们再决定是立刻就采用这个想法还是以后再说”。“时间进度”和“成本”这两个词往往比咖啡和泼冷水更管用，这样说，往往会把许多“立刻采用”变成“最好采用”。

如果你的组织机构还没有认识到需求分析的重要性，那么就请引述本章前面“进行创建活动前满足先决条件的安全和必要论据”一节的内容，告诉他们，在要求阶段变更设计是成本最低的办法。

建立一套更改控制过程

如果雇员们坚持更改的热情高涨，则可以考虑建立一个审查这种更改建议的正式委员会。用户改变主意，意识到他们的软件需要更强的功能是非常正常的。但如果他们频繁地改变主意以至于你无法跟上他们的速度，那就不正常了。这时如果拥有一套控制更改的正式过程，那将使大家都会感到宽慰。你感到宽慰是因为现在你只在特定的时候处理变动问题。顾客也感到宽慰是因为有专门机构处理他们的意见，会使他们感到自己倍受重视。

用开发的方法来容纳变动

一些开发方法可以极大地扩展你应付变更需求的能力。原型化开发的方法可能帮助你在全力以赴投入工作以前，首先了解系统的需求。渐进开发的方法是指按阶段公布系统。每次你只做一点儿，从用户那里得到一些反馈后，你再做一些调整的改动，然后再增加一些内容。这种方法的关键是使用短周期开发方法，以便你对顾客的需求变更迅速作出反应。

放弃项目

如果需求特别稀奇古怪或者反复无常，上面那些办法全都不起作用，那就放弃这个项目。即使你并不能真正地砍掉这个项目，你也可以考虑一下这样做会怎么样。考虑在你砍掉这个项目之前，事情会发展到什么地步。假如在某一情况下，的确可以把这个项目扔进垃圾箱，那么还可以考虑一下有或没有这个项目会造成什么区别。

3.3.4　检查表

需求

这个需求检查表包含一系列关于你的项目需求的自测题。本书并没有论及如何提出一份好的需求文件，这个检查表也同样没有。但用这个检查表，你可以检验一下在创建工作时，你的工作基础是否牢固可靠。

并不是表中所列出的每一个问题都适用于你的项目。如果你正在从事一个非正式项目，你会发现根本不需要考虑这个问题，你也会在其中发现一些需要考虑但并不需要回答的问题。但如果你正在从事一个大型的正视项目，我们建议你最好还是仔细考虑每一个问题。

需求内容

- 系统的所有输入都定义了吗？包括它们的来源、精度、取值范围和频率？
- 系统所有的输出都定义了吗？包括它们的目标、精度、取值范围、频率和格式？
- 所有的报告格式都定义了吗？
- 所有的硬件与软件接口都定义了吗？
- 所有的通信交界面都定义了吗？包括握手、错误检查以及通信约定？
- 是否从用户的观点出发，定义了所有必要操作的反应时间？
- 是否定义了时间问题，如处理时间、数据传输率以及系统吞吐能力？
- 是否对用户所要求完成的任务部作出了规定？
- 每项任务所需用到和产生的数据都规定了吗？
- 规定保密级别了吗？
- 规定可靠性了吗？包括软件出错的后果、在出错时要保护的至关重要的信息、以及错误测试和恢复策略。
- 规定所需最大内存了吗？
- 所需最大存储容量规定了吗？
- 对系统的维护性是否作出了规定？包括系统对运行环境、精度、性能以其与其它软件的接口等方面变化的适应能力规定了吗？
- 是否规定了相互冲突的设计之间的折衷原则，例如，在坚固性与准确性之间如何进行折衷？
- 是否制定了系统成败的标准？

关于需求的完善性

- 在开发开始前暂时得不到的信息是什么？是否规定了不够完善的区域？
- 需求定义是否已经完善到了可以成为软件标准的地步？
- 需求中是否有哪一部分令你感到不安？有没有根本不可能实现，而仅仅为了取悦老板和用户才加进来的内容？

关于需求的质量

- 需求是否是用用户的语言制定的？用户也这样认为吗？
- 需求中是否每一条之间都尽量避免冲突？
- 需求中是否注意了避免规定设计工作？
- 需求在详细程度方面是否保持了一致性；有没有应该更详细些的要求？有没有应该更简略些的？
- 需求是否明确得可以分为一些独立的可执行部分，而每一部分又都很明了？
- 是否每一条都与问题和答案相关？是否每一条都可以追溯到产生它的环境中？
- 是否每一条需求都可以作为测试依据？是否可以针对每一条进行独立测试以确定是否满足需求？
- 是否对可能的改动作出了规定？包括每一改动的可能性？

关于需求定义的进一步阅读

以下是一些给出了如何进行需求定义的书：

DeMarco, Tom 《 Structured Analysis and Systems Specification：Tools and Techniques 》

Englewood Cliffs，N.J：Prentice Hall，1979，这是关于需求定义的经典著作。

Yourdon，Edward 《Modern Structured Analysis》 New York：Yourdon Press，1989，这本新书论述了许多进行需求定义的文字和图表工具。

Hatley，Derek J 和 Imtiz A．Pirbhai《Strategies for Real－Time system Specification》Newyork : Dorset house，1988。这是一本替代 DeMarco 或 Yourdon 书的最佳选择。它重点论述了实时系统，并把 DeDarco 和 Yourdon 提出的图表法扩展到了实时系统中。

Shlaer，sally 和 Stephen Mellor《Object Oritented System Analysis－Modeling the World in Data》Englen wood Cliffs，N.J： Prentice Hall，1988。本书讨论了面向对象设计中的需求分析。

IEEE Std 830－1984（Guide for Software Requirements Specifications） in IEEE 1991。这份文献是 IEEE 为编制软件开发需求定义制订的指导性论述。它描述了需求定义中应该包括的内容并给出了几个例子。

Gibson，Elizabeth《objects－Born and Bred》Byte，1990 10:245－54。这篇文章是关于面向对象需求分析的入门书。

3.4　结构设计先决条件

软件结构设计是较高级意义上的软件设计，它是支持详细设计的框架。结构也被称为“系统结构”、“设计”、“高水平设计”或者“顶层设计”。一般说来，结构体系往往在一个被称为“结构定义”或者“顶层设计”的单一文件中进行描述。

由于本书是关于创建活动的，因此这部分也没有讲述如何开发软件结构。本部分的中心是如何确定一个现存结构质量。因为结构设计比求定义更接近于创建活动，因此对于结构设计的描述要比要求定义详尽得多。

为什么要把结构设计当成先决条件呢？因为结构设计的质量决定了系统概念上的完整性，而这又会决定系统的最终质量。好的结构设计可能使创建工作变得很容易，而坏的结构设计则使创建活动几乎无法进行。

在创建活动中，对结构设计进行变动也是很昂贵的。一般来说，在创建阶段修复结构设计错误要比修复要求错误耗时少，但比修正编码错误耗时多得多。从这个意义上来说，结构变动与变动要求差不多，所以，无论是出干修正错误还是提高性能的动机，如果要进行结构变动的话，那么越早越好。

3.4.1　典型的结构要素

有许多要素是一个好的系统结构所共有的。如果你是一个人在独自开发一个系统，那么你的结构设计工作，或者说顶层设计工作，将与你的详细设计工作重叠。在这种情况下，至少你应该考虑每一个结构要素。如果你正在从事一项由别人进行结构设计的系统工作，你应该不费什么劲儿就能找到其中的重要部分。下面是一些在两种情况下都需要考虑的要素。

程序的组织形式

一个系统结构首先需要一个总体上的概括性描述。如果没有的话，从成千个细节与几十个独立模块中勾画出一幅完整的图画将是一件十分困难的事情。如果这个程序仅仅是一个由十二

块积木组成的小房子，那么或许连你那两岁的儿子也会认为这很容易。然而，对于一个由十二个模块组成的软件系统，事情恐怕就困难得多了。因为你很难把它们组合到一起，而如果不能把它们组合到一起，你就不会理解自己所开发的这一个模块对系统有什么贡献。

在结构设计中，你应该能找出最终组织形式的几种方案，并且应该知道为什么选中了现在这种组织形式。如果开发模块在系统中不被重视，会使人产生挫折感。通过描述这些组织形式的替代方案，我们就可以从结构设计中找出选择目前方案的原因，并已知道每一个模块的功能都仔细考虑过了。回顾设计实践发现，设计理由对于维护性来说，与设计本身是同样重要的（Rombach 1990）

在结构设计中，应该在程序中定义主要模块。在这里，“模块”并不是指子程序。在结构设计中通常不考虑建立模块一级的子程序。一个模块是一个能完成某一高级功能的子程序的组合，例如，对输出结果进行格式化，解释命令，从文件中读取数据等。在要求定义中列出的每一项功能，都应该有至少一个模块覆盖这项功能。如果一项功能由两个或更多的模块覆盖，那么它们之间应该是互补的而不是相互冲突。

每一个模块作什么应该明确定义。一个模块应该只完成一项任务而且圆满完成。对于与它相作用的其它模块情况，你知道得越少越好。通过尽可能地降低模块之间的了解程度，就可能把设计信息都集中在一个模块中。

每个模块之间的交界面也应该明确定义。结构设计应该规定可以直接调用哪些模块，哪些模块它不能调用。同时，结构设计也应该定义模块传送和从其它模块接收的数据。

变动策略

创建一个软件系统，对于程序员和用户来说，都是一个逐渐学习的过程，因此在这个过程中作出变动是不可避免的。变动产生的原因可能是由于反复无常的数据结构，也可能是由于文件格式和系统功能改变，新的性能等而引起的。这些变动有时是为了增加新的能力以便强化功能，也有时是版本增加而引起的。所以结构设计所面临的主要挑战便是增强系统的灵活性，以便容纳这类变动。

结构设计应该清晰地描述系统应付变动的策略。结构设计应该表明：设计中已经考虑到了可能的功能增强变动，而且，应该使最可能的变动同时世是最容易实现的变动。比如，假设最可能的变动是输入或者输出格式、用户界面的方式或者处理要求，那么结构设计就应表明已经预先考虑到了这些变动，而且，其中每一个单一的变动，只会涉及到数量有限的几个模块。在结构设计中应付变动的手段可能是非常简单的，比如在数据文件中加入版本号，保留一部分区域以备将来使用，或是设计一些可以添加内容的文件。

结构设计中应该说明用于延缓变动的策略。比如，结构设计中可能规定应使用表驱动技术而不是手工编码技术。它还可能规定表所使用的文件应该保存在一个外部文件中，而不是编码在程序中，这样，可以不必重新编译就可以对程序作出调整。

购买而不是建造的决定

创建一个软件的最彻底的办法并不是创建——而是去购买一个软件，你可以购买数据库管理系统、屏幕生成程序、报告生成程序和图形环境。在苹果公司 Macintosh 或者微软公司 Windows 环境下编程的一个主要优点是你可以自动获得许多功能；图形程序，对话框管理程序，

键盘与处理程序，可以自动与任何打印机或者监视器工作的代码，等等。

如果计划中要求使用已有的程序，那它就该指出如何使这些重新被使用的软件适应新的要求，而且它应该证明这个软件可以通过改动来满足新的要求。

Barry Boehm 在 1984 年指出：从长远观点来看，重新使用旧软件是提高生产率的首要因素。购买代码可以降低计划、详细设计、测试和调试的工作量。 Caper Jones 在 1986 年报告如果购买的代码从 0 上升到 50%，那么生产率可以提高一倍。

主要的数据结构

结构设计应该给出使用的主要文件、表和数据结构。同时，还应给出考虑的替代方案并评审作出的选择。在《Software Maintenance Guidebook》一书中，Glass 和 Noiseux 认为数据结构对系统维护有举足轻重的影响，因而，它应该在经过全盘考虑之后，才能选定（1981 年）。如果某一应用需要维护一个用户识别表，而结构设计又选中了顺序存取表来实现，那它就该解释为什么顺序存取表要好于随机存取表、推栈和杂凑表。在创建阶段，这些信息可以使你对结构设计有一个比较深刻的理解。在维护阶段，这些信息也是非常宝贵的。如果没有它们，你就会有一种看一部不带字幕的外国电影的感觉。

不应该允许一个以上的模块访问数据结构，除非是通过访问子程序，以使得这种访问是抽象的而且是可控的。这将在 6.2 “信息隐蔽”部分中详细论述。

如果一个程序使用了数据库，那么结构中应该规定这个数据库的组织形式和内容。

最后，应该遵循数据守恒定律：每一个进入的数据都应该出去，或者与其它数据一道出去，如果它不出去，那他就没有必要进来。

关键算法

如果结构设计依赖于某一特定算法，那它应该描述或指出这一算法。同主要数据结构一样，结构设计中也应该指出考虑过的算法方案，并指出选中最终方案的原因。比如，如果系统的主要部分是排序，而结构设计中又指定了排序方式是堆排序，那它就要说明为什么采用堆排序的方法，以及未采用快速排序或插入排序的理由。如果是在对数据作出某种假定的基础上才选中堆排序的，那就该给出这个假定。

主要对象

在面向对象的系统中，结构中应指出要实现的主要对象，它应该规定每一个对象的责任并指出每个对象之间是如何相互作用的。其中应包括对于排序层次、状态转换和对象一致性的描述。

结构中还应该指出考虑的其它对象，以及选择这种组织形式的原因。

通用功能

除了特定程序的特定功能，绝大多数程序中都需要几种在软件结构中占有一席之地的通用功能。

用户界面。有时用户界面在要求定义阶段便已经规定了。如果没有的话，那就应该在结构设计中作出规定。结构中应该定义命令结构，输入格式和菜单。用户界面的精心结构设计，往

往是一个深受欢迎的软件与被人弃之不用的软件间的主要不同之处。

这部分结构应该是模块化的，这样，当用新的界面代替旧的时，就不致影响到处理和输出部分。比如，这部分结构应该使得用批处理接口替代交互式界面的工作非常容易。这种能力是很有用的，特别是在单元测试和子系统测试阶段。

用户界面设计本身就值得写一部专著，但本书并未涉及这一内容。

输入／输出。输入／输出是结构中另一个应引起重视的部分。结构中应规定采用向前看、向后看还是当前规则的查询方式。同时，还应该指出在哪个层次上检查输入／输出错误，是在区域层次、记录层次还是在文件层次上。

内存管理。内存管理是结构设计中应该处理的另一个重要部分，结构中应该对正常和极端情况下所需要的内存作出估计。例如，如果你正在写数据表，那么结构就应估计其中每一个单元所需的内存。它还应估计正常表格和最大表格所需要的内存。在简单情形下，这种估计应表明内存在某项功能的实现环境中是正常的。在复杂情况下，可能不得不建立自己的内存管理系统，如果是这样，那么内存管理程序的设计应和系统其它部分一样，需要认真对待。

字符串存储。在交互式系统中，字符串存储也应在结构设计阶段予以重视。在这种系统中，往往包含了大量的提示、帮助信息和状态显示。应该估计被字符串所占用的内存。如果程序是商用的，那么，结构中应该考虑到典型的字符串问题，包括字符串的压缩，不必修改代码即可保持字符串，以及保证在译成外文时对代码的影响将是最小的。结构设计可以决定字符串的使用方法，是编码在程序中，还是把它保存在数据结构中。是需要时通过存取子程序调用，还是把它存在一个源文件中，结构设计应该指明采用哪种方法及其原因。

错误处理

错误处理已成为当代计算机科学中最棘手的问题，没有谁能担负起频繁应付它的负担。有人估计，程序中有 90％的代码是为了应付例外的错误处理或者内务处理而编写的，就是说仅有10％的代码才是处理正常情况的。既然有如此多的代码是用于错误处理，那么在结构中阐明处理错误的策略就是十分必要的了。以下是些需要考虑的问题：

- 错误处理是纠正还是仅仅测试错误？如果是纠正错误，程序可以尝试从错误状态下恢复。如果仅仅是测试，那么程序可以继续运行，就像什么也没有发生一样，或者直接退出运行。但无论在哪种情况下，都应该提醒用户发现了错误。
- 错误测试是主动的还是被动的？系统可以积极地预防错误，如通过检验用户的输入是否合法，当然也可以消极地在无法回避它们时才做出反应。例如，用户的一系列输入产生了溢出，你可以清除，也可以滤除信息。同样，无论哪种方案，都要提醒用户。
- 程序是怎样对付错误的？一旦测试出错误，程序可以立刻抛弃产生错误的数据，也可以把它当作错误而进入错误处理状态，还可以等到全部处理完毕后再通知用户数据有误。
- 处理错误信息的约定是什么呢？如果结构设计中没有规定某种策略。那么用户界面在程序的不同部分就会像迷宫中的通道一样忽东忽西，让人摸不着头脑。为避免出现这类问题，结构设计中应建立一套处理错误信息的约定。
- 在程序中，应该在哪一个层次上处理错误呢？你可以在发现的地方立即处理，也可以把它交给一个错误处理子程序去处理，或者交给更高层次的子程序处理。

- 每一个模块检验输入数据合法性的责任级别有多高？是每一模块仅检验它自己的数据，还是由一级模块来检验整个系统的数据？是否每个层次上的模块都可以假定输入其中的数据是合法的？

坚固性 （Robustness）

坚固性是指在发现错误后，一个系统继续运行的能力。在结构设计中需要从几个方面表述坚固性。

裕度设计（over－engineering）。在结构设计中应该明确表述所要求的系统裕度有多大。结构设计中规定的裕度往往比需求定义中规定的要大。一个原因是由于系统是由许多部分组成的，这会降低其总体坚固性。在软件链条中，其强度不是由最薄弱的一环决定的，而是由所有薄弱环节的乘积决定的。

清楚地表述所要求的裕度级是非常重要的，这是因为程序员出于职业素养，会不自觉地在程序中留出裕度。通过清楚地规定裕度级，可以避免某一部分裕度过大，而另一部分又过小的现象发生。

断言（assertions）。结构中还应该规定断言的使用程度。断言是指一段放在代码中，当代码运行时可以使其自检的可执行语句。如果断言显示出正确信息，那么表明一切都正常运行。如果显示出错误信息，那么表明它在程序中发现了错误。比如，系统假定用户信息文件永远不会超过 5000 记录行，那么程序中可能会包含一段说明这个假定的断言。只要这个文件不超过5000，那么断言就保持沉默，而一旦断言发现此文件超过了 5000 个记录行，那它就会声称已发现了一个错误。

为了在程序中加入断言，你必须知道在设计系统时所做的假设，这也是在结构设计中应阐明采用假设的原因之一。

容错性（fault tolerance）。结构设计应指明所期望的容错性类型，容错性是指通过测试错误、修正错误或在不能修复时容错等一系列方法，来提高系统可靠性的技术。

例如，一个可以采用如下办法来容忍求算术平方根时的错误。

- 系统可以返回并重新开始。如果发现结构有误，系统可以返回到正常的部分并重新开始。
- 当发现错误时，系统可以用辅助代码来代替基本代码。如果第一个结果看起来是错的，系统将使用另一个备用求平方根子程序重新计算一遍。
- 系统可以采取投票算法。可以用三种不同的方法算平方根，每一个子程序求一个平方根，由系统作出比较。根据系统所采用的容错种类，最终结果可能是三者的平均，其中的中间值就是占优势的那一个值。
- 系统可以用一个假想值来代替错误的结果，以避免对程序其余部分的不良影响。

其它的容错方式包括：在测试出错误后，只让系统部分运行或者系统功能降级，关闭自己或者自动重新开始等，这些例子是非常简单的。容错性是一个非常诱人而又复杂的学科，但它也不在本书讨论之列。

性能

如果考虑到性能，那么在性能要求中应该考虑性能目标。性能目标包括速度和内存使用。

结构设计要对这些目标作出估计，并解释为什么这些目标是可以达到的。如果某个域可能有达不到目标的危险，或者，如果某个域要求使用特定的算法或者数据结构来达到某一目标，在结构设计中也应指出这点。结构设计还应该规定每一个模块或目标的时间和存储空间预算。

通用的结构设计质量准则

一个好的结构设计特征包括；对于系统中模块的讨论，每个模块中隐含的信息，选用和不选用某方案的原因。

这个结构应该是一个近乎完美的整体概念。关于软件工程的最权威的著作<<The Mythical Man－Month>>，其中心思想便是认为概念完整性是最重要的（Brooks，1975）。一个好的结构设计应满足这一条，当看到这个结构设计时，应该为其解决方案的自然和简单而折服。而不会有把问题和答案生拼硬凑到一起的感觉。

你或许知道在开发过程中变动结构设计的途径。每一次变动都应与总体和设计概念相符。不能使最终完成的结构设计看起来像是一个穿着由各种碎布拼凑起来的百家衣的乞丐。

结构的目标应该清楚地说明。一个以可变性为首要目标的结构设计可能与一个以性能为首要目标的结构设计差之千里，虽然二者的功能可能是完全一样的。

结构中作出每一个决定的动机都要阐明清楚。要当心“我们过去一直是这么干的”的理由。有这样一个故事，会给我们启迪。Beth 想按照她丈夫的家传方法做一道红烧牛肉。她的丈夫 Abdul 告诉她，要先把牛肉放在盐和调料中胞一下，再剁掉肉的两边，把中间部分放进锅里，盖上盖儿焖一下就可以了。Beth 问：“为什么要剁掉肉的两边？”Abdul 说：“我不知道，我总是这样做的，我们问一下妈妈吧”。便打电话问妈妈、Abdul 的妈妈则说是他的外祖母告诉她的。于是电话打到了 Abdul 的外祖母那儿，她的外祖母奇怪地说：“我也不知道你们为什么那样做，我那样做不过是因为肉块太大，放不进锅里”。

好的软件结构往往是机器和语言相互独立。当然，我们不能忽略系统的实现环境。然而，通过尽量减少对实现环境的依赖性，你可以避免过分地结构化系统，并且使你可以在创建阶段把工作做得更好。但如果程序专门是为某一机型或语言设计的，那么本条不适合。

结构设计应该恰好在过分定义和定义不足的分界线上。结构中不应该有任何部分受到了它不应受的重视。设计者不能以牺牲某一部分为代价来重视另一部分。

最后，结构中不应该有任何部分让你感到不舒服。它不应该含有任何仅仅为取悦老板而加上去的部分。你是最终实现它的人，如果你根本读不懂它，又谈何实现呢？

3.4.2　检查表

结构设计

一个好的结构设计应该阐明所有问题。这个表并不是用于指导结构设计的，而只是想提供一种方法，通过它，你可以估计处于软件食物链顶层的程序员可以从食物中获得多少营养。它可以作为建立自己的检查表的起点。同要求定义检查表的使用一样，如果你正在从事一个非正式的项目，那么其中有些条款是不必考虑的。但如果你正在开发一个较大的系统，那绝大部分内容都是非常有用的。

- 软件的总体组织形式是否清晰明了？包括对于结构设计的总体评论与描述。
- 模块定义是否清楚？包括它们的功能及其与其它模块的接口。

- 要求定义中所提出的所有功能，是否有恰当数量的模块覆盖？
- 结构设计是否考虑了可能的更改？
- 是否包括了必要的购买？
- 是否阐明了如何改进重新启用的代码来满足现在的结构设计要求？
- 是否描述并验证了所有主要的数据结构？
- 主要数据结构是否隐含在存取子程序中？
- 规定数据库组织形式和其它内容了吗？
- 是否说明并验证所有关键算法？
- 是否说明验证所有主要目标？
- 说明处理用户输入的策略了吗？
- 说明并验证处理输入／输出的策略了吗？
- 是否定义了用户界面的关键方面？
- 用户界面是否进行了模块化，以使对它所作的改动不会影响程序其它部分
- 是否描述并验证了内存使用估算和内存管理？
- 是否对每一模块给出了存储空间和速度限制？
- 是否说明了字符串处理策略？是否提供了对字符串占用空间的估计？
- 所提供的错误处理策略是不是一致的？
- 是否对错误信息进行了成套化管理以提供一个整洁的用户界面？
- 是否指定了坚固性级别？
- 有没有哪一部分结构设计被过分定义或缺少定义了？它是否明确说明了；
- 是否明确提出了系统目标？
- 整个结构在概念上是否是一致的？
- 机器和使用实现的语言是否顶层设计依赖？
- 给出做出每个重要决定的动机了吗？
- 你作为系统实现者的程序员，对结构设计满意吗？

3.5　选择编程语言先决条件

实现系统的语言对你来说是有重大意义的，因为从创建工作开始到结束你都要沉浸其中。

研究表明，程序语言选择可以通过几方面影响生产率和编码质量。

当程序员使用自己所熟悉的语言时，其工作效率要比使用陌生的语言高得多。TRW 公司的数据表明，两个水平和经验相当的程序员如果一个用一种他已用了三年的语言编程，而另一个则用一种他所陌生的语言编程，那么前者的效率要比后者高 30％。IBM 的调查表明，一个在某种语言上经验丰富的程序员，其效率要比在这种语言上没什么经验的程序员高三倍（Walston 和 Felix 1977）。

使用高级语言编程，其效率和质量要比使用低级语言高得多。Pascal 和 Ada 语言的效率、可靠性、简单性和可懂性是低级语言，如汇编和机器语言的 5 倍（Brooks 1987）。由于不必每次都为机器正确地执行了指令而欢呼，你当然可以节省许多时间。同时，高级语言的表达能力

比低级语言要高，这样，它的每一行代码就可以表达更多的内容。表 3－2 给出了在代码量相同的信况下，高级语言所表达的原指令与低级语言的比值（以汇编语言为代表）。

表 3-2　高级语言指令与低级语言指令比

语言	比值
汇编语言	1：1
Ada	1：4.5
Quick / Turbo Basic	1：5
C	1：2.5
Fotran	1：3
Pascal	1：3.5

IBM 公司的数据从另一个方面指出了语言特性是如何影响效率的，用解释语言工作的程序员往往比用编译语言工作的程序员的效率更高（Jones 1986）。许多种语言都有解释和编译两种形式（如多种版本的 c 语言），你可以用高效率的解释形式，然后再把它们转换成更容易执行的编译形式。

一些语言比其它语言更擅长解释编程思想。你可以把自然语言（如英语）和程序语言（如 Pascal 和汇编语言）进行对比。在自然语言中，语言学家 Sapir 和 Whorf 提出的假想指出，在一种语言的表达能力和其所能思考的问题之间存在着联系，你思考某一问题的能力取决于你所懂得的关于这一问题的词汇。如果你不懂那些词汇，那你也就不能表达那些思想，你甚至根本无法形成那些思想。

程序员也可能同样受到他所懂得的程序语言限制。在某种程序语言方面你所懂得的词汇，当然会决定你如何表达你的编程想法，还很可能决定你将表达什么样的思想。

程序语言影响程序员的思想方法。一个典型的故事是这样说的:“我们正用 Pascal 语言开发一个新的系统，而我们的程序员们却并不熟悉 Pascal 语言，他们都是搞 Fortran 语言出身的。结果他们写出的是用 Pascal 编译的代码，但是他们真正使用的却是变形的 Fotran 语言。他们用 Fortran 的不好的特性（goto 语句和全局数据）歪曲了 Pascal 语言，而同时又把 Pascal 丰富的控制和数据结构弃之不用”。这种现象在整个软件业都有报道（Hanson 1984，Yourdon 1986）。

3.5.1 语言描述

某些语言的发展史同其通用功能一样令人感兴趣。以下是关于一些在本书中所举的例程中所出现的语言的描述。

Ada 语言

是一种在 Pascal 语言基础上发展的通用高级语言，它是在国防部的要求和资助下发展起来的，特别适用于实时和嵌入式系统。Ada 强调数据抽象和信息隐蔽，迫使你区分模块的公共和局部部分。

把这种语言命名为“Ada”是为了纪念数学家 Ada lovelace，他被公认为世界上的第一个程序员，从 1986 年起，北约组织和国防部的所有关键任务嵌入式系统都采用 Ada 语言。

汇编语言

汇编语言，是一种低级语言，每一条语句都与一条机器指令相对应。由于语句使用特定的机器指令，所以汇编语言是针对特定处理器的，比如 Intel 80x86 或者 Motorala 680x0。汇编是第二代计算机语言，除非是执行速度或代码空间的需要，绝大多数程序员都避免使用它。

Basic 语言

Basic 是由 Dartmouth 学院的 John Kemeny 和 Thormas Kurtz 开发的一种高级语言。由字首组成的 BASIC 的意思是初学者的全功能符号指令代码（Beginner’ s All-Purpos Symbolic Instruction Code），Basic 主要用于教学生们编程。由于 IBM－PC 机包含了它而使其在微机中风行一时，Basic 原来是一种解释性语言，现在则解释性和编译性两种形式都有。

C 语言

C 是一种中级通用语言，本来是和 UNIX 操作系统相关的。C 有某些高级语言的特点，例如，结构化数据、结构化控制流、对于机器的独立性、丰富的操作指令等。它也被称作“可移植的汇编语言”，因为它广泛地使用了指针和地址，具有某些低级组成部分，如位操作，而且是弱类型的。

C 是在七十年代由贝尔实验室 Dennis Ritchie 开发的。C 本来是为 DEC PDP－11 设计的，它的操作系统、C 编译器和 UNIX 应用程序都是用 C 编写的。1988 年，ANSI 公布了 C 的编码标准，这成了微机和工作站编程的通用标准。

C++语言

C＋＋，是一种面向对象的语言，与 C 相似，由贝尔实验室的 Bjarne stroustruP 于 1980 年开发，除了与 C 兼容之外，C＋十提供了多形性和函数名称过载功能，同时，它还提供了比 C 更坚固的类型检查功能。

Fortran 语言

Fortran 是一种高级语言，引入变量和高级循环的概念。Fortran 代表 Formula Translation，即公式翻译的意思。 Fortran 最初是在五十年代由 Jim Bckus 开发，并且做过几次重大修订．包括 1977 所发表的 Fotran－77，其中增加了块结构化的 IF－THEN－ELSE 语句和字符串操作。Fortran－90 增加由用户定义的数据类型、指针、模块和丰富的数组操作。在写本书的时候（1992 年末）。Fortran 标准是如此引发争议，以致绝大多数语言商都没能最终完成它。本书中所引用的是 Fortran－77 标准。Fortran 语言主要在科学和工程计算中使用。

Pascal 语言

Pascal 是为了教学目的而开发的高级语言。其主要特征是严格的类型、结构化控制创建和结构化数据类型。它是在六十年代末由 Niklaus Wirth 开发，到了 1984 年，由于 Borland 国际公司引入了微机使用的低成本编译程序，Pascal 就流行起来了。

3.5.2　语言选择快速参考表

表 3－3 给出了关于不同语言适用范围的简略参考。它也可以帮你选择应该进一步了解的语言。但是，不要试图用它来代替对你某一特定计划进行语言选择时的详细评估。以下的排序是很粗略的，因此阅读时应仔细辨别，因为很可能会有许多例外。

表 3-3　适于不同种类程序的最差和最好语言

程序类型	最好语言	最差语言
结构化数据	Ada、 C++、 Pascal	汇编、Basic
快速而杂乱的项目	Basic	Pascal、Ada、 汇编
快速执行	汇编、C	解释性语言如 Basic
数学计算	Fortran	Pascal
易于维护的程序	Pascal 、Ada	C 、Fortran
动态内存使用	Pascal、C	Basic
在有限内存环境下运行	Basic、汇编、C	Fortran
实时程序	Ada、汇编、C	Basic 、Fortran
串操作	Basic 、Pascal	C

3.6　编程约定

在高质量软件中，你可以发现结构设计的概念完整性与较低层次实现之间的密切联系。这种联系必须与指导它的结构设计保持一致，而且，这种一致应该是内在的。这就是实现时在给变量和子程序命名、进行格式约定和注释约定时的指导方针。

在复杂的软件中，结构设计指导方针对程序进行结构性平衡，而实现指导方式则在较低层次上实现程序的和谐统一，使得每一个子程序都成为总体设计的一个可以信赖的组成部分。任何一个大的软件系统都需要结构控制，以便把编程语言的细节统一到一起。大型系统的完美之处便是它的每一个细节都体现了它的结构设计风格。如果没有一个统一约束，那么你的软件只能是一个由各种风格不同的子程序拼凑到一起的拼盘而已。

即使你有一个关于一幅画的美妙总体构思，但如果其中一部分是用古典手法的，另一部分是印象派的，其余则是超现实主义风格的，那么，再美妙的构思又有什么用呢？不论其中每一部分是如何密切联系主题的，这幅画的概念完整性都将荡然无存。同样，程序也需要较低层次上的完整性。

在创建工作开始之前，一定要写明你将要采用的编程约定、约定说明一定要写得非常详尽，使得在编程过程中无法对其进行改动。本书提供了许多非常详细的约定。

3.7　应花在先决条件上的时间

用于问题定义、需求分析和软件结构设计的时间，随项目需要的不同而不同。一般来说，一个运行良好的项目通常把 20～30％的时间用于先决条件。这 20～30％的时间中不包括进行详细设计的时间，因为它是创建活动的一部分。

如果你正从事一个正式项目，而要求又是不稳定的，那么，你将不得不与需求分析员一道解决要求定义问题，拿出你的一部分时间与需求分析员讨论，并给需求分析员一定时间以便让他重新征求用户意见，可以使要求定义更适合项目需要。

如果你从事的是一个非正式项目，而要求是不稳定的，应该给需求分析留出足够的时间，以免反复无常的要求定义影响你的创建工作。

如果要求对于任何项目——不管是正式还是非正式的，都是不稳定的，那你就该亲自从事需求分析工作。当完成需求分析后，再估计从事项目其余部分所需要的时间。这是一个很明智的办法，因为在你知道自己将作些什么之前，你是不可能知道需要多长时间来完成它的。打个比方，假设你是一个建筑承包商，你的顾客问："这项工程要花多少钱？"你则问他要干些什么，而他却接着说："我不能告诉你，我只想知道工程要花多少钱?"这时你最好对他说声谢谢，然后吹着口哨回家吧。

在建筑中，在知道要建什么之前，就进行工程预算显然是荒谬的。在设计师完成草图之前，老板是不会问要用多少水泥、钉子和木材的。但人们对于软件开发的理解往往不是如此清楚的，所以你的老板可能一时还弄不明白为什么要把需求分析当作一个单独的项目，这时你就需要作出解释。

3.8　改变先决条件以适应你的项目

先决条件随项目规模和正式性不同而变化。本章指出了大规模和小型项目之间先决条件的判别，可以根据项目的特点对先决条件作出合适的调整。要想详细了解大项目与小项目之间的不同，请参看第 21 章"程序规模是如何影响创建活动的"。

3.9　小 结

- 如果想开发一个高质量的软件，必须自始至终重视质量问题。在开始阶段强调质量往往比在最后强调质量更为有效。
- 程序员的份内工作之一便是向老板和同事宣传软件的开发过程，包括在编程开始前从事先决条件准备工作的重要性。
- 如果问题定义工作做得不好，那么在创建阶段，所解决的问题可能并不是用户真正要解决的问题。
- 如果需求分析工作做得不好，很可能因此而漏掉要解决问题中的重要细节。在创建工作后更改要求，要比在需求分析阶段进行更改的成本高 20 到 100 倍。所以，在开始编程前一定要确认要求定义工作一切正常。
- 在编程前规定好约定，在创建工作结束后再改变代码来满足约定几乎是不可能的。
- 在创建活动开始之前如果无法完成准备工作，可以尝试在不太稳固的基础上进行创建活动。

第四章　建立子程序的步骤

目录

相关章节

高质量程序的特点：见第 5 章
高层次设计；见第 7 章
注释方式：见第 19 章
创建工作先决条件：见第 3 章

本章详细讲述了在建立一个子程序时的典型步骤。虽然从广义上讲，你可以把本书所有的描述都当作是在讲如何建立程序，但本章把这些步骤放在同一背景下讲述。本章的中心内容是如何编写小规模的程序，以及编写对各种规模项目都十分关键的程序的特定步骤。本章也描述了从程序设计语言（PDL）到编码的转换过程，几乎没有哪些程序员充分利用了这一过程所带来的方便，这部分论述会给大家以启迪。

4.1　建立程序步骤概述

在建立程序过程中引入的许多低层次细节问题，并不需要按某一特点顺序来进行，但是一些主要活动——设计程序、检查程序、子程序编码、检查代码，则应该按图 41 的顺序来进行。

4.2　程序设计语言（PDL）

PDL（程序设计语言）是由 Came，Father 和 Gordon 共同开发的，在 1975 年发表之后．曾作过重大修改。因为 PDL 是在模仿英语，所以认为任何像是英语的 PDL，都可以正确表达思想是很自然的。但是，事实上 PDL 之间的好坏是有判别的。下面是有效使用 PDL 的一些方针：

- 用模拟英语的语句来精确描述每一个特定操作。
- 避免使用最终程序语言的语句。PDL 使你比在代码稍高级的层次上进行设计工作。当使用程序语言进行创建时，就又回到了低层次上，从而得不到由于在高层次上进行设计的好处，而且会受到不必要的程序语言语法规则的限制。
- 在设计意向这一层次上写 PDL。说明方法的意义，而不是描述如何用目标语言实现。

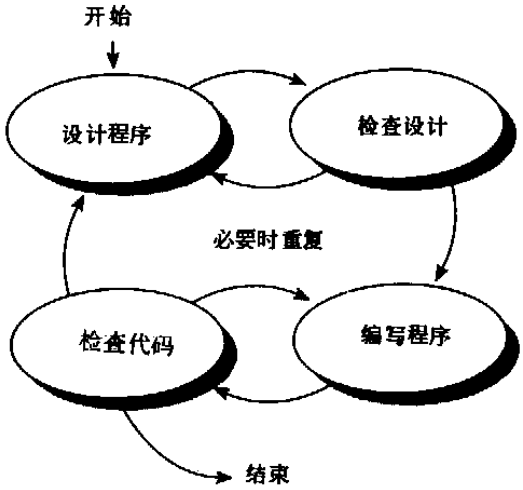


图 4-1　创建子程序过程中主要活动顺序示意图

- 在足够低的层次上写出 PDL，它几乎可以自动生成代码。如果 PDL 写得太简略，可能会在编码过程中忽略问题细节。应该精确地使用 PDL 以方便编码。

当 PDL 写好之后，就可以根据它来编码，而 PDL 则成为程序语言的注释。这可以省去大量的注释工作。如果 PDL 遵循了这一指导方针，那么注释将是非常完备而且富有意义的。

以下则是一个几乎违背了上述所有原则的错误使用 PDL 的例子:

```
Increment resource number by l
allocate a dlg struct using malloc
if malloc() returns NULL then return l
invoke OSrsrc _init to initialize a resource for the operation system
* hRstcPtr＝resource number
return 0
```

这个 PDL 的意图是什么？由于它写得很糟糕，因此很难说清楚。之所以称之为一个错误使用 PDL 的典型，是为它使用了像*hRSrcPtr 这种特定的 c 语言指针标志和 malloc( )这个特定的语言函数，即它采用了代码语句。这段 PDL 的中心是如何写代码，而不是说明设计意义。不管子程序返回 1 还是返回 0，这段 PDL 都引入了代码细节。如果从是否变为一个好的注释的观点来看这段 PDL，你就会发现它毫无意义。

以下是对同一个操作的设计，使用的是大大改进了的 PDL：

```
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
```

Reture TRUE if a new resource was created; else return FALSE

这段 PDL 要好于前一个。因为它完全是用自然语言写成的，没有使用任何目标程序语言语句。在第一段 PDL 中，它只能用 C 语言来实现，而第二段却并没有限制所使用的语言。同时，第二段 PDL 也是在意图层次上写成的。第二段 PDL 的意图是什么？其意图理解起来比前一个要容易多了。

尽管第二段 PDL 是完全用自然语言写成的，但它却是非常详细和精确的，很容易作为用程序语言编码的基础。如果把这段 PDL 转为注释段，那它则可以非常明了地解释代码的意图。

以下是你使用这种风格的 PDL 可以获得的益处：

- PDL 可以使评审工作变得更容易。不必检查源代码就可以评审详细设计。它可以使详细评审变得很容易，并且减少了评审代码本身的工作。
- PDL 可以帮助实现逐步细化的思想。从结构设计工作开始，再把结构设计细化为 PDL，最后把 PDL 细化为源代码。这种逐步细化的方法，可以在每次细化之前都检查设计，从而可以在每个层次上都可以发现当前层次的错误，从而避免名影响下一层次的工作。
- PDL 使得变动工作变得很容易。几行 PDL 改起来要比一整页代码容易得多。你是愿意在蓝图上改一条线还是在房屋中拆掉一堵墙？在软件开发中差异可能不是这样明显，但是，在产品最容易改动的阶段进行修改，这条原则是相同的。项目成功的关键就是在投资最少时找出错误，以降低改措成本。而在 PDL 阶段的投资就比进行完编码、测试、调试的阶段要低得多，所以尽早发现错误是很明智的。
- PDL 极大地减少了注释工作量。在典型的编码流程中，先写好代码，然后再加注释。而在 PDL 到代码的编码流程中，PDL 本身就是注释，而我们知道，从代码到注释的花费要比从注释到代码高得多。
- PDL 比其它形式的设计文件容易维护。如果使用其它方式，设计与编码是分隔的，假如其中一个有变化，那么两者就毫不相关了。在从 PDL 到代码的流程中，PDL 语句则是代码的注释，只要直接维护注释，那么关于设计的 PDL 文件就是精确的。

作为一种详细设计工具，PDL 是无可比拟的。程序员们往往愿意用 PDL 而不愿使用缺陷表。事实上程序员们愿意使用缺陷表以外的任何工具，调查表明，程序员们愿意使用 PDL，是因为它很容易用程序语言实现，而且 PDL 可以帮助发现详细设计中的缺陷，并且 PDL 也很容易写成文件，改动也很方便，PDL 并不是详细设计的唯一工具，但是 PDL 和 PDL 到代码流程的确是有用的工具。不妨试一下。在随后的几部分中，我们将告诉你如何使用它们。

4.3　设计子程序

创建一个子程序的第一步是设计。假设想设计一个根据错误代码输出错误信息的子程序，并且把这个子程序称为 RecordErrorMessge()，以下是关于 RecordErrorMessage()的要求定义：

RecordErrorMessage()的输入变元是非法代码，输出是与这个非法代码相对应的错误信息，它负责处理非法代码。如果程序运算方式是交互式，那么这个错误信息就打印给用户。如果运行方式是批处理式的，那么这个信息就送入一个信息文件。在输出信息后，这个子程序应该能返回到一种状态，指出程序是否成功。

在本章的其余部分，用这个子程序作为一个实际例子。这一部分的其余内容将论述如何设计这个子程序，设计这个子程序所需要进行的活动见图 4-2。

检查先决条件。在进行与子程序有关的任何工作之前，首先检查是否定义了这个子程序的工作任务，这项任务是否和整个结构设计融为一体？通过检查确定是否这个子程序被调用了？至少，在项目的要求定义中就涉及到它。

定义这个子程序将要解决的问题。应该足够详尽地规定它需要解决的问题，以便于创建。如果结构设计是非常详尽的，那么这项工作可能已经完成了，结构设计应该至少指出以下这些问题：

- 这个子程序将要隐含的信息。
- 这个子程序的输入。
- 这个子程序的输出，包括受到影响的全局变量。
- 这个子程序将如何处理错误？

图 4-2　设计程序中的所有实现步骤

下面是在 RecordErrorMessage()这个子程序中，上述考虑是如何得以阐明的。这个子程序隐含了如下两个事实；错误信息与现存的处理方式（交互式或者批处理），子程序的输入是非法代码，要求两种输出方式：第一是错误信息；第二是 RecordErrorMessass()子程序返回到调用它的程序时的状态。

问题说明之后，并没有直接给出解决方案。假设以这个例子来说，程序约定是在发现错误时立即报告。在这种情况下，这个子程序必须报告它所发现的每一个错误，假定其它错误都已经报告过了。根据要求，这时子程序应把状态变量设置为失败。

给子程序命名。给子程序命名似乎是小事一桩，但好的子程序名字往往是一个高质量软件的标志之一，而且，命名并不是件容易的事情。一般来说，子程序应该有一清楚的、不容易引起异义的名字。如果在给程序找一个好名字时感到困难，这往往意味着对程序的功能还不十分清楚。一个模棱两可的名字就像是一个在进行竞选辩论的政治家，似乎他在说着什么，可是当

你仔细听时，又分辨不出他的话到底有什么意义、应尽量将名字起得清楚。如果产生一个模棱两可名字的原因是模棱两可的结构设计，那么就应注意这个危险信号，这时应追回去改进结构设计。

在这个例子中，RecordErrorMessage()的含义是很清楚的，因此是个好名字。

决定如何测试子程序。在编写于程序时，最好能同时考虑如何测试。这对进行单元测试工作是很有益处的。

在这个例子中，输入很简单，就是错误代码。因此，可以计划用全部有效错误代码和一系列无效代码来进行测试。

考虑效率。根据所处的情形，你可以用一种或两种方式来说明效率。

在第一种情形下，程序的绝大部分，性能并不是主要的，在这种情况下，应该把子程序作成高度模块化而且具有很强的可读性，以便在今后需要时很容易对其作出改进。如果其模块化程度很高，就可以在需要时，用更好的算法或者汇编语言编写的子程序来代替速度较慢的程序而不致影响程序其它部分。

在第二种情形下，在大部分程序中，性能都是很重要的，这时，结构设计应该对子程序的运行速度和允许使用的内存作出规定，只要按照速度和空间指标设计子程序就可以了。如果速度和空间只有一方面是主要的，则可以牺牲一方面来满足另一方面的要求。在初始创建阶段，对子程序作出足够调整以使它满足速度和空间要求是合理的。

除了以上指明的情况以外，不必浪费精力去考虑个别子程序的执行效率。优化的收益主要来自高层次设计，而不是个别子程序、只有在高层次设计某方面有缺陷时，才需要进行微观优化，而这点只在程序全部完成时才会知道。除非必要，不要浪费时间进行增量改进。

研究算法和数据结构。同时提高编码质量和效率的最有效办法是重新使用好的代码。在学术文章中，已经有许多种算法被发明、讨论、检验和改进过。因此，与其花费时间去发明一种别人已经为之写过博士学位论文的东西，倒不如花几分钟测览一个算法论著，看有多少种算法可供选择。如果想使用某种已有的算法，切记要对其做出改进以适应你的程序语言。

编写 PDL。在做完上述工作之后，编写的时间可能已经不多了。本步骤的主要目的是，建立一种可以在实际编写子程序时提供思想指导的文件。

在主要工作步骤完成之后，可以在高层次 PDL 水平上编写子程序。可以使用编辑程序或者整体环境来编写 PDL，很快，这些 PDL 就将成为用程序语言编码的基础。

编写工作应该从抽象到具体。一个子程序最抽象的部分便是最开始的注释部分，这部分将说明要求于程序作什么；因此，首先应该写一个关于编写于程序目的的精确说明。编写这个说明也将帮你更清楚地理解这个子程序。如果在编写这部分说明时感到困难，那说明需要对这个子程序在整个软件中的地位和作用作出更深刻的理解。总之，如果感到编写抽象说明时有困难，应该想到可能是某一环节出了问题。以下是一个精确说明子程序作用的例子：

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

写完抽象说明后，再编写关于这个子程序的高层次 PDL。下面就是一个关于前述例子的高层次 PDL：

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

set the default status
look up the message based on the error code
if the error code is valid
    determine the processing method
    if doing interactive processing
       print the error message interactively and declare success
    else doing batch processing
       if the batch message file opens properly
            log the error message to the batch file，
               close the file，and declare success
else the message code is not valid
    notify the user that an interal error has been detected
```

应该注意的是这个 PDL 是在一个相当高的层次上写成的。它使用的不是程序语言，而是用自然语言来精确表达设计思想的。

考虑数据。可以在编写过程中的几个不同地方设计数据。在这个例子中，数据非常简单，因而数据操作并不是程序的主要部分。如果数据操作是程序的主要部分，那么在考虑程序的逻辑结构之前，考虑主要数据是必要的。如果在进行子程序的逻辑设计时，已经有了关键数据结构的定义，那将是大有梅益的。

检查 PDL。写好 PDL 并设计完数据之后，应该花一点时间来检查一下 PDL。返回来看着所写的 PDL，考虑一下应该怎样向别人说明。

请别人帮助看一下或听一下你的说明。也许你认为请别人看一个只有 11 行的 PDL 是很愚蠢的，但你会对这样作的结果感到惊奇。PDL 使假设和高层次错误比程序语言代码容易被发现。人们往往更愿意检查一个只有几行的 PDL，而不愿去检查一个有 35 行的 C 或 Pascal 子程序。

要确认对子程序做什么和将怎样做已经有了清楚透彻的了解。如果在 PDL 这一层次上对这点还没有概念上的了解，那么在编码阶段了解它的机会还有多少呢？如果连你都理解不了它的话，又有谁会理解呢？

逐步细化。在开始编码之前，要尽可能多使用 PDL 尝试一些想法。一旦开始编码，就会对所写的代码产生爱惜之情，这时，要再想把它扔掉重新开始是非常困难的。

通常的思想是逐步细化用 PDL 写成的子程序，直到可以在每行 PDL 语句下面添加一行代码而成为子程序为止，并把原来的 PDL 当作注释文件，或许最初的 PDL 的某些部分过于简略，需要进一步说明，那么切记一定要对其作出进一步说明。如果不能确认如何对某一部分编码，那么就继续细化 PDL，直到可以确认为止。要不断地细化 PDL 并对其作出进一步说明，直到你看到这样作是在浪费时间时，再开始实际的编码工作。

4.4　子程序编码

设计好子程序之后，就要开始实现。可以按照标准步骤实现，也可以根据需要作出改动。图 4-3 给出了实现一个子程序时的步骤。

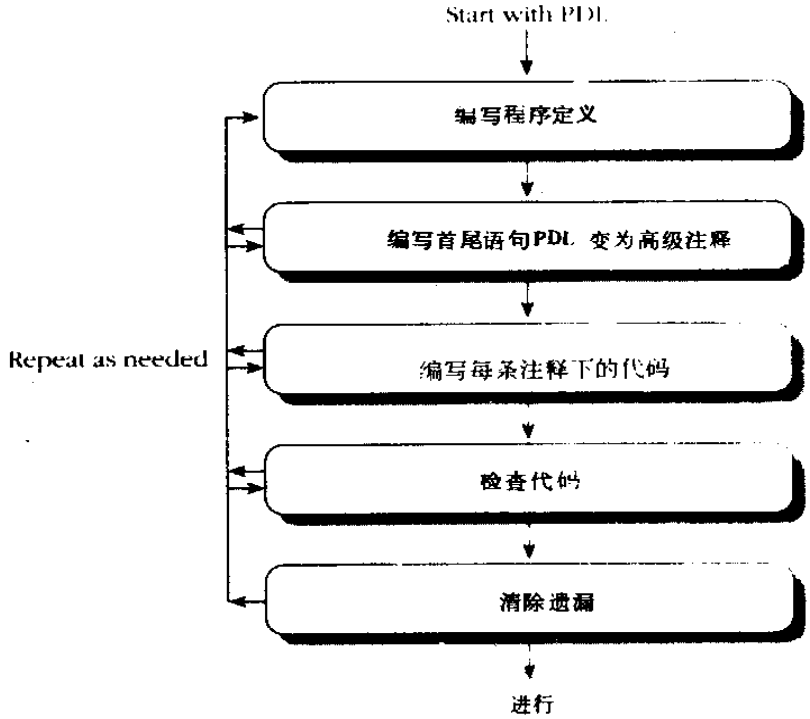


图 4-3 实现子程序的步骤

书写子程序说明。编写子程序的接口语句——编写过程或函数说明，应采用所需要的语言，无论 Pascal、C 或 Fortran 都可以，只要符合需要。把原来的抽象说明用程序语言实现，把它放在原来写好的 PDL 位置之上。以下是前述子程序的接口语句和抽象说明，它是用 Pascal 写成的：

```
procedure RecordErrorMessage
(
ErrorCode:ErrorCode_t ;
var Status:Status_t
);
```

这些是接口语句

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

已转化成 Pascal 风格注释的标题注释

```
  set the default staus
  look up the message based on the error code
```

```
if the error code is valid
    determine the processing method
      if doing interactive processing
      print the error message interactively and declare success
    else doing batch processing
      if the batch message file opens properly
          log the error message to the batch file,
          close the file, and declare success
else the message code is not valid
    notify the user that an internal error has been detected
```

这时是指出接口假设的好时机。在本例中，接口变量 ErrorCode 和 Status 是简明的，并且根据其特定用途排序，不含任何隐蔽信息。

把 PDL 转变成高层次注释。利用 Pascal 中的 Begin 和 End，或者 C 中的“{”和“}”，可以把 PDL 变成注释，以下是把前述的 PDL 变成了 Pascal 语言：

```
procedure RecordErrorMessage
    (
    Errorcode:ErrorCode_t;
      var Status:Status_t
    );

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

begin
    {   set the default status }
    {   look up the message based on the error code }
      {   if the error code is valid〕
      {   determine the processing method }
      {   if doing interactive processing}
        {   print the error message interactively and declare success}
      { else doing batch processing }
        {   if the batch message file opens properly}
          {log the error message to the batch file,
              close the file.   and declare success}
    {else the message code is not valid}
      { notify the user that an internal error has been detected}
end;   { RecordErrorMessage() }
```

这时，子程序的特点已经非常明显了，设计工作已经结束了，没看见任何代码，但已经知道子程序如何工作了。把 PDL 转换成程序语言代码是一件机械、自然、容易的工作。如果你不

觉得是这样，那么还需要进一步细化 PDL，直到有这种感觉为止。

在每一行注释下面填上代码。在每一行 PDL 注释语句下面填上代码。这有点像给报纸排版。首先画好轮廓线，然后再把每一篇文章填到空格中，每一个 PDL 注释行都相当于给代码画的轮廓线，而代码相当于文章。同文学文章的长度一样，代码的长度也是由它所要表达的内容多少决定的。程序的质量则取决于它的设计思想的侧重点和巧妙程度。

在本例中，前两行 PDL 注释产生了两行代码：

```
Procedure RecordErrorMessage
    (
    ErrorCode：ErrorCode_t;
    var status:Status_t
    ) ;

{ This routine outputs an error message based on an error code
  Supplied by the calling routine The way it outputs the message
  depends on the current processing state，which it retrieves
  on its own.    It returns a variable indicating success or failure. }
begin
{   Set the default status }
Status:=Failure;

{   look up the message based on the error code}
LookupErrorMessage(ErrorCode,ErrorMessage);

    {   if the error code is valid }
        { determine the processing method }
        { if doing interactive processing }
          { Print the error message interactively and declare success }
        {   else doing batch processing }
          { if the batch message file opens properly }
              { log the error message to the batch file,
                close the file，and declare success }
        {   else the message code is not valid }
          { notify the user that an internal error has been detected 〉
  end;  { RecordErrorMessage() }
```

这里是填充的代码

这里是新变量 ErrorMessage

这是一个编码的开始，使用了变量 Errormessage，所以需要说明它。如果是在事后进行注释，那么，用两个注释行来注释两行代码就不必要了。但是，采用目前这种方法，是注释的字面内容而不是它注释了多少行代码。现在，注释行已经存在了，所以还是将其保留。

代码需要变量说明，而且在每一注释行下面都要加人代码，以下是完整的子程序：

```
procedure RecordErrorMessage
    (
```

```
ErrorCode:ErrorCode_t;
var Status:Status_t
);

{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, whict it retrieves
on its own. It returns a variable indicating success or failure. }

var
    ProcessingMehod:  ProcessingMethod_t;
    ErrorMessage:     Message_t;
    FileStatus:       Status_t;
begin
    {set the default status}
    Status:=Failure;

    {look up the message based on the error code }
    LookupErrorMessage(ErrorCode,ErrorMessage);

    {if the error code is valid}
    if (ErrorMessage.ValidCode)   then begin

        {determine the processing method}
        ProcessingMethod := CurrentProcessingMehod;

        {if doing interaction processing}
        if (ProcessingMethod = Interactive)   then begin

            {print the error message interactively and declare success }
            PrintInteractiveMessage(ErrorMessage.Text);
            Status := Success
        end
        {else doing batch processing}
        else if (ProcessingMethod = Batch)   then begin

            {if the batch message file opens properly}
            FileStatus := OpenMessageFile;
            If (FileStatus = Success)   then begin

                {log the error message to the batch file,close the file,
                and declare success}
```

这里是变量声明

```
                LogBatchMessage ( ErrorMessage.Text ) ;
                CloseMessageFile;
                Status := Success
              end { if }
            end { else }
        end

        { else the message code is not valid }
        else begin

          { notify the user that an interanl error has been detected }
          PrintlnteractiveMessage ( 'Internal Error; Invalid error code' ,
                          'in RecordErrorMessage()' }
        end

    end;   { RecordErrorMessage () }
```

每一个注释都产生了一行或一行以上的代码，每一块都在注释的基础上形成了一个完整的思想。保留了注释以便提供一个关于代码的高层次解释，在子程序的开始，对使用的所有变量都作了说明。

现在，让我们再回过头来看一下前面的关于这个例子的要求定义和最初的 PDL，从最初的 5 行要求定义到 12 行的初始 PDL，接着这段 PDL 又扩大成为一个较大的子程序。即使要求定义是很详尽的。子程序的创建还是需要在 PDL 和编码阶段进行潜在的设计工作。这种低层次的设计工作正是为什么编码不是一件琐碎事的原因，同时，也说明本书的内容是很重要的。

非正式地检查代码。在注释下面填上代码之后，可以对每一块代码作一简略检查。尽力想一下什么因素可能破坏目前的块，然后证明这种情况不会发生。

一旦完成了对某一子程序的实现，停下来检查一下是否有误。在 PDL 阶段，就已经对其作了检查。但是，在某些情况下，某些重大问题在子程序实现之前是不会出现的。

使得问题直到编码阶段才出现的原因是多方面的。在 PDL 阶段引入错误可能到了详尽的实现阶段才会变得明显。一个在 PDL 阶段看起来完美无缺的设计，在用程序语言实现时可能会变得一塌糊涂。在详尽的实现阶段，可能会发现在结构设计或功能分析阶段引入的错误，最后，代码可能存在一种司空见惯的错误——混用语言，毕竟大家都不是尽善尽美的嘛。由于上述原因，在继续工作之前，要检查一下代码。

进行收尾工作。检查完代码是否存在问题后，再检查一下它是否满足本书所提到的通用质量标准。可以采取几个步骤来确认子程序的质量是否满足要求。

- 检查子程序的接口。确认所有的输入和输出数据都已作出了解释，并且使用了所有参数。关于细节问题，见 5.7 节“怎样使用子程序参数”。
- 检查通用设计质量。确认子程序只完成一项任务而且完成得很好，与其它于程序交叉是控制不严的表现。并且，应该采用了预防错误的设计。关于细节问题，见第五章“高质量程序的特点”。
- 检查子程序的数据。查找出不精确的变量名、没有使用的数据、没有说明的数据等

等。要了解详情，见关于数据使用的第八到第十二章。

- 检查子程序的控制结构。查找无限循环、不适当的嵌套等错误。详见关于使用控制结构的第 13 到 17 章。
- 检查子程序设计。确认已说明了子程序的表达式、参数表和逻辑结构。详见第 18 章“设计与风格”。
- 检查子程序的文档。确认被翻译成注释的 PDL 仍然是精确的。检查算法描述，查找接口假设和非显式依赖的文件资料，查找不清楚的编码等等。详见第 19 章“自我证明的代码”。

按需要重复步骤。如果程序的质量很差，请返回 PDL 阶段。高质量程序是一个逐步的过程，所以在重新进行设计和实现活动时，不要犹豫不决。

4.5　检查子程序

在设计并实现了子程序之后，创建活动的第三个主要步骤是进行检查，以确认所实现的软件是正确的。你或许会问，对代码进行的非正式检查和收尾工作难道不足以保证其正确性吗？的确，这两项工作会从某种程度上保证代码正确性，但不能完全保证，在这一阶段工作中漏掉的错误，只有在后面的测试工作中才会被发现。而到那时纠正它们的成本将变得很高，因此，最好现在就进行查错工作。

在心里对子程序进行查错处理。在前面提到过的非正式检查和清扫工作就是两种内心检查方法。另一方法是在心中执行每一个路径。在心中执行一个子程序是比较困难的，这个困难也是造成很困难保持子程序小规模的原因之一。要保证检查到每一个规定路径和中止点，同时还要检查所有的例外情况。可以自己进行这项工作，此时叫作“桌面检查”。也可以与同事们一道检查，这叫作“同事评审”、“过一遍”或者“视察”，这要取决于如何进行这项工作。

业余爱好者与职业程序员之间的最大区别就是迷信还是理解。在这里，“迷信”这个词指并不是指在月圆之夜产生各种错误或使你毛骨悚然的一段程序。它指的是你对代码的感觉代替对代码的理解。如果你总是认为编译程序或者硬件系统有故障，那说明你还处在迷信阶段。只有 5% 的错误是由编译程序、硬件或者是操作系统引起的（Brown and sampson, 1973, Ostrand and Weyuher, 1984）。进入理解境界的程序员总是怀疑自己的工作，因为他们知道 95％ 的错误出自这里。要理解每一行编码的意义，并且要明白为什么需要它。没有仅仅因为有效便是正确的东西。如果你不知道为什么它是有效的，那么往往它是无效的，只不过你没有发现罢了。

最后，要指出的是有了一个有效的子程序并非就完事大吉了。如果你不知道为什么它是有效的，那就研究并讨论它，或者用替代方案重新实现一次，直到你弄明白为止。

编译子程序。如果检查完了子程序，那就开始编译它。直到现在才开始编译工作，似乎是我们的工作效率太低了，因为早在几页之前程序就完成了。的确，如果早一些开始编译，让计算机去检查没有声明的变量，命名冲突等是可能会节约一些时间。

但是，如果晚一些开始编译，将会获得许多收益。其中的一个主要原因是，一旦开始编译，那么你脑袋里的秒表便开始嘀嗒作响了，在第一次编译之后，你就开始不停地想：下次编译一定让它全对。结果，在这种“就只再编译一次”的压力下，作了许多匆忙的、更易产生错误的修改，反而浪费了更多的时间。所以，在确信子程序是正确的之前，不要急于开始编译。

本书的重点之一，就是想告诉读者如何避免陷入把各种代码拼凑到一起，通过试运行检验它是否有效的怪圈。而在确信程序是正确的之前，就匆忙开始编译，恰恰是陷入了这种怪圈。如果你还没有进入这个怪圈，那最好还是当确信程序正确之后再开始编译。

以下是在编译时，尽可能地检查出全部错误的指导方针：

- 尽可能把编译程序的警告级别调到最高。只要允许，编译程序应尽量测试，将发现许多难以察觉的错误。
- 消除所有编译程序指出的错误和提出警告的原因。注意编译程序关于你的代码说了些什么。大量的警告往往意味着代码质量不高，所以应该尽量理解所得到的每个警 告。在实际中，反复出现的警告可能产生以下影响：你忽略掉它们，而事实上它们掩盖了更严重的错误。或者它们会变得使人痛苦，就像日本式的灌水酷刑。因此，比较安全或痛苦较小的办法，是消灭这些隐蔽的问题以消除警告。

使用计算机来检查子程序错误。子程序编译之后，将其放入调试程序，逐步运行每一行代码，要保证每一行都是按预期的运行。用这种简单的办法，你可以发现许多错误。

在调试程序中逐步运行程序之后，用在开发子程序时设计的测试用例对其进行测试。或许你将不得不搭些支架来支撑你的子程序——即那些仅在测试阶段用于支持子程序而最终不包括在产品中的代码。这些代码可能是调用子程序的程序，也可能是被子程序所调用的子程序。

消除子程序中的错误。一旦发现有错误，就要消除它。如果开发的子程序此时问题较多，那它将在这一阶段耗费较长的时间。如果发现程序中的错误异乎寻常的多，那就重新开发一个，不要试图修补它。修补往往意味着不充分的理解，而且肯定会在现在和将来产生更多的错误，而进行一个全新的设计将防止这一点。恐怕没有比重新写一个完美无缺的子程序来代替一个漏洞百出的子程序更能让人满意的事了。

4.5.l　检查表

创建子程序

- 是否检查过先决条件已经满足了？
- 定义子程序将要解决的问题了吗？
- 结构设计是否足够清楚，使得你可以给子程序起个好名字？
- 考虑过如何测试子程序了吗？
- 是否从模块化水平或者满足时间和内存要求角度考虑过效率问题？
- 是否查阅过参考书；以寻找有帮助的算法？
- 是否用详尽的 PDL 设计子程序？
- 在必要时，是否在逻辑设计步骤前考虑了数据？
- 是否检查过 PDL，它很容易理解吗？
- 是否注意到了足以使你返回到结构设计阶段的警告（使用了全局数据，更适合其它子程序的操作，等等）。
- 是否使用了 PDL 到代码流程，是否把 PDL 作为编码基础并把原有的 PDL 转为注释？
- 是否精确地把 PDL 翻译成了代码？
- 在作出假设时，验证它们了吗？
- 是从几个设计方案中选择了最好的，还是随意选择了一个方案？

- 是否彻底理解你的代码？它容易理解吗？

4.6 小　结

- 要想写好 PDL，首先要用易懂的自然语言，避免拘泥于某种程序语言，其次要在意向层次上写 PDL，描述设计作什么而不是如何作。
- PDL 到代码流程方法是详细设计的有力工具，而且使得编码非常容易。可以把 PDL 直接翻译成注释，但要注意保证注释是精确而有用的。
- 应该在工作的每一步中都检查子程序，并鼓励同事们检查。这样，可以在投入的资金和工作努力最少时便发现错误，从而极大降低改错成本。

第五章　　高质量子程序特点

目录

相关章节

生成子程序的步骤：见第 4 章
高质量模块的特点：见第 6 章
通用设计技术：见 7.5 节
软件结构设计：见 3.4 节

第四章讲述了生成子程序时应该采取的步骤，其重点是创建过程。本章的重点则是子程序本身，即区分好的子程序和低劣子程序的特征。

如果在进入现实而又困难的子程序细节问题之前，想阅读关于高层次设计的问题，那么请首先阅读第七章，然后再阅读本章。由于模块也要比子程序抽象，因此，也可在读完第六章后再阅读本章。

在讨论高质量子程序的细节问题之前，我们首先来考虑两个基本名词。什么叫“子程序”？子程序是具有单一功能的可调用的函数或过程。比如 C 中的函数，Pascal 或 Ada 中的函数或过程，Basic 中的子程序或 Fortran 中的子程序。有时，C 中的宏指令或者 Basic 中用 GOSUB 调用的代码块也可认为是子程序。在生成上述函数或过程中，都可以使用创建高质量子程序所使用的技术。什么是“高质量的子程序”？这是一个比较难以回答的问题。反过来最简单回答方式是指出高质量子程序不是什么。下面是一个典型的劣质子程序（用 Pascal 写成）：

```
Procedure HandleStuff ( Var InputRec:CORP_DATA,CrntQtr:integer,
        EmpRec:Emp_DATA, Var EstimRevenue:Real, YTDRevenue:Real,
        ScreenX:integer,ScreenY:integer,Var NewColor:Color_TYPE,
        Var PrevColor:COLOR_TYPE,Var Status:STATUS_TYPE,
        ExpenseType:integer);
```

```
begin
for i := 1 to 100 do begin
      InputRec.revenue[1]:= 0;
      InputRec.expense[i]:=CorpExpensse[CrntQtr,i]
      end;
UpdateCorpDatabase(EmpRec);
EstimRevenue:=YTDRevenue * 4.0 /real(CrntQtr)
NewColor:=PrevColor;
status:=Success;
if ExpenseType＝1 then begin
                        for i:= 1 to 12 do
                        Profit[i]:= Revenue[i]-Expense.Type[i]
                        end
else If ExpneseType= 2 then begin
                             Peofit[i]:=Revenue[i] - Expense.Type2[i]
                             end
else if ExpenseType= 3 then
begin
      Profit[i]:=Revenue[i] - Expense.Type3[i]
      end
end
```

这个子程序有什么问题？给你一个提示：你应该至少从中发现 10 个问题。当你列出所发现的问题后，再看一下下面所列出的问题；

- 程序的名字让人困惑。Handlestuff（）能告诉我们程序是干什么的吗？
- 程序没有被说明（关于说明的问题已经超出了个别子程序的范围，详见第 19 章“自我说明的子程序”）。
- 子程序的布局不好。代码的物理组织形式几乎没有给出其逻辑组织形式的任何信息。布局的使用过于随心所欲，程序每一部分的布局都是不一样的。关于这一点。只要比较一下 ExpenseType=2 和 ExpenseType=3 两个地方的风格就清楚了（关于布局问题，详见第十八章“布局与风格”）。
- 子程序的输入变量值 InPutRec 被改变过。如果它作为输入变量，那它的值就不该变化。如果要变化它的值，就不该称之为输入变量 InputRec。
- 子程序进行了全局变量的读写操作。它从 CorpExpense 中读入变量并写给 Profit。它应该与存取子程序通信，而不应直接读写全局变量。
- 这个子程序的功用不是单一的。它初始化了某些变量。对一个数据库进行写操作，又进行了某些计算工作，而它们又看不出任何联系。一个子程序的功用应该是单一，明了的。
- 子程序中没有采取预防非法数据的措施。如果 CrntQtr 的值为“0”，那么，表达式 YTDRevenue*4.0／real(CrntQtr)就会出现被零除的错误。
- 程序中使用了几个常数：100, 4.0, 12, 2 和 3。关于“神秘”（magic）数的问题见 11.1 节“常数”

- 在程序中仅使用了域的 CORP_DATA 型参数的两个域。如果仅仅使用两个域，那就该仅仅传入特定的域而不是整个结构化变量。
- 子程序中的一些参数没有使用过。ScreenX 和 ScreenY 在程序中没有涉及。
- 程序中的一个参数被错误标定了。PrevColor 被标定为变量型参数，然而在程序中又没有对其赋值。
- 程序中的参数太多。程序中参数个数的合理上限应该是七个左右。而这个程序中则多达 11 个。程序中的参数多得怕人，恐怕没谁会仔细检查它们，而且连数一下都不愿意。

除了计算机本身之外，子程序可以说是计算机科学最重大的发明。子程序使得非常好读而且也非常容易理解，编程语言中的任何特性都不能和这一点相比。像上例中那样使用子程序，简直就是对子程序的践踏，甚至可以说是一种犯罪。

子程序也是节省空间和提高性能的最好手段。想象一下，如果用代码段去代替程序中对子程序的每一次调用，那么程序将会有多么庞大。如果不是把多次重复使用的代码段存放在子程序中，而是直接把它放在程序中，那么对其进行性能改进将是一件很困难的事。是子程序使现代编程成为可能。

现在，你可能有些不耐烦。“是好，子程序的确很了不起，我一直都在使用它”。你说，“你的讨论似乎像是在纠正什么，你到底想让我做什么呢？”

我想说的是：有许多合理的原因使得我们去生成子程序。但是生成方法有好有坏。作为一个计算机专业的本科生，可以认为生成子程序的主要原因是避免代码段的重复。我所用的入门课本告诉我说，之所以使用于程序，是因为它可以避免代码段的重复，从而使得一个程序的开发、调试、注释和维护工作都变得非常容易。除了一些关于如何使用参数和局部变量的语法细节之外，这就是那本课本关于子程序理论与实践内容的全部。这实在不是一个完全而合理的解释。下面这节将详细描述为什么和怎样生成子程序。

5.1　生成子程序的原因

以下是关于为什么要生成于程序的一些合理原因，其中有些原因之间可能有互相重叠的地方。

降低复杂性。使用子程序的最首要原因是为了降低程序的复杂性，可以使用子程序来隐含信息，从而使你不必再考虑这些信息。当然，在编写子程序时，你还需要考虑这些信息。但是，一旦写好子程序，就可能不必再考虑它的内部工作细节，只要调用它就可以了。创建子程序的另外一个原因是尽量减小代码段的篇幅，改进可维护性和正确性。这也是一个不错的解释，但若没有子程序的抽象功能，将不可能对复杂程序进行明智的管理。

一个子程序需要从另一个子程序中脱离出来的原因之一是，过多重数的内部循环和条件判断。这时，可以把这部分循环和判断从子程序中脱离出来，使其成为一个独立的子程序，以降低原有子程序的复杂性。

避免代码段重复。无可置疑，生成子程序最普遍的原因是为了避免代码段重复。事实上，如果在两个不同子程序中的代码很相似，这往往意味着分解工作有误。这时，应该把两个子程序中重复的代码都取出来，把公共代码放入一个新的通用子程序中，然后再让这两个子程序调

用新的通用子程序。通过使公共代码只出现一次，可以节约许多空间。这时改动也很方便，因为只要在一个地方改动代码就可以了。这时代码也更可靠了，因为只需在一个地方检查代码。而且，这也使得改动更加可靠，因为，不必进行不断地、非常类似地改动，而这种改动往往又是认为自己编写了相同的代码这一错误假设下进行的。

限制了改动带来的影响。由于在独立区域进行改动，因此，由此带来的影响也只限于一个或最多几个区域中。要把最可能改动的区域设计成最容易改动的区域。最可能被改动的区域包括：硬件依赖部分、输入输出部分、复杂的数据结构和商务规则。

隐含顺序。把处理事件的非特定顺序隐含起来是一个很好的想法。比如，如果程序通常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，无论是读取用户数据的子程序，还是读取文件中数据的子程序，都不应该对另一个子程序是否读取数据有所依赖。如果利用两行代码来读取堆栈顶的数据，并减少一个 Stacktop 变量，应把它们放入一个 PopStack()子程序中，在设计系统时，使哪一个都可以首先执行，然后编写一个子程序，隐含哪一个首先执行的信息。

改进性能。通过使用子程序，可以只在一个地方，而不是同时几个地方优化代码段。把相同代码段放在子程序中，可以通过优化这一个子程序而使得其余调用这个子程序的子程序全部受益。把代码段放入子程序也使得用更快的算法或执行更快的语言（如汇编）来改进这段代码的工作变得容易些。

进行集中控制。在一个地方对所有任务进行控制是一个很好的想法。控制可能有许多形式。知道一个表格中的入口数目便是其中一种形式，对硬件系统的控制，如对磁盘、磁带、打印机、绘图机的控制则是其中另外一种形式。使用子程序从一个文件中进行读操作，而使用另一个子程序对文件进行写操作便是一种形式的集中控制。当需要把这个文件转化成一个驻留内存的数据结构时，这一点是非常有用的，因为这一变动仅改变了存取子程序。专门化的子程序去读取和改变内部数据内容，也是一种集中的控制形式。集中控制的思想与信息隐含是类似的，但是它有独特的启发能力，因此，值得把它放进你的工具箱中。

隐含数据结构。可以把数据结构的实现细节隐含起来，这样，绝大部分程序都不必担心这种杂乱的计算机科学结构，而可以从问题域中数据是如何使用的角度来处理数据。隐含实现细节的子程序可以提供相当高的抽象价值，从而降低程序的复杂程度。这些子程序把数据结构、操作集中在一个地方，降低了在处理数据结构时出错的可能性。同时，它们也使得在不改变绝大多数程序的条件下，改变数据结构成为可能。

隐含全局变量。如果需要使用全局变量，也可以像前述那样把它隐含起来、通过存取子程序来使用全局变量有如下优点：不必改变程序就改变数据结构；监视对数据的访问；使用存取子程序的约束还可以鼓励你考虑一下这个数据是不是全局的；很可能会把它处理成针对在一个模块中某几个子程序的局部数据，或处理成某一个抽象数据的一部分。

隐含指针控作。指针操作可读性很差，而且很容易引发错误。通过把它们独立在子程序中，可以把注意力集中到操作意图而不是机械的指针操作本身。而且，如果操作只在一处进行，也更容易确保代码是正确的。如果找到了比指针更好的数据结构，可以不影响本应使用指针的子程序就对程序作改动。

重新使用代码段。放进模块化子程序中的代码段重新使用，要比在一个大型号程序中的代码段重新使用起来容易得多。

计划开发一个程序族。如果想改进一个程序，最好把将要改动的那部分放进子程序中，将其独立。这样，就可以改动这个子程序而不致影响程序的其余部分，或者干脆用一个全新的子程序代替它。几年前，我曾经负责一个替保险推销员编写系列软件的小组，我们不得不根据每一个推销员的保险率、报价单格式等等来完成一个特定的程序。但这些程序的绝大部分又都是相同的：输入潜在客户的子程序，客户数据库中存储的信息、查看、计算价格等等。这个小组对程序进行了模块化，这样，随推销员而变化的部分都放在自己的模块中。最初的程序可能要用三个月的时间来开发，但是，在此之后，每来一个推销员，我们只改写其中屈指可数的几个模块就可以了。两三天就可能写完一个要求的程序，这简直是一种享受！

提高部分代码的可读性。把一段代码放入一个精心命名的子程序，是说明其功能的最好办法。这样就不必阅读这样一段语句：

```
if ( Node <> NULL )
   while ( Node.Next <> NULL ) do
         Node = Node.Next
   LeafName = Node.Name
else
   LeafName = " "
```

代替它的是：

```
LeafName = GetleafName(Node)
```

这个程序是如此简短，它所需要的注释仅仅是一个恰当的名字而已。用一个函数调用来代替一个有六行的代码段，使得含有这段代码的子程序复杂性大为降低，并且其功能也自动得到了注释。

提高可移植性。可以使用子程序来把不可移植部分、明确性分析和将来的移植性工作分隔开来，不可移植的部分包括：非标准语言特性、硬件的依赖性和操作系统的依赖性等。

分隔复杂操作。复杂操作包括：繁杂的算法、通信协议、棘手的布尔测试、对复杂数据的操作等等。这些操作都很容易引发错误。如果真的有错误，那么如果这个错误是在某个子程序中，而不是隐藏在整个程序中的话，查找起来要容易得多。这个错误不会影响到其它子程序，因为为了修正错误只要改动一个子程序就可以了。如果发现了一个更为简单迅速的算法，那么用它来代替一个被独立在子程序中的算法是非常容易的。在开发阶段，尝试几种方案并选择其中一个最好的是非常容易的。

独立非标准语言函数的使用。绝大多数实现语言都含有一些非标准的但却方便的扩展。使用这种扩展的影响是两面性的，因为在另外一个环境下它可能无法使用。这个运行环境的差异可能是由于硬件不同、语言的生产商不同、或者虽然生产商相同、但版本不同而产生的。如果使用了某种扩展，可以建立一个作为进入这种扩展大门的子程序。然后，在需要时，可以用订做的扩展来代替这一非标准扩展。

简化复杂的布尔测试。很少有必要为理解程序流程而去理解复杂的布尔测试。把这种测试放入函数中可以提高代码的可读性，因为：

(1) 测试的细节已经被隐含了。

(2) 清楚的函数名称已经概括了测试目的。

赋予这种测试一个函数，该函数强调了它的意义，而且这也鼓励了在函数内部增强其可读

性的努力。结果是主程序流和测试本身都显得更加清楚了。

是出于模块化的考虑吗？绝不是。有了这么些代码放入子程序的理由，这个理由是不必要的。事实上，有些工作更适合放在一个大的子程序中完成（关于程序最佳长度的讨论见 5.5 节“子程序长度”）。

5.1.1　简单而没有写入子程序的操作

编写子程序的最大心理障碍是不情愿为了一个简单的目的而去编写一个简单的子程序。写一个只有两或三行代码的子程序看起来是完全没有必要的。但经验表明，小的子程序也同样是很有帮助的。

小型子程序有许多优点，其中之一是改进了可读性。我曾在程序中采用过如下这样一个仅有一行的代码段，它在程序中出现了十几次：

```
Points = DeviceUnits * ( POINTS_PER_INCH / DeviceUnitsPerInch ( ) )
```

这决不是你所读过的最复杂的一行代码。很多人都明白它是用来转换的。他们也会明白程序中的每行这个代码都在作同一件事，但是，它还可以变得更清楚些，所以，我创建了一个恰当命名的子程序来作这些工作。

```
DeviceUnitsToPoints(DeviceUnits Integer): Integer;
begin
    DeviceUnitsToPoints = DeviceUnits *
                          ( POINTS_PER_INCH / DeviceUnitsPerInch ( ) )
end
```

在用这段子程序来代替那些十几次重复出现的代码行后，这些代码行基本上都成了如下的样子：

```
Points = DeviceUnitsToPoints ( DeviceUnits )
```

这显然更具可读性，甚至已经达到了自我说明的地步。

这个例子还暗示了把简单操作放入函数的另外一个原因：简单操作往往倾向于变成复杂操作。在写这个子程序时我还不知道这一点，但在某种情况下，当某个设备打开时，DeviceUnitPerInch()会返回零，这意味着我不得不考虑到被“0”除的情况，这样，又需要另外的三行代码；

```
DeviceUnitsToPoints( DeviceUnit:integer):integer;
begin
    if（ DeviceUnitsPerInch ( ) <> 0 ） then
        DeviceUnitsPoints = DeviceUnits *
                            ( POINTS_PER_INCH / DeviceUnitsPerInch ( ) )
    else
        DeviceUnitsToPoints= 0
end
```

如果原来的代码行仍然在程序中出现十几次，那么这一测试也要重复十几次，需要新增加 36 行代码。而一个简单子程序轻而易举地便把 36 变成了 3。

5.1.2　创建子程序的理由总结

以下是创建子程序理由概述：

- 降低复杂性
- 避免重复代码段
- 限制改动带来的影响
- 隐含顺序
- 改进性能
- 进行集中控制
- 隐含数据结构
- 隐含指针操作
- 隐含全局变量
- 促进重新使用代码段
- 计划开发一个软件族
- 改善某一代码段可读性
- 改善可移植性
- 分隔复杂操作
- 独立非标准语言函数的使用
- 简化复杂的布尔测试

5.2　子程序名称恰当

一个恰当的子程序名称应该清晰地描述出于程序所作的每一件事。以下是给子程序有效命名的指导方针：

对于过程的名字，可以用一个较强的动词带目标的形式。一个带有函数的过程往往是对某一目标进行操作。名字应该反映出这个过程是干什么的，而对某一目标进行操作则意味着我们应该使用动宾词组。比如，PrintReport），Checkotderlnfo（）等，都是关于过程的比较恰当的名字。

在面向对象的语言中，不必加上对象名，因为对象本身在被调用时就已经出现了。这时可求助于诸如 RePort.Print()，Orderlnfo.Check()和 MonthlyRevenu.Cafe()等名字。而像 RePort.PrintRePort 这类名字则是冗余的。

对于函数名字，可以使用返回值的描述。一个函数返回到一个值，函数应该用它所返回的值命名，例如 Cos()，PrinterReady()，CurrentPenColor()等等都是不错的函数名字，因为它精确地描述了函数将返回什么。

避免无意义或者模棱两可的动词。有些动词很灵活，可以有任何意义，比如 HandleCalculation()，ProcessInput（）等于程序名词并没有告诉你它是作什么的。这些名字至多告诉你，它们正在进行一些与计算或输入等有关的处理。当然，有特定技术情形下使用“handle”等词是个例外。

有时，子程序的唯一问题就是它的名字太模糊了，而子程序本身的设计可能是很好的。如

果用 FormatAndPrintOutput() 来代替 HandleOutPut() ，这是一个很不错的名字。

在有些情况下，所用的动词意义模糊是由于子程序本身要做的工作太模糊。子程序存在着功能不清的缺陷，其名字模糊只不过是个标志而已。如果是这种情况，最好的解决办法是重新构造这个子程序，弄清它们的功能，从而使它们有一个清楚的、精确描述其功能的名字。

描述子程序所做的一切。在子程序名字中，应描述所有输出结果及其附加结果。如果一个子程序用于计算报告总数，并设置一个全局变量来表示所有的数据都已准备好了，正等待打印，那么，ComputeReportTotal（）就不是一个充分的名字了。而 ComputeReport TotalAndSetPrintingReadyVar（）又是一个太长而且太愚蠢的命名。如果子程序带有附加结果，那必然会产生许多又长又臭的名字。解决的办法不应该是使用描述不足名字，而是采用直接实现每件事的原则来编程，从而避免程序带有附加结果。

名字的长度要符合需要。研究表明，变量名称的最佳长度是 9 到 15 个字母，子程序往往比变量要复杂，因而其名字也要长些。南安普敦大学的 MichaelRees 认为恰当的长度是 20 到 35 个字母。但是，一般来说 15 到 20 个字母可能更现实一些，不过有些名称可能有时要比它长。

建立用于通用操作的约定。在某些系统中，区分各种不同的操作是非常重要的。而命名约定可能是区分这些操作最简单也是最可靠的方法。比如，在开发 0S/2 显示管理程序时，如果子程序是关于直接输入的，就在其名称前面加一个“Get”前缀，如果是非直接输入的则加“Query”前缀，这样，返回当前输入字符的 GetlnputChar（）将清除输入缓冲区．而同样是返回当前输入字符的 QuerylnPutChar（）则不清除缓冲区。

5.3　强内聚性

内聚性指的是在一个子程序中，各种操作之间互相联系的紧密程度。有些程序员喜欢用“强度”一词来代替内聚性，在一个子程序中各种操作之间的联系程度有多强？一个诸如 Sin()之类的函数内聚性是很强的，因为整个子程序所从事的工作都是围绕一个函数的。而像 SinAndTan() 的内聚程度就要低得多了，因为子程序中所进行的是一项以上的工作。强调强相关性的目的是，每一个子程序只需作好一项工作，而不必过分考虑其它任务。

这样作的好处是可以提高可靠性。通过对 450 个 Fortran 子程序的调查表明，50％的强内聚性子程序是没有错误的，而只有 18％的弱内聚性子程序才是无错的(Card,carch 和 Agresti 1986)。另一项对另外 450 个子程序的调查则表明，弱内聚性子程序的出错机会要比强内聚性出错机会高 6 倍，而修正成本则要高 19 倍（Selby 和 Basili 1991）。

关于内聚性的讨论一般是指几个层次。理解概念要比单纯记住名词重要得多。可以利用这些概念来生成内聚性尽可能强的子程序。

5.3.1　可取的内聚性

内聚性的想法是由 wayne stevens,Glenford Myers 和 Larry Constantine 等人在 1974 年发表的一篇论文中提出来的，从那以后，这个想法的某些部分又逐渐得到了完善。以下是一些通常认为是可以接受的一些内聚类型：

功能内聚性。功能内聚性是最强也是最好的一种内聚，当程序执行一项并且仅仅是一项工作时，就是这种内聚性，这种内聚性的例子有： sin()， GetCustomerName(), EraseFile()，

CaldoanPayment()和 GetIconlocation()等等。当然，这个评价只有在子程序的名称与其实际内容相符时才成立。如果它们同时还作其它工作，那么它们的内聚性就要低得多而且命名也不恰当。

顺序内聚性。顺序内聚性是指在子程序内包含需要按特定顺序进行的、逐步分享数据而又不形成一个完整功能的操作，假设一个子程序包括五个操作：打开文件、读文件、进行两个计算、输出结果、关闭文件。如果这些操作是由两个子程序完成的，DoStep1()打开文件、读文件和计算操作，而 DoStep2()则进行输出结果和关闭文件操作。这两个子程序都具有顺序内聚性。因为用这种方式把操作分隔开来，并没有产生出独立的功能。

但是，如果用一个叫作 GetFileData()的子程序进行打开文件和读文件的操作，那么这个子程序将具有功能内聚性。当操作来完成一项功能时，它们就可以形成一个具有功能内聚性的子程序。实际上，如果能用一个很典型的动宾词组来命名一个子程序，那么它往往是功能内聚性，而不是顺序内聚性。给一个顺序内聚性的子程序命名是非常困难的，于是便产生了像 Dostep1()这种模棱两可的名字。这往往意味着你需要重新组织和设计子程序，以使它是功能内聚性的。

通讯内聚性。通讯内聚性是在一个子程序中，两个操作只是使用相同数据，而不存在其它任何联系时产生的。比如，在 GetNameAndChangePhoneNumber()这个子程序中，如果 Name 和 PhoneNumber 是放在同一个用户记录中的，那么这个子程序就是通讯内聚性。这个子程序从事的是两项而不是一项工作，因此，它不具备功能内聚性。Name 和 PhoneNamber 都存储在用户记录中，不必按照某一特定顺序来读取它们，所以，它也不具备顺序内聚性。

这个意义上的内聚性还是可以接受的。在实际中，一个系统可能需要在读取一个名字的同时变更电话号码。一个含有这类子程序的系统可能有些显得别扭，但仍然很清楚且维护性也不算差，当然从美学角度来说，它与那些只作一项工作的子程序还有一定差距。

临时内聚性。因为同时执行的原因才被放入同一个子程序里，这时产生临时内聚性。典型的例子有；Startup()，CompleteNewEmployee()，Shutdown()等等，有些程序员认为临时内聚性是不可接受的，因为它们有时与拙劣的编程联系在一切，比如，在像 Startup()这类子程序中往往含有东拼西凑的杂烩般的代码。

要避免这个问题，可以把临时内聚性子程序设计成一系列工作的组织者。前述的 Startup ()子程序进行的操作可能包括：读取一个配置文件、初始化一个临时文件、建立内存管理、显示初始化屏幕。要想使它最有效地完成这些任务，可以让这个子程序去调用其它的专门功能的子程序，而不是由它自己直接来完成这些任务。

5.3.2　不可取的内聚性

其余类型的内聚性，一般来说都是不可取的。其后果往往是产生一些组织混乱而又难以调试和改进的代码。如果一个子程序具有不良的内聚性，那最好重新创建一个较好的子程序，而不要去试图修补它。知道应该避免什么是非常重要的，以下就是一些不可取的内聚性：

过程内聚性。当子程序中的操作是按某一特定顺序进行的，就是过程内聚性。与顺序内聚性不同，过程内聚性中的顺序操作使用的并不是相同数据。比如，如果用户想按一定的顺序打印报告，而所拥有的子程序是用于打印销售收入、开支、雇员电话表的。那给这个子程序命名是非常困难的，而模棱两可的名字往往代表着某种警告。

逻辑内聚性。当一个子程序中同时含有几个操作，而其中一个操作又被传进来的控制标志所选择时，就产生了逻辑内聚性。之所以称之为逻辑内聚性，是因为这些操作仅仅是因为控制流，或者说“逻辑”的原因才联系到一起的，它们都被包括在一个很大的 if 或者 case 语句中，它们之间并没有任何其它逻辑上的联系。

举例来说，一个叫作 InputAll()的子程序，程序的输入内容可能是用户名字、雇员时间卡信息或者库存数据，至于到底是其中的哪一个，则由传入子程序的控制标志决定。其余类似的子程序还有 ComputeAll()，EditAll()，PrintAll()等等。这类子程序的主要问题是一定要通过传入一个控制标志来决定子程序处理的内容。解决的办法是编写三个不同的子程序，每个子程序只进行其中一个操作。如果这三个子程序中含有公共代码段，那么还应把这段代码放入一个较低层次的子程序中，以供三个子程序调用。并且，把这三个子程序放入一个模块中。

但是，如果一个逻辑内聚性的子程序代码都是一系列 if 和 case 语句，并且调用其它子程序，那么这是允许的。在这种情况下，如果程序的唯一功能是调度命令，而它本身并不进行任何处理，那么这可以说是一个不错的设计。对这种子程序的专业叫法是“事物处理中心”，事物处理中心往往被用作基础环境下的事件处理，比如，Apple Macintosh 和 Microsoft Windows。

偶然内聚性。当同一个子程序中的操作之间无任何联系时，为偶然内聚性。也叫作“无内聚性”。本章开始时所举的 Pascal 例程，就是偶然内聚性。

以上这些名称并不重要，要学会其中的思想而不是这些名词。写出功能内聚性的子程序几乎总是可能的，因此，只要重视功能内聚性以获取最大的好处就可以了。

5.3.3　内聚性举例

以下是几个内聚性的例子，其中既有好的，也有坏的：

功能内聚性例子。比如计算雇员年龄并给出生日的子程序就是功能内聚性的，因为它只完成一项工作，而且完成得很好。

顺序内聚性的例子。假设有一个按给出的生日计算雇员年龄、退休时间的子程序，如果它是利用所计算的年龄来确定雇员将要退休的时间，那么它就具有顺序内聚性。而如果它是分别计算年龄和退休时间的，但使用相同生日数据，那它就只具有通讯内聚性。

确定程序存在哪种不良内聚性，还不如确定如何把它设计得更好重要。怎样使这个子程序成为功能内聚性呢？可以分别建立两个子程序，一个根据生日计算年龄，另外一个根据生日确定退休时间，确定退休时间子程序将调用计算年龄的程序，这样，它们就都是功能内聚性的，而且，其它子程序也可以调用其中任一个子程序，或这两个部调用。

通讯内聚性的例子。比如有一个打印总结报告，并在完成后重新初始化传进来的总结数据的子程序，这个子程序具有通信内聚性，因为这两个操作仅仅是由于它们使用了相同的数据才联系在一起。

同前例一样，我们考虑的重点还是如何把它变成是功能内聚性，总结数据应该在产生它的地方附近被重新初始化，而不应该在打印子程序中重新初始化。把这个子程序分为两个独立的子程序．第一个打印报告，第二个则在产生或者改动数据的代码附近重新初始化数据。然后，利用一个较高层次的子程序来代替原来具有通讯相关的子程序，这个子程序将调用前面两个分出来的子程序。

逻辑内聚性的例子。一个子程序将打印季度开支报告、月份开支报告和日开支报告．具体

打印哪一个，将由传入的控制标志决定，这个子程序具有逻辑内聚性，因为它的内部逻辑是由输进去的外部控制标志决定的。显然，这个子程序不是按只完成一项工作并把它作好的原则。

怎样使这个子程序变为功能内聚性呢？建立三个子程序：一个打印季度报告，一个打印月报告、一个打印日报告，改进原来的子程序，让它根据传送去控制标志来调用这三个子程序之一。调用子程序将只有调用代码而没有自己的计算代码，因而具有功能内聚性。而三个被调用的手程序也显然是功能内聚性的。非常巧合，这个只负责调用其它子程序的子程序也是一个事务处理中心。最好用如 DispatchReporPrinting()之类带有“调度”或“控制”等字眼的词来给事务处理中心命名，以表示它只负责命令温调度,而本身并不做任何工作。

逻辑内聚性的另一个例子。考虑一个负责打印开支报表、输入新雇员名字并备份数据库的子程序，其具体执行内容将由传入的控制标志控制。这个子程序只具有逻辑内聚性，虽然这个关联看起来是不合逻辑的。

要想使它成为功能内聚性，只要按功能把它分成几部分就可以了。不过，这些操作有些过于凌乱。因此，最好重新建立一个调用各子程序的代码。当拥有几个需要调用的子程序时，重新组织调用代码是比较容易的。

过程内聚性的例子。假设有一个子程序，它产生读取雇员的名字，然后是地址，最后是它的电话号码。这种顺序之所以重要，仅仅是因为它符合用户的要求，用户希望按这种顺序进行屏幕输入。另外一个子程序将读取关于雇员的其它信息。这个子程序是过程内聚性，因为是由一个特定顺序而不是其它任何原因，把这些操作组合在一起的。

与以前一样，如何把它变为功能内聚性的答案仍然是把它分为几个部分，并把这几部分分别放入程序中。要保证调用子程序的功能是单一、完善的。调用子程序应该是诸如 GetEmployeeData()这样的子程序，而不该是像 GetFirstPartofEmployeeData()这类的子程序。可能还要改动其余读取数据的子程序。为得到功能内聚性，改动几个子程序是很正常的。

同时具有功能和临时内聚性的程序。考虑一个具有完成一项事物处理所有过程的子程序，从用户那里读取确认信息，向数据存入一个记录，清除数据域，并给计数器加 1。这个程序是功能内聚性的，因为它只从事一项工作，进行事物处理，但是，更确切地说，这个子程序同时也是临时内聚性的，不过当一个子程序具有两种以上内聚性时，一般只考虑最强的内聚性。

这个例子提出了如何用一个名字恰如其分地抽象描述出程序内容的问题。比如可以称这个子程序为 ConfirmEntryAndAdjustData()，表示这个干程序仅具有偶然内聚性。而如果称它为 CompleteTransaction()，那么就可能清楚地表示出这个子程序仅具有一个功能，而已具有功能内聚性。

过程性、临时或者可能的逻辑内聚性。比如一个进行某种复杂计算前 5 个操作，并把中间结果返回到调用子程序。由于 5 项操作可能要用好几个小时，因此当系统瘫痪时，这个子程序将把中间结果存入一个文件中，然后，这个子程序检查磁盘，以确定其是否有足够空间来存储最后计算结果，并把磁盘状态和中间结果返回到调用程序。

这个子程序很可能是过程内聚性的，但你也可能认为它具有临时内聚性，甚至具有逻辑内聚性。不过，不要忘了问题的关键不是争论它具有哪种不好的内聚性，而是如何改善其内聚性。

原来的子程序是由一系列令人莫名其妙的操作组成的，与功能内聚性相距甚远，首先，调用子程序不应该调用一个，而应该调用几个独立的子程序：l）进行前 5 步计算的子程序；2）把中间结果存入一个文件；3）确定可用的磁盘存储空间。如果调用子程序被称作

ComputeExtravagantNumber()，那么它不应该把中间结果写入一个文件，也决不该为后来的操作检查磁盘剩余空间，它所作的就仅限于计算一些数而已。对这个子程序的精心重新设计，将至少影响到一至两个层次上的子程序，对于这项任务的较好设计，见图 5—l。

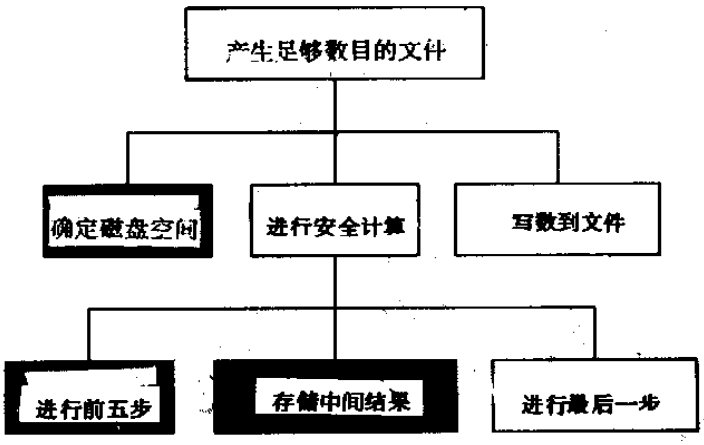


图 5-1 进行任务分解以获得功能内聚性举例

图中画阴影的部分是由原来的子程序从事的工作，在新组织结构中它们位于不同的层次上，这就是为什么为了把这些工作放人恰当的子程序中，要进行这么多重新组织工作的原因。用几个功能内聚性的子程序来代替一个具有不良内聚性的子程序是很平常的。

5.4　松散耦合性

所谓耦合性指的是两个子程序之间联系的紧密程度。耦合性与内聚性是不同的。内聚性是指一个子程序的内部各部分之间的联系程度，而耦合指的是子程序之间的联系程度。研究它们的目的是建立具有内部整体性（强内聚性），而同时与其它子程序之间的联系的直接、可见、松散和灵活的子程序（松散耦合）。

子程序之间具有良好耦合的特点是它们之间的耦合是非常松散的，任一个子程序都能很容易地被其它子程序调用。火车车箱之间的联接是非常容易的，只要把两节车箱推撞到一起，挂钩就会目前挂上，想象一下，用螺栓把它们固定到一起，或者只有特定的车厢之间才能联接到一起，那么事情将会有多么麻烦。火车车厢之间的联接之所以非常容易，是因为它们的联接装置非常简单。同样，在软件中，也应该使子程序之间的耦合尽量简单。

在建立一个子程序时，应尽量避免它对其它子程序有依赖性，应该使它们像商业上的伙伴一样相互分离，而不要使它们像连体婴儿一样密不可分。类似 Sin()的函数是松散耦合的，因为它所需要的只是一个传递进去的角度值。而类似 InitVars（varl，varZ，var3，……，varN）的函数则是紧密耦会的，因为调用程序事实上知道函数内部做什么。依靠使用同一全局变量联系在一起的子程序之间，其耦合程度则更高。

5.4.1　耦合标准

以下是几条估计子程序间耦合程度的标准：

耦合规模。所谓耦合规模是指两个子程序之间联系的数量多少。对于耦合来说，联系的数

量越少越好，因为一个于程序的接口越少，那么在把它与其它子程序相连接时，所要做的工作也越少。一个只有一个参数的子程序与调用它的程序间的耦合程序，要比有 6 个参数的子程序与调用它的程序间的耦合程度松散得多。一个使用整型参数的子程序与其调用程序之间的耦合程度，也要比一个使用有 10 个元素数组或者结构化数据的子程序与其调用程序之间的耦合程度松散得多。同样，使用一个全局变量的子程序与使用十二个全局变量的子程序相比，其耦合程度也要松散得多。

密切性。密切性指的是两个子程序之间联系的直接程度。联系越直接越好，两个子程序之间联系最密切的是参数表中的参数。这时，两个程序直接通讯。这时这个参数就像接吻时的嘴唇。联系密切程度稍低的是作用于同一全局数据的两个子程序。它们之间交流的直接性稍低。全局变量就像是两个子程序之间的爱情，它可能消失在信中，也可能到你想要它到的地方。联系程度最低的是作用于同一数据库记录或文件的两个子程序，它们都需要这个数据但却又羞于通知对方，这个被分享的数据就像是在课堂上传阅着的一张写有“你喜欢我吗？请回答‘是’还是‘不是’”的纸条。

可见性。可见性是指两个子程序之间联系的显著程度。编程不像是在中央情报局中工作，不会因为行动隐蔽而受到表彰，它更像是作广告，干得越是大张旗鼓，受到的表彰也就越多。在参数表中传递数据是明显的，因而也是好的。而通过改动全局数据以便让别的子程序来使用它，则是一个隐蔽的联系因而也是不好的。对全局数据联系进行注释以使其更明显，可能稍好些。

灵活性。灵活性是指改变两个子程序之间联系的容易程度。形象地说，你更喜欢电话上的快速插口装置，而不会喜欢用电烙铁把铜线焊到一起，灵活性可能有一部分是由其它耦合特性决定的，但也有一些区别。比如，有一个按照给定的被雇用日期和被雇用部门，寻找雇员的第一个监工的子程序，并命名它为 LookUpFirstsupervisor()。同时，还有一个对变量 EmpRec 进行结构化的子程序，变量 EmpRec 包括雇用日期、雇用部门等信息，第二个子程序把这个变量传给第一个子程序。

从其它观点来看，两个子程序之间的耦合是非常松散的。因为处于第一个和第二个子程序之间的变量 EmpRec 是公共的，所以它们之间只有一个联系。现在，假设需要用第三个子程序来调用子程序 LookUpFirstSupervisor()，但这个子程序中不含 EmpRec，却含有雇用部门和雇用日期信息。这时 LookUpFirstSupervisor()就不是那么友好了，它不情愿与第三个子程序进行合作。

对于调用 LookUpFirstsupervisor()的子程序来说，它必须知道 EmpRec 的数据结构。它可以使用一个仅有两个域的变元 EmpRec，但这又需要知道 LookUpFirstSupervisor()内部结构，即那些仅供它使用的域，这是一个愚蠢的解决方法。第二个方案是改动 LookUpFirstSupervisor，使它自带雇用部门和雇用日期信息，而不是使用 EmpRec。不管采用哪种方案，这个子程序都不像最初看起来那么灵活了。

如果愿意的话，一个不友好的子程序也是可以变为友好的。这种情况可以通过让它明确带有雇用部门和日期信息，而不再使用 EmpRec 来达到这一目的。

简而言之，如果一个子程序越容易被其它子程序调用，那么它的耦合程度也就越低。这样的好处是可以增强灵活性和维护性。在建立系统结构时，应该沿着相互耦合程度的最低线将其分开。如果把程序看成一块木头的话，就是要沿着它的纹理把它劈开。

5.4.2　耦合层次

传统上，把耦合层次称为非直觉性（unintuitive）。所以，在以下叙述中，将交替使用这两个名字。在以下叙述中，既有好的耦合，也有不好的耦合。

简单数据耦合。如果两个子程序之间传递的数据是非结构化的，并且全部都是通过参数表进行的，这通常称作“正常耦合”，这也是一种最好的耦合。

数据结构耦合。如果在两个程序之间传递的数据是结构化的，并且是通过参数表实现传递的，它们之间就是数据结构耦合的。这种耦合有时也称之为“邮票耦合”（stamp coupling）（不过我总觉得这种叫法非常奇怪）。如果使用恰当的话，这种耦合也不错，它与简单耦合的主要区别是它所采用的数据是结构化的。

控制耦合。如果一个子程序通过传入另一个子程序的数据通知它该作什么，那么这两个子程序就是控制耦合的。控制耦合是令人不快的，因为它往往与逻辑内聚性联在一起，并且，通常都要求调用程序了解被调子程序的内容与结构。

全局数据耦合。如果两个子程序使用同一个全局数据，那它就是全局数据耦合的。这也就是通常所说的“公共耦合”或“全局耦合”。如果所使用的数据是只读的，那么这种耦合还是可以忍受的，但是，总的来说，全局耦合是不受欢迎的，因为这时子程序之间的联系既不密切，又不可见。这种联系容易被疏漏掉，甚至可以认为它是一种由信息隐含带来的错误——信息丢失。

不合理耦合（pathological）。如果一个子程序使用了另外一个子程序中代码，或者它改变了其中的局部变量，那么它们就是不合理耦合的。这种耦合也称之为“内容耦合”。这一类耦合是不能接受的，因为它不满足关于耦合规模、密切性、可见性和灵活性中的任何一条标准。虽然这是一个很紧的联系，但是这种联系却是不密切的。改动另一个子程序中的数据无异于在其后背桶上一刀，而且，这背后一刀从表面上又是看不出来的。由于它是建立在一个子程序完全了解另一个程序内容的基础之上的，因此其灵活性也是很差的。许多结构化语言中，都有明确禁止不合理耦合的语法规则。但是，在 Basic 或汇编语言中，它却是允许的。因此，在用这种语言编程时，一定要小心！

以上所有类型的耦合，如图 5－2 所示。

5.4.3　耦合举例

以下是上述各种耦合的例子，其中有好的，也有坏的：

简单数据耦合的例子。比如，一个程序向 Tan（）子程序传递含有角度值的变量，那它们之间就是简单数据耦合的。

简单数据耦合的另一个例子。两个程序向另一个子程序传递姓名、住址、电话号码、生日和身份证号码等五个变量。

可接受的数据结构耦合的例子。一个程序向另一个子程序传递变量 EmpRec，EmpRec 是一个结构化的变量，包括姓名、住址、生日等等五个方面的数据，而被调用的子干程序则全部使用这五个域。

不可取的数据结构耦合举例。一个程序向另一个子程序传递同样的变量 EmpRec，但是，如果被调用的子程序只使用其中两个域，比如电话号码和生日。这虽然还是数据结构耦合，但却不是个很好的应用，如果把生日和电话号码作为简单变量来传递的话，将使联系更加灵活，

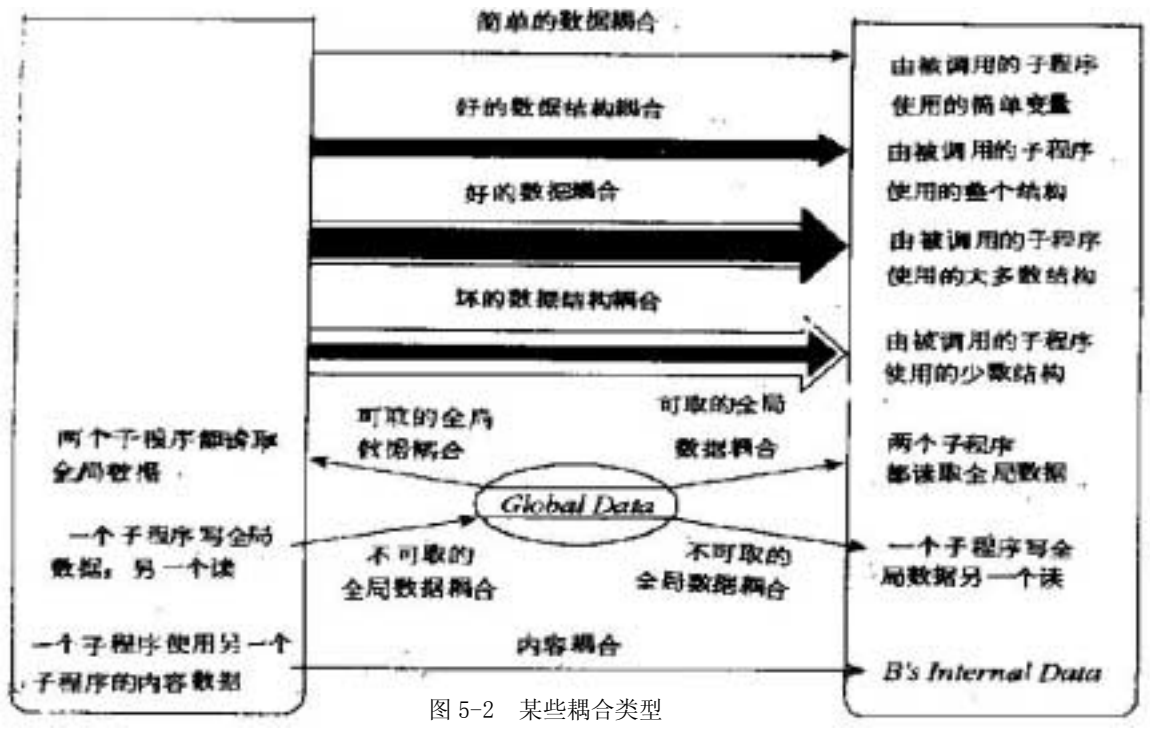


图 5-2　某些耦合类型

而且会使它们之间的两个特定域真正联系的可见性更好。

有问题的数据结构耦合的例子。一个程序向另一个子程序传递变量 OfficeRec。OfficeRec 有 27 个域，而被调用的子程序使用其中 16 个，这也是数据结构耦合，但是，它是一个好的数据结构耦合吗？决不是。传递 OfficeRec 使得联系是大规模的，这个事实非常明显，而传递 16 个单独参数，则又再次非常拙劣地表明了这一点，如果被调用子程序仅使用其中的 6 到 7 个域，那么单个地传递它们是个好主意。

在现在这种情形下，可以进一步对 OfficeRec 进行结构化，以使得在被调用程序中用得到的 16 个域包含在一个或两个亚结构中，这将是最简洁的解决办法。

简单数据耦合或可能数据结构耦合的例子。一个程序调用 EraseFile（）子程序，通过一个含有待删文件名的字符串确定将要删去的文件，这很可能是一个简单数据耦合。但也可以说这是一个数据结构耦合，因为字符串是一个数据结构。

我觉得，我们的结论是半斤与八两的关系，这两种叫法都是同样正确的，对其作严格区分是没必要的。

控制耦合的例子。比如，一个程序向另一个子程序传递控制标志，通知它到底是打印月报表、季度报表还是年度报表。

不可取的全局数据耦合的例子。一个程序改动一个表的人口作为全局变量，这个表是以雇员的识别卡作为索引的。然后，这个程序又调用另一个子程序并把雇员识别卡作为一个参数传递给它，而这个被调用的子程序则用雇员识别卡去读全局数据表，这是一个典型的全局数据耦合（虽然仅仅传递雇员识别卡形成的是简单数据耦合，但是第一个程序对表入口的改动，已经决定了这是一种最坏的耦合——全局数据耦合）。

可取的全局数据耦合的例子。一个程序把雇员识别卡传递给另一个子程序，两个程序都利用这个识别卡从一个全局表中读取雇员的名字，两个子程序都没有改变全局数据。

这通常也称为“全局数据耦合”，但事实上它更像“简单数据耦合”，我们也可称它为“可取的全局数据耦合”。与前述一个子程序改变另一个程序使用数据的例子不同，这两个程序并不是由全局数据联系在一起的。比较两个例子，这种对相同全局数据的只读使用是良性的。这两个从同一个全局表读取数值的程序，与上述那两个通过使用全局数据来掩盖它们之间联系的程序是完全不同的。

内容耦合的例子。在汇编语言中，一个子程序可以知道另一个子程序中说明为局部变量的表的地址。它可以命名用这个地址直接去改动这个表，而地址在两个子程序间并没有当作参数传递。

内容耦合另一个例子。一个 Basic 程序利用 GOSUB 语句来执行另一个子程序中的一段代码。

好的耦合关键是它可以提供一个附加的抽象层次——一旦写好它，就可以认为它是独立的。它降低了整个程序的复杂性，并且使你每次只致力于一件事情。如果在使用子程序时要求同时考虑几件事情——知道它的内部内容、对全局数据的改动、不确定的功能等，就会使其丧失抽象能力，那么使用子程序还有什么用呢？子程序本来是用于降低复杂性的工具，如果使用它没有使工作更简单，那说明没有用好它。

5.5　子程序长度

理论上，常把一个子程序的最佳长度定为一两页，即 66 到 132 行。按照这种原则，在七十年代，IBM 公司曾把子程序的长度限制在 50 行以下，而 TRW 公司则把这个标准定为 132 行（McCabe 1976）。几乎没有什么证据证明这一限制是正确的。相反，倒是证明较长子程序更有利的证据更有说服力，请参照以下几点：

- Basili 和 Perricone 1984 年的研究表明，子程序的长度与错误数量是成反比的。随着子程序长度的增加（长至 200 行的代码），每一行的错误数量开始下降。
- 另一个由 shen et al 1985 年进行的研究表明，程序长度与错误数是无关的，但错误数量会随着结构复杂性和数据数量的增加而增加。
- 一项由 Card、Church 和 Agresti 在 1986 年，以及 Card 和 Glass 在 1990 年进行的调查表明，小型子程序（32 行代码或更少）并不意味着低成本和低错误率，证据表明大型子程序（65 行或更多）的每行成本要低于小型子程序。
- 对 450 个子程序的一项调查发现小型子程序（包括注释行，少于 143 行源语句）的每行错误率要比大型子程序高 23%（Selby 和 Basili 1991）。
- 对计算机专业高年级学生进行的测验表明，学生们对一个被过度模块化的、由许多有 10 行左右代码子程序组成的软件，与同样内容但不含任何子程序的软件的理解程度是相同的。但若把整个程序分解成中等规模的子程序（每个有 25 个代码），学生们的理解程度会上升为 65%。
- 最近研究发现，当子程序长度是 100 到 150 行时，错误率最低(Lind 和 Variavan 1989)。

研究子程序长度有什么好处呢？如果你是一个经理，不要限制程序员们编写长于一页的子程序，刚才引用的资料和程序员们自己的经验都可以证明你这样作是正确的。如果想编写一个

长度是 100 行，150 行或 200 行的子程序，那就按照你想的去作吧。目前的证据表明，这种长度的子程序并不更易引发错误，而其开发更为容易。

如果要开发长于 200 行的子程序，就要小心了（这里的长度不包括注释行和空行）。目前还没有任何证据表明长于 200 行的子程序会带来更低的成本或更少的错误。而这样做却会使你达到可理解性的上限。在对 IBM 为 OS／360 操作系统及其它系统而开发代码研究中发现，最易出错的子程序是那些大于 500 行的子程序。在超过 500 行之后，错误数量会与子程序的长度成正比。而且，对一个有 148，000 行代码软件的研究发现，修改少于 143 行子程序错误所耗费的成本，要比修复长于 143 行的子程序中错误成本低 2.4 倍（Sely 和 Basin 1991）。

5.6　防错性编程

防错性编程并不意味着要对自己的程序提高警惕，这一想法是在防错性驾驶的基础上产生的，在这种驾驶方法中，必须在心中时刻认为其它驾驶员的行为都是不可预测的。这样，就可以在他们做出某些危险举动时，确保自己不会因此受伤。在防错性编程中，其中心思想是，即使一个子程序被传入了坏数据，它也不会被伤害，哪怕这个数据是由其它干程序错误而产生的。更一般地说，其思想核心是承认程序中都会产生问题，都要被改动，一个聪明的程序员就以这点为依据开发软件。

作为本书介绍的提高软件质量技术之一，防错性编程是非常有用的。最有效的防错性编码途径是一开始就不要引人错误。可以采用逐步设计方法、在编码前先写好 PDL、进行低层次设计、审查等都可以防止错误引入。因此，应优先考虑它们，而不是防错性编程。不过，你可以把防错性编程与这些技术组合起来使用。

5.6.1　使用断言

断言是一个在假设不正确时会大声抗议的函数或宏指令。可以使用断言来验证在程序中作出的假设并排除意外情况。一个断言函数往往大致带有两个内容：假设为真时的布尔表达式和一个为假时要打印出来的信息。以下是一个假定变量 Denominator 不为零时一个 Pascal 断言：

```
Assert （ Denominator<>0,'Denominator is unexpectedlg equal to 0.' ） ;
```

这个断言假定 Denominator 不等于”0”，第一部分 Denominator<>0 是一个布尔表达式，其结果为 True 或 False。第二部分是当第一部分的结果为 False 肘，将要打印出来的信息。

即使不愿让用户在最终软件中看到断言信息，在开发和维护阶段，使用断言还是非常方便的。在开发阶段，断言可以消除相互矛盾的假设，消除传入于程序的不良数值等等。在维护，可以表明改动是否影响到了程序其它部分。

断言过程是非常容易写的，下面就是一个用 Pascal 写成的例子：

```
Procedure Assert
        (
            Aseertionn: boolean;
            Message : string
        );
```

```
begin
  if（ not Assertion）
     begin
     writeln（Messase）;
     writeln('stopping the program.');
     halt（FATAL　ERROR）
     end
end;
```

一旦写好了这样一个过程，就可以用像第一个例子那样的语句来调用它。

下面是使用断言的一些指导方针：

如果有预处理程序的话，使用预处理程序宏指令。如果在开发阶段使用预处理程序处理断言，那么在最终代码中把断言去掉是非常容易的。

在断言中应避免使用可执行代码，把可执行代码放入断言，在关闭断言时，编译程序有可能把断言捎去。请看以下断言；

```
ASsert（FileOpen（InputFile<>NULL，’Coulldnt Oped input file’）;
```

这行代码产生的问题是，如果不对断言进行编译，也编译不了打开文件的代码，应把可执行语句放在自己的位置上，把结果赋给一个状态变量，然后再测试状态。以下是一个安全使用断言的例子：

```
FileStatus : FileOpen (InputFile);
Assert(FileStatus<>　NULL，'couldn't Opeh input file');
```

5.6.2　输入垃圾不一定输出垃圾

一个好的程序从来不会输出乱七八糟像垃圾似的东西，不管它被输入的是什么。一个好程序的特点是“输入垃圾，什么也不产生”，或“输入垃圾，输出错误信息”，也可以是“不允许垃圾进入”。从现在的观点来看“输入垃圾，输出垃圾”，往往是劣质程序。

检查所有外部程序输入的数值。当从用户文件中读取数据时，要确保读人的数据值在允许范围之内。要保证数值是可以取的，并且字符串要足够短，以便处理。要在代码中注释出输入数据的允许范围。

检查全部子程序输入参数值。检查子程序输入参数值，事实上与检查外部程序数据是一样的，只不过此时由子程序代替了文件或用户。

以下是一个检查其输入参数的子程序的例子，用 c 语言编写：

```
float tan
  (
       float OppositeLength;
       float AdjacentLength;
  )
{
 /*计算角度正切*/
Aseert( AdjancentLength != 0，”AdjanceLength deteced to be 0." );
```

```
return (OppsiteLenght/AdjancetLength);
}
```

决定如何处理非法参数。一旦发现了一个非法参数，该如何处理呢？根据不同情况，可以　希望返回一个错误代码、返回一个中间值、用下一个合法数据来代替它并按计划继续执行、与上次一样返回一个正确答案、使用最接近的合法值、调用一个处理错误的子程序、从一个子程序中调用错误信息并打印出来或者干脆关闭程序。由于有这样多的方案可供选择，所以当在程序的任一个部分处理非法参数时，一定要仔细，确定处理非法参数的通用办法，是由结构设计决定的，应该在结构设计层次上予以说明。

5.6.3　异常情况处理

应核预先设计好异常处理措施来注意意想不到的情况。异常处理措施应该能使意外情况的出现在开发阶段变得非常明显，而在运行阶段又是可以修复的，例如，在某种情况下使用了一个 case 语句，其中只预计到了五种情况，在开发阶段，应该能利用异常情况处理产生一个警告，提示出现了另外一种情况。而在产品阶段，应该利用异常情况处理做一些更完美的工作，比如向一个错误记录文件中写入信息等。总之，应该设计出不必费多大周折，就可以从开发阶段进入产品阶段的程序。

5.6.4　预计改动

改动几乎是每个程序都不可避免的现象。比如，开发一个旧的软件新版本，就需要对原有代码作出许多改动，不过，即使是在开发一个软件的第一版时，也不得不由于加入某些没有预计到的功能而对其进行改动。在开发软件时，应该努力作到使它很容易地进行改动。而且，越是可能的改动，越是要容易进行，把你在其中预想到的改动域隐含起来，是减少由于改动而对程序带来影响的最有力武器之一。

5.6.5　计划去掉调试帮助

调试帮助措施包括：断言、内存检查报告、打印语句等及其它一些为方便调试而编写的代码。如果所开发的软件是供自己使用的，那么把它们保留在程序中并无大碍。但是，如果是一个商用软件，那么这些措施留在程序中，则会影响其速度和占用空间等性能指标。在这种情况下，应事先作好计划，避免调试信息混在程序中，下面是几种方法。

使用版本控制。版本控制工具可以从同一源文件中开发出不同版本的软件。在开发阶段，可以设置包含全部调试辅助手段的版本控制工具，这样，到了产品阶段，就可以很容易地去掉在商用版本中所不希望出现的这些辅助手段。

使用内部预处理程序。如果在编程环境中带有预处理程序，如 C 语言，那么仅用一下编译程序开关，就可以加入或去掉这些辅助手段。可以直接使用预处理程序，也可以通过编写宏指令来进行预处理程序定义。下面是一个用 c 语言写成的，直接使用预处理程序的例子：

```
#define DEBUG
…
```

```
#ifdefined（DEBUG）
／*调试代码*／
…
#endif
```

这种思想可能有几种表现形式。不仅仅是定义 DEBUG，还可以赋给它一个值，然后再测试它的值，而不是测试它是否被定义了。用这种方法可以区分调试代码的不同层次。也可能希望某些调试用代码长期驻存在程序中，这时可以使用诸如＃if DEBUG＞0 之类的语句，把这段代码围起来，其它一些调试代码可能是有某些专门用途的，这时可以用＃if DEBUG—一PRINTER ERROR 这样的语句把这段代码围起来，在其它地方，还可能想设置调试层次，可用如下语句：

```
＃if DEBUEG＞LEVEL_A
```

要是不喜欢程序内部充斥着＃if defined（）这样的语句的话，可以用一个预处理程序宏指令来完成同一任务。以下是一个例子：

```
＃define DEBUG
＃if defined（ DEBUG）
＃define DebugCode（ code fragment）{code_fragment}
#else
#define DebugCode（code fragment）
#endif
DebugCode
 (
    statement 1;
    statement 2;
    …
    statment n;
 )
```

根据 DEBUG 是否定义，包含或不包含这段代码

与使用预处理程序的第一个例子一样，这种技术也有多种形式，可以使它更复杂一些，从而不是简单地全部包括或全部排除调试代码。

编写自己的预处理程序。如果编程语言中没有预处理程序，可以自己编写一个加入或去掉调试代码的预处理程序，这项工作是非常容易的。还要建立一个标识调试代码的约定，并编写自己的预编译程序来遵循这一约定。比如，在 Pascal 中，可以编写一个对如下关键字作出反应的预编译程序：／＃BEGIN DEBUG／和／＃END DEBUG／人并写一个批处理文件来调用这个预处理程序，然后再编译这段已经预处理过的代码。从长远观点来看，这样做可以节约许多时间，因为你不会误编译没有预处理过的代码。

保留使用调试程序。在许多情况下，可以调用一个调试子程序来进行调试工作。开发阶段

在控制返回调用程序之前，这个子程序可能进行几项操作。在最终软件中，可以用一个子程序来代替那个复杂的调试子程序，这个子程序将立即返回控制，或者在进行两个快速操作之后，返回控制。使用这种方法，对性能带来的影响很小，与编写自己的预处理程序来相比，其速度也要快得多。所以，有必要保留这个子程序在开发阶段和最终产品阶段的两个版本，以供在将来的开发和产品调试中使用。

比如，可以用一个检查传入其中指针的子程序作为开始：

```
Procedure DoSomething
(
  Pointer:PONITER_TYPE;
  ...
);
begin
{ check parameters passed in }
CheckPointer（Pointer）；  ——— 这行调用检查指针的子程序
  ...
end.
```

在开发阶段，CheckPointer（）子程序将对指针作全面检查，这项工作可能很费时，但却非常有效，它很可能是以下这个样子的：

Procdure CheCPOnter（ POinter： POINTER__TYPE）；——这个程序检查传入的每个指针，它可以在开发期间使用．完成格外检查

```
begin
      {perform check 1--maybe chech that it's not nil}
      {perfom check 3 --maybe check that what is point to isn't corrupted}
       {perform check n--...}
end;
```

当程序进入最终产品阶段时，可能并不希望所有的内务操作都与指针检查联系到一起，这寸，可以用下面这个子程序来代替刚才那个子程序：

```
Proccure CheckPointer (Pointer: POlNTER_TYPE );  ——— 本程序仅是即返回其调用程序
begin
  {no code;just return to caller}
end;
```

以上这些并不是去掉调试辅助工具的所有方案，但从提供一些在你的环境下能有效工作守案这个思路的角度来说，这些已经是足够的了。

5.6.6　尽早引入调试辅助工具

越早引入调试辅助工具，它们所起的作用也就会越大。一般说来，只有在被某一问题困扰

几次之后，你才会舍得花功夫去编写调试辅助工具，但如果你在第一次遇到问题时就这样做，或者引用一个以前遗留下的调试辅助工具，那么它将在整个项目中都会对你有很大帮助。

5.6.7　使用“防火墙”包容错误带来的危害

“防火墙”技术是一种包容危害策略，在建筑物中，防火墙的作用是防止火灾蔓延，把火隔离在一个地方。在轮船中使用分隔式水密能也是同样道理，如果船撞上了冰山，那么只有被撞的水密舱才会破损，而其它舱由于是与它分隔开来的，所以不会受到它的影响，这样就避免了由一个洞而带来的全船进水的灾难性后果。

信息隐蔽可以帮助在程序中建立防火墙。对另一个子程序的内容知道得越少，对它如何操作的假设也就越少，而假设越少，其中一个假设出错的可能性就会越小。

松散的耦合也是在程序内部修建防火墙的手段之一。两个子程序之间的耦合越松散，那么其中一个子程序中的错误影响到另外一个子程序的机会也越少。相反，如果两个子程序联系得非常紧密，那么一个子程序错误很可能会影响另外一个子程序。

在程序中建防火墙的最好办法是把某些接口标识成“安全”区边界。对穿越安全区边界的数据进行合法性检查，如果是非法的数据，就要作出合理的反应。基于这种想法的另一种技术是手术室技术，在数据被允许进入手术室之前，要对其进行消毒处理，手术室中的一切都认为是无毒安全的。当然，这个手术室是指一段代码，这样，在设计中，要作出的一个关键决定，就是在这个“手术室”中进入些什么？哪些要被放在它外面？应该把门放在哪儿？应该把哪些子程序放在安全区内，哪些放在外面？用什么对数据进行消毒？最简单的办法是在外部数据进入时对其进行消毒，但是，数据往往需要在几个层次上进行消毒，因此，有时需要进行多重消毒。

5.6.8　检查函数返回值

如果调用了一个函数，并且可以忽略函数退回值（例如，在 C 语言中，甚至不需要知道函数是否返回一个值），千万不要忽略这个返回值。要对这个值进行检查。如果不想让它出错的话，一定要对其进行检查。防错性设计的核心就是防止常错误。

对于系统函数来说，这个原则也是适用的，除非在结构设计中制定了不检查系统调用的返回码，而是在每次调用后检查错误代码。如果发现错误，C 语言中的 Perror（），或其它语言中等效的部分，能同时也查出错误个数和这个错误的说明。

5.6.9　在最终软件中保留多少防错性编程

防错性编程带来的矛盾是，在开发过程中，你希望出现错误时越引人注意越好，惹人讨厌总比冒险忽视它好得多。但在最终产品中，你却希望它越不显眼越好，程序运行不管成功与否，都要看起来十分优雅。以下是帮助你决定在最终软件中应该保留哪些防错性编程的一些原则：

保留查找在要错误的代码。首先要确定哪些域可以承受本测试到的错误，而哪些域不能。例如，你正在编写一个表格程序，程序中的屏幕更新区就可以承受未测试到的错误，因为发生这种情况的后果，不过是一个混乱的屏幕而已，而在计算部分，就不能发生这种情况。因为这会在表格中产生难以察觉的错误。绝大多数用户都宁愿忍受一个混乱的屏幕而不是错误的表格。

去掉那些无关紧要错误的代码。如果一个错误的后果是无关紧要的，那就去掉检查它的代码。在上例中，你很可能会去掉检查屏幕更新区错误的子程序。“去掉”并不是指从物理上把这

段代码删掉，它指的是版本控制预编译开关，或其它不编译那段特定代码的技术。如果不存在空间限制问题，你也可以保留这段查错代码，并让它向一个错误记录文件隐蔽地传送信息。

去掉那些引起程序终止的代码。在开发阶段，程序发现了一个错误时，你会希望这个错误越引人注意越好，以便你能修复它，通常，达到这一目的最好办法是让一个程序在发现错误时打印出错误信息然后终止。即使对于微小错误来说，这样做也是很有用的。

而在最终软件中，在程序终止前，用户总是希望有机会将其工作存盘。他们往往愿为达到这一目的而忍受一些反常现象，用户们不会感激那些使其工作付诸东流的东西，不管这些东西在调试阶段多么有用，哪怕最终极大提高了软件质量。如果程序中含有会使千百万数据丢失的调试代码，那么在最终产品中应将其去除掉。

保留那些可以使程序延缓终止的代码。同时，那些相反的代码也应该保留。如果程序中含有测试潜在致命错误的信息，那么用户会为能在它们最终发展起来之前将自己的工作有盘而感到高兴。我所使用的文字处理机在溢出内存之前会亮起“SAVE”提示灯进行警告，当发现这一情况后就立即存盘并退出。当重新启动程序时，一切又变得正常了。从理论上说，程序不应该溢出内存，而且，在用同一台机器重新启动程序运行同一文件时，它也不应该用更多内存。产生了内存溢出问题说明程序有缺欠，但是，程序员想得很周到，在程序中保留了内存检查代码，我也宁愿得到一个警告信息，而不愿失去我前面所做的工作。

保证留在程序中的错误提示信息是友好的。如果在程序中保留了内部错误提示信息，要确保它是用友好的语言表达。在我早期编程工作中，我曾经收到一个用户的电话，说她在屏幕上看到了这样的信息“你的指针地址有错，笨蛋!”幸亏她还有一些幽默感，这对我来说是很幸运的。通常的办法是通知用户存在“内部错误”，并告诉用户一个她可以投诉的电话号码。

要对防错性编程提高警惕。过多的防错性编程会带来它自身的问题，如果你对每一种可以察觉的参数传递，在每一个可以察觉的地方都进行检查，那么程序将变得臃肿而笨拙。更糟的是，附加的用于防错性编程的代码本身并非是完善无缺的，同其它代码一样，你也会在其中发现错误，而且，如果你是随意写它的，那么错误也会更多。考虑好需要在哪里预防错误，然后再使用防错性编程。

5.7　子程序参数

子程序间的接口往往是一个程序中最容易出错的部分，由 Basili 和 Perricone 进行的一项研究表明，程序中 39%的错误都是内部接口错误，即予程序间的通信错误。以下是尽量减少这类错误的一些准则：

确保实际参数与形式参数匹配。形式参数，即哑参数，是在子程序定义中说明的变量，实际参数是在调用程序中使用的变量和参数。

常见的错误是在子程序调用时变量类型有误，比如，在需要使用实型数时使用了整型数（这种情况只在像 C 这种弱类型的语言中，才会遇到。例如，在汇编语言或在 C 语言中未使用全部编译程序的全部警告时就可能产生这种问题。而在 Pascal 中，当变元是单纯输入的时候，几乎不会产生这个问题）。通常编译程序在把实参传入子程序之前，会把它转为形参的类型。如果产生这个问题，编译程序通常会产生警告。但是在某些情况下，特别是当变元既用于输入也用于输出时，你可能由于传递了错误的变元类型而受到惩罚。

要养成检查参数表中参数变元类型和注意编译程序关于变量类型不匹配警告的习惯。在C语言中，使用ANSI函数的原型，以便编译程序会自动检查变元类型，并在发现类型错误时发出警告。

按照输入一修改一输出的顺序排列参数。不要随机地或者按照字母表的顺序排列参数，应该将输入参数放在第一位，既输出又输入的参数第二位，仅供输出的参数第三位这样来排参数。这种排列方法示示子程序中操作进行的顺序——输入数据、修改数据、输出一个结果、下面是一个Ada语言中的参数排列顺序。

```
procedure InvertMatrix
(
OringinalMatrix :     in MATRIX;
ResultMatrix :       out MATRIX;
);

procedure ChangeStringCase
(
DesiredCase :            in STRING_CASE;
MixedCaseString:       in out USER_STRING
);
…
procedur PringPageNumber
(
PageNumber :     in INTEGER;
Status:              out STATUS_TYPE
);
```

这种排列约定与C语言中把被修改的参数放在首位的规定是冲突的。不过我仍然认为上述排列顺序至少对我来说是十分明智的。但如果你一直接某种特定方式对参数排序，这也是可以帮助提高程序可读性的。

如果几个子程序今使用了相似的参数，应按照不变的顺序排到这些参数。子程序中参数的排列顺序可以成为一种助记符，而不停变动的排列，会使得这些参数非常难记。比如，在C语言中，fptintf()函数与printf()函数相比，除了多了一个文件作为第一变元之外，两者其余都是一样的。而函数fputs()与函数puts()相比，也只是前者多了一个文件作为最后变元。这实在是一个糟糕的区别，因为它使得这些函数的参数的难记程度比实际要高多了。

我们来看一个例子，同样是C语言中的函数，函数Strncpy（）是按照目标字符率、源字符率和字节的最大数目来排列变元的，而函数memcpy（）是按同样的顺序来排列变元的，这种相似性使得两个函数中的参数都非常好记了。

使用所有的多数。如果向某个子程序中传入了一个参数，那就要在其中使用；如果不用它的话，就把它从子程序接口中去掉。因为出错率是随着未用参数个数的增加而升高的，一项调查表明，在没有未用参数的子程序中，有46％是完全无错的。而在含有未用参数的子程序中，

仅有 17％到 29％是完全正确的（Card，Church，Agresti 1986）

不过，这个去掉未用参数的规则有两个特例。首先，如果你使用了 c 语言中的指什函数或 Pascal 中的变量过程，那可能会有一些子程序拥有完全相同的参数表，而在这其中又可能有几个子程序没有完全用到这些参数，这是允许的。其次，当你按照某种条件对程序进行部分编译，可能会使用某些参数编译部分程序。但如果你去掉这部分是正确有效的，那这也是允许的。一般来说，如果你有充分的理由不使用某一参数的话，那就按照你想的大胆去干吧。但如果理由不是很充分的话，就要保留这个参数。

把状态和“错误”变量放在最后。根据约定，状态变量和表示程序中有错误的变量应该放在参数表的最后。这两种变量对于子程序来说是不很重要的。同时又是仅供输出的变量，因此把它们放在最后是非常明智的。

不要把子程序中的参教当作工作变量。把传入子程序中的参数用作工作变量是非常危险的。应该使用局部变量来代替它。比如，在下面这个 Pascal 程序段中，不恰当地使用了 InnutVal 这个变量来存放中间运算结果。

```
Proceoure ffemPe
        (
          VAR     InputVal :        Integer;
                  OutputVal:        Integer;
    );
  begin
    InputVal:=InputVal * CurrentMultiplier（ InputVal）;
    HputVal:= InputVal + CurrentAdder（ InputVal）
    …
    OutputVal := Inputval;
  end
```

在这个程序段中，对 Inputval 的使用是错误的，因为在程序到达最后一行时，InnutVal 不再保持它输入时的值。这时它的值是程序中计算结果的值，因此，它的名字被起错了。如果你以后在更改程序，需要用到 InPutVal 的输入值时，那很可能会在 InputVal 的值已经改变后还错误地认为它保留原有值。

该如何解决这个问题呢？给 InputVal 重新命名吗？恐怕不行。因为假如你将其命名为 WorkingVal 的话，那么这个名称是无法表示出它的值是来自于程序以外这个事实的。你还可以给它起一个诸如 InputvalThatBeComesAWorkinsVal 之类荒唐的名字或者干脆称之为 X 或者 Val，但无论哪一种办法，看起来都不是很好。

一个更好的办法是通过显式使用工作变量来避免将来或现在可能由于上述原因而带来的问题。下面的这个程序段表明了这项技术：

```
procedure Sample
   (
     VAR   InputVal:        Inteqer;
           OutputVal:     Inteqer;
   );
```

```
var
   WorkingVal: Integer;
begin
   WorkingVal := InputVal;
   WorkingVal := WorkingVal * CurrentMultipier (WorkingVal );
   WorkingVal := WorkingVal + CurrentAdder( WorkingVal );
   ...
        ———  如果需要在这里或其他地方使用输入原始值，它还存在
   ...
   OutputVal := WorkingVal;
end;
```

通过引入新变量 WokingVal，即保留了 InputVal 的作用，还消除了在错误的时间使用 InputVal 中值的可能性。在 Ada 语言中，这项原则是通过编译程序进行强化的。如果你给某个参数的变量名前缀是 in，则不允许在函数中改变这个参数的值。不过，不要利用这个理由来解释把一个变量很具文学性地命名为 WorkingVal，因为这是一个过于模棱两可的名字，我们之所以这样使用它，仅仅是为了使它在这里的作用清楚一些。

在 Fortran 语言中，使用工作变量是一个非常好的习惯。如果在调用于程序参数表中的变量被调用于程序改动了，那么它在调用程序中的值也将被改变。在任何语言中，把输入值赋给工作变量的同时都强调了它的来源。它避免了从参数表中来的变量被偶然改变的可能性。

同样的技术也被用于保持全局变量的值。如果你需要为全局变量计算一个新值，那应该在计算的最后把最终值赋给全局变量，而不要把中间值赋给它。

说明参数的接口假设。如果假定被传入子程序的数据具有某种特性，那么需要对这个假设作出说明。在子程序中和在调用程序的地方都需要说明这一假设，这绝不是浪费时间。不要等到写完子程序后再回过头来说明这些假设，因为那时很可能你已经忘记这些假设了。如果能在代码中放入断言的话，那么其效果要好于说明这些假设。

- 关于参数接口的哪些假设需要作出说明呢？
- 参数是仅供输入的，修改的还是仅供输出的？
- 数值参数的单位（英尺、码、还是米等）。
- 如果没有使用枚举型参数的话，应指出状态参数和错误变量值的意义。
- 预期的取值范围。
- 永远不该出现的某些特定值。

应该把一个子程序中的参数个数限制在 7 个左右。7 对于人的理解能力来说是一个富于魔力的数字。心理学研究表明人类很难一次记住超过 7 个方面的信息，这个发现被应用到不计其数的领域中，因此，如果一个子程序中的参数个数超过 7 个，人们就很难记住，这样会更安全一些。

在实践中，把一个子程序中的参数个数限制在多少，取决于你所用的程序语言是如何处理复杂数据结构的。如果你所用的是一种支持结构化数据的先进语言，你可以传递一个含有 13 个域的数据结构，而把它只看成是一个独立的信息。如果你使用的是一种比较原始落后的语言，那你就不得不把这个复合数据结构分解成 13 个单独参数分别传送。

如果你发现自己总是在传递比较多的变元，则说明程序之间的耦合就有些过于紧密了。这

时应重新设计子程序或子程序群，来降低耦合的紧密性。如果你把同一数据传给不同的子程序，应当把这些子程序组织成一个模块，并把那些经常使用的数据当做模块数据。

考虑建一个关于输入、修改和输出参数的命名约定。如果发现区分输入，修改和输出参数是非常重要的，则你可以建立一个关于命名的约定，以便区分它们，比如可以用 i_,m_,o_作前缀。要是你不觉得冗长的话，可以用 INPUT，MODIFY 和 OUTPUT 来作前缀。

仅传递子理序器要的那部分结构化变量。如同在 5.4 节关于耦合中讨论过的那样：如果子程序不是使用结构化变量中绝大部分的话，那么就只传递它所用得到的那一部分。如果你精确规定了接口，在别的地方再调用这个子程序会容易些。精确的接口可以降低子程序间的耦合程度，从而提高子程序的使用灵活性。

不过，当我们使用抽象数据类型（ADT）时，这一精确接口规则使不适用了。这种数据类型要求我们跟踪结构化变量，但这时你最好不要过分注意结构内部，在这种情况下，应把抽象数据类型子程序设计成将整个记录作为一个参数来接收的，这可以使你把这个记录当成 ADT 子程序之外的一个目标，并把这个记录的抽象水平保持在与 ADT 子程序的相同高度上，如果你通过利用其中的每一个域来打开结构，那你就丧失了由 ADT 所带来的抽象性。

不要对参数传递作出任何设想。有些程序员总是担心与参数传递有关的内部操作，并绕过高级语言的参数传递机制，这样做是非常危险的，而且使得程序的可移植性变坏。参数一般是通过系统堆栈传输的，但这决不是系统传递参数的唯一方式。即使是以堆栈为基础的传递机制，这些参数的传递顺序也是不同的，而且每一个参数的字长都会有不同程度的改变。如果你直接与参数打交道，事实上就已经注定了你的程序不可能在另一个机器上运行。

5.8　使用函数

像 C、Pascal 和 Ada 等先进的语言，都同时支持函数和过程，函数是返回一个值的子程序，而过程则是不返回值的子程序。

5.8.1　什么时侯使用函数，什么时侯使用过程

激进者认为函数应该像数学中的函数一样，只返回一个值。这意味着函数应接受唯一的输入数据并返回一个唯一的值。这种函数总是以它所返回的值来命名的，比如 sin()，cos()，CustomerID()等等，而过程对于输入、修改、输出参数的个数则没有限制。

公用编程法是指把一个函数当作过程来使用，并返回一个状态变量。从逻辑上说，它是一个过程，但由于它只返回一个值，因此从名义上说，它又是函数。你可能在语句中使用过如下一个称为 Fotmatoutput（）的过程：

```
if (Formatoutput(Input，Formatting，Output)　=　Success　)　then　...
```

在这个例子中，从它输出参数的角度来看，是一个过程。但是从纯技术角度来说，因为程序返回一个值，它又是一个函数。这是使用函数的合法方式吗？从保护这个方法的角度出发，你可以认为这个函数返回一个值与这个子程序的主要目的——格式化输击无关。从这个观点来看，虽然它名义上是一个函数，但它运行起来更像是过程。如果一贯使用这种技术的话，那么用返回值来表示这个过程的成功与否并不会使人感到困惑。

一个替换的方案是建立一个用状态变量作为显式参数的子程序，从而产生了如下所示的

代码段：

FormatOutput（ Input， Formatting， Output， status ）

if（Status ＝ Success） then …

我更赞成使用第二种方法，这倒并不是因为我是个坚持严格区分函数与过程的教条主义者，而是因为它明确区分了调用和测试状态变量值的子程序。把调用和测试状态值的语句写成一行增加了语句的代码密度，也增加了其复杂性。以下这种函数用法也是很好的：

```
Status：＝FormatOutput（ Input， Formatting， output ）
if( status = success ) then...
```

5.8.2 由函数带来的独特危险

使用函数产生了可能不恰当值的危险，这常常是函数有几条可能的路径，而其中一条路径又没有返回一个值时产生的。在建立一函数时，应该在心中执行每一条路径，以确认函数在所有情况下都可以返回一个值。

5.9 宏子程序

特殊情况下，用预处理程序宏调用生成子程序。下面的规则和例子仅限于在 C 中使用预处理程序的情况。如果你使用的是其它语言或处理程序，应调整这些规则以适应你的要求：

把宏指令表达式括在括号中。由于宏指令及其变元被扩展到了代码中，应保证它们是按照你想要的方式被扩展的。在下面这个宏指令中包含了一种最常见的错误：

```
#define product（a，b）a*b
```

这个宏指令的问题是，如果你向其中传了一个非基本数据（无论对 a 还是 b），它都无法正确地进行乘法运算。如果你使用这个表达式来算（x＋1,x＋2），它会把它扩展到 x+1*y+2，由于乘法运算对加法具有优先佳，因此输出结果并不是你想要的结果；一个好一些但并非完美的同样功能的宏指令如下：

```
＃define　product　（ a，b） （a）*（b）
```

这一次的情况要稍好些，但还没有完全正确，如果你在 preduct（）中使用比乘法具有优先权的因子。这个乘运算还是要被分割开，为防止这一点，可以把整个表达式放人括号：

```
＃define　preduct　( a，b)　((a)*(b))
```

用斜线将多重语句宏指令包围起来。一个宏指令可能具有多重语句，如果你把它当作多重语句来对待的话就会产生问题，以下是一个会产生麻烦的宏指令例子：

```
＃define　LookupEntry (Key，Index )\
  Index=(key-10)/5;\
  Index=min（Index,MAX_INDEX）;\
  Index=max( Index,MIN_INDEX);
  …
  for　( Entrycount=0;　Entrycount<NumEntries;　Entrycount ++)
```

用子程序的命名方法来给扩展为代码的宏命名，以便在必要时用子程序代替它。在 C 语言中给宏命名的规定是应该使用大写字母来命名，如果可以使用子程序来代替它，那么就使用子程序命名规定来代替 C 中的宏命名规定。这样，你只要改变所涉及的子程序，就可以非常容易地对宏和子程序进行互相替换。

采纳这种建议也会带来一些危险，如果你一直使用＋＋和一，当你误把宏当作子程序来用时就会产生副作用，考虑到副作用带来的问题。如果你采纳这个建议避免副作用的话，你就可以干得更好。

5.8.2　检查表

高质量的子程序

总体问题

- 创建子程序的理由充分吗？
- 如果把一个子程序中的某些部分独立成另一个子程序会更好的话，你这样做了吗？
- 是否用了明显而清楚的动宾词组对过程进行命名？是否是用返回值的描述来命名函数？
- 子程序的名称是否描述了它做的所有工作？
- 子程序的内聚性是不是很强的功能内聚性？它只做一件工作并做得很好吗？
- 子程序的耦合是不是松散的？两个子程序之间的联系是不是小规模、密切、可见和灵活的？
- 子程序的长度是不是它的功能和逻辑自然地决定的：而不是由人为标准决定的？

防错性编程

- 断言是否用于验证假设？
- 子程序对于非法输入数据进行防护了吗？
- 子程序是否能很好地进行程序终止？
- 子程序是否能很好地处理修改情况？
- 是否不用很麻烦地启用或去掉调试帮助？
- 是否信息隐蔽、松散耦合，以及使用“防火墙”数据检查，以使得它不影响子程序之外的代码？

- 子程序是否检查返回值？
- 产品代码中的防错性代码是否帮助用户，而不是程序员？

参数传递问题

- 形式参数与实际参数匹配吗？
- 子程序中参数的排列合理吗？与相似子程序中的参数排列顺序匹配吗？
- 接口假设说明了吗？
- 子程序中参数个数是不是 7 个或者更少，
- 是否只传递了结构化变量中另一个子程序用得到的部分？
- 是否用到了每一个输入参数？
- 是否用到了每一个输出参数?
- 如果子程序是一函数，是否在所有情况下它都会返回一个值？

5.10　小　结

- 建立子程序的最重要原因是加强可管理性（即降低复杂性），其它原因还有节省空间、改进正确性、可靠性、可修改性等等。
- 强调强内聚性和松散耦合的首要原因是它们提供了较高层次的抽象性，你可以认为一个具备这种特性的子程序运行是独立的，这可以使你集中精力完成其它任务。
- 有些情况下，放入子程序而带来巨大收益的操作可能是非常简单的。
- 子程序的名称表明了它的质量，如果名称不好但却是精确的，那么说明它的设计也是非常令人遗憾的。如果一个子程序的名称既不好又不精确，那它根本就无法告诉你程序作了些什么。无论哪种情况，都说明程序需要改进。
- 防错性编程可以使错误更容易被发现和修复，对最终软件的危害性显著减小。

第六章　模块化设计

目录

相关章节

高质量子程序的特点：见第 5 章
高层次设计：见第 7 章
抽象数据类型：见第 12.3 节

“你已经把你的子程序放入我的模块中”
“不，你已经围绕着我的子程序设计好了模块”

人们对于子程序和模块之间的区别往往不很注意，但事实上应该充分了解它们之间的区别，以便尽可能地利用模块所带来的便利。

“Routine”和“Modu1e”这两个单词的意义是很灵活的，在不同的环境下，它们之间的区别可能会变化很大。在本书中，子程序是具有一定功能的，可以调用的函数或过程，关于这一点在第五章已经论述过了。

而模块则是指数据及作用于数据的子程序的集合。模块也可能是指，可以提供一系列互相联系功能的子程序集合，而这些子程序之间不一定有公共的数据。模块的例子有：C 语言中的源文件，某些 Pascal 版本中的单元及 Ada 语言中的“包”等等。如果你所使用的语言不直接支持模块，那么可以通过用分别编程技术来模仿它，这也可以得到许多由模块带来的优点。

6.1　模块化：内聚性与耦合性

“模块化”同时涉及到子程序设计和模块设计。这是一种值得研究的，非常有用的思想方法。

在 1981 年出版的《Software Maintenance Guidebook》一书中，Glass 和 Noiseux 认为模块化给维护性带来的好处要比给结构带来的好处多得多，它是提高维护性的最重要因素。Lientz 和 Swanson 在《Software Maintenance Management》一书中引用的一项研究表明，89%的代码使用者认为使用模块化编程改进了维护性（1980）。在一次理解测验中发现，采用模块化设计程序的可读性要比不采用这种设计的程序可读性高 15%（1979）。

模块化设计的目标是使每个子程序都成为一个“黑盒子”，你知道进入盒子和从盒子里出来的是什么，却不知道里边发生什么。它的接口非常简单，功能明确，对任何一个特定的输入，

你都可以精确地预测它相应的输出结果。如果你的子程序像一个黑盒子，那么它将是高度模块化的，其功能明确，接口简单，使用也灵活。

使用单独一个子程序是很难达到这一目的的，这也正是引入模块的原因。一组子程序常常要使用一套公用的数据，在这种情况下，由于子程序间要共享数据，因而它们不是高度模块化的，作为一个单个的子程序，它们的接口也不简单。但是，作为一个整体，这组子程序则完全有可能为程序的其它部分提供一个简单的接口，也完全有可能达到高度模块化这一目标。

6.1.1　模块内聚性

模块的内聚性准则，与单个子程序的内聚性准则一样，都是十分简单的。一个模块应该提供一组相互联系的服务。

比如一个进行驾驶控制模拟的模块，其中应含有描述汽车目前的控制设置和目前速度的数据。它可以提供像设定速度、恢复到刚才的速度、刹车等功能。在其内部，可能还有附加的子程序和数据来支持这些功能，但是，模块外的子程序则不需对它们有任何了解。如果这样的话，那么这个模块的内聚性将是非常强的，因为模块中的每个子程序都是为提供驾驶控制模拟服务的。

再比如一个进行三角函数计算的子程序，模块中可能含有 Sin()、Cos()、Tan()、Arcsin()等全部密切相关的三角函数子程序。如果这些子程序都是标准的三角函数，那么它们无须共享数据，但这些子程序间仍然是有联系的，因此这个模块的内聚性仍然是非常强的。

下面是一个内聚性不好的模块例子，设想一个模块中含有几个子程序为实现一个堆栈：init_stack()、push()和 pop()；模块中同时还含有格式化报告数据和定义子程序中用到的所有全局数据的子程序。很难看出堆栈与报告子程序或全局数据部分有什么联系，因此模块的内聚性是很差的。这些子程序应该按照模块中心的原则进行重新组织。

在上例中，对模块内聚性的估计是以模块数据和功能为基础进行的。它是在把模块作为一个整体的层次上进行的。因而，模块中的子程序并不会因为模块内聚性好而一定具有良好的内聚性。所以模块中的每个子程序设计，也要以保证良好内聚性为准则。关于这方面的问题，见 5.3 节“强内聚性”。

6.1.2　模块耦合

模块与程序其它部分间的耦合标准与子程序间的耦合标准也是类似的。模块应被设计成可以提供一整套功能，以便程序的其它部分与它清楚地相互作用。

在上述的驾驶控制例子中，模块担任了如下功能：SetSpeed()、GetCurrentSettings()、ResumeFormerSpeed()和 Deactivate()。这是一套完整的功能，因而程序的其它部分与它的相互作用完全是通过规定的公用接口进行的。

如果模块所提供的功能是不完善的，其它子程序可能被迫对其内部数据进行读写操作。这就打开了黑盒子盖而使其成为透明的了，这实际上破坏了模块化。结构化设计的先驱 Larry Constantine 指出，模块提供的功能必须是完整的，以便它的调用者们可以各取所需。

为了设计出强内聚而又松散耦合的模块，必须在设计模块和设计单个子程序的标准之间进行平衡与折衷。降低子程序之间耦合性的重要措施之一，就是尽可能减少使用全局变量。而创建模块的原因之一则是为了让子程序可以共享数据；你若想使同一模块中的子程序不必通过参

数表进行传递，可以采用对其中所有数据进行直接存取来实现。

从所有模块中的子程序可以对它进行存取的角度来说，模块中数据很像是全局数据。但从不是程序中所有的子程序都可以对它进行存取的角度来说，它又不像是全局数据，它只对模块中的子程序来说，才是可以存取的。因此，在模块设计中的最重要决定之一，便是决定哪个子程序需要对模块中数据进行直接存取。如果某个子程序仅仅是由于可以对模块中数据进行存取的原因才留在模块中的，那么，它应该被从模块中去掉。

6.2　信息隐蔽

如果你阅读了书中所有推荐参阅文献的注释，你就会发现其中有 400 多个是关于信息隐蔽的。拥有这么多参考文献的内容一定是非常重要的吧？是的，它的确非常重要。

进行信息隐蔽的设计思想贯穿了软件开发的每一个层次，从使用命名的常量而不是使用自由常量到子程序设计、模块设计和整个程序设计。由于这一思想往往是在模块这一层次得到最充分体现的。因此，我们在本章详细讨论它。

信息隐蔽是为数不多的几个在实践中无可辩驳地证明了自己价值的理论之一（Boehm 1987）。研究发现，应用信息隐蔽进行设计的大型程序容易更改指数要比没采用这一技术的高 4 倍。同时，信息隐蔽也是结构化设计和面向对象设什的基础之一。在结构化设计中，黑盒子思想便来源于信息隐蔽。在面向对象设计中，也是信息隐蔽引发了抽象化和封装化的设计思想。

6.2.1　保密

信息隐蔽中的关键概念是“保密”。每一个模块的最大特点都是通过设计和实现，使它对其它模块保密。这个秘密或许是可能被改动的区域、某个文件的格式化、一个数据结构的实现方式、或是一个需要与程序其它部分隔离开来，以便其中的错误产生的危害最小的区域。模块的作用是将自己的信息隐蔽起来以保卫自己的隐私权。信息隐蔽的另一个称谓是“封装”，其意思是一个外表与内容不一样的盒子。

无论管它叫什么，信息隐蔽都是设计子程序和模块的一种方法，它对模块的意义更重要些。当你隐藏秘密时，你就设计了一组存取同一套数据的子程序。对一个系统的改动可能涉及到几个子程序，但是，它只应涉及一个模块。

在设计模块时，一项重要任务就是决定哪些特性应该是对模块外部公开的，哪些应该是作为秘密隐藏起来的，一个模块中可能使用 25 个子程序而只暴露出其中的 5 个，其余 20 个都只在内部使用。模块中也可能用到了几个数据结构，但却把它们全部隐藏起来。它可能会也可能不会提供把数据结构信息通知给程序其余部分的子程序。模块设计的这一方面一般被称作“可见性”，因为它主要涉及了模块的功能特性是否是对外部分开或暴露的。

模块的接口应该尽可能少地暴露它的内部内容。一个模块应该像是一座冰山，你只看到它的一角，而它其余 7/8 的部分则藏在水面下。

与设计的其它方面一样，设计模块的接口也是一个逐渐的过程，如果接口在第一次是不正确的，可以再试几次直到它稳定下来；如果它稳定不下来，那么就需要重新设计它。

可以用各种不同的图形来代表模块。模块表示图的关键是，它应该区分开仅供模块内部使用的功能和对外开放的功能。这种图形通常称之为“积木图”，是由 Erody Boock 在开发 Ada

语言过程中提出来的。图 6-1 表示出了一种模块图。

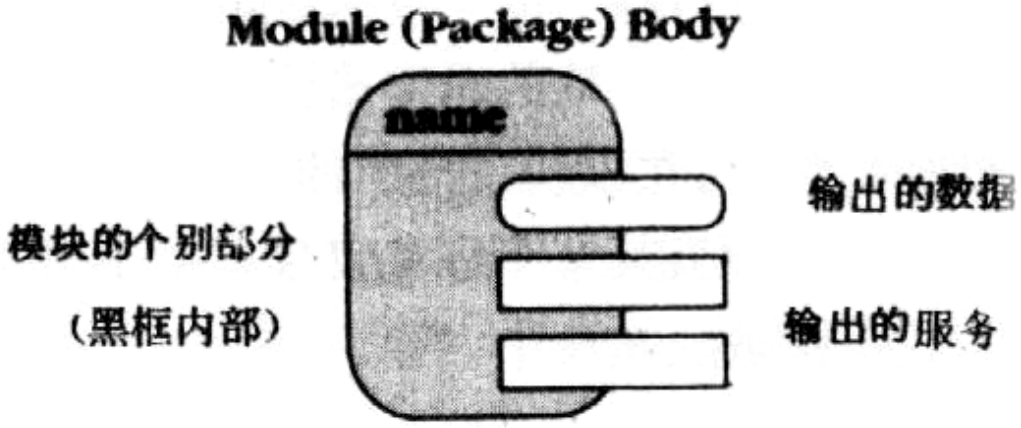


图 6-1　一个模块中公用和个别部分

其中公用部分是矩形块，个别部分如黑盒子那样表示。

信息隐蔽不必暗示出一个系统的形状；系统可能具有分层结构，也可能像图 6-2 中所示那样具有网状结构。

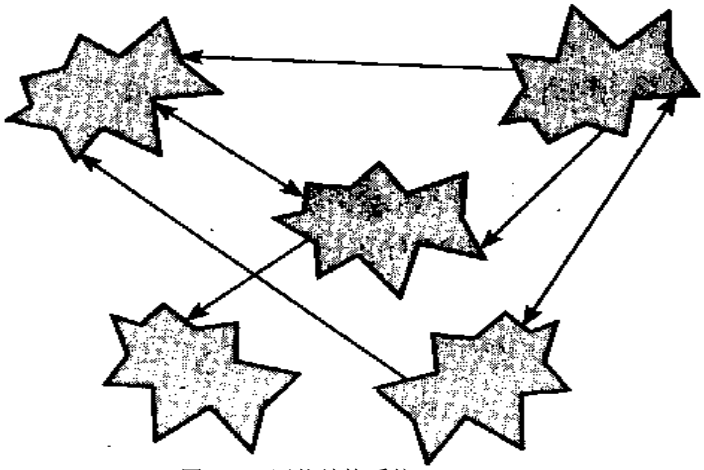


图 6-2　网状结构系统

在网状结构中，你只要规定哪些模块可以与其它模块通信，这种特定的通信是如何进行的，然后再进行联接的就可以了，如图 6-3 所示，积木图也可以用在网状结构中。

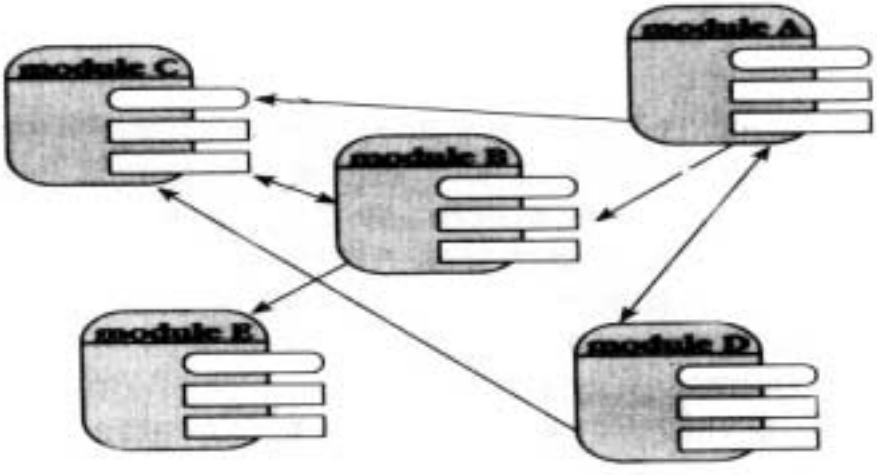


图 6-3　用包含信息隐蔽思想的符号表示网状系统

6.2.2　信息隐蔽举例

几年前我曾写了一个中型系统（有 20K 行代码），在其中广泛使用了链表结构。问题域是由数据结点构成的，每一个结点又与亚坐标、实坐标和等同点相联接。由于我选用了链表结构，因此在程序中到处都是类似这样的语句：

```
node = node.next
and
phone = node.data.phone
```

这些语句直接对链表数据结构进行操作。尽管链表非常自然地将问题进行了模块化，但是这种方法对内存的使用效率却非常低，于是我想使用整型数组索引来代替内存指针，因为这样可以提高内存利用率，并且为在其它区域进行性能优化创造机会。但是，由于刚才提到的那种编码语句充满了程序，因而修改工作非常困难。因为我无法在 20000 多行代码中把它们一一找出来。如果当初我采用了含有如下存取子程序的模块的话，我只要在一个地方即存取子程序中改动代码就可能了。

```
node = NearestNeighbor(node)
phone = EmergencyContact(node)
```

我到底赢得了多少内存？我将会赢得或者失去多少速度，我不知道，但是如果当初我隐含了数据结构的细节并且使用了存取子程序，我就可以很容易地找到答案。而且，我还可以尝试一下另外几种方法。我本来可以从许多方案中挑选一个最好的，可是，由于我把数据结构的细节暴露给了整个程序，我不得不使用我所厌恶的方案。

除了方便修改，隐含复杂数据结构细节的另一个重要原因是：隐含细节可以澄清你编写某段代码的意图。在上例中，一个富有经验的程序员不难读懂下面这条语句的：

```
node = node.next
```

显然，这个语句指的是一个链表结构，但除此之外，它什么也不能告诉你。然而，一个像 node = NearestNeighbour(node)这样的存取子程序，则清楚描述了链表所代表的内容，因而这是很有用的，并且提醒你应该对 node 这个名称进行改进（node 与其邻居有什么关系），node = node.text 这样的语句与实际相脱离，你根本无法想到应该改进它们的名称以说明实际问题。

隐含数据结构的最后一个原因是出于对可靠性的考虑。如果你用一个专门的子程序来存取数据结构，你只需在其中设置一个安全验证就可以了。否则，你就不得不在所有这个子程序访问变量的地方设置安全验证。比如，如果你使用了链表，并且想读取链表中的下一个元素，并且要注意不超过链表的最后一个元素，你可能用如下的代码：

```
if   ( node.text<>null )   then
node = node.text
```

如果在某种情形下，为了更谨慎一些，你可以使用如下代码：

```
if   ( node<>null )   then
     if ( node.next<>null )   then
```

```
node = node.text
```

如果你的程序中充斥着 node=node.next 这样的语句，你可能要在其中某些地方进行测试而跳过其余部分。但是，如果这个操作是独立在一个子程序调用中：

```
node=NearestNeighbor(node)
```

那么，你只要在子程序中一个地方进行测试，那么这一测试就会在整个程序中都起作用。如果有时你想在整个程序中都对使用 node 的地方进行测试，也很容易漏掉其中某些地方。然而，如果你把这一操作独立在一个子程序中，那么是不可能有遗漏的，因为此时这项工作完全是自动进行的。

隐含数据结构细节的另一个好处是容易调试，比如，你发现 node 值在某处变得有问题了，但却不知道是在哪里。如果存取 node 的代码充斥了整个程序的话，那么找到出问题的地方不亚于大海捞针。但如果它是被孤立在一个子程序中的话，那么你可以在其中加入一段检查 node 的调试码，从而在每次存取它时都进行测试，这样事情就解决了。

应用存取子程序最后一个优点是，可以使所有对数据的存取所遵循的是一种平行的组织形式；或者通过存取子程序、或者直接对数据进行存取，不会两者兼而有之。当然，在负责数据的模块内部，对数据的存取都是直接的，在这种情况下失去平行性是不可避免的，这样做的目的是不在公共场合吹脏肥皂泡。这常常伴随着对在存取程序中进行直接数据操作这一拙劣设计的隐含。

6.2.3　常见需要隐含的信息

在你所从事的项目中，你可能与不计其数需要隐含的信息打交道，但是，其中只有几种是你要反复遇到的：

- 容易被改动的区域
- 复杂的数据
- 复杂的逻辑
- 在编程语言层次上的操作

以上每一项都将在下面的部分中给予详细论述。

容易被改动的区域

容易改动是好的程序设计中一项最富于挑战性的工作。目的是将不稳定的区域孤立起来，以便使改动带来的影响仅限于一个模块中。以下是你在为应付改动的工作中要遵循的步骤。

1. 识别出那些可能被改动的地方。如果分析工作做得很好的话，其中应该附有可能改动的地方和改动内容的明细表。在这种情况下，找出可能的改动是非常容易的。如果需求分析中没有进行这项工作，可以参阅下面关于在任何项目中都可能被改动的区域的讨论。
2. 把可能被改动的地方分离出来。把第一步中发现的每一个可能改动的地方分隔到自己的模块中，或者将其与其它可能一起被改动的要素一起，独立到一个模块中。
3. 独立可能被改动的地方。应把模块间的接口设计成对可能变动不敏感，同时，接口应该把变动限制在模块内部，外部不会受到内部变动影响。而其它调用这个被改动过模块的模块，不应感受到这个模块被修改过。模块的接口应该能保护模块的隐私权。

以下是一些可能变动的区域：

对硬件有依赖的地方。对于监视器、打印机、绘图机等，要清楚在尺寸、颜色、控制代码、图形能力及内存等方面可能的变化。其余对硬件有依赖性的方面包括与磁盘、磁带、通讯口、声音器件等接口的变化等等。

输入和输出。在比原始的硬件接口稍高一些的设计层次上，输入/输出是另外一个反复无常的区域。如果某一应用产生它自己的数据文件，那么当这一应用变得复杂起来时，文件的格式可能也要变化。用户层次上的输入和输出格式也有可能变化，比如，在打印纸上边界的位置、每页上边界的数量、域的排列顺序等等。总之，检查所有的外部接口以寻找可能的变化是个好主意。

非标准语言特性。如果在程序中使用了非标准扩展，应该把这些扩展隐含在一个模块中，以便当运行环境变化时你可以很容易地替换它。同样，如果你使用了不是在所有环境下都存在的库子程序，应该把实际的库子程序放在另一个环境下也可以使用的接口后面。

难于设计和实现的域。最好把难于设计和实现的域隐含起来，因为此处的工作可能作得很糟，你可能不得不返工。把它们分隔起来，以便使由于拙劣设计或实现对系统所带来的危害最小。

状态变量。状态变量指示程序的状态，往往比其它数据更容易被改动。在典型的情形下，你可能最初把某一错误状态变量定义成逻辑变量。但后来又发现如果把它赋成具有 NoError，WarningError 和 FatalError 三个值的枚举型变量来实现会更好。

你至少可以在使用状态变量时，加上两个层次的灵活性和可读性。

- 不要使用逻辑型变量作为状态变量，应使用枚举型变量。赋予状态变量一种新状态是非常常见的，给枚举型变量赋一个新的类型只需要重新编译一次，而对于逻辑型变量则需要重新编写每行检查状态变量的代码，谁难谁易是很明显的。
- 使用存取子程序检查变量，而不要对其直接检查，通过检查存取子程序而不是状态变量，可以进行更复杂的状态测试。例如，如果想检查一个错误状态变量和一个当前函数状态变量，那么把测试隐含在子程序中来进行，要比用充斥着程序的复杂代码进行测试容易得多。

数据规模限制。如果你说明一个数组中含有 15 个元素，那么你就把系统不需要的信息暴露给了它。应该保护其隐私权，信息隐蔽并不总是意味着把一系列功能装入模块这类复杂的工作，有时，它简单到就是用一个像 MAX_EMPLOYEES 之类的常量来代替 15，以便隐含它。

商业规则。商业规则指法律、政策、规定、惯例等编入一个计算机系统中的东西。如果你在编写一个工资发放系统，你可能把 IRS 关于允许的扣留数和估计税率等规则编入程序。其余附加的规则是由工会规定的关于加班率、节假日付酬等方面的规定。如果你正在编写一个引用保险率的软件，其规定来源于州关于信誉、实际保险率等的管理规定。

这些规定往往是数据处理系统中频繁变动的部分。因为国会可能修改法律，保险公司会调整保险率。如果你遵从信息隐蔽原则，那么当规则变动时，建立在这些规则上的逻辑关系不会完全垮掉。这些逻辑关系会隐藏在系统中唯一一个阴暗角落里，直到需对其作出改动为止。

预防到改动。当考虑一个系统中潜在的改动时，应该按照使得改动范围或大小与其改动可能性成反比的原则来设计系统。如果改动很可能发生，要确保系统可以容易地容纳这一特征。只有极其不可能发生的变动，才应该被允许在变动时，会影响到系统中一个以上的模块。

一个寻找可能发生变动域的技术是，首先分析程序中可能会被用户用到的最小的子单元，这些子单元组成了程序的核心，而且很可能被改变。其次，规定对系统的最小增值。它们可以小到看起来完全是琐碎的程度。这些潜在改进域组成了对系统的潜在改进。应使用信息隐蔽原则对这些域进行设计。首先分析核心，可以发现哪些要素事实上是后加上的，从而从那里推测并隐含改进。

复杂的数据

所有的复杂数据都很可能被改动；如果它很复杂而对它使用得又很多，那么在实现层次上与其打过交道后，可能会发现实现它的更好方式。如果应用信息隐蔽来隐含数据实现，就可以付出较少的努力而获得更好的实现方法。如果不是这样，那么你每次与这些数据打交道时，你可能都会在后悔，如果当初进行了信息隐蔽，改动实现将会是多么容易啊！

对复杂数据的使用程度，主要取决于程序。如果是一个只有几百行代码的小程序，你想在其中对变量进行直接操作，那就这样干吧，这样可能影响程序，但也可能不会。在担心由于对数据直接操作而带来的维护问题之前，应首先考虑这个小程序的特点。如果你正在编写一个大一些的程序或使用了全局数据，那么就该考虑使用存取子程序。

复杂的逻辑

隐含复杂的逻辑可以改善程序的可读性。复杂的逻辑并不总是程序的最主要方面，把它隐含起来可以使得子程序的活动更清楚。与复杂数据一样，复杂逻辑也是很可能变动的部分。所以，把程序的其它部分从这种变动里隔离出去是非常有益的。在某些情况下，你可以将所使用的逻辑种类隐含起来，例如，你可以通过一个大的 if 语句、case 语句或查表方式来进行测试。除了这些进行测试的代码外，其余的代码不需要知道这些细节。如果程序中的其余代码只需要知道结果，那么你就应该仅仅告诉它们结果。

在程序语言层次上的操作

你的程序越是像一个实际问题的解决方案，它就越是不像程序语言结构的组合，那么，其质量也就越好，应该把过于专业化的信息隐含起来，比如，下面的语句：

```
EmployeeID = EmployeeID+1
CurrentEmployee ＝ EmployeeList [ EmployeeID ]
```

这是一段很不错的程序，但是它是用过于专业化的语言来表达的，应该用较高程度抽象的语言来进行这个操作：

```
CurrentEmployee = NextAvailableEmployee()
```

或者用：

```
CurrentEmployee = NextAvailableEmployee( EmployeeList, EmployeeID )
```

通过加入一个隐含了用专业化语言解释正在发生什么的子程序，使得在一个更高的抽象层次上处理这个问题。这使得你的意图更清楚，而且使得代码更容易理解和改动了。

如果用图表来实现一个排序问题。函数 HighestPriorityEvent()，LowestPrioritryEvent()和 NextEvent()是抽象函数, 隐含了实现细节；而 FrontOfQueue(), BackOfQueue()和 NextInQueue()

并没有隐含多少细节，因为它们提到了实现，暴露了它们该隐藏的秘密。

一般来说，在设计一组在程序语言语句层次上操作数据的子程序时，应该把对数据操作隐含在子程序组中，这样程序的其余部分就可能在比较抽象的层次上处理问题了。

6.2.4　信息隐蔽的障碍

绝大多数信息隐蔽障碍都是心理上的，它主要来自于在使用其它技术时形成的习惯。但在某些情况下，信息隐蔽也的确是不可能的，而一些看起来像是隐蔽障碍的东西，但仅仅是借口而已。

信息过度分散

信息隐蔽的一个常见障碍是系统中信息过于分散。比如在一个系统中到处分布着常数100。把 100 当作一个常数，降低了引用它的集中程度。如果把信息隐蔽在一个地方会更好，因为这样它的值将只在一个地方改变。

另一个信息过于分散的例子是程序中分布着与用户交互的接口。如果需要改变交互方式，比如，从命令行方式改为格式驱动方式，那么所有的代码事实上都要被改动。因此，最好把用户交互接口放入一个单独的模块中，这样，你不必影响到整个系统就可以对交互方式进行改动。

而还有一个例子则是全局数据结构，比如，一个在整个系统中四处被存取的拥有多达 1000 个元素的雇员数据数组。如果程序直接使用这个全局数据，那么这个数据结构的实现信息——它是一个数组且拥有最多 1000 个元素——将充斥着整个程序。如果这个程序只通过存取子程序来使用这个数据结构，那么就只有这个存取子程序才知道这些细节。

交叉依赖

一个不易察觉的信息隐蔽障碍是交叉依赖。比如模块 A 中的某一部分调用了模块 B 中的一个子程序，而模块 B 中又有一部分调用了模块 A 中的子程序。应避免这种交叉依赖现象。因为只有在两者都已准备好的情况下，你才能测试其中的一个。当程序被覆盖时，必须使 A 和 B 同时驻留在内存中，才能避免系统失败。通过寻找两个模块中被其它模块使用的部分，把这些部分放入新的模块 A’和 B’中，用模块 A 和 B 中的其余部分来调用 A’和 B’，基本上可以消除这一问题。

误把模块数据当成全局数据

如果你是个谨慎的程序员，那么信息隐蔽的障碍之一便是误把模块数据当作全局数据而避免使用它，因为要避免由于使用全局数据而带来的麻烦。但是，如同在 6.1 节“模块化：内聚性与耦合性”中所说的那样，这两种数据是不同的。由于只有在模块中的子程序才可以对其进行存取，因而由模块数据带来的麻烦要比全局数据小得多。

如果不使用模块数据，就不会知道了解由模块所带来的巨大收益。如果一个子程序向模块传递了只有它才能处理的数据的话，那么就不该由模块来承担拥有数据集合并对其进行操作的罪责。比如，在前面列举的建议利用如下语句来提高抽象程度的例子中：

```
CurrentEmployee = NextAvaliableEmployee()
```

或使用：

CurrentEmployee=NextAvailableEmployee(EmployeeList, EmployeeID)

这两个赋值语句间的区别是：在第一种情形下，NextAvailableEmployee()拥有关于雇员表和目前表中的入口是哪一个入口的信息，而在第二种情况下，NextAvailableEmployee (EmployeeList, EmployeeID)只是从向它传递数据的子程序中借用这些信息。当你使用NextAvailableEmployee()时，为了提供全部的抽象能力，不必担心它所需要的数据，只要记住它负责自己的问题就可以了。

全局数据主要会产生两个问题：(1)一个子程序在对其进行操作时并不知道其它子程序也在对它进行操作；(2)这个子程序知道其它子程序也在对其进行操作，但不知道它们对它干了什么。而模块数据则不会产生这些问题，因为只有被放在一个单独模块中的有限几个子程序才被允许对模块数据进行直接存取操作，当一个子程序进行这种操作时，它知道别的子程序也在进行同样操作，并确切知道这些是哪几个子程序。如果你还不相信的话，试一下，结果会令你满意的。

误认为会损失性能

信息隐蔽的最后一个障碍是在结构设计和编码两个层次上，都试图避免性能损失。事实上，在两个层次上你都不必担心这一点。在结构设计层次上，这种担心之所以不必要是因为，以信息隐蔽为目标进行结构设计，与以性能为目标进行结构设计是不矛盾的，只要你同时考虑到这两点，那么就可以同时达到这两个目标。更常见的担心是在编码层次上，这种担心主要是认为间接而不是直接地存取数据结构会带来运行时间上的损失，因为这样做附加了调用层次。当测试了系统的性能并在瓶颈问题上有所突破时，这种担心是不成熟的。为提高软件性能做准备的最好手段之一就是模块化设计，这样，在发现了更好的方案之后，不必改变系统其余部分，就可以对个别子程序进行优化。

6.3　建立模块的理由

即使不经常使用模块，凭直觉也很可能会对可以放入模块的数据和子程序种类有所了解。从某种意义来说，模块并不是人们的目标，它只是数据及对数据所进行的操作的集合，并且支持面向对象的概念——抽象和封装。模块不支持继承性，因而它也并不完全支持面向对象编程，描述它的这种有限的面向对象特性的词汇是 Booch 1991 年提出来的“基于对象”编程。

以下是一些适合使用模块的域：

用户接口。可以建立一个模块来把用户接口要素独立起来。这样，不会影响程序其它部分，你就可以进行改进。在许多情况下，用户接口模块中往往包含有几个模块来进行诸如菜单操作、窗口管理、系统帮助等。

对硬件有依赖的区域。把对硬件有依赖的区域放入一个或几个模块中。这些区域常见的有：与屏幕、打印机、绘图机、磁盘驱动器、鼠标等的接口。把这些对硬件有依赖的区域独立起来可能帮助把程序移植到新环境下运行。设计一个硬件经常变动的程序时，这也是很有帮助的，可以编写软件表模拟与特定硬件的交互作用，硬件不存在或不稳定时，让接口子程序与这些模拟软件打交道。然后在硬件稳定时，再让接口子程序与硬件打交道。

输入与输出。把输入/输出封装起来，可以使程序其余部分免受经常变动的文件和报告格式的影响。把输入/输出放入模块，也使得程序很容易适应输入/输出设备的变动。

操作系统依赖部分。把对操作系统有依赖的部分放入模块的原因与把对硬件有依赖部分放入模块的原因是相同的。如果你正在编写一个在 Microsoft Windows 下运行的软件，为什么要把它局限于 Windows 环境下呢？你完全可以把对 Windows 的调用放在一个 Windows 接口模块中。如果以后想把程序移植到 Macintosh 或者 OS/2 环境下，你所要做的只是改动一下接口模块而已。

数据管理。应把数据管理部分放入模块中，让其中的子程序去与那些杂乱的实现细节打交道。而让模块外的子程序用抽象的方式与数据打交道，这种方式应该尽可能避免实际处理问题，如果你认为将数据管理模块化是将其放入一个单独模块中，那你就错了。通常，每一种主要的抽象数据类型，都需要一个单独的模块来管理。

真实目标与抽象数据类型。在程序中，需要为每个真实目标创立一个模块。把这一目标所需要的数据放入模块中，然后再在其中建立对目标进行模块化的子程序。这就是所谓抽象数据类型。

可再使用的代码。应把计划在其它程序中再用的程序部分进行模块化。建立模块的一个优点是，重新使用模块要比重新使用面向功能的程序实用得多。在面向对象设计和面向功能设计方法中，刚开始的项目都不能从以前的项目中借用许多代码，因为以前项目还不够多，无法提供足够的代码基础。使用面向功能设计方法开发的程序，大约可以从以前的项目中借用 35%的代码：而在使用面向对象设计方法开发的项目中，则大约可以从以前的项目中借用 70%的代码。如果可以通过深思远虑而在以后的项目中避免重写 70%的代码，那为什么不这样做呢？

可能发生变动的相互联系的操作。应该在那些可能发生变动的操作周围修建一道隔墙。这事实上是容错原则的一种，因为这样可以避免局部的变动影响到程序的其余部分。在 6.2 节中，给出了一些经常发生变动的区域。

互相联系的操作。最后，应把互相联系的操作放到一起。在绝大多数情况下，都可以发现把看起来互相联系的子程序和数据放在一起的更强的组织原则。在无法隐蔽信息的情况下，比如共享数据或计划增强灵活性时，仍然可以把成组操作放在一起，比如，三角函数、统计函数、字符串操作子程序、图像子程序等。通过精心地成组放置相关操作，还可以在下一个项目中重新使用它。

6.4　任何语言中实现模块

有些语言直接支持模块化，但有些语言则需要补充一些编程标准才可以。

6.4.1　模块化所需的语言支持

模块包括数据、数据类型、数据操作以及公共和局部操作的区分等。为了支持模块化，一种语言必须支持多种模块。如果没有多模块，其它任何要求都是空谈。

数据需要在三个层次上可以被存取和隐含，在局部，在模块中及在全局中，绝大多数语言都支持局部数据和全局数据。如果想要使某些数据仅对模块中的子程序才是可以存取的，那么就要求语言支持模块数据，即只有某些而不是全部子程序都可以存取的数据。

对于数据类型的可存取性和可隐含性的要求，与对数据的要求是类似的。某些类型应该隐含在某一特定模块中，而另一些类型应该是对其它模块开放的。模块需要能够对那些可以知道其它类型的模块进行控制。

对模块层次上的子程序的要求也与上述相类似。有些子程序应该只有在模块内部才能调用，而且模块应该对某一子程序是专用的还是公用的可以进行控制。在模块之外，不应该有其它模块或子程序知道这个模块中存在专用子程序。如果模块设计得很好，那么其它模块或子程序不应该有任何理由来关心专用子程序的存在。

6.4.2　语言支持概述

在下表中，对几种语言支持信息隐蔽的必要结构进行了总结：

通用 Basic 和通用 Pascal 不支持多模块，所以被排在支持模块化的前列，Fortran 和

语言	多模块	数据			数据类型			源程序	
		局部	模块	全局	局部	模块	全局	专用	模块/全局
Ada	•	•	•	•	•	•	•	•	•
C	•	•	•	•	•	+	•	•	•
C++	•	•	•	•	•	•	•	•	•
Fortran77	+	•	+	•	–	–	–	–	•
通用 Basic	–	–	–	•	–	–	–	–	•
通用 Pascal	•	•	•	•	•	•	•	•	•
Turbo Pascal	–	•	–	•	•	–	•	–	•
QuickBasic	•	•	•	–	•	–	–	–	•

QuickBasic 不能控制支持模块化的数据类型。只有 Ada、C++和 C 以及 Turbo Pascal 才允许模块限制对某一子程序的调用，从而使这个子程序真正是专用的。

简而言之，除非使用 Ada、C、C++或 Turbo Pascal，否则，就不得不通过命名或其它约定来扩充所使用语言的能力，以便模拟使用模块。以下部分简单论述了直接支持模块化的语言，并且将告诉你在其它语言中怎样模拟使用模块。

Ada 与 Modula-2 支持

Ada 通过“包”的概念来支持模块化。如果用 Ada 编过程序，那么就已经知道如何建立包了。Modula-2 通过模块的概念来支持模块化。虽然 Modula-2 并不是本书的特性，它对模块化的支持仍然是非常直接的，以下给出了一个用 Modula-2 进行排序的例子：

```
definition module Events;
    export
        EVENT,
          EventAvailable,
          HighestPriorityEvent,
          LowestPriorityEvent;
```

这些是公共的

```
type
        EVENT = integer;
    var
        EventAvailable:boolean;        { true if an event is available }
    function HighestPriorityEvent:Event;
    function LowestPriorityEvent:Event;
end Events;
```

面向对象的语言支持

面向对象的语言，如 C++，对模块化的支持是直接的，模块化是面向对象编程的核心。以下是一个 C++来实现排序的模块的例子：

```
class    Buffer
    {
    public;
    typedef intEVENT;
    BOOL Eventavailable;          /*true if an event is available */

    EVENT HighestPriorityEvent(void);
    EVENT LowestPriorityEvent(void);

    Private;
    ...
    };
```

Pascal 的支持

某些版本的 Pascal，即 4.0 版和随后的 Turbo Pascal，利用单元的概念来支持模块化。“单元”是一个可以包括数据、数据类型和子程序的数据文件。单元中有一个说明了可供模块外部使用的子程序和数据接口。数据也可以被说明为在这个文件内部的函数和过程使用，而且是仅在其内部使用。这为 Turbo Pascal 提供了可以在局部、模块和全局层次上的数据可存取性。以下是在 Turbo Pascal 5.0 版中的排序模块形式：

```
unit    Events;
INTERFACE
    type
        EVENT=integer;
    var
        EventAvailable:boolean；{ true if an event is available }
```

```
    function HighestPriorityEvent: Event;
    function LowestPriorityEvnet：Event；

IMPLEMENTATION
...                 ——文件中这部分的子程序数据，如果没有在上面 INTERFACE 中说明
                        的话，则对其它文件来说是隐蔽的
end．{unit Events}
```

属于 Generic Pascal 标准的 Pascal 实现并不直接支持模块化，不过你可以通过扩展它们来达到模块化，这将在后面讨论。

C 的支持

虽然用 C 语言编程的程序员们并不习惯在 C 中使用模块，但事实上 C 也直接支持模块化。每一个 C 源文件都可以同时含有数据和函数，可以把这些数据和函数说明为 Static，这将使它们只在源文件内部才能使用。也可以不把它们说明为 Static，此时它们在源文件外也可以使用。当每一个源文件都被当作模块时，C 就完全支持模块化了。

由于源文件和模块并不完全相同，你需要为每一个源文件创建两个标题文件——一个作为公用、模块标题文件，另一个作为专用的、源文件标题文件。在源文件的公用标题文件中，只放入公用数据和函数说明，下面是一个例子：

```
/*   File:Event.h        本文件仅包含公共的可用类型、数据和函数说明
     Contains public declarations for the "Event" module. */
typedef int EVENT;

extern BOOL EventAvailable;      /* true if an event is available */

EVENT HighestPriorityEvent(void);
EVENT LowestPriorityEvent(void);
```

而在源文件专用标题文件，只放入供内部使用的数据和函数说明。用#include 命令把标题文件只放入组成模块的源文件中，不要允许其它文件含有它。以下是一个例子：

```
/*   File:_Event.h
     Contains private declarations for the "Event" module. */

/*   private declarations   */  ——这里是专用类型、数据和函数
...
```

C 源文件中，要使用#include 来把两个标题文件都包含在其中。在其它使用了 Event 模块中的公用子程序模块中，用#include 命令来只把公用的模块标题包含进去。

如果在单独一个模块中,需要使用一个以上源文件，可能要为每一个源文件都加一个专用标题文件；但对组成模块的子程序组，应该只加一个公共的模块标题文件。对于模块中的源

文件来说，利用#include 来包含同一个模块中其它源文件的标题文件是可以的。

Fortran 的支持

Fortran90 为模块提供了全部支持。而 Fortran77 如果在规定范围内使用，则只对模块提供了有限的支持。它拥有创建一组对数据具有独占存取权的机制，这个机制就是多重入口点，对这种机制的随便使用，往往会产生问题。而如果小心使用的话，它则提供了一种不必使数据成为公用的，就可以使一组子程序对同数据进行存取的途径。

可以说明数据为局部的、全局的或者是 COMMON 的。这个特性又提供了使子程序对数据进行受限制存取的方法，定义一组组成一个模块的子程序。Fortran 编译程序无论如何也不会认为它们组成了一个模块，但是从逻辑上来说，它们的确组成了一个模块。对模块中每一个允许对数据进行直接操作的子程序，都应该用 COMMON 作为其前缀来命名。不要让模块以外的子程序使用 COMMON 作为前缀。利用编程标准来弥补程序语言不足这一办法，将在后面讨论。

6.4.3　伪模块化

在像通用 Basic，通用 Pascal 和其它既不直接也不间接支持模块化的语言中，该如何进行模块化呢？答案在前面已经提到过了，利用编程标准来代替语言的直接支持。即使你的编译程序并不强制你采用好的编程应用，还可以采用能够达到这一目的的编码标准。以下的讨论涉及了模块化所要求的每一方面。

把数据和子程序装入模块

这个过程的难易程度取决于编程语言，即它是允许使用各种源文件的还是要求所有的代码必须是在一个源文件中的？如果你需要用 10 个模块，那就创建 10 个源文件。如果环境要求所有代码都要在一个源文件中，那就按模块把代码划分为几部分，同时用注释来区分每一个模块的开头和结尾。

保证模块的内部子程序是专用的

如果所采用的程序语言不支持限制内部子程序的可见性，使得所有的子程序对其它子程序来说都是公用的，可以利用编码规定来限定只有标明公用的子程序才能被模块外的子程序使用。不管是什么样的编译程序，下面都是你可以做的：

通过在说明的地方加注释，来明确区分公用和专用子程序。明确区分与每个子程序相关的模块。如果别人不得不通过查询子程序说明来使用子程序，要确保在说明中注释了它是公用的还是专用的。

不允许子程序调用其它模块的内部子程序。使用注释把属于同一个模块的子程序联系到一起。

采用命名约定来表明一个子程序是内部的还是外部的。可以让所有的内部子程序名称前面都带有下划线（_）。虽然这种方法无法区分不同模块的内部子程序，但是绝大多数人还是可以区分一个子程序是否是属于它们自己的模块。如果它不是自己模块中的子程序，那么显然它是其余模块中的子程序。

采用表明它是内部的还是外部的命名规定。所采用约定的细节，往往取决于编程语言所赋予你对子程序命名的灵活性。例如，一个 DataRetrieval 模块的内部子程序名称可能会以 dr_作为前缀。而 UserInterface 模块内部子程序的名称前缀则可以是 ui_。而属于同一模块的外部子程序则分别会以 DR_和 UI_作为前缀，如果对名称的字节长度是有限制的（如，ANSI FORTRAN77 中的 6 个字母），那么命名的约定就会用掉其中相当一部分，这时，在制定命名约定时就要注意这一长度限制。

保证子程序的内部数据是专用的。

保证模块层次上的内部数据的专用性与保证模块层次上子程序的专用性是相似的。通常，要采用清楚表明只有特定数据才能在文件外部使用的编码标准。不管编译程序允许你做的是什么，以下是一些可以采用的步骤：

首先，利用注释来说明数据是公用的还是专用的。明确区分模块所涉及到的每一个数据的可存取性。

其次，不允许任何子程序使用其它模块中的专用数据，即使编译程序把此数据作为全局变量也不可以。

第三，采用可以使你注意一个文件是专用还是公用的命名约定。为了保持连贯性，应该使这一命名原则与子程序的命名原则类似。

第四，采用表明数据属于哪一个模块，即它是外部还是内部的命名约定。

6.4.4　检查表

模块的质量

- 模块是否有一个中心目的？
- 模块是否是围绕着一组公用数据进行组织的？
- 模块是否提供了一套相互联系的功能？
- 模块功能是否足够完备，从而使得其它模块不必干预其内部数据？
- 一个模块相对其它模块是否是独立的？它们之间是松散耦合的吗？
- 一个模块的实现细节，对其它模块来说，是隐含的吗？
- 模块的接口是否抽象到了不必关心其功能实现方式的地步？它是作为一个黑盒子来设计的吗？
- 是否考虑过把模块再划分为单元模块？是否对其进行了充分的再划分工作？
- 如果用不完全支持模块的语言编程，你是否制定了编程约定以使这种语言支持模块？

6.5　小　结

- 不管调用哪一个，子程序与模块的不同是很重要的，要认真考虑子程序与模块的设计。
- 从模块数据是被几个子程序使用的这一角度来说，它与全局数据是相同的，但从可以使用它的子程序是有限的，而且清楚地知道是哪些子程序可以使用它这一角度来说，模块数据与全局数据又是不同的。因此，可以使用模块数据而没有全局数据的危险。

- 信息隐蔽总是有益的。其结果是可以产生可靠的易于改进的系统，它也是目前流行的设计方法的核心。
- 创建模块的原因有许多是与创建子程序相同的。但模块概念的意义要比子程序深远得多，因为它可以提供一整套而不是单独一个功能，因此，它是比子程序更高层次的设计工具。
- 可以在任何语言中进行模块设计。如果所采用的语言不直接支持模块，可以用编程约定对其加以扩展，以达到某种程度的模块化。

第七章　高级结构设计

目录

相关章节

高质量子程序的特点：见第 5 章
高质量模块的特点：见第 6 章
软件结构设计：见 3.4 节

有些人可能认为高层次设计不是真正的实现活动，但是在小规模项目中，许多活动都被认为是实现活动，其中包括设计。在一些较大的项目中，一个正式的结构设计可能只是分解系统，而其余大量的设计工作往往留在创建过程中进行。在其它的大型项目中，设计工作可能是非常详细的，以致于编码不过是一项机械的工作而已，但是设计很少有这样详尽的。编码人员常常要进行一些设计工作。

高层次设计是一个很大的主题，同时也由于它只部分地与本书主题有关系，因此，我们将只论述其中的几个方面。模块设计和子程序设计的好坏在很大程度上取决于系统的结构设计好不好，因此，要确保结构设计先决条件（如 3.4 节所述）已经被满足了。也有许多设计工作是在单个子程序和模块层次上进行的，这已在四、五、六章中论述过了。

如果已经对结构化设计和面向用户设计非常熟悉了，你可能想阅读下一部分的介绍，再跳到 7.4 节关于两种技术的比较，最后阅读 7.5 节。

7.1　软件设计引论

“软件设计”一词的意思是指，把一个计算机程序的定义转变成可以运行程序的设计方法。设计是联系要求定义和编码与调试的活动的桥梁。 它是一个启发的过程而不是一个确定的过程，需要创造性和深刻的理解力。设计活动的绝大部分活动都是针对当前某特定项目进行的。

7.1.1　大型和小型项目设计

在大型的、正规的项目中，设计通常是与需求分析、编码等活动分立的，它们甚至可能是由不同人分别完成的。一个大型项目可能有几个级别的设计工作——软件结构设计、高层次模块设计和实现细节设计。结构设计具有指导的意义,在小规模编码阶段往往也是很有帮助的。如

果进行了通用或高层次设计，那么其指导方针则在大规模编码阶段是非常有意义的。不管是出于什么原因，即使名义上设计工作已经结束了，但事实上它远没有停止。

在小型的、非正式的项目中，大量的设计工作都是由程序员们坐在键盘前面完成的。“设计”可能只是用程序语言编程，用 PDL 来编写子程序。也可能是在编写子程序之前画一下流程图。不管是怎样进行的，小型项目与大型项目一样，都会从良好的设计工作中受到大量好处。而使得设计活动明显化，则会极大地扩展这一好处。

7.1.2　设计的层次

软件系统中，设计是在几个不同层次的细节上进行的。有些技术适用于所有的层次，而另外一些，则往往只会适合其中的一或两个层次，下面是这些层次：

层次 1：划分成子系统

在这一层次上，设计的主要成果是划分系统的主要子系统。这些子系统可能是很大的——数据库接口、用户接口、命令解释程序、报告格式程序等等。这一层次的主要设计活动是如何将系统划分成几个主要要素，并且定义这些要素间的接口。在这一层次上的划分工作主要是针对那些耗时在几天以上的项目。在图 7-1 中，设计是由三个主要元素和它们的交互作用组成的。

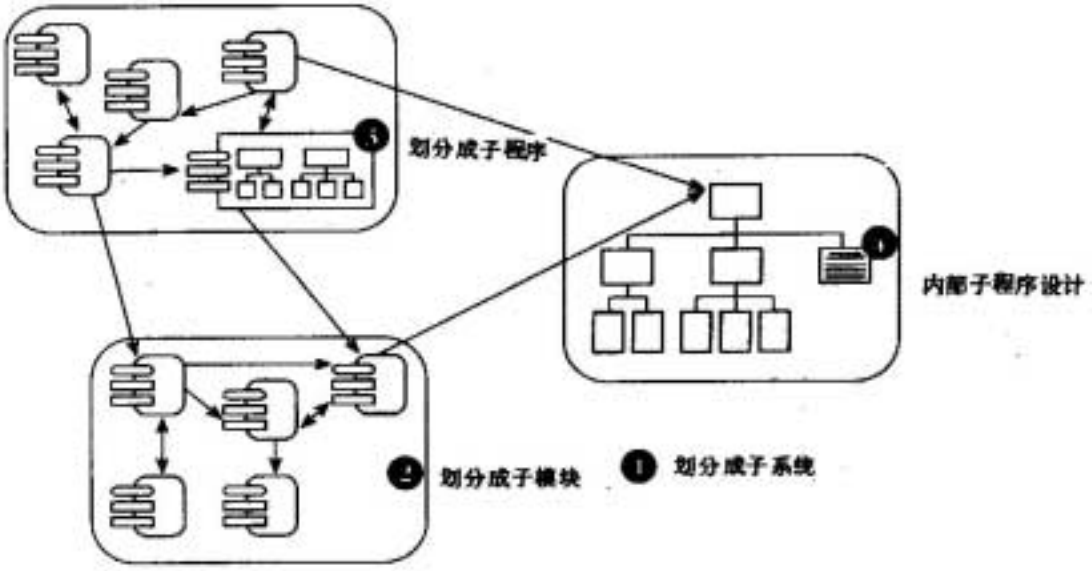


图 7-1　一个程序的设计层次

其中程序划分为系统（1），子系统更进一步划分为模块（2），一些模块划分为程序（3），（4）为每个子程序内部设计。

“子程序”和“模块”的特定意义已经在前几章中引入了。“Subprogram”一词将在本章中使用，我们可以称它为“亚程序”，它是指小于整个程序的任何程序，其中也包括子程序和模块。

层次 2：划分成模块

这一层次的设计包括识别系统中的所有模块。在大型系统中，在程序分区层次上划分出来

的子系统往往太大，难以直接翻译成代码。例如，一个数据库接口子系统可能非常复杂，需要有十几个子程序来实现。如果出现这种情况，那么还需要将这个子系统再划分为模块：如数据存储、数据恢复、问题解释等模块。如果分解出来的模块还是太复杂，那么将对它再次划分。在许多程序的设计中，在层次 1 中分解出来的子系统将直接转变成层次 2 中的模块，而不再区分这两个阶段。

在定义模块时，同时也要定义程序中每个模块之间的相互作用方式。例如，将定义属于某一模块的数据存取函数。总之，这一层次的主要设计活动是确保所有的子系统都被分解成为可以用单个模块来实现它的每一部分。

与把一个系统分解成子系统一样，把子系统分解成模块也是主要针对耗时在几天以上的项目的。如果项目很大，这一层次的分解活动是与上一层次严格区分的。如果项目较小，层次 1 与层次 2 可能是同时进行的。在图 7-1 中，分解为模块的活动已经包含于每个元素中了。正如图中所示的那样，对于系统不同部分的设计方法是也不同的。对某些模块之间关系的设计可能是以网络思想为基础的，也可能是以面向对象思想为基础的。如同图中左面两个子系统那样。而其它模块的设计则可能是分级的。如图中右侧的子系统那样。

层次 3：划分成子程序

这个层次的设计包括把每个模块划分成各种功能，一旦一个子程序被识别出来，那么就同时规定它的功能。由于模块与系统其它部分是相互作用的，而这一作用又是通过功能子程序进行的，所以，模块与系统其余部分的作用细节是在这一部分规定的。例如，将严格定义如何调用解释程序。

这一层次的分解和设计工作对于任何耗时超过几个小时的项目都是需要的，它并不一定需要被正式地进行，但起码是要在心中进行。在图 7-1 中，在左上角的一组中的一个模块中，给出了划分成子程序的工作活动。当你揭开黑盒子的盖子时，如同图 7-1 中标有 3 的模块，你可以发现由模块提供的功能是由层次组织的子程序组成的。这并不是意味着每个黑盒子中都含有层次结构，事实上只有某些黑盒子中才有。其余组织形式的子程序可能没有或很少有层次结构。

层次 4：子程序内部的设计

在子程序层次上的设计，包括设计单个子程序中的详细功能等。子程序内部设计往往是由程序员进行的。这一设计包括编写 PDL，在参考书中寻找算法，在子程序中组织代码段落，编写编程语言代码等活动。这种层次的工作在任何一个项目中都是要进行的，无论是有意识的还是无意识的，是作得好还是作得坏。如果缺少了这一层次的工作，任何程序都不可能产生。在图 7-1 中，在标有（4）的一组中，表现了这个层次的工作。

7.1.3　创建中的设计工作

对于设计层次的讨论，是论述本章其余部分内容的前提，当人们在提到“设计”一词时，他们事实指的可能是许多不同的活动。这是一个非常重大的主题，也是非常重要的。以下是关于这一活动，但这次是按照从细节到总体的顺序进行的。

内部子程序设计

在第四章中间明确讨论过内部子程序设计问题，在第五章“高质量子程序的特点”中，对这一问题又作了进一步的讨论。在第五章关于数据和数据控制部分中，从个别程序语句和子程序中模块的层次，对这一问题进行了讨论。在本章中，这一问题的讨论主要分布在各个部分中。

划分成子程序

在结构化设计方法中，“设计”往往是指设计程序的结构，而不指单个子程序内部的设计。关于程序结构问题，完全可以写一本专著，本章中论述的只是构造简单程序集合的技术总结，这些子程序集合将是你在设计中经常要实现的。

划分成模块

在面向对象设计中，“设计”指的是设计一个系统中的不同模块。模块定义也是一个很大的足以写成一部书的主题，在第六章中已经论述过了。

划分成子系统

对于中小型程序来说（一般是 10000 条语句左右），定义模块和子程序的技术往往隐含在整个程序的设计中，在更大一些程序中，往往需要特殊的设计方法，这对于本书来说（本书重点是实现）是难以详尽论述的。在许多情况下，特别是在一些小项目中，设计工作是键盘上完成的，因此事实上是一种实现活动，虽然它应该在早期就完成了。本书之所以涉及了模块和子程序设计，是因为它们处在实现的边缘上。而关于程序划分的其它讨论，则不在本书之列。

7.2　结构化设计

结构化设计这一概念是1974年在《IBM 系统日报》（IBM System Journal）一篇论文中出现的。在后来由 Ed Yourdon 和 Larry Constantine 写进《Structured Design:Fundamentals of a Discipline of Computer Program and Systems Design》书中（1971），对其作了全面补充与扩展。 Constantine 是最初那篇论文的作者之一，而“自顶向下设计”一词则是指一种非正式的结构化设计，类似的词还有“逐步求精”和“分解”等，指的基本都是同一意思。结构化设计是与其它结构化设计方法一道使用的。

结构化设计是由以下部分组成的：

- 系统组织，系统将被设计成几个黑盒子，明确定义的子程序和模块、接口的实现细节对其它子程序来说都是隐含的。
- 开发设计的策略。
- 评估设计准则。
- 关于问题的明确说明，这是解决问题的指导原则。
- 表达设计的图形和语言工具，包括 PDL 和结构图。

在下面的内容中，将对这些内容作比较详细的论述。

7.2.1　选择需进行模块化的要素

在前面几章论述了程序和模块相关好坏的标准，并提供了确定子程序和模块质量的检查表，但并没有给出识别子程序和模块的方法，在本节中，将论述这一问题的指导原则。

自顶向下分解

把程序分解为子程序的一种流行方法是自顶向下分解，也称为自顶向下设计或逐步求精。其特点是从关于程序功能的粗略说明出发，逐步推进到程序要做的每一项特定的工作。从粗略的层次出发往往是指从程序中的“主要”子程序出发，通常，把这个子程序画在结构图的顶部。

以下是几项在进行自顶向下分解时要牢记的原则：

- 设计高层次。
- 避免特定语言细节。从设计中，不应该看出打算在程序中使用什么语言，或者说当在设计中更换要用的语言时，不会产生任何麻烦。
- 暂时不指出下一层次的设计细节（与信息隐含类似）。
- 正规化每个层次。
- 检验每个层次。
- 转移到下一个层次，进行新的求精工作。

自顶向下设计指导原则的依据是：人脑一次只能考虑有限数量的细节。如果你从一个较简略的子程序开始，逐步把它分解成更加详细的子程序，就不必每次考虑过多的细节。这种方法也常称之为“分而治之”战术。它对于分层结构往往是最有效的。如图 7-2 所示。

从两个方面来看，这种分而治之战术是一个逐次迭代逼近的过程，首先，这是由于往往在一次分解之后你并不会马上停止，还要进行下几个层次的分解。其次，是由于分解并不是一蹴而就的，采用某种方法分解一个程序，再看一下效果，然后，又用另外一种方法来分解这个程序，看效果是否会好些，在几次尝试之后，就有了一个很好的办法，同时也知道为什么这样做。

需要把一个程序分解到什么程度呢？要持续不断地分解，直到看起来下一步进行编码要比再分解要容易为止，或者到你认为设计已经非常明了详细，对再分解已经感到不耐烦为止，到这时，可以认为分解已经完成了。由于你比任何人都熟悉这个问题，而且也比任何人都清楚，因此，你要确保其解决方案是很容易理解的，如果连你都对解决方案有些困惑的话，那么，试想一下，又有谁会理解它呢？

自底向上合成

有时候，自顶向下方法过于抽象以至于让人不知从何下手。如果想要进行一些具体的工作，那么可以试一下自底向上的设计方法，如图 7-2 所示。你可以问自己，“这个系统需要做什么？”毫无疑问，你能够回答这个问题。你可以识别出系统需要具备的较低层次的功能，例如，你可能知道这个系统需要进行报告格式化、计算报告总数、用不同颜色在屏幕上显示字母等等。当你识别出某些低层次功能后，再从事较高层设计可能会有把握些。

以下是一些在进行自底向上合成时要牢记的原则：

- 问自己，关于系统要做什么你都知道哪些？
- 利用这一问题识别出某些低层次功能。

- 识别出这些低层次功能共同的方面，将其组合到一起。
- 向上一个层次，进行同样的工作，或回到顶端开始自顶向下。

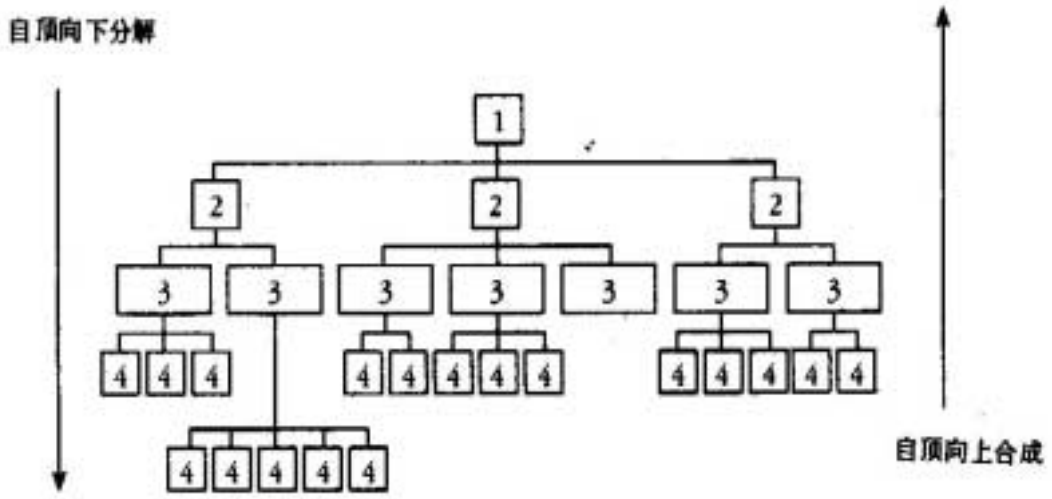


图 7-2　自顶向下分解与自底向上合成

其中自顶向下是从一般到特珠，自底向上是从特殊到一般。

自顶向下与自底向上

自底向上与自顶向下策略的首要区别是前者是合成，而后者则是分解。一个是从可控制的零散问题出发，把它们合成到一起从而获得一个总体解决方案，而另一个则从总体问题出发，将其分解成可控制的零散问题。两种方法各有优缺点。在实际使用时要详加比较。

自顶向下设计方法的特点是比较简单，人们往往擅长把大问题分解成小问题，而程序员们则更是擅长这点。当一个问题是用层次结构模型化的时候，自上而下的分解方法恰好与其相符。

自顶向下设计的另一个优点是你可以避开实现细节。由于系统常常受到实现细节变动的干扰（比如文件结构或报告格式的变化），因此把这些细节隐含在层级结构的底部，而不是让它在顶部起支配作用，是非常有益的。

这种设计方法也有它的缺点。其中之一是系统的总体功能可能是很难识别的。关于系统所作的最重要决定之一就是如何进行第一步分解工作，而在自上向下设计中，刚开始接触系统，对其了解还很少时，便不得不做出这一决定，这是很危险的。它的另一个缺点是：由于许多系统本身并不是层级结构的，因此是很难清晰地分解。或许这种设计方法的最大缺点就是它要求系统在顶层要有一个单一而又清楚的功能，而对于现代事件驱动系统来说，这是很难想象的。

自底向上设计方法的优点是它在早期就可以识别出有用的功能子程序，结果是坚实可靠的设计。如果已经开发了相似的系统，那么可以参阅一下旧系统，看看有什么可以借用的。

这种文件的弱点是很难单独地使用它。因为大多数人都不善于从小的概念出发形成综合的设想。这就像一个自己组装的玩具，我想我已经组装完了，怎么盒子中还有零件呢？好在，你不必单独使用它。

它的另一个弱点是,有时从你识别出的细节出发，无法建造出整个程序，就像你无法用砖头造出一架飞机一样。而且当你知道在底部需要什么功能时，你可能已经不得不进行顶层设计了。

不过，这两种方法并不是互相矛盾的。设计是一个启发的过程，就是说没有一种百试不爽的设计方法，它总是一个尝试的过程。因此，在找到好方法之前，尽可以大胆尝试，可以用自顶向下工作一会儿，再用自底向上工作一会儿。

设计也是一个逐次迭代逼近的过程。因此，你在第 n 次用自底向上方法学到的东西，将在第 n＋1 次用自顶向下方法设计时起到很大帮助作用。

7.3　面向对象

面向对象设计方法的特点是通过对实际问题的分析，从中抽象出目标，然后再用程序语言来表现它，其过程主要是：识别目标中的分目标并识别出对于分目标的操作，然后再根据分目标的操作开发出一个系统。面向对象设计是在程序中设计目标或模块的一种方法。在较低的程度上说，它也是设计单个子程序的一种方法。

虽然有些鼓吹者把计算机历史划分为面向对象设计出现前阶段和面向对象设计出现后阶段，但事实上面向对象设计与其它设计方法并不冲突。特别地，面向对象设计与结构化编程所提供的低层次结构化并不是不兼容的，但它与高层次结构化的确不兼容。在更高的层次上，面向对象设计方法在简单的功能性层次结构上，添加了类、群和非层次结构等新的概念。对这些高层次的组合思想进行研究和标准化工作，将会使编程技术再向前产生一次飞跃。

在本书，对于面向对象设计的讨论是非常浅显的。与结构化设计方法相比，面向对象设计的抽象化程度更高。本节着重论述的只是在较低层次上起作用的抽象方法，其中主要是在个别语句、子程序和有限数量的子程序这个层次上的。这种设计方法相对来说也是一种新的设计理论。它还没有完全成熟，关于这方面积累的设计经验也还不够丰富，但是，它是很有前途的。

7.3.1　关键思想

面向对象设计是建立在如下主张之上的，即：一个程序模型越是真实地反映了实际问题，那么，由此产生出的程序质量越好，在多数情况下，关于项目的数据定义要比功能稳定得多，因此应象面向对象设计一样，根据数据来进行设计，这可以使设计更稳定。对于现代编程来说，面向对象设计中有许多思想是很重要的。

抽象

抽象所带来的主要好处是可以忽略掉无关紧要的细枝末节问题，而专注于重要的特性。绝大多数现实世界中的目标都是抽象的，房屋是木材、钉子、玻璃、砖和水泥等的抽象，是把它们组织起来的一种特殊形式。而木材本身，则又是纤维、细胞及某些矿物质的抽象，而细胞呢，则又是各种各样的物质分子的抽象。

在建造房屋时，如果你从分子层次上与木材、钉子等打交道，是永远不可能建成房屋的。同样，在建造一个软件系统时，如果总是在相当于分子的层次上工作，那是不可能建成软件系统的。在面向对象设计中，你尽力创造出与解决真实问题某一部分抽象程度相同的编程抽象，以便解决编程问题中的类似部分，而不是用编程语言实体来解决问题。

面向对象设计擅长使用抽象，但因为它所使用的“智力砖块”，要比结构化设计中功能方法所使用的“智力砖块”大得多。在结构化设计中，抽象的单位是函数；而在面向对象设计中，

抽象的单位是目标。由于目标包括函数及受函数影响的数据，从而使得在比函数更高层次上对问题进行处理成为可能。这种抽象能力使你可以在更高层次上对问题进行考虑，而且，不必把神经绷得太紧，就可以一次考虑很多问题。

封装

封装是对抽象不存在地方的补充。如果抽象对你说“你应该在较高层次上看一个目标”，而封装则会说“你只能在这个层次上看一个目标”。这事实上就是 6.2 节所述的信息隐蔽的重复。你对于一个模块所知道的只是它让你知道的那些，别的什么也没有。

我们继续用房屋比拟来说明问题：封装是一个使你可以看到房屋的外表但不能走进去的办法，当然，或许你可以透过窗户看到一小部分内部情况。在较为过时的语言中，信息隐蔽完全是自愿的行为，因为大门上没有“禁止入内”的标志，房门没有上锁，窗户也是敞开的，而对于 Ada 语言来说，信息隐蔽则是强制性的：门被牢牢地锁上了，窗户紧闭，而且警报系统也在工作，你所看到的就是你所得到的，而且是你得到的一切。

模块化

面向对象设计中的模块与结构化设计中模块的含义是一致的。相联系的数据和功能被放入模块，在理想情况下，模块是高度内聚而又松散耦合的。同信息隐蔽一样，当模块内部发生变化时，其接口保持不变。

层次结构和继承性（inheritance）

在设计软件系统时，你经常可以发现两个之间非常相似，其差别非常小的目标。例如，在一个会计系统中，你可能既要考虑正式雇员，也要考虑兼职雇员，与两种雇员相联系的绝大多数数据都是相同的，但也有一小部分是不同的。在面向目标编程中，你可以定义一种广义雇员，把正式雇员当作一种广义雇员，但二者之间稍有差别。把兼职雇员也看作一种与种类无关的广义雇员，那么这种操作就按广义雇员模式进行。如果某种操作是与雇员种类有关的，那么就按雇员种类不同，分别进行操作。

定义这种目标间的共同和不同点称为“继承性”，因为兼职和正式雇员都从广义雇员那里继承了许多特点。

继承策略的好处是它与抽象的概念是一致的，抽象在不同层次的细节上与目标打交道。在某一层次上调用某种分子；在下一个层次上调用纤维与植物细胞的组合，在最高层次上调用一片木头。木头有某种特性（如你可以用锯子锯它或用斧头劈它），不管是松木还是加州红杉木，它们有许多共同的特性，如同它们有不同特性一样。

在面向对象编程中，继承性简化了编程，因为你只要写一个通用子程序来处理目标间的共同特性，再编写几个专用子程序去处理它们间的不同特性就可以了。有些操作，例如 Getsize()，可能在任何抽象层次上都适用。程序语言支持 Getsize()这种直到运行时才需要知道操作对象的子程序特性称为“多形性”。像 C++等面向对象的语言，自动支持继承性和多形性。而以目标为基础的语言，如 Ada 和过程性语言如 C 和 Pascal 则不支持这种特性。

目标与类

在面向对象设计中，最后一个关键概念是目标与类。目标是程序在运行时其中的任何一个实体，而类则是当你看程序清单时存在的一个静态实体。目标是在程序运行时具有特定值和属性的动态实体。例如，你可以定义一个具有名字、年龄、性别等属性的人，而在运行时则会遇到 Nancy，Tom 等人，也就是说，目标是类的特例，如果你对数据库术语很熟悉的话，它们的区别与其中“模式”与“事例”的区别是类似的，在本节其余部分将不严格区分这些词，常会把两种实体都称为“目标”。

7.3.2　面向对象设计的步骤

面向对象设计的步骤是：

- 识别目标及其属性，它往往是数据。
- 确定对每个目标可以做些什么。
- 确定每一个目标可以对其它目标做些什么。
- 确定每个目标对其它目标来说是可见的部分——哪一部分是开放的，哪一部分是专用的。
- 确定每个目标的公共接口。

这些步骤下一定非要按某一特定顺序来进行，但是却需要重复。逐次迭代逼近对面向对象设计是与其它设计方法同样重要的，下面将分别论述这些步骤。

识别目标及其属性。计算机程序通常是以客观世界实体为基础的，例如，你可以用客观世界中的雇员，时间卡及帐单为基础来建造一个时间——记帐系统。图 7-3 中表示了从面向对象观点来看，这一系统的表现形式。

在图形系统中的目标将包括：窗口、对话框、按钮、字体、绘图工具等。对问题域进行进一步研究，可能会发现比这种一对一方式更好的软件目标识别方法，但是，从客观世界中的真实目标出发总是一个比较好的方法。

识别目标属性并不比识别目标困难。每一个目标都有与计算机程序相关联的特性。例如，在时间——记账系统中，一雇员目标将具有姓名、标题和记账率；顾客目标将有姓名、支票地址、收支状况及存款余额等；账单目标具有欠账数量、顾客姓名、日期等等。

图中目标的图形符号与第六章讲述的模块符号类似。

确定可以对一个目标做些什么。对每一个目标都可以进行一系列操作，在时间——记账系统中，雇员的记账率等可以变动，可以要求提供雇员的奖励情况等，顾客目标可以更改其存款额或地址等。

确定目标之间可以互相做些什么。这一步骤与其字面意义是相似的。在时间——记账系统中，雇员可以对其它目标做些什么呢？做不了什么。雇员所能做的一切便是输入时间卡信息。而账单则可以接受时间卡信息，在更复杂的系统中，其它相互作用更为明显。

确定每一个目标中对其它目标来说是可见的部分。在设计中的一个关键问题就是决定目标的哪些部分应该是可见的，哪些部分应该是隐蔽的。对于数据和功能来说，都要做出这种确定。

在表示时间——记账系统的图 7-2 中只表示出了可见的部分，隐蔽的部分则被藏在黑

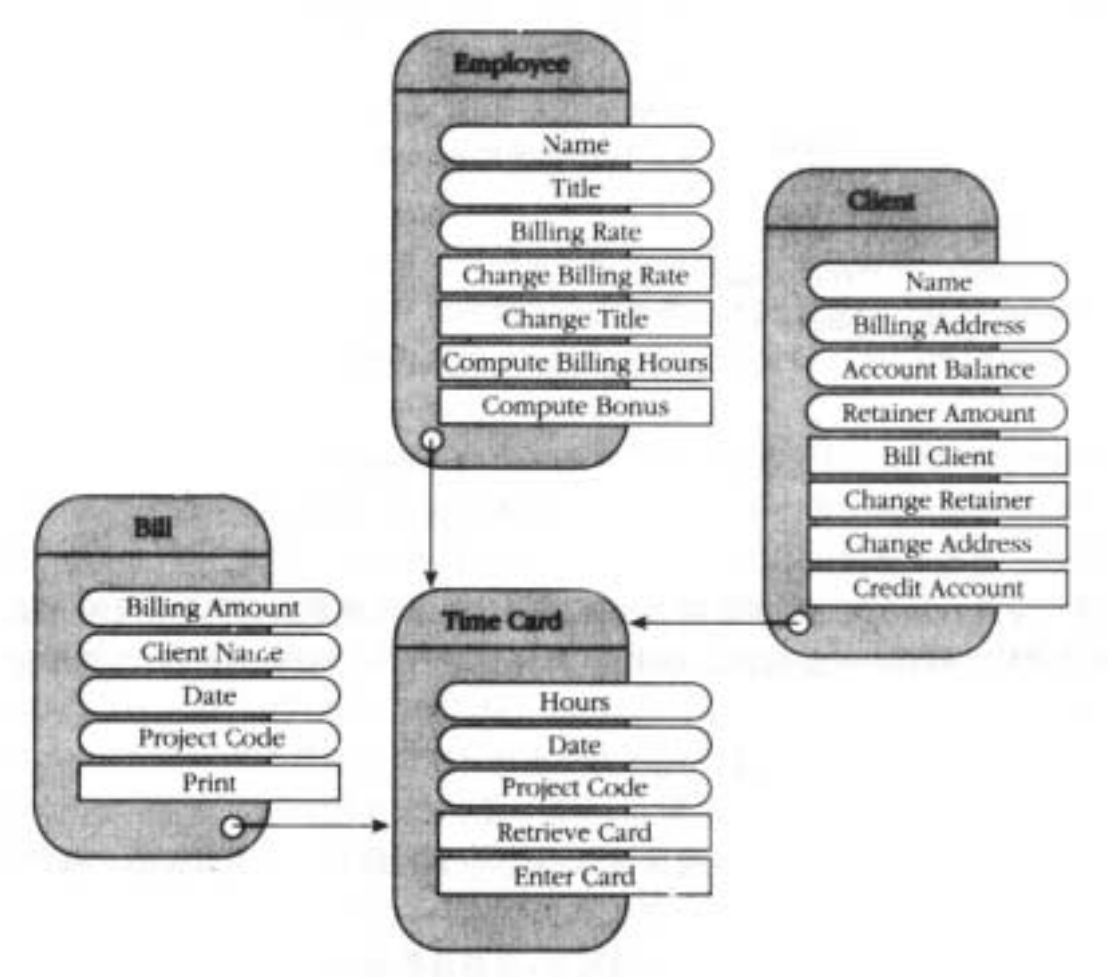


图 7-3　四个主要目标组成的记帐系统

盒子中。顾客与雇员目标看起来是非常复杂的，因为它们每一个都具有七八个可见的特性。这种复杂的表现形式是图示法的一个弱点，这种情况会随着可见特性的增加而恶化。而一个精心设计好的目标往往有许多附加的可见特性。

定义每一个目标的接口。在设计目标中的最后一个步骤是为每个目标定义一个正式的、语法的、在程序层次上的接口。这包括由目标提供的功能和目标与类之间的继承关系。特别地，这一步将包括函数和子程序说明。例如：时间卡的接口（用 C++编写）可能是这样的：

```
class TimeCard
    {
      public:

      int Enter
        (
        EMPLOYEE_ID    Employee,
        DATE           Date,
        CLIENT_ID      Client,
        PROJECT        ProjectCode,
        int            Hours
        );
```

```
int Retrieve
    (
    int&            Hours,
    DATE&           Date,
    CLIENT_ID&      Client,
    PROJECT         ProjectCode,
    EMPLOYEE_ID     Employee
    );

protected:
…
 };
```

当你进行完这一步的工作，到达高层次的面向对象系统组织时，可以用两种方法进行迭代，以便得到更好的目标——类组织。你也可以对定义的每一个目标进行迭代，把设计推向更详细的层次。

7.3.3　典型面向对象设计的评论

一个面向对象系统通常有至少四类目标。如果你不知道这其中每类目标的一些情况，你很可能会漏掉某类目标。

- 问题域要素。这个要素直接指向问题域，在前述的记账系统中，问题域包括客观世界中的目标，如：雇员、顾客，时间卡和账单等。
- 用户接口要素。这个要素指的是系统中负责人机交互的部分。它包括数据入口类型、窗口、对话框、按扭等等。正如 6.2 节中提到的那样，最好把系统的用户接口部分隐蔽起来以适应修改。
- 任务管理要素。这类要素指的是计算机本身的目标。包括实时任务管理程序、硬件接口、通讯协议等。
- 数据管理要素。这部分要素包括保持一致的数据。它包括数据库以及其相联系的所有存储、维护和检索等功能。

7.4　对目前流行设计方法的评论

如果你仔细观察日前流行的设计方法——包括结构化设计和面向对象设计——你会发现每种方法都包括两个主要部分：

- 把一个系统分解成子系统的准则
- 解释分解的图形式语言符号
- 有些方法还包括第三个要素
- 防止你使用其它方法的规定

把“设计”的意义限制在前两个要素上说明设计的核心是把系统分解成亚程序 (Subprogram) ，同时也说明亚程序的设计并不具备足够的挑战性，不值得讨论。

一个好的系统分解的确是很有价值的，但并不是说一旦确立了好的结构，设计就可以停止了。在确认出子程序的模块之后，还有许多设计工作要做。

伴随着某些设计方法的第三个要素，即强调应该只使用一种方法的思想，是非常有害的。没有一种方法囊括了设计系统所需的全部创造力和洞察力。强调使用一种方法将破坏设计中的思维过程。

但是，设计方法的选择往往会成为一种宗教似的问题——你去参加了一个宗教复兴会议，听一些结构化设计的先知讲道，然后你回到自己的圣地在石碑上写下一些神圣的程序，从此以后，你不再允许自己进入非基督教的领域。你应该知道，软件工程不是宗教，不应该引人宗教的狂想性。

如果你是个建筑工人，你不会用同样的方法建造每一幢建筑物。在周一你在浇注水泥，而到了周末你则在修建一幢木屋。你不会用水泥来修木屋，也不会在一幢新建好的摩天大楼门口挂上“成人禁止入内”的牌子。你会根据不同的建筑物而采取不同的施工方法，从建造房子中，你应该得到关于编程方法选择的启示，应该选择与问题相适应的方法，这种世俗方法的合理性已经被许多例子和研究所证明。每种方法都有其优点，但同时也有其弱点，应分析使用(Peter 和 Tripp 1977，Mannino 1987，Kelly 1987，Loy 1990)。

但是，有些方法的障碍是由它们自己的复杂的术语产生的。比如，你想学习结构化设计，你必须懂得如下词汇：输入流与模块、输出流与模块、正规、控制、公用、全局和内容耦合、功能的、顺序的、通讯的、过程的、临时的、逻辑的和偶然性内聚。输入、输出、事务处理中心、事物处理分析和事物处理模块，甚至 Benuzerfreundlichkeit（多可怕！）一词也出现了。字典也无法给出这些词的解释。

结构化设计，以隐蔽信息为目标的设计和面向对象设计等方法提供了看问题的不同角度。图 7-4 给出了使用它们的典型方法。

从事结构化设计的人与从事面向对象设计的人会发现他们进行交流非常困难，原因是他们没有意识到是在不同层次上讨论设计的，因此主题也是不同的。从事结构化设计的 Tom 说：“我想这个系统应该分解成 50 个子程序。”面向对象设计的 Joh 则说：“我认为系统应划分成 7 个目标”。如果你仔细观察，可能会发现这 7 个目标中可能共含有 50 个子程序，而 Tom 的子程序或许可以分成 7 组。

7.4.1　何时使用结构化设计

结构化设计主要是一种把程序分解成子程序的方法。它强调功能但不强调数据。一个面向功能的问题域的例子是一个以批处理方式读入数据，按照可以预计的顺序对数据进行可以预计的处理并且写数据。

结构化设计并没有把子程序组成一个协同工作子程序组的概念，也没有子程序内部设计的概念，除非这个子程序的内部会影响到整个系统。因此，结构化设计非常适用于具有许多互不作用的独立功能的系统。同时，它也适于那些只有几百行代码的小型程序，因为这些程序过于简单，没有建立类、目标和属性的必要。

结构化设计的最先提出者 Larry Constantine，曾经发表过一篇“目标、函数和程序可扩展性”的文章，论述了把结构化设计和面向对象设计组合到一起的设计方法。如果数据变动可能性很大，那么采用面向对象设计比较合适，因为它将变动可能性较大的数据独立在目标（模块）

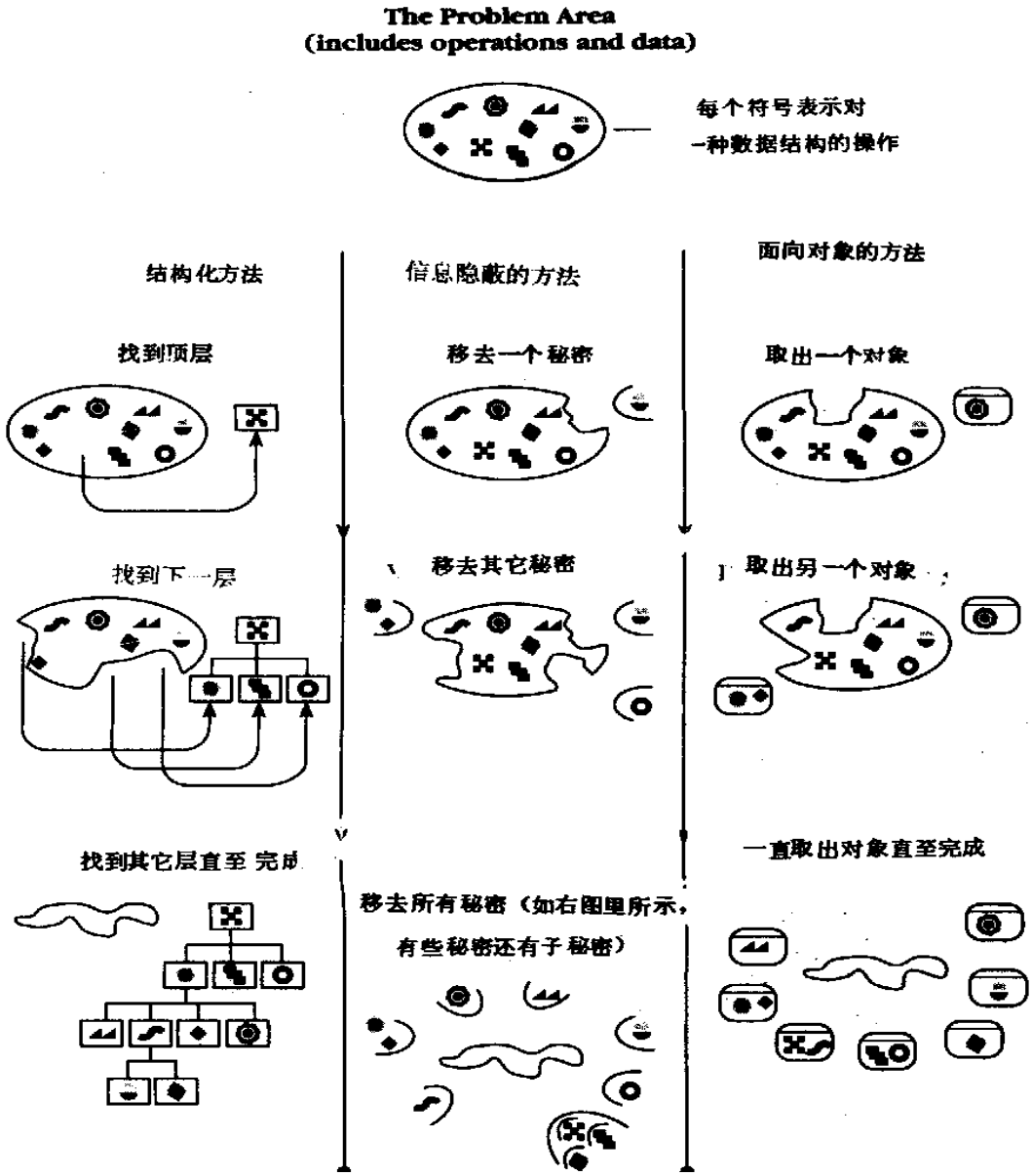


图 7-4　对于一个问题的不同设计（不同设计导致不同的解决方法，每一种都是正确的）

中。如果功能变动的可能性较大，采用面向对象设计就不太适合了。因为功能散布在许多目标（模块）中。如果功能变化的可能性比数据要大，那最好采用分解功能的结构化设计。

7.4.2　何时采用信息隐蔽

无论什么问题领域，都应该尽量采用信息隐蔽。使用它没有任何危险。到目前为止，联邦卫生委员会还没有发现它会发生危险，不论是设计子程序、模块，还是目标程序，它都是很有效的，因此你尽可以放心使用它。

7.4.3　何时采用面向对象设计

面向对象设计与结构化设计的主要区别是：面向对象设计在较高抽象层次上要比结构化设

计有效。这是具有重大历史意义的。因为结构化最初开发起来时程序员们正在建立定义大而复杂的系统，到现在已经远不及如今的大规模系统那样复杂了。

面向对象设计主要是设计模块数据和对数据操作的集合。它非常适于从最顶层分解系统。而当你识别出目标的接口并开始编码时，你往往会转向结构化设计。如果你用面向对象的语言编程，那么很难认为你没有在面向对象设计，因为你用面向对象方法、信息或其它进行结构设计工作。如果你是在用比较传统的过程性语言进行设计，则很容易认为你是在用较旧的结构化设计方法，在这时，使用这一方法是很合适的。

面向对象设计适合于任何客观世界中的目标。这类系统的例子包括高度交互化的窗口、对话框、按钮等程序；面向对象的数据库；需要对随机事件做出反应的事件驱动系统等等。

关于面向对象设计技术的研究工作主要是针对代码在 105～1005 行以上的系统的。对于这种规模的系统，结构化设计基本上是无能为力的，而面向对象设计则往往比较有效。然而，除了这些超大型项目之外，稍显陈旧的结构化设计还是比较有效的，而面向对象设计对于较小型项目的有效性，却还有等待证明。

7.5　往返设计

通过组合使用主要设计方法来扬长避短是完全可能的。每种设计方法都只是程序员工具箱中的一件工具，不同的工具适合不同的工作，你将从研究所有方法的启发中获益无穷。

下面一小节结论述了软件设计为什么困难的某些原因，并指出了如何组合使用结构化设计，面向对象设计和其它设计方法。

7.5.1　什么是往返

你可能会有这样的体验：当你编写程序快结束时，你非常希望能有机会再重新编写一次，因为在编写过程中你对问题又有了更深的理解。这对设计也是同样适用的，只不过在设计中这个循环的周期更短，带来的好处也更大，因此，你完全可以在设计过程中进行几次往返。

“往返设计”一词抓住了设计是个迭代过程这一特点：通常你不会只从 A 点走到 B 点，往往还需要再返回 A 点。

在用不同的设计方法对各种设计方案进行尝试的进程中，将从高层次的总体上和低层次的细节上对问题进行观察。在从事高层次问题时获得的总体印象将会对你在低层次细节中的工作有很大帮助；同时，在从事低层次问题时所获得的细节将为你对高层次的总体理解和作出总体设计决定奠定下良好的基础。这种在高层次和低层次之间往返思维过程是非常有益的，由此而产生的结构，将比单纯自顶向下或自底向上产生的结构要稳定得多。

许多程序员，都会在这一往返过程中遇到麻烦。从对系统的一个观察点转到另一个观察点上，的确是很困难的，但这是进行有效设计所必需的，你可以阅读一下 Adams 于 1980 年写的一本叫“Conceptual Blockbusting”的书，来提高自己思维的灵活性。

7.5.2　设计是一个复杂的过程

J.P Morgon 曾经说过人们在做事情时常常有两个原因：表面上冠冕堂皇的原因和真正的原因。在设计中，最终结果往往看起来是井井有条的，似乎设计者从未犯过任何设计错误，事

实上，设计过程很少有像最终结果那样井井有条。

设计是一个复杂的过程。因为你很难把正确答案与错误答案区分开来。如果你让三个人分别设计同一个程序，他们带回来的往往是三个大相径庭的方案，而且其中每一个看起来都非常适用。它是一个复杂的过程还因为你在设计过程中曾钻过许多死胡同、犯过许多错误。说它是一个复杂的过程也是因为你不知道什么时候设计方案已经足够完善了。什么时候算完成呢？对这个问题的通常答案是“当你没有时间时”。

7.5.3　设计是一个“险恶”的过程

Horst Rittel 和 Melvin Webber 把“烦人”的问题，定义成只有通过解决它或者部分解决它，才能给出明确定义的问题。这个似是而非的定义事实上暗示着你不得不首先“解决”这个问题，对其有一个清楚的定义和理解，然后再重新解决一遍，以获得正确的解决办法。这一过程对软件开发就像母爱和面包对你我一样必不可少。

在现实中，关于险恶问题的一个富于戏剧性的例子便是托卡马大桥的设计。在修建大桥时，主要考虑的便是它应该能承受设计载荷并能抗 12 级大风。然而，没人想到风在吹过桥时会产生“卡门旋涡”——一种特殊的空气动力学现象，从而使桥产生横向简谐振动。结果，在 1940 年的一天，只在 7 级风的作用下，桥便因振动而坍塌了。

说它是险恶问题的典型例子是因为直到桥坍塌时，也没有一个设计师想到应该在这个设计中考虑空气动力学问题。只有在建成大桥（解决问题）之后，才使得他们意识到了要考虑的这一“额外”问题，经过重新设计，新桥至今依然屹立在河上。

你在学校中设计的程序和在实际工作中设计的程序最重要的不同是：在学校中遇到的程序设计问题，几乎没有哪个是险恶的，教师留给你的程序作业都是预先想好让你一次即可完成的。如果哪位教师留给你们一个程序作业，当你们完成后他又突然改变了作业题目，接着，当你即将完成那个程序时，他又改变了主意，会怎么样呢？我想，如果有谁胆敢这样的话，你们肯定会把它绞死。但在实际工作中，几乎总是这样。

7.5.4　设计是一个启发的过程

进行有效设计的关键是要认识到它是个启发的过程。设计中，总是吃一堑，长一智的。往返设计的概念事实上解释了设计是个启发过程这一事实，因为你要把任何设计方法都只当成一种工具。一种工具只对一种工作或者一种工作的某一部分才有效，其余的工具适合其它的工作，没有一种工具是万能的。因此，你往往要同时使用几种工具。

一种很有效的启发工具就是硬算。不要低估它。一个有效的硬算解决方案总比优雅却不能解决问题的方案要好。以搜索算法的开发为例，虽然在 1946 年就产生了对分算法的概念，但直到 16 年后，才有人找到了可以正确搜索各种规模的表的算法。

图示法是另一种有力的启发工具。一幅图抵得上一千个单词。你往往不愿用那一千个单词而宁愿用一幅图，因为图形提供了比文字更高的抽象水平，有时或许你想在细节上处理某一问题，但是，更常见的是在总体上处理问题。

往返设计的一个附加的启发能力是你在设计的头几次循环中，可以暂时对没有解决的细节问题弃之不管，你不必一次就对一切都做出决定，应记住还有一个问题有待做出决定，但同时要意识到，你目前还没有充分的信息来解决这个问题。为什么在设计工作的最后 10%的部分苦

苦挣扎呢？往往在下一循环中它们会自然获得解决。为什么非要在经验和信息都不足的情况下草率决定呢？你完全可以在以后等经验和信息丰富时做出正确决定。有些人对一次设计没能彻底解决问题会感到很不舒服，但与其很不成熟地勉强解决问题，不如把问题暂放一个，待到信息足够丰富时，再解决它。

最重要的设计原则之一是不要死抱着一种方法不放。如果编写 PDL 无效的话，那么就作图，或用自然语言写出来，要么就写一小段验证程序，或者使用一种完全不同的方法，比如硬算解决法，坚持用铅笔不停地写和画，大脑或许会跟上。如果这一切都无效，暂时放开这个问题。出去自由自在地散散步，或者想一下别的，然后再回到这个问题上。如果你已经尽了全力但还是一无所获，那么暂时不考虑这个问题往往会比坚持冥思苦想更快得到答案。最后，可以借鉴其它领域中的方法来解决软件设计中的问题。关于问题解决中的启发方法的最初的一本专著是 G. Polya 的《How To solve in》一书（1957），Polya 的书推广了数学中解决问题的方法，表 7-1 就是对其所用方法的总结，本表摘自 Polya 的书中的类似的总结表：

表 7-1　怎样解决问题

1．理解问题，你必须理解要解决的问题

问题是什么？条件是什么？数据是什么？有可能满足条件吗？已知条件足以确定未知吗？已知条件是否不够充分？是否矛盾 7 是否冗余？

画一个图，引入恰当的符号，把条件的不同部分分解开。

2．设计一个方案。找到已知数据和未知之间的联系。如果不能找出直接联系的话，你可能不得不考虑一些辅助问题，但最后，你应该找到一个解决方案。

以前你是否遇到过这个问题？或者是见过与它稍有不同的问题？是否知道与其相关的问题？是否知道在这个问题中有用的定理？

看着未知！努力回忆起一个有着相同或类似未知的问题。这里有一个与此相关的你以前解决过的问题，你能利用它吗？是能利用它的结论还是能用它的方法？是否该引入辅助要素以使这个问题可以再用？

能否重新表述一下问题？能用另外一种方式表述它吗？返回到定义。

如果你无法解决这个问题,可以先试着解决一些别的问题,是否能想象出一个容易解决的相关问题；一个广义些的问题或是一个更特殊的问题？一个相似的问题呢？能否解决问题的一部分呢？仅保留一部分条件,忽略其余条件；未知可以被决定到什么程度？会发生什么变化？能否从数据中推导出一些有用的东西？能否找出适于确定未知的其余数据？能否改变数据或未知？同时改变两者呢？这样做能否使新的未知和新的数据更接近些？是否使用了全部的数据？使用全部条件了吗？是否考虑了这个问题的全部必要条件？

3．执行你的计划。

执行你解决问题的计划，同时检查每一步工作。你是否可以认定每一步都是正确的？你能证明这点吗？

4．回顾，检查一下答案。

你能检查一下答案吗？能检查一个论证吗？能否用另外一种方法推导出答案？能否一眼就看出答案？

能否在其它问题中再利用本题的答案或者结论？

7.5.5　受迎的设计特点

高质量的设计往往有一些共同的特点。如果你能达到这些目标，那么可以认为你的设计也是非常成功的。有些目标是互相矛盾的。但是这是设计的挑战所在，在相互矛盾的目标之间做

出合理的折衷。某些高质量设计的特点同时也是高质量程序的特点——可靠性。其余的则是设计所独有的。

以下是设计所独有的一些特点：

智力上的可管理性。对于任何系统来说，智力上的可管理性都是其重要目标之一。它对于整个系统的完整性是非常重要的，并且会影响程序员们开发和维护系统的难易程度。

低复杂性。低复杂性实际上是智力上的可管理性一部分，由于上述同样的原因，这点也很重要。

维护的方便性。维护的方便性意味着设计时要为负责维护的程序员着想。在设计中，要不停地想象维护程序中将会对你的设计提出的问题。应该把维护程序员当作你的听众，同时把系统设计成明白易懂的。

最小的联系性。最小的联系性指的是按照保持子程序之间的联系最少的原则来设计，应该利用强内聚，松散耦合和信息隐蔽等作为指导原则来设计系统，使其内部的联系性尽可能少。最小的联系性可以极大地减小综合、测试和维护阶段的工作量。

可扩充性。可扩充性指的是不必影响系统的内部结构，就可以对系统的功能进行强化，你可以改变系统的某一部分而不影响其余部分，使得最大可能性变动对系统带来的影响最小。

可重复使用性。 可重复使用性指的是把系统设计成其中许多部分是可以被其它系统借用的。

高扇入。高扇入指的是对于一个给定的子程序来说，应该有尽可能多的子程序调用它。高扇入表明一个系统在低层次上充分利用了功能子程序。

低或中等程度扇出。低或中等扇出指的是对一个确定的子程序来说，它所调用的子程序应该尽可能地少。高扇出（大约 7 个以上）说明一个子程序控制了许多其它子程序，因此可能是很难理解的。而中等扇出则表明一个子程序只把任务交给了数量较少的其它子程序，因此是比较容易理解的。低扇出（少于 4 个）看起来像是一个子程序没有把足够的任务交给其余的子程序去做，但经验表明并不是这样。一项研究表明有 42%只调用一个子程序的子程序是没有错误的，有 32%的调用 2～7 个子程序是没有错误的,而在调用 7 个以上子程序的情况中，只有 12%是没有错误的（Card, Church 和 Agresi，1986）。由此，Card 认为 0～2 个扇出是最优的。

可移植性。可移植性指的是把系统设计成很容易转到另外的环境下运行。

简练性。简练性指的是把系统设计得没有任何多余部分。Voltaire 曾说过，当一本书不能删掉，而不是不能添补任何内容时，才可以认为它已完成了。在软件中，这也是非常正确的，因为当你对系统进行改进时，你不得不对冗余的代码进行开发、评审、测试和维护等等工作，而且在开发软件的新版本时，新版本也不得不与这些冗余的代码兼容。最有害的观点是“多加入些又不会有害，怕什么呢？”

成层设计。成层设计指的是尽量分解的层次是成层的，这样你可以在每一个单独的层次上观察系统，同时也可以使观察的层次是连续的。也就是说当你在某一层次上观察系统时，不会看到在其它层次上看到的东西。你会经常遇到某些子程序和软件在几个层次上起作用。这样会使系统很混乱，应尽力避免。

如果在编写一个先进系统时，不得不借用许多旧的、设计得不好的代码，那么你可以在新系统中建立一个层(layer)，与那些旧代码相联接。精心设计这个层使它把旧代码的缺点隐含起来，从而使新层表现了一整套连续的功能。然后，让程序的其余部分调用些子程序而不是直接

调用旧代码。成层设计的好处是：（1）它可以使你避免与拙劣的旧代码直接打交道；（2）一旦你想废弃那些旧代码中的子程序的话，只要修改一下接口层就可以了。

标准化技术。标准化技术是深受欢迎的。一个系统使用的奇特的、非标准的技术越多，当别人第一次读它时就会越感到可怕，也越难理解。应该通过采用常用的、标准化的技术使得人们在阅读它时是一种熟悉的感觉。

7.5.6　检查表

高层次设计

本表给出了在评估设计质量时，通常要考虑一些问题。本表是 3.4 节中结构设计检查表的补充，这个表所考虑的主要是设计质量。3.4 节中的检查表则侧重于结构设计和设计内容。这个表中的某些内容是相互重合的。

- 是否使用了往返设计方法，应从几个方案中选择最好的，而不是首次尝试就确定方案。
- 每个子程序的设计是否都和与其相关的子程序设计一致？
- 设计中是否对在结构设计层次上发现但并未解决的问题作了充分说明？
- 是否对把程序分解成目标或模块的方法感到满意？
- 是否对把模块分解成子程序的方法感到满意？
- 是否明确定义了子程序的边界？
- 是否是按照相互作用最小的原则定义子程序的？
- 设计是否充分利用了自顶向下和自底向上法？
- 设计是否对问题域要素、用户接口要素、任务管理要素和数据管理要素进行了区分？
- 设计是智力上可管理的吗？
- 设计是低复杂性吗？
- 程序是很容易维护的吗？
- 设计是否将子程序之间的联系保持在最低限度？
- 设计是否为将来程序可能扩展作了准备？
- 子程序是否是设计成可以在其它系统中再使用的？
- 低层次子程序是高扇入的吗？
- 是否绝大多数子程序都是低或中等程度扇出的？
- 设计易于移植到其它环境吗？
- 设计是简练的吗？是不是所有部分都是必要的？
- 设计是成层的吗？
- 设计中是否尽量采用了标准化技术以避免奇特的、难以理解的要素？

7.6　小结

- 设计是一个启发的过程。固执地坚持某一种方法只会抑制创造力，从而产生低质量的程。坚持设计方法上有一些不屈不挠的精神是有益的，因为这可以迫使你对这种方法进行充分理解。但是，一定要确信你是在不屈不挠而不是顽固不化。

- 好的设计是通过迭代逼近得到的：你尝试过的设计方案越多，你最终所确定的设计方案也越好。
- 结构化设计比较适合于小规模的子程序组合，同时，它对于功能变化可能性比数据大的问题也是较适用的。
- 面向对象设计更适于子程序与数据的组合，通常在比结构化设计抽象程度更高些的层次上适用。它尤其适合于数据变动可能性大于功能变动可能性的问题。
- 设计方法仅是一种工具，你对工具运用得好坏决定了你所设计的程序的质量。利用不好的设计方法，也可能设计出高质量的程序。而即使是好的方法，如果运用不当的话， 也只能设计出拙劣的程序。但不管怎样，选择正确的工具更容易设计出高质量的软件。
- 许多关于设计的丰富而有用的信息都是在本书之外的。在这里所论述的，不过是冰山的一角而已。

第八章　生成数据

目录

相关章节

变量命名：见第 9 章
使用变量时的一些考虑：见第 10 章
使用基本数据类型：见第 11 章
使用复杂数据类型：见第 12 章
在结构设计阶段定义数据：见 3.4 节
说明数据：见 19.5 节
数据结构布置：见 18.5 节

本章的内容既包括高层次的子程序、模块和程序设计中要考虑的问题，也包括对数据实现问题基本要素的讨论。

数据结构在创建阶段能带来的收益大小，在某种程度上是由它对创建前的高层次工作影响大小决定的。好的数据结构所带来的收益往往是在需求分析和结构设计阶段体现出来的。为了尽可能地利用好的数据结构带来的收益，应在需求分析和结构设计阶段就定义主要数据结构。

数据结构的影响同时也是由创建活动所决定的。由创建活动来填平需求分析与结构设计之间的壕沟是很正常也是很受欢迎的。在这种微观问题上，仅靠画几张蓝图来消除一致性的缺陷是远远不够的。本章的其余部分将论述填平这一壕沟的第一步——生成进行此项工作的数据。

如果你是一位专家级的程序员，本章的某些内容对你来说可能已是司空见惯了。你可以浏览一下标题和例子，寻找你不熟悉的内容看就可以了。

8.1　数据识别

有效生成数据的第一步是应该知道该生成什么样的数据结构。一张良好的数据结构清单是程序员工具箱中的一件重要工具。对数据结构基本知识的介绍不在本书范围之内。但你可以利用下面的“数据结构识别测试题”来看一下自己对其知道多少。

8.1.1　数据结构识别测验题

每遇到一个你所熟悉的词可以计 1 分。如果你认为你知道某个词但并不知道其确切内容，计 0.5 分。当作完题后，把你的得分加到一起，然后再根据测验题后面的说明来解释你的得分。

____抽象数据类型
____数组
____B 树
____位图
____逻辑变量（布尔变量）
____字符变量
____命名常量
____双精度
____枚举流
____浮点
____散列表
____堆
____索引
____整型
____链表
____文字型
____局部变量
____查找表
____指针
____队
____记录
____回溯
____集合
____堆栈
____字符串
____结构化变量
____树
____联合
____数值链
____变体记录
____最后得分

以下是得分解释办法（可随便些）：

0～14 分　你是个初级程序员，可能是计算机专业一年级的学生，或者是一个正在自学第一种语言的自学者。如果读一下后面列出的书的话，你将会学到很多。本章的许多论述都是针对高级程序员的，如果你读完那些书再阅读本书将更有益。

15～19 分　你是个中级程序员或是个健忘的富有经验的程序员虽然你对表中许多概念都已经很熟了，但最好还是先阅读一下后面列出的书。

20～24 分　你是个专家级的程序员，你的书架上很可能已经插上了后面所列出的书。

25～29 分　关于数据结构，你知道的比我还多！可以考虑写一本你自己的专著（请别忘了送我一本）。

30～32 分　你是个自大的骗子。“枚举流”、“回溯”和“数值链”并不是指数据结构的，是我故意把它们加进去以增加难度的，在阅读本章其余部分之前，请先阅读引言中关于智力诚实性的内容。

以下是关于数据结构的一些不错的书：

Aho, Alfred V., John E. Hopcroft 《Data Structure and Algorithms, Reading，Mass》Addison-Wesley，1983。

Reingold, Edward M 和 Wilfred J.Hansen《Data Structures》, Boston: Little, Brown, 1983。

Wirth, Niklaus,《Algorithms and Data Structures》, Englewood Cliffs,N.J.;Prentice Hall, 1986。

8.2　自建数据类型的原因

程序语言所赋予你的阐明自己对程序理解的最强有力的工具之一便是程序员定义的变量类型。它们可以使程序更容易阅读。如果使用 C、Pascal 或其它允许用户定义类型的语言，那么一定要利用这一点。如果使用的是 Fortran，Generic Basic 等不允许用户定义变量类型的语言，那接着看下去，或许读完本节后你就想换一种语言了。

为了说明自定义类型的效力，让我们来看一个例子。假设你正编写一个把 x、y、z 坐标转换为高度、经度、纬度坐标的程序，你认为可能会用到双精度浮点数，但是在十分确定之前，你只想用单精度浮点数。这时，可用 C 中的 typedef 语句或 Pascal 中的 type 说明来为坐标专门定义一种变量（当然也可用其它相当的语言）。以下是在 C 中是如何建立这种变量的：

```
typedef float Coordinate_t;          /* for coordinate variables */
```

这个类型定义说明了一种新的类型，Coordinate_t，在功能上它与 float 是一样的，为了使用这种新类型，应用它来说明变量，就像用 float 来预先定义一种类型一样。以下是一个用 C 写成的例子：

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

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

在这段代码中，变量 x，y，z 和变量 latitude, longitude, elevation 都是 Coordinate_t 类型的。

现在，假设程序发生了变化，发现最终还是得用双精度变量。由于你为坐标专门定义了一种类型，因此所要做的就是改变一下类型定义而已。而且只需在一个地方进行改动；在 typedef 语句中。下面是改变后的类型定义：

```
typedef double Coordinate_t; /* for coordinate variables */   ——原浮点已改为双精度类型
```

下面来看第二个例子，它是用 Pascal 写成的。假设你正在生成一个工资发放系统，其中雇员的名字至多有 30 个字母。你的用户告诉你从来没有任何人的名字长度超过 30 个字母。你是否该在程序中把数字 30 作为文字型变量来使用呢？如果你这样作的话，那你就过于轻信你的用

户了。更好的办法是为雇员名字定义一种类型：

```
Type
    EmployeeName_t = arrayy[1..3] of char;
```

当引入数组或字符串时，最好定义一个命名常量来表示数组或字符串的长度，然后在类型定义中使用这个命名常量，在程序中，你会许多次用到了这个常量——以下是你将使用它的第一个地方，它看起来是这样的：

```
Const
    Namelenght_c = 30；——这里是命名常量的说明
…
Type
    EmployeeName_t = array[1..NameLength_c] of Char；——这里是命名常量使用的地方
```

一个更有说服力的例子是将自定义类型与信息隐蔽的思想结合在一起。在某些情况下，你想隐蔽的信息是关于数据类型的。

在坐标例子中用 C 写成的程序已经在走向信息隐蔽的途中了。如果你总是用 Coordinate_t 而不是用 float 或 double，事实上已经隐蔽了数据的类型。在 C 或 Pascal 中，这些便是语言本身能为信息隐蔽所做的一切，其余部分便是你或后来的使用者必须遵守这个规则，不查看 Coordinate_t 的定义。C 和 Pascal 所赋予的都是广义的而不是狭义的信息隐蔽能力。

其它像 Ada 和 C++等语言则更进一步支持狭义的信息隐蔽。下面是在 Ada 语言中，用包来定义 Coordinate_t 的代码：

```
package Transformation is
    type Coordinate_t is private;      ——这个语句说明 coordinate_t 作为包的专用说明
…
```

以下是在另一个包中使用 Coordinate_t 的代码：

```
with Transformation:
…
procedure Routine1（…）…
    latitude:      Coordinate_t;
    longitude:     Coordinate_t;
begin
    -- statements using latitude and longitude
    …
end Routine1;
```

注意 Coordinate_t 在包定义中是说明为专用的，这意味着只有 Transformation 包中的专用部分才知道 Coordinate_t 的定义，而程序其余部分则都不知道。在有一群程序员的开发环境中，只有包的定义部分才是开放的。对于从事另一个包的程序员来说，他是不可能查寻 Coordinate_t 的类型的。在这里，信息是狭义隐蔽的。

这些例子已经阐明了建立自己的类型的几条理由：

- 使得改动更加容易。建立一种新类型工作量极小，但这却可以带来极大的使用灵活性。

- 避免过度分散的信息分布。硬性类型会使程序中充斥着数据类型细节，而不是使其集中在一个地方。这正是 6.2 节中所讨论的集中化原则。
- 为了增加可靠性。在 Ada 和 Pascal 中，可以定义类似 Age_t = 1~99 的类型。然后，编译程序会产生运行检查信息，以保证 Age_t 类型总是在 1~99 的范围内。
- 为了补偿语言的弱点。如果语言中不具备某种定义好的类型，可以自己定义它。例如，C 中不含逻辑类型，通过建立你自己的类型，很容易弥补这个缺点：

```
typedef int Boolean_t;
```

8.3　自建数据类型的准则

以下是几条生成自己的类型时应牢记的准则：

建立具有面向功能名称的类型。应避免用暗指计算机本身数据类型的名称。要使用代表实际问题某一部分的名称。在上例中，为坐标建立的新类型命名就很恰当，因为它代表了客观世界中的实体。同样，你也可以为现金、工资发放代号、年龄等客观世界中的实体建立新变量。

要避免使用含有已定义变量类型的名称。比如像 BigInteger 和 LongString 等指的是计算机数据而不是客观世界中实体的名称就应避免使用。建立自己的类型其最大优点是，可以在程序及其实现语言之间建立一个绝缘层，而指向程序语言类型的名称则会破坏这个绝缘层，使用已定义的类型不会带来任何优点。面向问题的名称可以增加易改进性，并且使数据说明成为自注释的。

避免使用已定义类型。如果类型存在变动的可能性，那么除了在 typedef 和 type 定义之外，不要再使用已定义的类型。建立面向功能的类型是非常容易的，而改变程序中该类型的数据是非常困难的。而且，当使用自建的面向功能类型说明变量时，也同时对变量进行了说明。Coordinate_x 所告诉你的关于 x 的信息要比 float x 多得多。因此应尽可能使用自建类型。

不要对已定义类型重新定义。改变标准类型的定义往往令人困惑。例如，语言中已经有了 Integer 类型，而你又自建了叫作 Integer 的类型。这样，程序的阅读者往往会记住你所定义的 Integer 的含义，而仍把它当作语言中的标准 Integer 类型。

定义替换类型以增强移植性。与避免重新定义标准类型的建议相反，你可能想为标准类型定义一种替换类型，从而使得在不同的硬件环境下变量所代表的都是同一实体。例如，你可以定义 INT 类型来代替标准的 int 类型，它们之间的唯一区别便是字母的大小写。但当你把程序移到新的硬件环境下时，你只要重新定义一个 INT 类型，就可以在新的硬件环境下使得数据类型与旧环境下相容。

如果你所用的语言是不区分大小写的，你将不得不使用其它办法来对标准类型和替换类型加以区别。

使用其它类型来建立新类型。你可以在已经建立的简单类型的基础上建立复杂类型。这种变量类型可以进一步推广你用原来类型所达到的灵活性。

8.4　使变量说明更容易

本节论述的是怎样才能更顺利地进行变量说明。显然，这是件很容易的事情，你甚至会认

为在书中专门用一节来论述这个问题是小题大做。但不管怎样，你毕竟在建立变量上花费了许多时间，因此，养成这方面的良好习惯以避免不必要的失败和徒劳的努力是非常必要的。

8.4.1　使用模框（template）进行变量说明

在一个文件中存储一个变量说明模框。需要说明新的变量时，你可以把这个模框调入程序，对其进行编辑以说明新变量。下面是用 C 写成的一个模框：

```
extern    *    *;        /*   */
static    *    *;        /*   */
          *    *;        /*   */
```

这个模框有几个优点。首先，很容易从中选择出与你要求最接近的行，然后删掉其余各行；第二，每行中“*”的作用是占有位置，这使得进入每行的编辑位置都非常容易；第三，如果你忘记了更改“*”，那么一定会产生语法错误，从而起到了提醒的作用；第四，使用模框可以保持说明形式的一致性；最后，预留的注释空格将提醒你在说明变量时进行注释，这简化了以后的程序注释工作。

不过，不要以为你必须采用与上例完全相同的模框，尽管按自己的想法去建立自己的模框，只要它能降低变量说明工作量、增强代码可读性并且使调试更容易就可以了。我的一个朋发给自己找出了一个注释变量的理由。她名叫 Chris，她的模框是这样的：

```
*    *    { Chris    is a jerk! }
```

8.4.2　隐式说明

有些语言具有隐含变量说明功能。例如，如果在 Basic 或 Fortran 中你没有说明就使用了某些变量，那么编译程序将自动说明。

隐式说明是所有语言特性中的最具危害性的特性之一。

如果你用 Fortran 或 Basic 编过程序，那你一定知道要找出 ACCTNO 的值不正确是多么的困难。而你最后却发现这是由于 ACCTNUM 被重新初始化为零的原因，如果你所用的语言不要求对变量做出说明，那么，这种错误是很常见的。

而如果你所用的程序语言要求对变量作出说明，那么在出现上例中的错误之前，你必须首先先犯两个另外的错误才行。首先，你要错误地同时把 ACCTNUM 和 ACCTNO 放入了子程序中；其次，你要错误地在子程序的说明段中同时说明了这两个变量。第二个错误是很难出现的，因此，要再犯上例中的错误几乎是不可能的。要求你对变量进行显式说明的语言其主要优点之一，便是可以使你在使用变量时更加谨慎。但如果你用的是具备隐式说明功能的语言，那你该怎么办呢？以下给出了一些指导原则：

关闭隐式说明功能。有些编译程序允许关闭这一功能。比如，在 Fortran 中，可以使用一个叫 IMPLICIT NONE 的语句，这并不是 ANSI FORTRAN77 中的标准语句，但许多 Fortran 中都有这一扩展。

说明所有变量。即使编译程序不要求，但你每次使用新变量时，都要坚持对它做出说明。这样并不能避免所有的错误，但是至少可以避免某些错误。

使用命名约定。建立一个关于常用后缀（如 NUM 和 NO）的命名约定，这样，当你想要

用一个变量时就不会误了两个。

检查变量名。利用你的编译程序或其它工具软件产生的参考表。许多编译程序都会列出你在子程序中使用的所有变量，从而使你发现 ACCTNUM 和 ACCTNO 中的错误。同时，它也会指出说明但并未使用的变量。

8.5　初始化数据的准则

在编程中，最大的错误原因之一便是对数据不恰当的初始化。开发有效地避免初始化错误的技术，可以节约大量的调试时间。

不恰当初始化产生的问题，是由某一变量带有你没有预料的初值引起的。这可能是由下述原因中的任何一种引起的：

- 这个变量未被赋过任何值。其值是在程序开始运行时，由在它的内存中的位置偶然的值决定的。
- 变量的值已经过时了。变量在某点中被赋过值，但是这个值已不再有效了。
- 变量的一部分被赋了值，而另一个部分没有被赋值。如果你使用的是指针变量，那么常见的错误是用指针分配了内存，却忘记了对指针所指的变量进行初始化。这与对变量根本不赋值的效果是一样的。

这种情况往往有几种表现形式。你可能初始化了结构的某一部分而没有初始化另一部分；也可能会分配内存，然后初始化变量，但指向变量的指针却没有被初始化，这意味着你是随意地拿一段内存并赋予它们一些值，这段内存可能是存储数据的，也可能是存储代码的，也可能是被操作系统所占用的。这种错误的表现形式是多种多样的，而且它们之间往往每次都有着天壤之别。这使得调试指针错误要比调试其它错误困难得多。

下面是如何避免初始化错误的一些准则：

检查输入参数的有效性。初始化的另一种有价值的形式是检查输入参数的有效性。在赋给输入参数任何值之前，要首先确认它是合理的。

在使用变量的位置附近对其进行初始化。有些程序员喜欢在子程序开始的一个地方对所有变量进行初始化，以下是用 Basic 语言写成的例子：

```
' initialize all variables
Idx = 0
Total = 0
Done = False
...

'lots of code using Idx and Total
...

'code using Done
while not Done
   ...
```

另外一些程序员则尽可能在每一次用到变量的位置附近对其进行初始化，下面也是用 Basic 写成的例子：

```
Idx = 0
'code using Idx
...

Total = 0                    —— Total 在使用位置附近初始化
'code using Total
...

Done = False                 —— Done 在使用位置附近初始化
'code using Done
while not Done
        ...
```

第二个例子要好于第一个，其中有几个原因。在第一个例子中，当运行到使用 Done 的代码时，Done 很有可能已经被改变过了。即使在初次写程序时不会出现这种情况的话，那么以后你对其进行修改时，很可能会出现这种情况。第一种方法的另一个问题是，把所有变量集中进行初始化，会给人以这些参数将在程序中随处都被用到的印象，而事实上 Done 只在程序结束前才被用到；最后，当对程序进行改动时（这是很可能的，起码在调试时要修改），可能会把 Done 包含在循环中，从而需要对 Done 重新进行初始化，在这种情况下，第二个例子中的代码不会有什么太大的变化。而第一个例子中的代码则可能会产生讨厌的初始化错误。

这也是邻近原则的一个例子：把相关的操作放在一起。这一原则也适用于把对代码的注释放在相应的代码附近，把循环设定代码放在循环附近，把注释放在非循环代码中。

要特别注意计数器和累加器。变量 i、j、k 和 Sum、Total 等往往代表计数器和累加器。常见的错误是在下次用到它们时，忘记了对其进行清零操作。

查找需要重新进行初始化的地方。首先问自己一下有哪些地方需要重新进行初始化。重新初始化的原因可能是由于变量在循环中被用过多次，也可能是由于变量在对子程序的调用中间要保持原值且需清零。如果需要重新初始化的话，要确保初始化语句是在被重复的代码段中。

对命名常量只初始化一次，用可执行代码初始化变量。如果要用变量来模仿命名常量，那么在程序的开始对它们进行一次初始化是可以的，在 Fortran 中可以用 Data 语句来做到这点。在其它语言中，可以在 Startup()子程序中初始化它们。

应在使用变量的位置附近的可执行代码对其进行初始化。最常见的改动之一是改动某一子程序，变一次调用它为多次调用它。在 DATA 语句或 Startup()子程序中被初始化的变量，在程序中不能被重新初始化。

按照所说明的对每个变量进行初始化。虽然其地位无法与在变量使用位置附近对其初始化相比，但是按照所说明的初始化变量，仍然是防错性编程中的一件有力工具。如果你养成习惯，那就可以有效地防止初始化错误。下例就保证了 student_name 在每次调用含有它的子程序时，都将被重新初始化。

```
char student_name[NAME_LENGTH+1] = {'\0'};          /* full name of student */
```

利用编译程序的警告信息。许多编译程序都会对使用未初始化的变量进行警告。

设置编译程序使其自动初始化所有变量。如果你的编译程序支持这种选择项，那么让它对所有变量进行初始化是非常简单的。但是，由于对编译程序的依赖性。当把程序移到另一台编译程序不同的机器上时，则有可能产生问题。这时要确保你对所用的编译程序进行了注释，否则是很难发现程序对编译程序有依赖性。

使用内存存取检查程序来查找无效的指针。在某些操作系统中，操作系统代码会自动查找无效指针，在其它情况下，就只有依靠自己了。不过，你也不一定非得靠自己。可以买一个内存存取检查程序来检查程序中的指针操作。

在程序开始初始化工作内存。把工作内存初始化到某一个值可以帮助发现初始化错误。可以采取以下任何一种方法：

- 可以用程序内存填充程序来赋予内存某一已知值，这个值用 0 比较好，因为它保证未初始化指针指向低内存，比较容易发现它们，在 80X86 处理器中，16 进制 0CCH 比较好，因为它是机器的断点中断码。如果是在调试中运行数据而不是代码，你就会进入断点。使用值 0CCH 的另一个优点是在内存转储中它很容易辨认。
- 如果你使用的是内存填充程序，可以每次改变一个填充值，用这种方法来检查一下运行环境下隐藏的错误。
- 可以用程序在软件运行时初始化工作内存。虽然使用内存填充程序的目的是把错误暴露出来，但这种技术的目的则是隐藏它们。通过每次把工作内存由相同的值充满，可以保证在每次运行时程序不会被启动时的随机因素干扰。

8.5.1　检查表

数据生成

生成类型

- 程序中是否对每种可能变动的数据都使用了不同的类型？
- 类型名称是面向客观世界中的实体类型而不是面向程序语言中的类型吗？
- 类型名称是否有足够的表现力来帮助说明数据？
- 避免重新定义已定义的类型了吗？

说明数据

- 是否使用了模框为简化数据说明工作？并用其来统一说明形式？
- 如果所用的语言支持隐式说明，是否采取了补救措施？

初始化

- 是否每个子程序都对输入参数进行了检查以保证其有效性？
- 是否在使用变量的位置附近对其进行初始化的？
- 是否恰当地对计数器和指针进行了初始化？是否在必要时对其进行了重新初始化？
- 在反复执行的代码段中，是否对变量进行了恰当地重新初始化？
- 用编译程序编译代码时是否是无警告的？

8.6　小结

- 在你的工具箱中需要一张全部数据结构的清单，以便用最合适的方法处理每一种问题。
- 建立自己的数据类型，以增加程序的可变动性，并使其成为自说明的。
- 数据初始化很容易产生错误，因此应采用本章推荐的技术来避免由意外初始值所产生的错误。

第九章　数据名称

目录

相关章节

生成数据：见第 8 章
使用变量时要考虑的问题：见第 10 章
说明数据：见 19.5 节
格式化数据说明：见 18.5 节

本章论述的是如何对数据进行恰当命名。这个主题对有效编程是非常重要的，但是在进行恰当命名时要考虑的几十个问题中，我却只读过对其中两到三个的讨论。大多数编程课本只用几段来论述名称缩写的选择问题，讲的也都是些关于这方面的陈词滥调，而完全寄希望于你自己去解决问题。对于另一个极端，使用过多关于命名的信息将你淹没，我认为简直就是犯罪。

9.1　选择名称

在给变量命名时，你不能像给小狗起名时那样仅仅挑有趣或好听的名字。但除了实体不同之外，变量与变量名和狗与狗名实际是同一回事。

因此，一个变量的好坏在很大程度上是由其名字决定的。所以在选择变量名时一定要谨慎。

以下是一段使用不恰当变量名的例子（用 C 写成）：

```
X       = X - XX;
XXX     = Aretha + SalesTax( Aretha );
X       = X + LateFee（X1，X）+ XXX;
X       = X + Interest(X1，X);
```

这段代码是干什么的？X1，XX 和 XXX 代表的是什么呢？Aretha 的意思是什么？如果某人告诉你，这段程序是根据收支平衡和新的购买情况来计算顾客和全部账单的，那么你将使用什么变量来代表那些新购买活动呢？

以下是一个同样内容的新程序，这里，上述问题就很容易回答了。

```
Balance         = Balance - LastPayment;
MonthlyTotal    = NewPurchases + SalesTax( NewPurchases );
Balance         = Balance + LateFee( CustomerID, Balance ) + MonthlyTotal;
Balance         = Balance + Interest( CustomerID，Balance );
```

通过比较两段代码，我们发现好的变量名是易读、易记而且是恰当的。可以使用几条通用原则来达到这些目的。

9.1.1　命名时要考虑的最重要问题

在给变量命名时，考虑的问题是变量名称是否完全而又准确地描述了变量所代表的实体。一个有效的方法是用自然语言（如英语）将变量所代表的实体描述出来。往往这一描述本身便是最好的名称，因其不含缩写它很容易读懂，又由于它是对实体的全面描述。因此不会与其它实体相混淆，而且因为它与概念相近，所以也很容易记。

比如要用一个变量来代表美国奥林匹克队的人数，你可以对其命名为NumberofPeople0nTheUSOlympicTeam。对代表自 1896 年以来国家队在奥林匹克运动会上最多得分的变量可以用 MaximumNumberofPointsSince1986 作为其名称。而用 InterestRate 或 Rate 作为代表目前利率的变量名要比用 r 或 x 好得多。

你应该可以发现这些名字的两个特点。首先，它们很容易解释。事实上，你根本不需要解释，它们的意思是一目了然的；第二条则是其中有些名字很长，长得根本不实用。稍后我们将讨论这一问题。

下面是一些变量名的例子，同时列出了好的和坏的。

变量代表的实体	恰当的名称	不恰当的名称
火车速度	Velocity、TrainVelocity、VolocityInMPH	VELT，V，TV，X，X1
今天日期	CurrentDate、CrntDate	CD，Current，C，X
每页行数	LinesPerPage	LPP , Lines , X , X1

CurrentDate 和 CrntDate 是恰当的名字，因为它们全面准确地描述了“今天日期”这一含义。而且，它们用的是明显的单词。程序员们往往忽视使用平常的词，而事实上这是最简单的解决办法；CD 和 C 太简短了，说明不了任何问题；Current 并没有说明现在的什么？是总统还是赌马的结果？Date 像是一个不错的名字，但究竟是什么时候的 Date？是基督出生那天吗？X，X1 在任何情况下几乎都是不好的名字，因为它们通常都是代表未知量的，如果你要它代表其它实体时，往往会引起误会。

9.1.2　面向问题

一个好记的名字通常是面向问题而不是解决问题的。一个恰当的名字往往说明是“什么”而不是“怎样”。通常，如果一个名称指向计算的一方面而不是指向问题，那么可以认为之是个“怎样”而不是“什么”的名称。要避免使用这种名称而要使用面向问题的名称。

雇员数据的记录可称为 InputRec 或 EmPloyeeData，InputRec 是一个计算机术语，指的是输入和记录；EmPloyeeData 指的是问题域而不是计算方面。同样，对一个表示打印机状态的变量

来说，BitFlag 要比 PrinterReady 专业化得多。在计帐系统中，CalcVal 要比 Sum 更加专业化。

9.1.3　最佳名称长度

名称的最佳长度应介于 MaximumNumberOfPointsSincel896 和 x 之间。太短 的名字 往 往表示不出完整的意思，而用 x1 和 x2 来表示的问题，即使你能找出 x1 代表的是什么，也很难发现 x1 和 x2 之间有什么联系。太长的名字难以输入，而且会对软件的可视结构产生破坏作用。

Gorla 和 Benander 在 1990 年发现当变量名长度在 10 到 16 个字母时，COBOL 程序最容易调试。而变量名称长度在 8 到 20 个字母时，程序调试容易几乎是一样的。这一准则并不是说你必须把所有变量名长度都限制在 9 到 15 或 10 到 16 个字母之间。但这确实要求你检查一下，程序中变量名长于这个标准的那些变量，确保清楚程度符合需求。

以下是对一些变量名的评价，或许会给你一些启迪：

名称太长的：

NumberofPeopleOnTheUS0lympicTeam

NumberOfSeatsInTheSaddleDome

MaximumNumberOfPointsSince1896

名称太短的：

N，NP，NTM

N，NS，NSISD

M，MP，Max，Points

合适的：

NumTeamNumbers，TeamMbrCount，cTeamMbrs

NumSeatslnDome，SeatCount，cSeat

MaxTeamPoints，Record Points，cPoints

9.1.4　变量名的作用域

短的变量名总是不好的吗？当然不总是。当你用 i 作为一个变量名时，这个较短的长度就可以说明某些问题，比如，这个变量是一个临时变量，只在有限的操作范围内才是有效的。

当程序员读到这样一个变量时，他应该能猜到这个变量只在几行内使用。如果你把某个变量称为“i”，你就等于在说“这个变量仅作为循环计数器或数组索引数使用，在这几行代码外没有任何意义”。

由 W．J Hansen 进行的一项调查表明，较长的名字适于较常使用的变量或全局变量。而较短的名字则适于局部变量或循环变量（1980）。但短名字会产生许多问题，有些程序员把避免使用它们作为防错性编程的准则。

9.1.5　变量名中的计算值限定词

许多程序中含有带有计算值的变量：totals，averages，maximums 等等。如果你用限定词诸如（Ttl，Sum，Avg 等）来改动变量，那要把它们放在最后。

这种方法有几个优点。第一，变量名中最有意义的部分——表达变量名大部分意义的部分，

被放在最前面；第二，通过建立这种约定，在同一程序中同时使用 TtlRevenue 和 RevenueTtl 会避免因此而引起的混淆；第三，像 RevenueTtl，ExpenseTtl，RevenueAvg 和 ExpenseAvg 这样的一系列名字有一种令人愉快的相似感。而像 TtlRevenue，ExpenseTtl，RevenueAvg 和 AvgExpense 则不具备这种相似性。最后，一致性可以改善可读性并简化了维护。

这个准则的例外是 Num 放在变量名前面时表示全部，如 NumSales 表示的全部商品的数额；当 Num 放在变量名后面时，它表示下标，SaleNum 表示现在标出的商品是第几个。Numsales 末尾的 S 也是表示两者意义区别的一个标志。但是，由于过分频繁地使用 Num 会产生混淆，所以往往用 Count 来表示总数，而用 Index 来表示下标。这样，SalesCount 表示的是卖出的总数，而 Saleslndex 则指的是卖出的某一种特定商品。

9.1.6　变量名中的反义词

应恰当使用反义词。使用关于反义词的命名约定可以帮助保持连续性，同时也可以提高可读性。像 first/last 这样一组反义词是很容易理解的，但 first/end 就有些让人摸不着头脑了，以下是一些比较容易理解的反义词。

add/remove	begin/end	create/destroy
insert/delete	first/last	get/release
increment/decrement	put/get	up/down
lock/unlock	min/max	next/previous
old/new	open/close	show/hide
source/destination	source/target	start/stop

9.2　特定数据类型命名

除了对数据命名通常需要考虑的一些问题之外，对特殊数据类型必须给予特殊的考虑。本书将论述循环变量、状态变量、临时变量、逻辑变量、枚举变量和命名常量的命名问题。

9.2.1　循环变量命名

由于几乎每个程序中都含有循环，因此，对循环变量的命名问题加以专门考虑是十分必要的。

简单循环中的循环控制变量的名字往往也是十分简单的，常用的是 i、j、k 等。下面是一个 Pascal 循环的例子：

```
for i:＝FirstItem to LastItem do
    Data[i] := 0；
```

如果这个变量还要在循环外使用，那么应该用比 i、j、k 更能说明问题的名称。比如，你正从文件中读取记录，并且要知道已经读取了多少个记录，那么用 RecordCount 作为其名称似乎更合适些，请看下面的这个 Pascal 程序：

```
RecordCount:= 0
while not eof(InputFile) do
```

```
        begin
        RecordCount := RecordCount + 1;
        ReadLn ( Score [ Recordcount ] )
        end;

{ lines using RecoudCount }
…
```

如果循环体长度较长的话，那就很容易使人忘记它代表的是什么，因此最好给循环控制变量一个富有意义的名字。由于经常进行更改，扩展和拷贝等代码到另一个程序中，因此，大多数有经验的程序员都避免用 i、j、k 这类的名字。

使循环体变长的一个常见原因是嵌套。因此，对于一个有多重嵌套的循环，最好给循环控制变量以较长的名字以便改善其可读性：

```
    for TeamIndex := i to TeamCount to begin
      for EventIndex := 1 to EventCount[ TeamIndex ] do
          Score[ TeamIndex, EventIndex ] := 0
    end;
```

通过精心对循环控制变量进行命名，可以避免它们的交叉：当你想用 i 时误用了 j，或者想用 j 时却又误用了 i。这样做也可以使对数组的存取操作更为明了。Score [TeamIndex, EvenIndex]显然要 Score[i,j]更能说明问题。总之，应尽量避免使用 i、j、k 来命名。如果不得不使用它们的话，那除了把它们用作循环控制变量之外，最好不再用作别的变量名。这一约定是众所周知的，如果不遵守它只会引起别人的困惑。

9.2.2　状态变量命名

状态变量描述的是程序所处的状态。下面论述了它们的命名原则。

用比 flag 更好的名称来命名变量，最好不用 flag 作为状态变量的名字。之所以要避免使用 flag 作为标志名称，是因为它不能告诉你关于这个标志的任何信息。为了清楚起见，应该给标志赋值，并且用枚举类型、命名常量或当作命名常量使用的全局变量对其进行测试。下面是一个在 C 语言中不恰当命名标志的例子。

```
if（ Flag ）...
if（ StatusFlag & 0x0F ）...
if（ PrintFlag ＝＝ 16 ）...
if（ ComputerFlag ＝＝ 0 ）...

Flag          =  0x1;
StatusFlag      =  0x80;
PrintFlag     =  16;
ComputerFlag =  0;
```

如果这个程序不是你写的，而且也没有注释告诉你的话，你是无法知道类似 statusFlag = 0x80 之类的语句到底是要干什么的, 而且你也无法知道 Statusflag 和 0x80 到底是什么意思。以

下是功能与内容相同但清楚得多的程序：

```
if ( DataReady )...
if ( CharacterType  &  PRINTABLE_CHAR )...
if ( ReportType     ==  AnnualRpt )...
if ( RecalcNeeded  ==  TRUE )...

DataReady          = TRUE;
CharacterType      = CONTROL_CHARACTER;
ReportType         = AnnualRpt;
RecalcNeeded       = FALSE;
```

显然，第二个例子中 CharacterType = CONTROL_CHARACTER 的意义要比第一个中 StatusFlag = 0x80 的意义要清楚得多。同样，第二个例子中的条件语句 if (ReportType == AnnualRpt) 也显然要比第一个中的 if (PrintFlag == 16)清楚得多。第二个例子表明，你可借助预先命名的常量或枚举类型来使用这种方法。以下是如何利用枚举类型和命名常量来设置上例中用到的值，仍用 C 来实现：

```
/* values for DataReady and RecalcNeeded */

#define TRUE       1
#define FALSE      0

/* values for CharacterType */

#define LETTER                  0x01
#define DIGIT                   0x02
#define PUNCTUATION             0x04
#define LINE_DRAW               0x08
#define PRINTABLE_CHAR          ( LETTER | DIGIT | PUNCTUATION | LINE_DRAW )

#define CONTROL_CHARACTER       0x80

/* values for ReportType */

Typedef enum {     DailyRpt，MonthlyRpt,，QuarterlyRpt，
                   AnnualRpt，AllRpts〕REPORT_TYPE_T;
```

当你发现自己“侦破”了一段代码时，应该考虑对变量重新命名。侦破一桩凶杀案是可以的，但你不应该去“侦破”一段代码。代码应该是具有良好可读性的。

9.2.3　临时变量命名

临时变量用来保存中间运算结果，如用作暂时保留某个位置或保留内务操作值。通常把它们叫做 TEMP 或 X 等没有什么意义的名字。临时变量的使用往往标志着程序员还没有完全理解

程序。而且，由于名义上给了它一个“临时”的状态，因而程序员们在处理它们时往往会采取漫不经心的态度，从而增大了出错机会。

要警惕“临时”变量。通常，暂时保留某些值是完全必要的。但如果在你的程序中出现很多临时变量的话，则说明你对它们在程序中作用和使用它们的目的还不清楚。先让我们看一下下面用 C 语言写成一个例子：

```
/*  Compute root 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 );
```

把由公式 sqrt (b^2 - 4 * a * c) 计算出来的值储存起来是个不错的想法，尤其是在后面还有两处用到了它的情况下。但是用 TEMP 作为它的名称不能告诉你关于这个变量意义的任何信息。一个较好的作法是下面这个例子：

```
    /*  Compute roots of a quadratic equation.
        This assumes 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 );
```

事实上两段代码是完全一样的，但是通过采用更准确也更能说明问题的变量名，大大改善了其可读性。

9.2.4　逻辑变量命名

以下命名逻辑变量的几条准则：

记住一些典型的逻辑变量名。以下是些非常有用的逻辑变量名：

Done。用 Done 来表示某项工作已经完成了。这个变量可以表示子程序或循环是否已经完成。当某项工作没有完成时，把 Done 的值赋为 False；当某项工作已经完成时，把 Done 的值赋为 True。

Error。用 Error 来表示发生了错误。当没有错误时将 Error 的值赋为 False；当有错误时将其
赋为 True。

Found。用 Found 来表示是否找取了某个值。当搜寻数组来查找某一值或搜寻某一文件表查找某一雇员的识别卡时，没有找到某一值时将其值赋为 False，而一旦找到这个值。则把 Found 值赋为 True。

Success。用 Success 来表示某一操作是否已成功地完成。当某一程序无法完成时，将 Success 的值置为 False；而当某一操作已经完成时，将 Success 的值置为 True。如果可能的话，可以用比 Success 更准确更具有表达力的名称来代替它，用这个新名称应可以精确地表达出到底是什么已 成功地完成了。比如当某一程序完成处理后就可以认为是成功时，就可以用 Processingcomplete 来代替 Success。如果当找到某一值就可以为程序是成功的时，可以用 Found 来代替它。

用旧含非真即假的名字来给逻辑变量命名。像 Done 或 Success 等之所以是恰当地逻辑变量名，是因为它们的状态是非真(True) 即假 (False)的。某项工作或者完成了或者没完成；或者是成功的或者不成功，不会有第三种状态。而类似 Status 或 SourceFile 等则是不恰当的名字，因为看不出它们的状态是非真即假的。如果 Status 的值是 True，那它是否意味着某种物质有状态呢？任何物质都有状态。或者说当其值是 True 时，意味着某种物质的状态很好，而为 False 时则意味着状态不好？仅从 Status 这个名称是无法回答这些问题的。

为清楚起见，最好用 Error 或 Status_OK 等来代替 Status；用 SourceFileAvailable 或 SourceFileFound 来代替 Source。

有些程序员喜欢用 Is 用为逻辑变量名的前缀，于是变量名就成了一个问句：IsDone？IsError？IsSuccess？当用 True 或 False 来回答上述问题时，就等于给变量赋了值。这样做的好处是可以避免那些不恰当的名称。如 IsStatus 显然没有任何意义。

使用肯定的逻辑变量名。否定式的变量名如 NotFound、NotDone 和 Notsuccessful 等在“非”运算中是很难读懂的，如：

```
if not NotFound
```

这类名字应该用 Found，Done，Successful 等来代替，以方便“非”运算。

9.2.5　枚举类型命名

当使用枚举型变量时，可以通过使用相同的前缀或后缀表示某一类型的元素都是属于同一组的，如下面的这段 Ada 程序所示：

```
type COLOR is ( COLOR_RED, COLOR_GREEN，COLOR_BLUE );
type PLANET is ( PLANET_ERATH, PLANET_MARS, PLANET_VENUS );
```

9.2.6　常量命名

对常量来说，应该用它所代表的抽象实体而不是数值来命名。FIVE 是一个很不恰当的常量名称（不管它代表的数值是否是 5.0）；CYCLES_NEEDED 则是个恰当的名称，CYCLES_NEEDED 可以等于 5.0 也可以等于 6.0，而 Five = 6.0 则是个荒唐的语句。由于同样的原因，BAKERS_DOZEN 也是很不恰当的变量名，而 MAX__DONUTS 则要恰当得多。

9.3 命名约定

有些程序员往往坚持标准和约定，这是有其原因的。然而，某些原则和约定过于刻板而且往往是无效的，这只会破坏你的创造力和程序的质量，这实在是个不幸，因为有效的标准和约定是你的工具箱中最为有效的工具之一。这一节将讨论为什么及什么时候和怎样建立你自己的命名标准。

9.3.1 为什么要建立约定

约定可以带来如下好处：

- 它们可以使更多的东西成为独立的。通过做出一个总体决定而不是许多局部决定，你

可以把精力放到更重要的程序特性上。

- 它们可以帮助借鉴其它项目的经验并移植自己的经验。相似的名字可以使你更容易并且更自信地猜测陌生变量的功用。
- 它们可以使你更快地熟悉新项目的代码。与一套连贯的而不是各式各样、东拼西凑的　　代码打交道显然要容易得多。
- 防止一变量多名。如果没有命名约定，你很可能给一个变量取两个或更多的名字。例如，你可以把所有点的个数称作 PointTl 又称作 Ttl_Points。这对于你来说可能没什么，因为你是程序的作者。但对以后要读这个程序的程序员来说，这很可能会使人困惑。
- 弥补语言的缺陷。你可以利用命名约定来仿效命名常量或枚举类型，这一约定可以区分局部、模块和全局变量，也可以并入编译程序所不支持类型的信息。
- 命名约定还可以强化相关项之间的联系。如果你使用的是结构化数据，那么编译程序会自动考虑到这一点；如果所用的语言并不支持结构化数据，可以通过命名约定来补充它。像 AddrmPhone 和 Name 等名称并不表示变量是相联系的。但如果你决定所有的雇员数据变量名都要用 EmP 作为前缀，那么毫无疑问，EmpAddr、EmpPhme 和 EmpName 就是联系的变量了。通过建立伪结构化数据，弥补语言的缺陷。

关键是有约定总比没有约定好，哪怕约定是随意的也罢，约定的效力并不是由某一确定的约定，而是由约定存在决定的，它可以增加代码的结构并减少你的担心。

9.3.2　什么时候使用命名约定

关于这个问题并没有一成不变的答案。但在以下几种情况下，使用命名约定我认为还是值得的。

- 当同时有几个程序员从事一个项目时。
- 计划把程序交给另一个程序员进行修改和维护时（这时命名约定是必不可少的）。
- 当你的程序将由其它程序员来评审时。
- 当程序规模过大，需要按部分来考虑它时。
- 当一个项目中要频繁使用某些不常见的词汇，而又想开始编码时。

命名约定总是有益的，上述准则可以帮助你确定在某一项目中命名约定使用的广泛程度。

9.3.3　正式程度

不同约定的正式程度是不同的。一个简单的约定可能只有一句话，“使用有意义的名称”。略微正式的约定将在下节讨论。更正式些的约定将 9.5 节中论述。一般来说，约定正式程度是由从事一个程序的程序员人数、程序的规模和程序的预测生存期决定的，在小型程序中，严格的约定往往是不必要的。如果是需要几个人协作（可能是在开始，也可能是在程序生存期内的某个时间）的程序，那么可读性往往要依赖正式的命名约定来保证。

9.4　非正式命名约定

绝大多数项目都采用如下所述的非正式命名约定。

9.4.1　与语言无关的约定准则

以下是一些与语言无关的约定准则：

标识全局变量。常见的编程问题之一是误用全局变量。可以在所有的全局变量前面都加上 g_作为前缀来解决。比如看到 g_Running Total 时，程序员就会知道这是一个全局变量，从而把它作为全局变量对待。

标识模块变量。模块变量是在模块内部供几个子程序使用的变量。要能清楚地表明它既不是全局变量也不是局部变量。这可以用在变量前加 m_作为前缀来解决。在 C 中，你可以通过在任何子程序外说明 Static 变量来建立模块层次的数据。这样的变量对文件中子程序来说都是可用的，但文件外的子程序则无法使用。

标识类型定义。类型的命名约定需要有两个功能：它们要明确地指出某一名称是类型名称，同时要可以避免类型名称与变量名称相冲突。为了达到这两个要求，使用前缀或后缀不失为一个好办法。在 Pascal 中，可以使用小写的_t 来表示类型名称，例如 Color_t 或 Menu_t。在 C 中用这种办法稍有些困难，常见的办法是用大写字母组合如 COLOR 或 MENU 来表示类型名，但这可能与命名预处理程序常量相混淆。而“_t”这一约定已经被标准类型 size_t 所采用了，因此可以用“_T″表示类型名称，如 COLOR_T 和 MENU_T。

标识命名常量。需要对命名常量加以标识，以便可以使你知道是在用一个变量（其值可能变动）还是在用一个命名常量给某个变量赋值。在 Pascal 或 C 中，你还可能是在用函数给某一变量赋值。这些语言，并不要求函数使用括号。而在 C 中，即使是不带参数的函数也必须使用括号。

给命名常量命名的一种办法是“_C”作为其名称的后缀。在 Pascal 中，用这种方法可以产生出像 MaxRecs_C 或 MaxlinesPerPage_ C 之类的名字。在 C 中，你可以用“_C”作为其后缀。

标识枚举类型。与命名常量同样的原因，枚举类型也需要被标识出来。即将其与变量、命名常量和函数加以区别。常用的方式是用“_e”或“_E”作为后缀。

标识输入参数。有时输入参数会被错误改动。在像 C 或 Pascal 这样的语言中，当一个被改变过的值返回调用子程序时，必须予以明确地说明。在 C 中，这是用“*”说明的，在 Pascal 中用的是 VAR，Ada 语言中使用的是 out 限定词。在其它语言如 Fortran 中，如果你改变了一个输入值，那么不管你是否愿意，它都将被返回，但假如你建立了命名约定，在约定中规定只用于输入的参数前面要采用 IP 作为前缀，那么当发现在等号左边出现了带有 IP 前缀的变量时，你就可以知道发生了错误。比如在程序中看到 IPMAX = IPMAX＋l 语句，就可以立刻认定它是错误的，因为前缀 IP 表明 IPMAX 的值是不允许变动的。

对名字作格式化以增强可读性。增强可读性常见的两项技术是用分隔字符或大写字母将单词分隔开来。例如，GymnasticsPointTotal 或 Gymnastic_Point_Total 的可读性显然是要强于 GYMNASTICSPOINTTOTAL。C、PASCAL、Ada 和其它语言都允许大小写混用的方式。C、Pascal 和其它语言都允许使用的下划线“_”作为分隔符。

尽量不要混合使用这两项技术，那样会降低可读性。如果坚持采用其中任何一种，那么你的代码的可读性将大为改观，关于变量名第一个字母是否应该大写的问题，激烈的争论已经持续了很长时间（是 TotalPoints 好还是 totalPoints 好？）。但是只要保持一致性，我认为二者区别实际上并不大。

9.4.2　与语言有关的命名约定

应当遵守所使用语言的标准命名约定。

你可以发现许多书中都讲述了语言的形式准则。关于 C、Pascal 和 Fortran 的准则将在下面予以论述。

C 约定

有些命名的约定是只适用于 C 的，可以在 C 中使用这些约定，也可以改变它以使其适应其它语言。

- c 和 ch 是字符变量
- i 和 j 整型下标
- n 是数量
- p 是指针
- s 是字符串
- 预处理程序宏指令是以全部大写来表示的，这通常扩展到包含 typedef
- 变量和子程序名称都是小写的
- 下划线“_”用做分隔符

这些是在 UNIX 操作系统下用 C 语言编程的一些通用约定。在不同的环境下，这些约定是有区别的。在 Microsoft Windows 环境下，C 程序员们往往愿意使用匈牙利命名约定，并且对变量名称是大小写混用的。在 Macintosh 环境下，C 程序员们往往乐于使用类 Pascal 的命名约定给子程序命名。因为 Macintosh 工具箱和操作系统子程序都是为 Pascal 接口设计的。

Pascal 约定

Pascal 中只有几条特殊的约定，你可以在 Pascal 中使用它们，也可以改进它们以适应其它语言需要。

- i、j 和 k 是整型下标
- 变量和子程序名称是以大小写混用的形式出现的

Fortran 约定

Fortran 有一些语言本身固有的命名约定，在 Fortran 中可以使用它们。但我想如果不把它们扩展到其它语言的话，全世界都会因此而感激你的。

- 以字母 I 到 N 开头的变量名是整型的
- I、J 和 K 只能作为循环控制变量
- X、Y 和 Z 是浮点数

9.4.3　命名约定举例

当命名约定准则长达几页时，它们看起来是非常复杂的。事实上，命名约定是完全不必要复杂到可怕程度的，你可以改进它们以适应你的需要。变量名应包括三个方面的信息：

- 变量内容（它代表的是什么）

- 变量数据类型（整型、浮点等）
- 变量在程序结构中的位置——例如，定义变量的模块名称或者一个表示变量是全局的前缀。

以下是 C 和 Pascal 中采用前面讲过的命名原则进行命名的一些例子。我并不是想把这些约定推荐给你，而是想给你一些变量名应包括哪些信息的想法。

Pascal 命名约定示例

约定类别	约定内容
LocalVariable	局部变量名是大小写混用的。这个名称应与潜在的数据类型无关，且应指出这个变量代表的到底是什么
RoutineName()	子程序也是大小混用的（具体讨论见 5.2 节）
m_ModuleVariable	仅供某一模块子程序使用的变量以 m_作为前缀
g_GlobalVariable	全局变是用 g_为前缀
Constant_C	命名常量以_C 作为后缀
Type_t	类型以_t 作为后缀
Base_EnumeratedType	枚举类型是以其基本类型的助记符作为前缀的，如 Color_Red, Color_Blue

C 命名约定示例

约定类别	约定内容
GlobalRoutineName()	公用程序名称是大小写混合使用的
_FileRoutineName()	供某一个模块专用的子程序以下划线作为前缀
LocalVariable	局部变量名是大小写混用的。其名称应与数据类型无关且应指明变量代表的究竟是什么
_FileStaticVariable	模块（文件）变量以一个下划线作前缀
GLOBAL_GlobalVariable	全局变量是以定义它的模块（文件）名称的全大写形式助记符作为前缀的，如 SCR_Dimension
LOCAL_CONSTANT	仅供某一子程序或模块（文件）专用的命名常量是全大写的
GLOBAL_CONSTATNT	全局命名常量名字是全大写的，且以它的模块（文件）名称助记符的大写形式来作前缀的，如 SCR_MAXROWS
TYPE	类型定义是全大写的
LOCAL_MACRO()	仅供某一子程序或模块（文件）专用的宏定义是大写的
GLOBAL_MACRO()	全局宏定义是大写的且以它的模块（文件）名称助记符大写形式为前缀

9.5　匈牙利命名约定

匈牙利命名约定是一整套对子程序和变量进行命名的详细约定。这一约定广泛应用在 C 语言中，特别是在 Microsoft Windows 环境下用 C 编程时，之所以称其为“匈牙利”是因为遵循这一　　约定的命名看起来像是外文。而且其发明者 Charles Simonyi 原来是匈牙利人。

匈牙利命名主要包括三个部分：基本类型、一个或更多的前缀、一个限定词。在下面的例子是在 C 中采用匈牙利命名约定写成的，你很容易就可以对其加以改进以扩展到其它语言中。

9.5.1 基本类型

基本类型是指待命名变量的数据类型。基本类型名称通常不指向程序语言中任何一种已定义的标准数据类型。基本类型是一种更抽象的数据类型，基本类型名称指向的可能是窗口、屏幕区和字体等实体。在匈牙利名称中只能使用一种基本类型。

基本类型是用你为某一特定程序建立的简写代码来表示的，然后对其作标准化以便在这一程序中继续使用。以下是在一个文字处理程序中你可能用到的基本类型：

基本类型	含义
wn	窗口
scr	屏幕区
fon	字体
ch	字符（不是 C 意义上的字符，而是这个文字处理程序将用来在文件中代表字符数据结构意义上的）
pa	段落

使用匈牙利基本类型时，同时也定义了与基本类型使用同样缩写的数据类型。因此，如果使用了如上表列出的基本类型，就会看到像这样的数据说明：

```
WN      wnMain;
SCR     scrUserWorkspace;
CH      chCursorPosition;
```

9.5.2　前缀

前缀放在基本类型前面并描述了变量的使用。与基本类型不同的是前缀在某种程度上是标准化的。表 9-1 中示出了一级标准匈牙利前缀：

表 9-1　匈牙利变量名称前缀

前缀	含义
a	数组
c	数目（如记录、字符等等的个数等）
e	数组的元素
g	全局变量
h	处理
i	数组下标
m	模块层次上的变量
p（lp,np）	指针（长指针、近指针——对于 Intel 机器）

前缀是小写的，并且放在变量名中基本类型的前面。如果需要的话，可以把它与基本类型或与其自身组合使用。例如，一个表示窗口的数组，可以用“a”来表示它是个数组，用“wn”表示窗口，因此可以用“awn”来作这个数组的名称；而对窗口的操作可称为 hwn；cwn 是窗口的数目；cfon 是字体的种类等等。

9.5.3　限定记号

匈牙利名称的最后一个要素是限定词。限定调是名称的描述部分。没有使用匈牙利约定

时也可利用这部分来补偿。在上前面给出的例子——wnMain、scrUserWorkspace 和 chCursorPosition 中，Main、UserWorkspace 和 CursorPosition 都是限定词。在本章前面给出的关于变量命名的准则也适用于限定词。

除了自己生成的限定词以外，匈牙利约定中还对容易在处理时产生混淆的概念给出了标准化限定词。在表 9-2 中，表示了些标准限定词。

表 9-2　匈牙利约定中的标准限定词

限定词	含义
Min	数组或其它表中绝对的第一个元素
First	数组中需要进行处理的第一个元素。First 含义与 Min 相近，但 First 是相对操作的第一个，而 Min 则是指数组本身的第一个元素
Last	在一个数组中需要最后进行处理的元素。Last 是 First 的反义词
Lim	在一个数组中需要处理的元素上界。与 Last 一样，Lim 也是 First 的反义词，但与 Last 不同的是, Lim 代表的是数组的不完全上界；而 Last 代表的则是最后一个元素。通常，Lim 等于 Last 加 1
Max	Max 指的是数组或表列中的绝对最后一个元素而不是相对操作的最后一个元素

限定词能够而且应该与基本类型和前缀联合使用。例如，PaReformat 指的是要重新格式化的段落；apaRefotmat 指的是要重新格式化的段落数组。而 ipaReformat 则指的是需要重新格式化的段落的数量。一个 for 循环来重新格式化段落的 C 程序如下:

```
for  (
     theReformat = paFirstReformat;
     ipaReformat <= paLastReformat;
     ipaReformat++;
     )…
```

可以用 PaLimReformat 来代替上例中的变量名 PaLastReformat 重写相同的循环，这时，由于 Lim 和 Last 的区别，在确定循环是否结束的检查中，将用“<”来代替“<=”。即用：

```
ipaReformat <   PaLimReformat
```

来代替

```
ipaReformat <= PaLastReformat
```

9.5.4　匈牙利名称举例

以下是利用匈牙利约定产生的变量名。其中变量名的基本类型部分采用的是前面文字处理程序示例中的基本类型。

变量名	意义
ch	字符变量（并不是 C 语言意义上字符，而是指文字处理程序中用来在文件中代替字符的数据结构）
achDelete	要删去的一个字符数组
ich	字符数组的下标
ichMin	字符数组中绝对第一个元素的下标

ichFirst	字符数组中第一个需要进行某种操作的元素的下标
echDelete	字符数组中产生的某一元素，如 ecbDelete = acbDelete [icbFirst]的结果
pachInsert	要插入字符数组中的指针
ppach	指向某一字符数组指针的指针
cchInser	要插入字符的数量
cscrMenu	用作菜单屏幕区的数量
hscrMenu	对用作菜单屏幕区的操作
mhserUserInput	用户输入屏幕区的模块层次上的操作（所有在这一模块中的子程序都可对这一变量进行存取操作）
ghscrMessages	为获得信息而对屏幕区进行的全局操作

注：上表中某些变量名不含限定词。虽然省略限定词是很常见的，但我们并不提倡这样做，应尽可能使用限定同。

9.5.5　匈牙利约定优点

匈牙利约定与其它命名约定一样，拥有由命名约定所带来的一切共同优点。由于有这样多的标准名称，因此在任何一个单个子程序或程序中要特殊记忆的名字是非常少的。匈牙利约定完全可以在不同项目中采用。

匈牙利约定可以使得在命名中容易产生定义的区域变得准确清楚。特别是约定中对 First，Min，Last，Max 和 Lim 的准确区分在实际中是尤其有帮助的。匈牙利约定可以使人对编译程序无法检查的抽象数据类型进行检查：cpaReformat[i]很可能是错误的，因为 cpaReformat 不是数组，而 apaReformat[i]则可能是正确的，因为 apaReformat[i]是数组。

匈牙利约定可以在类型不严格的语言或环境中对类型进行说明。例如，在 Windows 环境下编程时，需要你放弃许多类型，这极大地限制了编译程序进行严格类型检查的能力。而建立约定则可以对环境的这一弱点作出补偿，匈牙利约定还可以使名称更简洁，可以用 CMedals 而不用 TotalMedals 来代表奖牌的数量，使用 pNewScore，而不是用 NewScorePtr 命名一个新分数指针。

9.5.6　匈牙利约定缺点

一些版本的匈牙利约定事实上忽视了用抽象数据类型作为基本类型。它们以程序语言中整型、长整型、浮点数和字符串为基础来建立基本类型。匈牙利约定基本类型事实上是没有什么价值的，因为它使得程序员陷入对类型进行人工检查的困扰之中，而不是让编译程序对类型进行更加快速而又准确的检查。

这种形式匈牙利约定的另一个问题是它把数据的意义与其表现联系在一起。比如，说明某一变量是整型的，把它改为长整型的时，不得不改动这一变量的名称。

匈牙利约定的最后一个问题是它鼓励了懒惰、不含什么信息的变量名的出现。当程序员用 hwnd 来命名对窗口的操作时，往往忽视了他所指的到底是哪种窗口、对话框、菜单还是帮助区的屏幕？显然用 hwndmenu 要比 hwnd 清楚得多。以变量的意义为代价来获得对其类型的精确描述显然是愚蠢的。不过好在可以用加限定词的办法来同时获得完整的意义和精确的类型。

9.6　短名称

从某种程度上来说，使用短名称的意向是早期语言的遗留物。较老的语言如汇编、Basic 和 Fortran 语言把变量名长度限制在七到八个字母之间，从而迫使程序员们不得不使用短名称。在现代语言如 C、Pascal 和 Ada 中，事实上可以使用任意长度的名称，因此，此时已没有任何必要再使用短名称。

如果确实不得不使用短名称的话，要注意某些使用短名称的方法要好于其它的。可以通过去掉不必要的单词、使用短符号或者使用其它缩写技术来建立恰当的变量短名称。可以使用任意一种缩写技术。但最好多熟悉几种缩写技术因为没有任何一种方法是万能的。

9.6.1　缩写使用的总体准则

以下是使用缩写的几项准则，其中有某些准则是与其它准则相密的，因此不要试图一次使用其中所有的技术。

- 使用标准的缩写（常用缩写，如列在字典缩写表中的）。
- 去掉所有的非大写元音字母（如 Computer 写成 Cmptr，Screen 写成 Scrn，Integer 写成 Inter 等）。
- 使用每个单词的头一个或头几个字母。
- 截掉每个单词头一至三个字母后面的其余字母。
- 使用变量名中每一个有典型意义的单词，最多可用三个单词。
- 每个单词的第一个和最后一个字母。
- 去掉无用的后缀——ing，ed 等等。
- 保留每个音节中最易引起注意的发音。
- 反复交替地使用上述技术，直到变量名长度缩短至 8 到 20 个字母为止，或者到你所用语言规定的长度为止。

9.6.2　语音缩写

有些人喜欢根据单词的发音而不是拼写来进行缩写。如把 skating 写成 sk8ting，brightlight 写成 bilite，before 写成 b4 等等。这更像是在让人们破译密码，因此我不主张采用这种方法，但作为一种技术，你可以尝试一下“破译”出如下语音缩写的含义：

ILV2SK8　　XMEQWK　S2DTM8O　NXTCd　TRMN8R

9.6.3　关于缩写的建议

进行缩写时很容易陷入误区。以下是避免出现这种情况的几条准则：

不要通过拿掉单词中一个字母进行缩写。多敲一个字母费不了多少精力，而由此损失的可读性却往往是巨大的。如把 June 写成“Jun”或“July”、“Jul”等都是不值得的。而且如果总是使用这种只省略一个字母的缩写，很容易使人忘记你是否省略掉了一个字母。因此，或者多省略几个字母，或者使用全称。

缩写应保持一致性。应坚持使用同一缩写。例如，是使用 Num 或者 No，但不要两者混用。

同样，也不要时而缩写某一名称，又时而不缩写。比如，假设已经用了 Num 就不要再同时使用 Number 了。

使用容易发音的缩写。如应使用 xPos 而不是 Xpsn，用 CurTotal 而不是 ntTtl。可以用能否在电话中让对方明白的方法来检验缩写名称，如果不能的话，最好换一个比较容易说的缩写。

避免会引起错误发音的组合。如为表示 B 的结束，应使用 ENDB 而不要使用 BEND。如果使用了良好的分隔技术，则不必理会这条准则。如 B_END，BEnd 或 b_end 都不会导致错误发音。

近义词来避免命名冲突。在使用短名称常碰到的一个问题是命名冲突——对不同的名称使用了同一缩写。如 fired 和 fullrevenuedisbursal，假设对缩写的要求是限定在三个字母的话，那么很可能两者都会被缩写成 fri，从而产生冲突。

避免这个问题的方法之一是使用近义词。如可以用 dismissed 来代替 fired，用 complete revenuedisturb 来代替 fullrevenuedistural 便可以解决上面的问题。

用注解表来说明短名称。在只允许使用简短名称语言中，可以通过加入一个注解表来对变量名称的缩写加以说明，可以把注解表作为代码开始的注释块，以下是一个 Fortran 中使用注解表的例子：

```
C*************************************************************
C    Translation Table
C
C    Variable   Meaning
C    --------     -------
C    XPOS          X-Coordinate Position ( in meters )
C    YPOS          Y-Coordinate Position ( in meters )
C    NDSCMP            Needs Computing ( =0 if no computation is needed;
C                           =1 if computation is needed )
C    PTGTTL        Point Grand Total
C    PTVLMX            Point Value Maximum
C    PSCRMX            Possible Score Maximum
C*************************************************************
```

多从读程序者而不是写程序者的角度去考虑变量名称。你可以通过隔一段时间再阅读一下程序的方法来检查一下是否需要费很大精力才能弄清变量的含义。如果是这样的话，应改进命名技术来解决这一问题。

9.7　要避免的名称

以下是应该避免的几种名称：

避免容易产生误会的名称或缩写。要确认变量名称是清楚的。FALSE 通常是指 TRUE 的反义词，如果用它当“Fig and Almond Season”的缩写显然是不合适的。

避免含义相同或相近的名字。如果可以交换两个变量的名称而不致影响程序，那说明这两个变量都应重新命名。如 Input 和 InVal，RecNum 和 NumRecs 等每两个的意义都很相近，如果

在同一程序中同时使用它们来命名两个变量，就非常容易引入某些难以察觉的错误。

避免使用含义不同但拼写相似的名称。如果发现两个变量名称拼写相似但含义不同，那应对其中一个重新命名或改变缩写技术，比如 ClientsRecs 和 ClientsReps 这样的名称就应避免。因为它们之间只有一个字母不同，而且这一字母很难辨别，两个名称之间至少应有两个字母不同，或者把不同的字母放在词首或词尾。如用 ClientRecords 和 ClientsReports 来分别代替上述两个名称显然要好得多。

避免使用发音相同或相近的名称。如 Wrap 和 rap。因为这将使你在与同事讨论问题时遇到很多麻烦。

避免在名称中使用数字。如果变量名中的数字的确很有意义的话，应使用数组而不应使用单个变量。如果使用数组不合适的话，那么使用含有数字的变量名更不合适。比如，应避免使用 FILE1、FILE2 和 Total1、Total2 这类名字。可以用很多办法来区分两个变量，但不要采用在变量名末尾加数字的方法。我不敢说应绝对禁止在变量名中使用数字，但起码你应尽全力避免这种用法。

避免在名称中改写字母。记住单词的拼写是一件困难的事情，而记住改了字母的单词则更困难。例如，通过改写字母把 highlight 写成 hilite 以节省三个字母，将使得读者很难记住这个单词被改写成什么样了，是 Hilite，还是 Hai-a-lai-t？谁知道呢？

避免常见的容易拼写错的单词。Absense、acummulate、acsend、calender、conceive、defferred、definate、independance、occassionally、prefered、reciept、superseed 等是英语中经常容易拼写错误的，绝大多数英语词典中都列有常见的容易拼写错的单词。应避免在变量名中使用这些单词，以避免因拼写错造成程序中的错误。

不要单纯通过大写来区分变量名。如果使用的是可以区分大小的语言，可能会试图用 Frd 来代替 fired，用 FRD 来代替 final review duty，用 frd 来代替 full revenue disbursal，应放弃这种做法。尽管每个名字都是唯一的，但其中每个名称所代替的意义则是任意且容易混淆的。谁能知道 Frd，FRD 和 frd 分别对应的是 fired，final review duty 和 full revenue disbursal 而不是按其它顺序来对应的呢？

避免使用标准子程序名和已定义的变量名。所有的语言都要求保留其标准子程序名和已定义变量名，应注意避免使用这些子程序和变量。比如，下面这段代码在 PL/I 中是合法的，但若你真的这样的话，那你一定是一个十足的傻瓜：

```
if if = then then
    then = else;
else else = if;
```

不要使用与变量所代表的实体没有任何联系的名字。像 Margaret 或 Coolie 之类的变量名事实上保证了除你之外没有其它任何人能理解它的。不要用你的女朋友、妻子或朋友的名字作为变量名，除非这个程序是关于你的男朋友、妻子或朋友的。即使真的是这样的话，你也应该意识到他们是有可能变化的，因此用通用些的名字如：BoyFriend、wife 或 FavoriteBeer 会更好。

避免使用含有难以辨认字符的变量名称。要知道有些字符是非常相象的，很难把它们区分开来。如果两个变量名的唯一区分便是一个或两个这种字符，那么你区分这些变量时就会感到

十分困难。例如，请尝试一下把下表每一组中与其它两个变量名不同的一个找出来。

变量名表

EyeChart1	EyeChartI	EyeChart1
TTLCONFUSION	TTLC0NFUSION	TTLCONFUSION
Hard2Read	HardzRead	Hard2Read
GRANDTOTAL	GRANDTOTAL	6RANDTOTAL
Ttl5	TtlS	TtlS

如上表所示，难以区分的字符有"l"和"1"、"1"和"I"、"."和"，"、"0"和"o"；"S"和"5" 、"G"和"6"等。

9.8　小结

- 恰当的变量名是可读性好的必要条件之一。特殊的变量如循环变量和状态变量要予以特殊考虑。
- 命名约定可以区分局部、模块和全局变量。同时它还可以区分类型名称，比如可以对命名常量、枚举类型和变量加以区分。
- 不管你从事的是哪种项目，都应该采用命名约定。所采用的命名约定取决于程序的规模和从事这一程序的程序员的人数。
- 匈牙利约定是一种非常有效的命名约定，比较适于大规模项目和程序。
- 在现代编程语言中几乎不需要采用缩写技术。

9.8.1　检查表

通用命名约定

- 变量名称是否完全准确地描述了变量代表的是什么？
- 变量名是否指向是客观世界中的问题，而不是关于这问题的用程序语言表达解决方案?
- 变量名称是否是够长，使得你不必破译它？
- 变量名中如果含有计算限定词的话，是否将其放在最后？
- 是否在名称中用 Count 或 Index 来代替了 Num？

对特殊类型数据的命名

- 循环变量的名称是有意义的吗？（如果循环体较长是嵌套循环的话，应用有含义的名称来代替 i、j、k 之类的名称）
- 是否用更富于含义的名称来代替了被叫作"tempotarg"的临时变量？
- 当逻辑变量的值是"True"时，它的名称是否充分表达了其含义？
- 是否用前缀或后缀来表明了某些枚举类型是一类的？如用 Color 来作 ColorRed，ColorGreen，ColorBlue 等枚举类型的前缀。
- 命名常量的名称是否是指向它们代表的实体而不是它们所代表的数值的？

命名约定

- 命名约定是否区分了局部、模块和全局数据？

- 命名约定是否对类型名称、命名常量、枚举类型和变量进行了区分？
- 在不支持强化仅供子程序输入参数的语言中，命名约定是否对这类参数进行了标识？
- 命名约定是不是与程序语言的标准约定尽可能地相容？
- 对于语言中没有强制的子程序中仅做输入的参数，是否约定将它标识了？
- 是否对名称进行了格式化以增强程序的可读性？

短名称

- 代码是否使用了长名称？（除非有必要使用短名称）
- 是否避免了只省略一个字母的缩写？
- 所有单词保持缩写的连续性了吗？
- 所有的名称都是容易发音的吗？
- 是否避免了会引起错误发音的名称？
- 是否在注释表中对短变量名进行了注释？

避免如下这些常见的命名错误了吗

- 易引起误会的名称
- 含义相似的名称
- 仅有一或两个字母不同的名称
- 发音相似的名称
- 使用数字的名称
- 对单词作改写以使其比较短的名称
- 英语中常拼写错的名称
- 标准库子程序或已定义的变量名又定义了
- 完全是随意的名称
- 含有难以辨识字母的名称

第十章　变　量

目录

相关章节

生成数据：见第 8 章
数据命名：见第 9 章
使用基本数据类型：见第 11 章
使用复杂数据类型：见第 12 章
格式化数据说明：见 18.5 节
说明数据：见 19.5 节

由于前面一章叙述的都是数据名称问题，你可能会认为恰当地命名变量之后便完事大吉了。绝非如此！命名仅仅是开始，你使用变量的方法也是非常重要的。

如果你是个富有经验的程序员的话，那么本章的内容对你尤其有用。在你完全清楚替代方案之前，很容易在开始时使用有害的技术。然后，即使在你知道如何避免它们时，出于习惯仍会继续使用它们。有经验的程序员们将会发现 10.5 节“变量功能单一性”和 10.6 节“全局变量”的讨论对他们来说是非常有趣的。

10.1　作用域

作用域指的是变量名称的影响范围，也可称之为可见性，即在程序中某一变量被知道和提及的范围。作用域有限或很小的变量只在程序的一小部分中被知道，如：一个只有在一个小循环中用到的循环变量。作用域大的变量则在程序中的许多地方被知道，如：在一个程序中被到处使用的雇员信息表。

不同的语言处理作用域的方式是不同的。在 Basic 某些实现中，所有变量都是全局的。因此在这种情况下你无法对变量作用域进行任何控制，这也是 Basic 的主要缺点之一。在 C 中，变量可以是对块（用大括号括起来的部分）可见的，也可以是分别对子程序、源文件或整个程序可见的。在 Ada 中，变量可以分别是对块、亚程序、包、任务、单元或整个程序中可见的。

以下是一些关于作用域（可见性）的常用准则：

尽可能减小作用域。你所采取的方法往往取决于你“方便性”和“可管理性”这两个问题的看法。许多程序员喜欢用全局变量。因为全局变量存取很方便而且程序员们不必围着参数表和模块命名规则转。事实上，这种存取方便性是与由全局变量引入的危险共存的。

另一些程序员则尽可能使用局部变量，因为局部变量可以提高可管理性。你所隐含的信息越多，那么需要记在心中的东西就越少，而需要记住的东西越少，那么犯错误的机会也就越少，因为许多细节都不需要再进行记忆了。

“方便性”和“可管理性”之间的区别可以理解为是强调写程序还是强调读程序。扩大变量作用域事实上的确可以方便写程序，但是一个任意子程序都可以在任意时刻访问任一个变量的程序，往往要比使用模块化子程序的程序难懂得多。在这种程序中，你无法单纯理解一个子程序，你必须同时也理解与这个子程序分享全局变量的其它子程序。这样的程序不仅难读，而且也很难调试和修改。

因此，你必须尽可能地减小变量的作用域。如果能将变量的作用域限制在一个子程序之内的话，那是再好不过的了，如果你无法把它限制在一个模块中的话，那就利用存取子程序来使几个模块分享这一数据。总之，应尽量避免使用全局变量以减小作用域。

把对某一变量的引用集中放置。某些研究人员认为把对某一变量的访问放得越近，那么对程序阅读者的精神压力也就越小。这一想法有很大的直觉吸引力——你每次只需注意比较少的变量。以下是由这一想法产生的几项准则：

应恰好在某一循环前初始化循环中用到的变量，而不是在含有这个循环的子程序开头对其中用到的变量进行初始化。这样作可以使你在修改循环时，同时想起对相应的变量的初始化进行修改；或者在这一循环外再嵌套一个循环时，此时外部循环每执行一次时，都会对内部循环用到的变量进行初始化，而不会出现只初始化一次的错误。

要在用到某一变量时才对它进行赋值。你可能有过费尽心机地寻找某一变量到底是在哪被赋值的体验。因此，越清楚地表示出变量赋值的地方越好。

下例指出了在一个计算日收入的子程序中，是怎样把对同一变量的引用集中放置，以便方便地寻找它们的。第一个例子是违反这一原则的一个C语言子程序：

```
void SummarizeData（...）
  {
   ...
   ...
   GetOldData(OldData, &NumOldData);
   GetNewData(NewData, &NumNewData);
   TtlOldData = Sum(OldData， NumOldData);
   TtlNewData = Sum(NewData,NumNewData);
   PrintOldDataSummary(OldData,TtlOldData,NumO1dData);
   PirntNewDataSummary(NewData,TtlOldData,NumNewData);
   SaveOldDataSummary(TtlOldData,NumOldData);
   SaveNewDataSummary(TtlNewData,NumNewData);
   ...
   ...
  }
```

语句使用两组变量

在上例中，你不得不同时注意 OldData、NewData、NumOldData、NumNewData、TtlOldData

和 TtlNewData 六个变量，而且又是在这样短的一段程序中。下面的例子指出了如何把这一数量减少到只有三个：

```
void SummariseDaily（ ... ）
    {
     GetOldData(OldData, &NumOldData);
     TtlOldData = Sum(OldData, NumOldData);
     PrintOldDataSummary(OldData,TtlOldData,NumOldData);
     SaveOldDataSummary (TtlOldDataNumOldData);
     ...
     GetNewData( NewData, &NumNewData);
     TtlNewData = Sum( NewData, NumNewData);
     PrintNewDataSummary( NewData,TtlNewData,NumNewData);
     SaveNewDataSummary( TtlNewData, NumNewData );
     ...
    }
```

使用 OldData 的语句

使用 NewData 的语

如果像上例这样把程序分用两块，那么每一块都要比原来的块要短，而且其中的变量也要少得多。这两个块都很容易理解，而且如果需要把这段程序分成几个子程序的话，两个具有较少变量的块本身就是定义得很好的子程序。

10.2 持久性

“持久性”指的是某一数据的使用寿命。持久性有几种表现形式。如下所示：

- 与某一个块或子程序的生存期一样长。C中的auto变量或Pascal中的局部变量就属于这种情况。
- 其生存期取决于你的意愿。Pascal中用New()产生的变量直到dispose()它们时才失效。在C中用malloc()产生的变量也将持续到free()它们时才失效。
- 与程序的生存期一样长。绝大多数语言中的全局变量都属于这种情况。比如c中的static变量和 Turbo Pascal中“类型化的常量”（类型化常量是对Pascal的非标准推广）。
- 永远有效。这些变量可能包括你在程序再次执行之间存储在数据库中的变量。例如，在一个交互式的程序中用户可以定义屏幕的颜色，可以把这些颜色存在一个文件中，在每次程序加载时再将其调出，现在有少数几种语言支持这种持久性。

假定的某个变量的持久性要长于其实际持久性时，就会出现问题。变量就像放在冰箱中的牛奶，你认为它可以保存一星期，但有时可以保存一个月，有时则五天就坏了。变量的持久性也是同样不可测的。当变量的有效生存期已经结束时，还试图重新使用它，它还会保持原值吗？有些情况下变量中的值已经被改变了。这可以使你意识到自己的错误，而在另一些情况下，计算机还让变量保持原值，从而让你认为自己正确地使用了变量。

以下是可以使你避免这种错误的几个步骤；

- 在程序中加入调试代码来检查变量的值是否合理。如果不合理的话。产生一个警告

信息来提示检查不恰当的变量初始化。

- 在写代码时假定变量已经失效。比如，退出子程序时某一变量等于某个值，当再次进入子程序时不要假定这个变量仍保持原值。当然，当你用语言的特定功能来使变量保持原值时这一原则不适用，比如用 C 中的 static 如来实现这一功能。
- 养成在恰好使用某一变量之前对其进行初始化的习惯。如果发现使用的变量附近没有对它的初始化，那么你就要小心了。

10.3 赋值时间

一个对程序的维护性和可读性有深远影响的主题是“赋值时间”——把变量的值和变量联系在一起的时间。是在写程序时把它们联系在一起？还是在编译、加载或者程序运行时把它们联系在一起？

应该尽可能地晚一些将它们联系在一起。通常，越是晚一些给变量赋值，代码的灵活性便越大。下面是在可能的最早时间——程序写成时对变量进行赋值例子（用C写成）：

```
TestID＝47；
```

由于47是一个程序的常数值，因此在代码写好时47便与TestID联系在一起了。像上例那样编码赋值的想法是很有害的，因为如果当这里的47改变时，程序其余用到47且必须与TestID的值相同的地方很可能会出现语法错误。

以下是一个稍晚些赋值的例子，即在编译时进行赋值：

```
#define MAX_ID 47
...
TestID = MAX_ID；
```

MAX_ID是一个宏或是命名常量，当编译时编译程序会用一个值来代替它。如果所用的语言支持这种用法的话，应尽量这样用，因为这种方法要好于前面的用47来硬性赋值。它使得改变MAX_ID值变得很容易，因为你只需要在一个地方作出改动就可以了，而且不会影响程序性能。

下面是在最后时刻赋值的例子，即在运行时赋值。

```
TestID ＝ MaxID;
```

程序中MaxID是一个在程序运行时被赋值的变量。这样做的灵活性和可读性也要好于前面的硬性编译赋值。下面是另一个在运行时赋值的例子：

```
TestID ＝ ReadFileForMaxID()；
```

上例中的ReadFileForMaxID()是一个在程序运行时从一个文件中读取数值的子程序。这一例子假定在程序开始运行之前要用到的值已经被放在文件中了。显然这样作的灵活性和可读性也要好于前述的硬性编码赋值的例子。不必通过改动程序来改变TestID的值，而只要改动一下存储该值的文件就可以了。这种方法常用于用户可以定义应用环境的交互式系统中，用户的定义被存储在一个文件中，当程序运行时从文件中读取定义。

下面是在程序运行时进行赋值的最后一种形式：

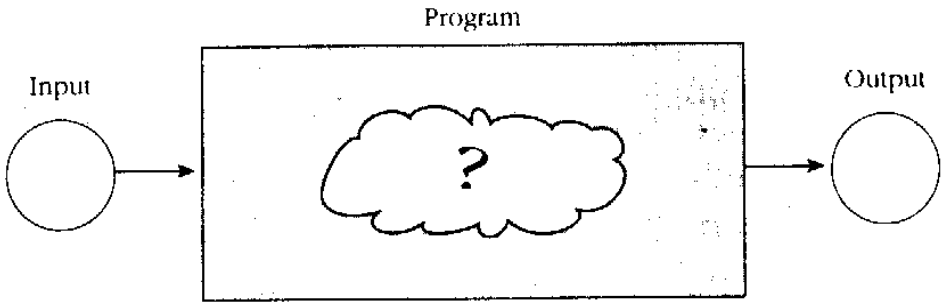
```
TestID = GetMaxIDFromUser();
```

上例中 GetMaxIDFromUser()是一个采用交互方式从用户那里读取数值的子程序。这种方

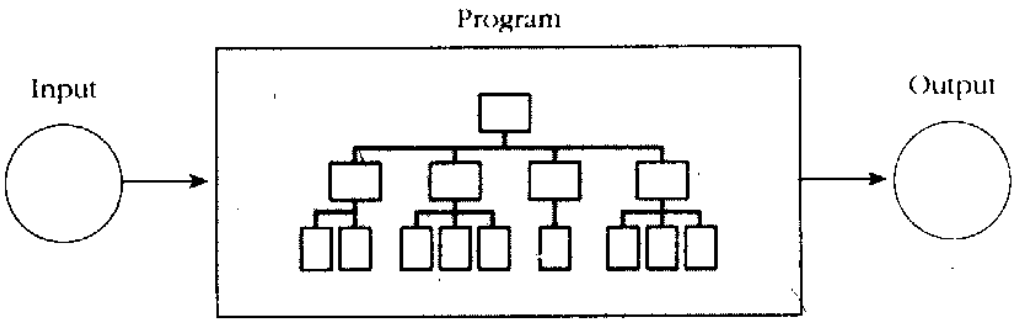
法的可读性和灵活性要远远好于硬性编码赋值。为改变TestID值根本不需作出任何改动，只要在程序运行时由用户输入另外一个值就可以了。从上述在运行时赋值的例子可以看出，即使同样是在运行时赋值的方式，变量与其值联系到一起的具体时间也是不同的。最后一个例子中的赋值可以在程序运行中任一时刻进行，它取决于用户被要求输入TestID值的时间。

10.4　数据结构与控制结构的关系

许多研究者都曾试图努力找出数据结构与控制结构之间的通用关系，这其中最成功的是英国计算机学家 Michael Jackson。Jackson的技术，主要是通过一种系统的方法把数据结构变换为控制结构。他的方法在欧洲得到了充分发展并被广泛应用。本书无法详细论述Jackson的理论，但可以对这种理论所基于的数据和控制流之间的调节关系作一概述。



从可用的数据和输出该是什么样子的想法开始

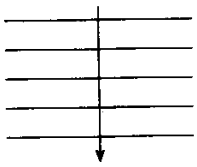


然后对程序进行定义，使其把输入转化为输出

Jackson勾画出了三种类型数据与相应的控制结构之间的关系。

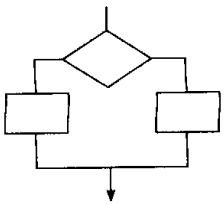
程序中顺序性数据可以转化为顺序性语句。数列是由一组按某一特定顺序使用的数据组成的。如果用排成一列的五条语句来处理五个不同的数值，那么它们就是顺序性语句，如果需要从某一文件只读取雇员的名字、社会保险号码、地址、电话号码和年龄等五个数据，那你将在程序中使用顺序性语句来从文件中读取这些顺序性数据。

程序中的选择性数据可以转换为if和case语句。通常，选择性数据指的是在任一特定时刻，几个数据中的某一个会出现——将选定其中的某一个数据。相应的程序必须用If-Then-Else语句或Case语句进行选择操作。比如在一个工资发放系统中，你可能需要按某一雇员是按小时计酬



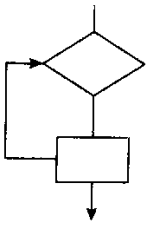
顺序性数据是指按某一指定顺序进行处理的数据

还是按固定薪水计酬的对其进行不同的处理。程序中的模式就与数据中的模式相容。



对选择性数据你可以使用两个中的任何一个，但不能同时

程序中的重复性数据可以转化为for、repeat和while循环结构。重复性数据指的是同一类型要重复几次的数据。通常，这类数据是作为记录存储在文件或数组中的，比如从文件中读取的一列社会保险号码。重复性数据将与读取它们的循环相容。



重复性数据的操作需要重复

真实的数据可能是上述几种类型数据的组合，这时可以用上述几种操作的组合处理复杂的数据结构。

10.5　变量功能单一性

可以通过几种“巧妙”的办法使变量具有一个以上的功能，但最好不要使用这种所谓的巧妙办法。

应使每一个变量只具有一个功能。有时人们会试图在两个不同的地方使用同一变量来进行两个不同的活动。通常，变量名对其中的一个活动来说是不恰当的，或者在两种情况下都充当了“临时变量”的角色（且是用无意义的X或Temp来命名的）。下面给出了一个用C语言写成的一个临时变量具有两个功用的例子：

```
/* Compute roots of a quadratic equation.
```

```
        This code assumes 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;
```

这段程序的问题是：头几行代码中的Temp与后几行代码中的Temp是什么关系呢？事实上它们之间没有任何关系，可是，这样使用它会使人们误以为它们之间存在着某种联系，从而产生困惑，以下是对上面程序的改进：

```
    /* Compute roots of a quadratic equation.
        This code assumes that (b^2 - 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;
```

避免使用具有隐含意义的变量。一个变量具有一个以上功用的另一种情况是同一变量取值不同时，其代表的意义也不同。如变量PageCount代表是已经打印的页数，但当它等于-1时则表示发生了错误；变量CustomerID代表的是顾客号码，但当它的值超过500,000时，你要把CustomerId的值减掉500,000以得到一个过期未付款的帐号；而BytesWritten一般表示的是某一输出文件的字节数，但当其值为负时，则表示的是用于输出磁盘驱动器数目。

应避免使用上述具有隐含意义的变量。技术上把这种滥用称之为“混合关联”。使用同一变量来承担两项任务意味着对其中一项任务来说变量类型是错误的。比如上例中的PageCount，在正常情况下代表页数时是整型的，而当它等于-1表示发生了错误时，则是把整型变量当作逻辑型来用了。

即使你清楚这种用法，别人也往往很难理解。如果用两个变量来分别承担两项工作的话，其意义的清晰性将会使你感到满意，并且，也不会在存储上有额外的麻烦。

保证所有说明的变量。与用一个变量来承担两项工作相反的另一个极端是根本不用它，而研究表明未被使用的变量往往是与出错率高相联系的。因此，要养成确实用到每一个说明变量的习惯。某些编译程序会对说明但未用到的变量给出警告。

10.6 全局变量

全局变量在程序的任何地方都可以进行存取。有时它也被非正式地用来指可存取范围大于局部变量的变量，如在某个单一文件中可以随处存取的模块变量。但是，在单独一个文件中随处可存取，本身并不能表示某一变量是全局的。

绝大多数有经验的程序员都认定使用全局变量要比局部变量危险，同时他们也认为利用几个子程序来存取数据是非常有益的。尽管对使用全局数据的危险性有许多附和的意见，但研究发现全局变量的使用与错误率上升之间并无联系。

即使使用全局变量是没有危险的，使用它也决非最好的编程方法。本书其余部分将讨论由此而引入的问题。

10.6.1 伴随全局变量的常见问题

如果不加选择地使用全局变量，或者不使用它们就感到很不方便，那么你很可能还没有充分意识到信息隐蔽和模块化的好处。模块化和信息隐蔽可能并不是最终真理，但它们对程序的可读性和维护性的贡献是令其它技术望尘莫及的，一旦你懂得了这一点，你就会使用与全局变量关系尽可能少的子程序和模块。

对全局数据的疏忽改变。你可以会在某处改变全局变量的值，而在别处又会错误地以为它仍保持着原值，这就是所谓的副作用，例如在下面的Pascal程序段中，TheAnswer就是个全局变量：

```
TheAnswer := GetTheAnswer;        ——TheAnswer是一个全局变量
OtherAnswer := GetOldAnswer;      ——GetOtherAnswer改变了TheAnswer
AverageAnswer := (TheAnswer + OtherAnswer )/2;   ——AverageAnswer是错误的
```

你可能以为对GetOtherAnswer的调用并没有改变TheAnswer的值，如果真的是这样的话，那么第三行中的平均就是错误的。事实上，对GetOtherAnswer的调用，的确改变了TheAnswer的值，因此程序中有一个需要改正的错误。

伴随全局变量的奇怪的别名问题。“别名”指的是用两个或更多的名称来叫某一变量。当全局变量被传入子程序，然后又被子程序既用作全局变量又用作参数时，就会产生这种问题。以下是一个用到全局变量的Pascal子程序：

```
Procedure WriteGlobal (VAR InputVar:Integer);
begin
   Globa1Var := InputVar + 1;
   Writeln( 'InputVariable:', InputVar);
   writeln( 'GlobalVariable:', Globe1Var);
end;
```

下面是一个把全局变量当作变元来调用上面子程序的程序：

```
   WriteGlobel（GlobalVar）;
```

由于WriteGlobal把InputVar加1后得到了GlobalVar，你会以为GlobalVar要比InputVar大1，但下面

却是足以令你大吃一惊的运行结果：

Input Variable： 2
G1obal Variable： 2

这里令人难以置信的是：事实上GlobalVar和InputVar是同一个变量，由于GlobalVar是通过调用子程序被传入WriteGlobal()的，因此它被用两个不同的名字提及了，或者说它是被“别名”了，从而Writeln()的结果与你想要的便是大相径庭的了。虽然你用两个不同的名字来区分全局变量，但它们还是将同一变量打印了两次。

有全局数据的代码重入问题。通过不止一个控制进入代码，已经变得越来越普遍了。在Microsoft Windows、Apple Macintosh和OS/2 Presentation Manager及递归子程序中都用到了这种代码。代码重入使得全局变量不仅可能被子程序们分享，而且可能被同一程序的不同拷贝所分享。在这种环境下，你必须保证即使在某一程序的多个拷贝都在运行的情况下，全局数据仍保持着它原来的意义。这是一个很有意义的问题，你可以根据后面推荐的技术来避免这一问题。

全局数据妨碍重新使用的代码。为了从另一个程序中借用某段代码，首先你要从这个程序中把这段代码取出来，然后把它插入要借用它的程序中。理想的情况是你只需把要用的模块或单个子程序拿出来放入另一个程序中就可以了。

但全局变量的引入则使这一过程变得复杂化了。如果你要借用的模块或子程序使用了全局变量，你就不能简单地把它拿出来再放入另一个新程序了。这时你必须对新程序或旧的代码进行修改以使得它们是相容的。如果想走捷径的话，那最好对旧代码进行修改，使其不再使用全局数据。这样作的好处是下次再要借用这段代码时就非常方便了。如果不这样作的话，那你就需要改动新程序，在其中建立旧有的模块或子程序要用到的全局数据。这时全局变量就像病毒一样，不仅感染了旧程序，而且随着被借用的旧程序中的模块或子程序传播到了新程序中。

全局变量会损害模块性和可管理性。开发大于几百行规模软件的一个主要问题便是管理的复杂性，使其成这可管理的唯一办法便是将程序成为几个部分，以便每次只考虑其中的一个部分。模块化便是将程序分为几部分的最有力工具之一。

但是全局数据却降低了你进行模块化的能力。如果使用了全局数据，不能做到每次只集中精力考虑一个子程序吗?当然不能,这时你将不得不同时考虑与它使用了相同全局数据的其余所有子程序，尽管全局数据并没有摧毁程序的模块性，使它减弱了模块性，仅凭这一点就该避免使用它。

10.6.2 使用全局数据的理由

在某些情况下全局数据也是很有用的：

保存全局数值，有时候需要在整个程序中都要用到某些数据。这些数据可能是反映程序状态的——在交互式状态时还是批处理状态?是正常状态还是错误修正状态？它们也可能是在整个程序中都要用到的信息，如程序中每一个子程序都要用到的一个数据表。

代替命名常量。尽管C、Pascal等绝大多数现代语言都支持命名常量,但仍有一些语言不支持，这时,可以用全局变量来代替命名常量。例如，你可以通过分别给TRUE和FALSE赋值”1”和”0”采用它们代替常量型值1和0,或者通过LinesPerPage = 66这一语句把每页行数（66行）赋给LinesPerPage，从而用LinesPerPage来取代66。通过使用这种方法,可以改善代码的可读性和

易改动性。

方便常用数据的使用。有时候需要非常频繁地使用某一个变量，以至于它几乎出现在每一个子程序的参数表中，与其在每个子程序的参数表中都将这个变量写一次，倒不如使它成为全局变量方便。在这种情况下，这一变量几乎是到处都被存取的，不过，很少有这种情况，更多的情况是它是被一组有限的子程序存取的，这时你可以将这些子程序及数据装入一个模块，这将在稍后详细讨论。

消除“穿梭”数据。有时候把某个数据传入一个子程序中仅仅是为了使它可以把这一数据传入另一个子程序中，当这个传递链中的某个子程序并没有用到这个数据时，这个数据就被叫做“穿梭数据”。使用全局变量可以消除这一现象。

10.6.3　怎样降低使用全局数据的危险

你可能会认为下面这些准则让人感到很不自由也很不舒服，或许的确是这样，但我想你一定知道“良药苦口利于病”这句话吧？

先使所有变量都成为局部的，然后再根据需要把其中某一些改为全局变量。首先使所有变量针对单个子程序来说都是局部的。如果发现还需要在别的地方使用它，那么在使它成为全局变量之前应先使它成为模块变量。如果最后发现必须使它成为全局变量，你可以这样作，但必须确认这是迫不得已的。如果开始就使某一变量成为全局的，那么你决不会再把它变成局部的，而很可能如果开始把它用成局部的话，你永远也不会再把它变为全局的。

区分全局和模块变量。如果某些变量是在整个程序中存取的，那么它们就是真正的全局变量。而某些变量只在一组子程序中存取，事实上是模块变量。在指定的那组子程序中，对模块进行任何存取操作都是可以的，如果其它子程序要使用它的话，应通过存取子程序来进行。即使是程序语言允许，也不要像对待全局变量那样对模块变量进行存取操作。要在自己的耳边不停地说“模块化！模块化！模块化!”。

建立使你一眼即可识别出全局变量的命名约定。如果使用全局变量的操作是十分明显的，可以避免许多错误。如果不只是出于一个目的使用全局变量（如既作为变量又用替换命名常量），那一定要保证命名约定可以区分这些不同目的。

建立一个清楚标出所有全局变量的注释表。建立标识全局变量的命名约定，对表明变量的功用是非常有帮助的，而一个标有所有全局变量的注释表则是读你程序的人最强有力的辅助具之一（Glass 1982）。

如果你用的是 Fortran 语言，那么仅使用标号 COMMON 语句，不要使用空白 COMMON。空白 COMMON 可以使任意一个子程序存取任意一个变量。使用命名的 COMMON 来详细规定可以存取特定 COMMON 数据的特定子程序，这是一种在 Fortran 中模拟使用模块数据的方法。研究表明这种方法是很有效的。

使用加锁技术来控制对全局变量的存取。与多用户数据库环境下，当前值的控制方式类似，在全局变量被使用或更新之前锁定要求，这个变量必须被“登记借出”，在变量被使用过之后，再被“登记归还”，在它不能被使用期间（已经被登记借出），如果程序其它部分企图要求将它登记借出，那么加锁／开锁子程序将打印出错误信息。

加锁技术在开发阶段是有用的。当程序最终成为产品时，程序应该被改动来做比打印更有意义的工作，使用存取子程序来存取全局变量的好处之一，就是使你可以方便地实现加锁／开

锁技术，而如果不加限制地存取全局变量的话，就很难实现这一技术。

不要通过把数据放人庞大的变量，同时又到处传递它来掩盖你使用了全局变量的事实。把什么都放入一个巨大的结构，可能从字面上满足了避免使用全局变量这准则，但这只是表面文章，事实上这样做，得不到任何真正模块化的好处，如果你使用了全局变量的话，那就分开使用它，不要企图用臃肿的数据结构来掩盖它。

10.6.4 用存取子程序来代替全局数据

用全局数据能作的一切，都可以通过使用存取子程序来做得更好，存取子程序是建立在抽象数据类型和信息隐蔽的基础上的。即使不愿使用纯粹的抽象数据类型，仍然可以通过使用存取子程序来对数据进行集中控制并减少因改动对程序的影响。

存取子程序的优点

以下是使用存取子程序的优点：

- 可以对数据进行集中控制。如果你以后又找到了更合适的数据结构，那么不必在每一处涉及到数据的地方都进行修改，而只修改存取子程序就可以了，修改的影响可以被限制在存取子程序内部。
- 可以把所有对数据的引用分隔开来，从而防止因其错误造成的影响蔓延。使用类似 Stack.array[stack.top]new_element 的语句来把元素压入堆栈时，很容易忘记检查堆栈是否溢出从而造成错误。如果使用存取子程序，例如 push stack(new_element)，就可以把检查堆栈是否溢出的代码写入 push_stack()子程序，这样每次调用子程序都可以对堆栈自动进行检查，而你则可以不必再考虑堆栈溢出问题。
- 你可以自动获得信息隐蔽带来的好处。即使你不是为了信息隐蔽才使用存取子程序的，它也是信息隐蔽的范例性技术之一。你可以改变存取子程序的内部而不必改动程序的其余部分。打个比方，存取子程序使你可以改变房屋的内部陈设而不会变动房屋的外观，这样你仍然可以很容易便找着你的家。
- 存取子程序很容易转换为抽象数据类型。存取子程序的一个优点是：它可以得到很高的抽象级，而直接存取全局变量却难以做到。例如，你可以用存取子程序 ifPageFull（）来代替语句 if lineCount > Maxlines，虽然这是个很小的收益，但是大量的这类差别便积聚成了高质量软件与东拼西凑到一起的软件之间的不同之处。

怎样使用存取子程序

以下是关于存取子程序理论与应用简短论述：将数据隐含在模块中，编写可以使你访问并修改数据的子程序，数据所在模块之外的子程序要求存取数据时，应让它通过存取子程序而不直接地存取模块内的数据。比如，假设有一个状态变量你可以通过两个存取子程序 **Getstatus()** 和 **SetStatus()**来对其进行存取操作。

以下是关于存取子程序使用的几条较为详细的准则：

要求所有子程序来对数据进行存取操作。通过存取子程序将数据结构隐含起来。通常需要两个子程序，一个读取数据的值，而另一个用于赋给它新值。除去几个可以直接存取数据的服务性子程序，其它子程序都应通过存取子程序接口来对数据进行存取。

不要把所有的全局数据都放入同一个模块中。如果你把所有的全局数据都归成一个大堆，并编写对其存取的子程序，的确可以消除由全局数据带来的问题，但同时也抛掉了信息隐蔽和抽象数据类型所带来的优点。编写存取子程序之前，应先考虑一下每一全局数据应属于哪一个模块，然后把这个全局数据、相应的存取子程序和其关联的子程序放入那个模块中。

在存取子程序中建立某种程度的抽象。在数据所代表的意义层次上而不是计算机本身的实现细节层次上建立存取子程序，可以使你更容易应付可能的变动。

请比较下列两组语句：

直接使用复杂数据	通过存取子程序使用复杂数据
node=node. next	node=nearestNeighbor(node)
node=node.next	node=nextEmployee(node)
node=node.next	node=nextRatele(node)
Event=EventQueue[QueueFront]	Event=HighestprioirtyEvent( )
Event=EventQueue[QueueFront]	Event=LowesPriotityEvent( )

表中前三个语句对中，抽象的存取子程序告诉你的信息要比数据结构所告诉你的多得多。如果直接使用数据结构，那么一次需要做工作就太多了。你必须在表示出数据结构本身正在做什么（移到链表中的下一个链）的同时，表示出正在对数据结构所代表的实体做什么（调用一个邻居、下一个雇员或税率），这对于简单数据结构来说是很重的负担。把信息隐蔽在存取子程序后面，可以使代码自己指出这些，并且可以使得其它人从问题域而不是实现细节的层次上来阅读程序。

把对数据的所有存取保持在同一抽象水平上。如果你用了某一存取子程序对某一数据进行了一项操作，那么对这一数据的其它操作也应通过存取子程序来实现。比如若是通过存取子程序来从数据结构中读取数值的，那么对该数据结构的写操作也应通过存取子程序来实现。又比如假设你通过调用 initstack()子程序将元素压入堆栈，但你接着又用 value=array（strack.top）来获得堆栈的一个入口，那么此时对数据的观察点便不连续了。这种不连续性使别人很难理解你的程序。因此，要保持对数据所有存取操作抽象水平的一致性。

在上表中的后两个语句对中，两个事件排队的操作是平行进行的。在队列中插入一个事件将比表中其它操作都更复杂，因为你不得不改变队列的前后顺序，调整现存事件以便为它腾出空间，再写几行代码以便找到插入它的地方等等，从一个序列中移出一个事件几乎是同样麻烦的。因此，在编码时如果把复杂操作放入子程序，而其余操作直接对数据进行，将产生对数据结构拙劣的、非平等的使用。请比较下面的两组语句：

对复杂数据的非平行使用对复杂数据的平行使用

对复杂数据的非平行使用	对复杂数据的平行使用
Event=EventQueue[QueueFront]	Event=HightestPriorityEvent( )
Event=EventQueue[QueueBack]	Event=LowestPriorityEvent( )
AddEvent(Event)	AddEvent(Event)
EventCount=EventCount-1	RemoveEvent(Event)

应注意这些准则适用许多模块和子程序构成的大型程序。在小一些的程序中，存取子程序

的地位也会相应降低。但不管怎样，实践已经证明，存取子程序是增强程序灵活性并避免由全局变量带来问题的有效手段之一。

10.6.5　检查表

使用数据时通常要考虑的一些问题

一般数据

- 是否已经使变量的作用域尽可能地小？
- 是否把对变量的使用集中到了一起？
- 控制结构与数据结构是相对应的吗？
- 每个变量是否有且仅有一个功能？
- 每个变量的含义都是明确的吗？是否保证了每个变量都没有隐含的意义？
- 每一个说明过的变量都被用到了吗？

全局变量

- 是否是在迫不得已的情况下，才使某些变量成为全局的？
- 命名约定是否对局部、模块和全局变量进行了区分？
- 是否说明了所有全局变量？
- 程序中是否不含有伪全局变量——传往各个子程序的庞大而臃肿的数据结构？
- 是否用存取子程序来代替了全局数据？
- 是把存取子程序和数据组织成模块而不是随意归成一堆的吗？
- 存取子程序的抽象层次是否超过了程序语言实现细节？
- 所有相互有联系的存取子程序，其抽象程度都是一致的吗？

10.7　小结

- 尽量减小变量的作用域。把对变量引用集中到一起，应尽量使变量成为局部或模块的，避免使用全局变量。
- 使每个变量有且仅有一个功能。
- 并不是因为全局数据危险才避免使用它们，之所以避免用它是因为可以用更好的技术来代替它。
- 如果全局数据确实不可避免的话，应通过存取子程序来对其进行存取操作。存取子程序不仅具备全局变量和全部功能，而且可以提供更多的功能。

第十一章　基本数据类型

目录

相关章节

生成数据：见第8章
数据命名：见第9章
使用变量时通常要考虑的问题：见第10章
格式化数据说明：见18.5节
说明数据：见19.5节

基本数据类型是其它各种数据类型的基本构成部分。本章叙述了使用整型数、浮点数、字符数据、逻辑变量、枚举类型、常量、数组和指针等的一些窍门。数据结构，即含有一种以上简单类型的数据类型组合，将在下章讨论。

本章同时也涉及了与基本数据类型有关的一些问题的解决方法。如果你已经有了这方面的知识，可以直接跳到本章末尾的检查表中，看一下需要避免的问题，然后直接进行下一章。

11.1　常　数

以下是一些可以减少数据使用错误的措施：

避免"奇异数"(magic numbers)。“奇异数”指的是出现在程序中间的不加解释的常数。如100或47524。如果你所用的语言支持命名常量的话，那就用命名常量来代替它。如果无法使用命名常量的话，应考虑使用全局变量。

避免”奇异数”可以带来三个好处：

- 可以更可靠地进行修改。如果使用命名常量的话,就不会漏掉要修改的几个“100”中的一个，或者错误地修改一个本不该改动的“100”。
- 可以使修改更容易。假设要把人口的最大值从 100 变为 200，如果你用的是奇异数的话，

你将不得不找出所有的“100”，并把它们改为“200”。如果在程序中你使用了100+1或100-1 的话，那同时你还要找出所有的101或99并把它们改为201或199。但若你使用的是命名常量的话，则只需在定义命名常量的地方把100改为200就可以了。

- 可以增强代码可读性。确实，对语句

  for　i=1　to　100

  你可以猜测 100指的是入口的最大数，但是对语句

  for　i=1　to　MaxEntries

  你可以肯定地知道 to 后面的是入口最大数，而无须猜测，即使你可以绝对保证不改变某个数，那么使用命名常量来代替它也可以提高可读性。

在需要时可以使用常数“0”或“1”。″0″或″1″往往被用来增加或减少循环变量的值，也用于数组的第一个元素下标或循环变量的初始值。如:

```
for　i=0　to　CONSTANT
```

其中的″0″和

```
Total=Total+1
```

中的″1″都是允许的。在程序中是允许″0″或″1″作为常数出现的，但其它数值则不允许以这种方式出现。

采取预防被″0″除的措施。每当你在程序中使用了除号(大多数语言中都用″/″表示)时，都要考虑除数是否有可能是零。如果存在这种可能性的话，应加入防止被“0”除的代码。

明显进行类型转换。当程序中有不同类型间的转换时，要确保这种转换是明显的。在 Pascal 中，你可以用:

```
x := a + float(i)
```

在 C 中，可以用:

```
x = a +(float)i
```

这样作也可以保证类型转换确实是按照你的意愿进行的。不同编译程序的类型转换是不同的，如果不这样作的话就有出错的危险。

避免混合类型比较。如果 x 是浮点数而 i 是一个整型数，那么以下检验:

```
if (i=x) then……
```

几乎可以认定是不会起作用的。在编译程序确定出比较要用的类型，把一种类型转换为另一种、进行一连串的访问并最终确定答案之后，你的程序要是仍在运行的话，那真可说是“瞎猫遇上死耗子了”。应该通过人工转换来使编译程序可以对同一类型的数进行转换，以便你可以确切知道正在被比较的是什么。

注意编译程序的警告。许多先进的编译程序都会对在同一表达中使用不同数值类型进行警告。应注意这些警告！几乎每一个程序员都有过帮助别人去寻找某个麻烦的错误，而最终却往往发现编译程序早已对此提出过警告的经历。优秀的程序员总是力争消除所有的编译程序警告信息。让编译程序去查错毕竟比自己干容易得多。

11.2　整型数

以下是一些在使用整型数时应该牢记的准则:

检查整型相除。当你使用的是整型数时，7/10并不等于0.7，它等于0,这也同样适于中间结果。在客观世界中10*(7/10)=7，而在整型算法中10*(7/10)却等于0，因为(7/10)等于0，解决这个问题最简单的办法是调整计算顺序，如上例的表达式可以改写为：(10*7)/10，使得除法运算在最后进行。

检查整型是否溢出。在进行整型加法或乘法运算时，应明确可能的最大整型数。通常不带符号的最大整型数是65535，或说是2^{16}-1。当两个整型数相加的结果超过可以的最大整型数时就会出现问题。比如，250*300，正确的答案是75000。但由于整型溢出,你得到的答案很可能是9464，(75000-65536=9464)。下面是一些常见类型的整型数的范围:

整型数类型	范围
有符号8位	-128到127
无符号8位	0到255
有符号16位	-32768到32767
无符号16位	0到65535
有符号32位	-2，147，483，648到2，147，483，647
无符号32位	0到4，294，967，295

防止整型溢出的最简单办法是先笔算一下表达式中每一项的值是否溢出。例如，在整型表达式 M=J*K 中，J 的最大可能值是 200，而 K 的最大值是 25，从而 M 的最大值为 5000，小于 65535，因而这一运算是可行的，但若 K 的最大值为 2000 的话，那么此时 M 的最大值便是 200 000，大于65535，因而运算是不可行的。这时，你将不得不采用长整型或者浮点数来容纳 M 的最大值。

同时还要考虑到将来的程序扩展，如果 M 的值永远不超过 5000 的话是最好的，但如果 M 的值在几年之内都将是逐步增长的，那应把这点考虑在内。

检查中间结果是否溢出。公式的最后结果并不是你要考虑的唯一的数。比如用 Pascal 写出下面的一段代码:

```
var
   TermA:      integer;
   TermB:      integer;
   Product:    integer;
begin
   TermA :=1000;
   TermB :=1000;
   Product :=(TermA*TremB)div   1000;
   writeln('(', TermA, '*', TermB, ')div 1000=', Product);
```

如果你认为 Product 的值与(1000*1000)/1000 相同，你可能会认为它的值是 1000，但是在 1000*1000 的结果最终被 1000 除之前,必须计算出 1000*1000 的值,而此时的结果为 1,000,000，显然已经溢出。你能猜到最后的运行结果是什么吗？下面是结果:

```
(1000*1000)div 1000=16
```

如果你所用的机器的整型上限是32767，那么1,000,000这一中间结果显然是太大了，因而其实际结果为 16,960，它再被 1000 除，最后结果就是 16 了。

可以用处理整型溢出相同的方法来处理中间结果整型溢出，即使用长整型或者将其转换为浮点类型。

11.3　浮点数

使用浮点数时要考虑的主要问题是许多十进制的分数不能用浮点数精确地表示出来，像1/3或1/7这样的无限小数，其浮点数形式通常只有7到15位有效数字。在许多的 Pascal 版本中，一个占有4个字节的1/3的浮点数表示形式为0.33333334267440796，它精确到7位。在一般情况下，它是足够精确的。但是，在某些情况下，它的不精确性也足以令人迷惑。

以下是使用浮点数时需要特殊考虑的一些问题：

不要在数量级相差太大的数之间进行加减运算。假设变量都是 4 个字节的浮点变量，那么 1,000,000.00+0.01 的结果很可能仍然是 1,000,000.00 而不是 1,000,000.01，因为这里的浮点变量只有 4 个字节，因而在结果中 0.01 事实上是无效数字。同样，5,000,000.02-5,000,000.01的结果也很可能是0.0而不是0.1。

怎么办呢？如果你不得不对像上例那样相差巨大的浮点数进行加减运算的话，可以先检查一下所要运算的数，然后从最小的数开始运算。同样，如果你需要对无穷数列进行求和运算，应从最小一项开始进行，事实上是从末尾向开头进行运算，这并不一定能消除上述问题，但可以使由此带来的危害最小。许多算法书中都有关于这方面的论述。

避免相等比较。应该相等的浮点数事实上往往是不相等的。主要问题是：算法不同但结果应该相同的浮点数运算的结果事实上是不同的。例如，把0.1 累加10次的结果很少是1.0。下面的例子便表示出了两个应该相等的变量。Sum 和 Nominal，事实上是不相等的：

```
var
   Norminal:   single; ——变量 Nominal 是4字节实数
   Sum:        single;
   i:          integer;
begin
   Norminal:=1.0;
Sum:=0;
for   i:=1   to   10   do
      Sum:=Sum+0.1;   ——Sum 进行10*0.1计算，结果为1.0
    if(Nominal=Sum) then  ——这是错误的比较
       writeln('Numbers are the same.')
    else
       writeln(' Numbers are different.')
end;
```

正如你所预料的那样，这个程序的输出结果是：

Numbers are different.

因此，最好用另外一种方法来代替相等比较。一种方法是确定一个可以接受的精度范围，然后用逻辑函数来确定两个数是否接近。比如，你可以编写一个Equal()函数，当两个值足够接近时Equal( )返回的值为True，否则其返回的值为False，在Pascal 中，函数是这样的：

```
const
        AcceptableDelta = 0.00001;

function Equals(Terml:single; Term2:single): boolean ;
begin
        if(abs(Terml-Term2)<AccptableDelta)    then
              Equals :=True;
        else
              Equals :=False;
end;
```

如果用这个子程序来代替前述提到的不恰当相等比较，新的比较将是这样的:

```
if ( Equals ( Nominal，Sum ) ) then ……
```

这个例子的运行结果是这样的:

```
Numbers are the same
```

用常数来给 AcceptableDelta 硬性赋值可能并不适合你的要求，你可能需要根据比较的两个数大小来计算 AcceptableDelta 的值。

防止舍入误差。舍入误差问题产生的原因与数量级相差过大的数之间加减运算产生问题的原因是一致的，因而，两者的解决方法也是类似的。以下是用于解决舍入误差问题的一些方法:

首先，将变量转换为精度更高的变量类型。如果你使用的是单精度实数，那么就把它转换为双精度的。

第二，将变量转换为二——十进制（BCD）变量。这种方法较慢而且会占用更多的空间，但却可以完全消除这一问题。特别是变量代表美元和美分或者其它需要精确平衡的变量时，这一技术尤为有用。

第三，将变量从浮点型转化为整型的。这时往往需要用长整型以得到所要求的精度。这种技术要求你自己知道数值的分数部分。比如你知道某些款项是用浮点数来表示美元，并用分数来作为其美分部分的，这是一种表示美元和美分的很常用的方法，当需要将其转化为整型时，你必须知道美分是用整型表示的并且用 100 美分来表示 1 美元。换句话说,你用 100 来乘以款项的美元部分并将美分部分的整型数保持在 0～99 美分之间，这种表示方法初看起来可能会令人感到别扭，但无论从速度还是精度角度来看，这都是一种很有效的方法。

可以通过一组转换手程序来使这一过程简单些，这些子程序包括:（1）从同时含有美元和美分的数据中得到美元的子程序；(2)从中得到美分的子程序；(3)把美元变量和美分变量合成为美元——美分变量的子程序。这些子程序同时也为你提供了检查是否有整型溢出的机会。

11.4　字符和字符串

以下是使用字符串时应注意的几个问题，其中第一条适于所有的语言:

避免“奇异字符和字符串”。奇异字符串则是指常量字符串（如“Gigamatic Accounting Program”）。如果你所用的语言支持命名变量的话,应用命名变量来代替它，否则可以用全局变
量来代替它。避免使用常量字符串的原因如下:

- 对常用的字符串如程序的名称、命令名称、报告题头等等往往要经常改变串内容。例如,把“Gigamatic Accounting Program”改为“New and Improved!Gigamutic Accounting Program”等等。
- 国际市场正变得日益重要。把存储在资源文件中的字符串翻译成外文要比在程序中到处寻找并翻译这些字符串容易得多。
- 常量字符串往往会占用很多空间。如果你在菜单、帮助屏幕、入口形式、提示信息等处都使用它们，它们就会因太多而失去控制，从而可能产生内存问题。而当字符串相对源代码是独立的时，这一问题就比较容易解决。
- 常量字符率和字符往往是神秘的，而命名常量则可以清楚地表明你的意图。在下面的C语言例子中,“\027”到底是什么？恐怕不会有谁知道。而使用“ESCAPE”来代替它意义就清楚多了:

  if(input_char==‘\027’)…　　——这种表示不好!

  if(inpnt_char==ESCAPE)…　　——这种表示好!

警惕边界错误。由于子串可以像数组差不多一样可以加下标，在读写超过串结束时，应注意边界错误。

11.4.1　C 语言字将串

在绝大多数语言中，使用字符和字符串数据都不容易出什么问题，但是不幸的是，在 C 中却很容易出问题，因此，其余部分重点讨论 C 语言中字符和字符串。

要清楚字符串指针和字符数组之间的区别。字符串指针和字符数组的问题是由 C 处理字符串的方式引起的。应该注意两者之间在两个方面的不同:

- 要注意任何含有字符串且又带有引号的表达式。在 C 中，对字符串的操作几乎都是由strcmp()、strcpy(),strlen()及其它相关联的子程序完成的。等号的出现往往意味着有指
  针错误。在 C 中赋给一个字符串变量。如下面的这个语句:

  StringPtr ="Some Text String"

在这种情况下，“Some Text Siring”。是一个指向字符串的指针，这个赋值语句只是使得指针StringPtr指向了字符串而并没有把它的内容赋给StringPtr。

- 使用命名约定来表示变量究竟是字符数组还是字符串指针。在这里第九章中描述过的匈牙利命名约定便大有用武之地了，使用 psz 作为字符串指针名称的前缀，而使用 ach 作为字符数组的前缀。虽然这样并不能消除所有的错误，但是可以提醒你在使用含有以”ach”或”psz”为前缀的变量表达式时要提高警惕。

将字符串的长度说明为 CONSTANT+1。在 C 中，由于字符串而产生的边界错误是非常常见的。因为很容易忘记一个长度为 n 的字符需要 n+1个字节的空间，因为空结束符（在字符串末尾由 0 占用的字节）也需要一个字节。避免这个问题的简单而有效的方法便是使用命名常量来说明所有的字符串。这种方法的关键是每次都用如下方法来使用命名常量：把字符串的长度说明为 CONSTANT+1，然后用 CONSTANT 作为字符串的长度。请看下面的例子：

```
/*     Declare the string to have length of "constant+1"
       Every other place in the program，"constant" rather
       than"constant+1"is used.*/

char string[NAME_LENGTH+1]={0};/* string of length NAME_LENGTH */
     这个串的长度说明为 NAME_LENGTH+1

/*     Example 1: Set the string to all 'A's using the constant.
       NAME_LENGTH，  as the number of 'A's that can be copied.
       Note that NAME_LENGTH rather than NAME_LENGTH+1 is used.   */

for(i=0;i<NAME_LENGTH;i++) ——这里，串操作使用 NAME_LENGTH
     string[ i ]='A';

…

/*   Example2:Copy another string into the first string using
     the constant as the maximum length that can be copied.   */

strcpy(string，some_other_string，  NAME_LENGTH);
```

如果没有处理这一问题的约定，那么有时你可能会把其长度说明为 NAME_LENGTH，而使用时认为其长度为 NAME_LENGTH-1，而有时又会说明成 NAME_LENGTH+1而操作时认为是 NAME_LENGTH。这样，每次使用字符串时，你都不得不努力回忆是如何说明的。

而如果你每次都用同一种方法来使用字符串，就不必分别记住每个字符串是怎样说明的了。从而也就避免了由于记错某个字符串的说明方式而引起的错误。因此建立约定可以减少工作量和程序错误。

把字符串初始化为“0”以避免无限长字符串。C 是通过字符串结尾的零结束符——字符串末尾被 0 占用的字节来确定字符串结束的。不管你认为字符串有多长，只要 C 没有发现零结束符，它就认为字符串没有结束，如果你忘记了在字符串末尾放置一个0，那么你对字符串的操作
将可能不会按照预期的进行。

可以用两种方法来避免这个问题。首先，当你说明字符数组时，可以把它初始化为0，如下所示:

```
    char EventName[MAX_NAME_LENGTH+1]={0};
```

第二，对字符串进行动态分配时， 用 calloc( )代替 malloc( )来把它们初始化为0。calloc( )

的功能是分配内存，并马上初始化为0。而 malloc( )的功能只是分配内存而不进行初始化。

在 C 中用字符数组来代替指针。如果内存空间不是主要问题的话（通常是这样），应把所有的字符串变量都说明为字符数组。这样作可以避免指针问题，并且在你出错时，编译程序会产生警告信息。

用 Strncpy()代替 strcpy()以避免无限长字将串。C 中的字符串子程序有安全的也有危险的。像 strcpy()和 strcmp()之类的子程序是很危险的,因为只有当遇到零结束符时它们才会停止。而 strncpy()和 strncmp()相对来说,则要安全得多,因为它们把某一参数当作最大长度,因此即使某些字符率是无限长的，你的调用函数也不会持续不断地永远进行下去。

11.5　逻辑变量

逻辑变量的使用很少会出问题，但是仔细使用它们可以使程序更清楚。

使用逻辑变量来说明程序。与其单纯地判断某一逻辑表达式倒不如把表达式赋给某一变量，以便使得判断的意义不会被误会。例如，在下面的程序段中，if 判断的目的到底是什么并不清楚，是为了结束？为了一个错误条件？还是别的什么？

```
If ( ( ElementIdx<0 ) || ( MAX_ELEMENTS<ElementIdx ) ||
     ElementIdx==LastElementIdx)
    {
     …
    }
```

而在下面的程序段中，通过使用逻辑变量，if 的意义就十分清楚了:

```
Finished =((ElementIdx<0)||(MAX_ELEMENTS<ElementIdx));
RepeatedEntry =(ElementIdx == LastElementIdx);
if( Finished || RepeatedEntry)
   {
    …
   }
```

使用逻辑变量来简化复杂的判断。编写一个复杂的判断时，往往需要尝试几次才能成功，而当以后想要修改这个判断时，往往又不知道这个判断到底是干什么的，使用逻辑变量可以简化判断。在上面的例子中，程序要判断的事实上是两个条件：子程序是否已经结束和它处理的是否是一个重复入口。通过建立逻辑变量 Finished 和 RepeatedEntry，可以使得 if 判断更简单、更易读、更不容易错误而且也更容易修改了。

下面是另一个复杂判断的例子，它是用 Pascal 来实现的:

```
If((eof(InputFile)and(Not InputError))and
   (MinAcceptableElmts<ElmtCount)and(ElmtCount<=MaxElmts))then
  begin
  { do something or other }
    …
```

```
end;
```

上面的判断是非常复杂的，但并不少见。它使读程序的人产生了很大的思想负担。我猜想你甚至根本不会去试图理解这个 if 判断，而只会说："在需要时再看看它到底要干什么吧！"对此要引起注意，因为别人在读你的程序时，如果遇到类似判断，也会有与你相同的想法的。

下面是通过引入逻辑变量对上述程序进行简化后的程序：

```
AllDataRead          := eof(InputFile)and(Not InputError);
LegalElementCount:=(MinAcceptableElmts<ElmtCount)and
                        (ElmtCount<=MaxElmts);
If(AllDataRead and LegalElementCount) then  ——这里是简单的判断
  begin
 { do something or other}
 …
 end;
```

这段程序要比改动前简单得多。你可以很容易看到 if 判断的条件。

如果必要的话，立自己的逻辑类型。某些语言如 Pascal,含有已经定义的逻辑类型。而 C 等语言则没有。在像 C 这种语言中，可以定义自己的逻辑类型。在 C 中，你可以像这样来作：

```
typedef   int   BOOLEAN;      /* define the boolean type */
```

把变量说明为 BOOLEAN 而不是 int，可以使得使用它们的意图更清楚，并且在某种程度上使程序成为自说明的。

11.6　枚举类型

枚举类型是允许对某一类对象的每一个成员都用英语来进行描述类型。Ada、C 和 Pascal 都支持这种类型，通常在知道一个变量的所有可能值，并且想用词将它们表达出来时使用枚举它们。以下是 Ada 中枚举类型的几个例子：

```
type COLOR is
      (
       COLOR_INVALID，
       COLOR_RED，
       COLOR_GREEN，
       COLOR_BLUE，
       );

type COUNTRY is
      (
       COUNTRY_INVALID，
       COUNTRY_US，
       COUNTRY_ENGLAND，
       COUNTRY_FRANCE，
       COUNTRY_CHINA，
```

```
  COUNTRY_JAPAN，
      );
type OUTPUT is
    (
      OUTPUT_INVALID，
      OUTPUT_SCREEN，
      OUTPUT_PRINTER，
      OUTPUT_FILE，
      );
```

枚举类型是某些陈旧说明方法的有力替换工具，使用它你就不必再说:“1代表红色，2代表绿色，3代表蓝色…”。下面提供了使用枚举类型的几条准则:

使用枚举类型来提高可读性。你可以用下面的语句:

```
  if ChosenColor=COLOR_RED
```

来代替

```
  if ChosenColor=1
```

显然，第一个语句的含义要比第二个清楚得多。无论何时看到数值型方案时，你都应考虑一个用枚举类型来代替是不是会使可读性更好些。

使用枚举类型来提高可靠性。在 Pascal 中，枚举类型可以使得编译程序比在使用整值和常量对类型进行更加彻底的检查。使用命名常量时，编译程序无法知道只有 COLOR_REDCOLOR_BLUE 和 COLOR_GREEN 才是合法值。Output=COLOR_RED 和 COLOR=COUNTRY_ENGLAND 这两个语句都会被认为是合法的。而当你使用枚举类型时，如果说明某一变量为 COLOR，那么编译程序将只允许 COLOR 取 COLOR_RED，COLOR_BLUE 和 COLOR_GREEN 三个值。

使用枚举类型来改善易修改性。枚举类型可以使你更容易地改动代码。我在“1代表红色、2代表绿色、3代表蓝色”中发现了一个错误，那么你就不得不在整个程序中找出所有的1，2，3并进行修改。而如果使用枚举类型的话，你只要修改一下类型定义并重新编译一下就可以了。

用枚举类型来代替逻辑变量。逻辑变量往往无法充分表达出需要它表达的含义。比如，假设当某个子程序成功地完成了它的任务时其返回值为 True，否则为 False。而后来你又发现事实上 False 分两种情况，第一种情况是任务失败但其影响仅限于子程序内。第二种情况是任务失败并且产生了将传播到程序其余部分的致命错误。在这种情况下，使用有 success，Warning 和 FatalError 三个值的枚举类型将比使用仅有 True 和 False 两个值的逻辑变量有效和清楚得多。而且，当失败和错误的种类再增加时，对其进行扩展以区分这些情况也是非常容易的。

检查无效值。如果在 if 或 case 语句中测试枚举类型,那么应检查无效值。在 case 语句中使用 else 来捕捉无效值：

```
case (ScreenColor)
  ColorRed:    …
  ColorGreen: …
  ColorBlue:   …
  else          ——这里是无效值判断
```

```
    PrintErrorMsg('Internal Error 752，Invalid color.')
  end;{case}
```

把枚举类型的第一个入口保留为无效的。在说明枚举类型时应把第一个值保留为无效值。这方面的例子请参见前面 Ada 例程中对 COLOR、COUNTRY 和 OUTPUT 的说明。许多编译程序都把枚举类型中第一个元素的值赋为零。把被赋为0值的那个元素说明为无效，可以帮助找出不恰当初始化的变量，因为当它们失效时更容易为0。

11.6.1　如果所用的语言不支持枚举类型

如果你所用的语言不支持枚举类型，那么可以用预处理程序宏或全局变量来模拟它。下面是一个在 Basic 中模拟枚举类型的说明：

```
' set up COLOR enumerated type
ETColorInvalid      =0
ETColorRed          =1
ETColorGreen        =2
ETColorBlue         =3
' set up COUNTRY enumerated type
ETCountryInvalid    =0
ETCountryUS         =1
ETCountryEngland    =2
ETCountryFrance     =3
ETCountryChina      =4
ETCountryjapan      =5
' set up ANSWER enumerated type
ETAnswerInvalid     =0
ETAnswerYes         =1
ETAnswerNo          =2
ETAnsweMaybe        =3
```

有了上面这段说明，你就可以使用其中的变量而不必再用数字常量了，从而可以改进程序的可读性。

非常巧合，上例同时也是一个使用命名约定的例子。在上例中，前缀 ET 用来表示这一个变量是模拟枚举类型，这一命名约定清楚地表明以 ET 为前缀的变量在初始化之后不应再被赋值，同时，它也降低了当你使用与枚举类型某一值相近的名称时发生命名冲突的可能性。

11.7　命名常量

命名常量很像一个变量，只是一旦你给它赋值之后便不能再改变它的值了。命名常量可以

使你通过名称而不是具体的数来引用某一固定的数值，例如，用 MaximumEmployees，而不是1000。

使用命名常量是一种对程序进行参数化的方法——把程序可能变动的地方放入参数中，一旦真的需要变动时，只要在一处修改参数而不必在整个程序中到处进行修改。假设按照你认为需要的长度说明了某一数组，接着便产生了越界错误因为数组不够大，这时你就会了解命名常量的优点了。当需要改变某一数组的大小时，你只要改变用来说明数组的常量就可以了，这种“单点控制”技术在朝着使软件真正“软”起来的方向上迈出了一大步——使修改工作非常容易。

在数据说明中使用命名常量。在数据说明和需要知道所处理数据规模的语句中使用命名常量，可以改善程序的可读性、可维护性。在下面的 Pascal 程序中，是用 PhoneLength_c 而不是值7来描述雇员的电话号码长度的：

```
Const
    PhoneLength_c = 7;     ——PhoneLength 作为一个常量说明
Type
    EmployeePhone_t=array[l..PhoneLength_c] of char;   ——它在这里使用
  …
{ make sure all characters in phone number are digits }
for   i:=1   to   PhoneLength_c   do   ——它也在这里使用
      if   PhoneNumber[i]<‘0’ or   PhoneNumber[i]＞’9’   then
        {do some error processing}
        …
```

这是一个很简单的例子，但是不难想象，程序中许多地方要用到电话号码长度。

在编写上面这个程序时，所有的雇员都住在同一城市里，因此电话号码长度只有7位，但随着公司的开发，在其它城市和州中出现了公司的雇员，因此需要更长的电话号码。如果你已经对程序进行了参数化，你只要在定义命名常量 PhoneLength_c 的地方进行修改就可以了。

正如你所预料的，使用命名变量也极大地改善了程序的可维护性。事实上，任何有助于对程序中可能发生变动部分进行集中控制的技术，都可以降低程序的维护工作量。

避免常数值。应该使用命名常量来代替它，可以使用编辑程序来搜寻程序中的“2，3，4，5，6，7，8，9”等常数值，并用命名常量来代替它们，并且要保证没有偶然地用到它们。

使用全局变量来模拟命名常量。如果你所用的语言不支持命名常量，可以用全局变量来模拟它们。通过借鉴前面例子中模拟枚举类型的方法来模拟命名常量，即使语言不直接支持命名常量，你仍然可以获得由命名常量带来的大部分好处。

一致地使用命名常量。在程序中一会儿使用命名常量，一会儿使用常数值，而且它们代表的又是同一实体的话，那将是非常危险的。如果说，有些编程方法是在自找错误的话，那么这种方法就是其中最典型的一种，当需要改变实体的值时，很可能会在改完命名常量的定义之后便认为完事大吉了，而把程序中的常数值仍遗留在那里，从而使程序产生神秘的错误，而且，这种

错误是非常难以修改的。

11.8　数　组

数组是最简单也是最常见的结构化数据。在某些语言中，数组也是唯一的一种结构化数据。数组包括同属于一种类型的一组元素，并且是通过下标直接存取的。下面是使用数组时应注意的一些问题。

确保所有的数组下标都没有越界。可以认为，数组所有使用问题都是由数组元素被随机存取而引起的。最常见的问题是程序试图对越界的数组元素进行存取。在某些语言中，这将导致运行或编译错误，而在其它语言中，这样只会导致不正确的结果。

把数组当作顺序性结构。许多计算机科学家建议决不要随机地对数组进行存取。他们认为对数组的随机存取与程序中的 goto 语句一样有害:这种存取往往是不受限制的，容易产生错误，而且很难验证是否正确。因此，他们建议用其元素以顺序存取的集合、堆栈或者队列来代替数组。

将来的程序语言中或许会直接支持这些结构。现在的程序员们不得不用存取子程序来实现它们。实验表明用这种方法进行的设计往往引入较少的变量和变量引用，因此可以提高软件效率和可靠性。

检查数组边界。如同检查循环结构的边界一样，在检查数组边界的过程中你也会发现许多错误。在心中问自己:代码对第一个元素的存取是正确的吗？是否错误地存取了它前面或者后面元素？最后一个元素呢？是否有越界错误？最后再检查程序是否正确存取了数组中间的元素。

对于多维数组，要保证其下标的顺序是正确的。通常是很容易把 Array[j,i]错写成 Array[i,j]的，因此应仔细检查多维数组的下标顺序是否正确，在 i,j 表示意义易混淆的情况下应该用意义更清楚的名称来命名数组下标。

警惕下标的错误交叉。当使用嵌套循环时,往往容易把 Array[i]错写成 Array[j]从而使内外循环的目的颠倒，在程序中应注意这一问题。不过更好的办法是使用比 i，j 更有说明意义的名称以彻底避免这类错误。

使数组的长度留有一定裕度。数组的越界错误是非常常见的。即使是在存取数组时只是发生了一个元素的越界也会引起严重错误。因此在说明数组时使其长度比实际需要的稍长些，可以使越界错误的后果不致很严重。

这是一种可以容忍的不太正规的技巧，在这样作之前，心中一定要有数。这样才能使这种方法发挥出应有效力。

在 C 中，使用 ARRAY_LENGTH（ ）宏来处理数组。采用如下例所示的通过定义 ARRAY_LENGTH（）宏的方法来处理数组，可以提高程序的灵活性。

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

当对数组进行操作时，可以使用 ARRAY_LENGTH（ ）宏代替命名常量作为数组的上界，请看下面的例子:

```
ConsistencyRatios[ ] =
    { 0.0，0.0，0.58，0.90，1.12，
      1.24，1.32，1.41，1.45，1.49，
```

```
        1.51，1.48，1.56，1.57，1.59 };
…
for(RatioIdx = 0;RatioIdx < ARRAY_LENGTH(ConsistencyRatios);
      RatioIdx++)   ——这里使用宏
```

这种技术尤其适用于类似上例中的那种无维数组。当你需要增加或减少人口时，不必改变描述数组大小的命名常量，当然，这种技术也适于有维数组，这时你使用这种技术便可以免去
为定义数组而建立额外的命名常量。

11.9　指　针

指针是现代编程中最容易出错的区域。使用指针是非常复杂的，只有你对所用编译程序内存管理方式有非常深刻理解之后，才能正确使用它。

11.9.1　理解指针

概念上，每个指针包括两部分：内存存储单元及对这个存储单元中内容的解释。

内存中的存储单元

内存中的存储单元就是地址，通常是用十六进制数来表示的。在 Intel 分段式处理器中，地址是由段及对段的位移联合组成的，如 02BF:0010。而在 Motorola 处理器中，地址就是一个 32 位值，如 0001EA40。指针本身只含有地址，要使用指针指向的数据，你必须到达那个地址并解释那个内存存储单元中的内容。如果你看一下那个存储单元中的存储信息，会发现其中只不过是一些位的组合而已，必须对它加以解释才能使它有意义。

怎样解释存储单元中的内容

解释存储单元中内容的基础是指针的基本类型。如果指针指向一个整数，它的真实含义是编译程序把由指针提供的存储单元解释为一个整数。当然，可能会出现整数指针、字符串指针和浮点数指针同时指向同一个存储单元的情况，而此时只有一个指针正确地解释了该单元中的内容。

在考虑指针时，应该想到内存本身并不具有与之相对应的解释。只有通过使用某一特定类型的指针，才能把某一存储单元中的位解释成有意义的数据。

在图11-1中，表示了用几种方法解释同一存储单元而获得的不同内容。

在上图中，每种情况下指针所指的存储单元中都含有十六进制数0A，而0A 以外所采用的字节数多少则取决于对存储信息的解释方式。而存储信息被使用的方式也取决于它被解释的方式（它也取决于你所使用的处理器，因此不要认为上述结果也适合于你的 PC_CRAY）。同样的存储信息可以被解释成字符串、浮点数、整数或其它任何东西，这要取决于指向这一存储信息的指针基本类型。

11.9.2　关于使用指针的几点建议

对许多错误来说，找出它是很容易的，但要改正它则往往比较困难。而指针错误则不然。指

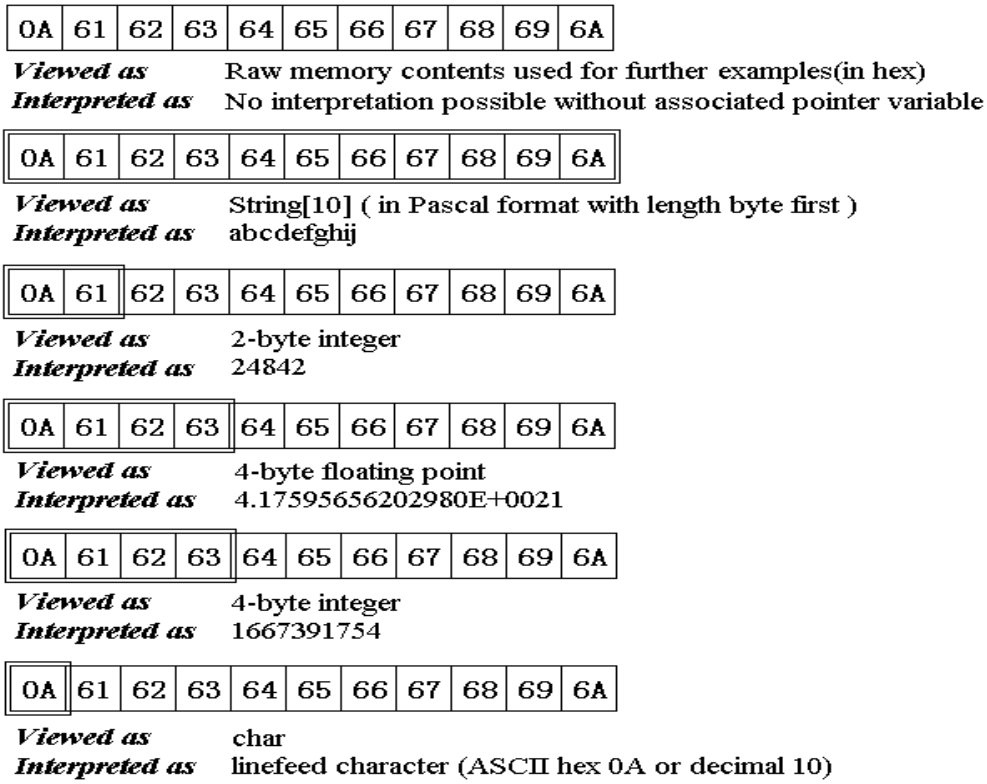


图11-1　各种数据类型使用的内存空间

针错误主要是由于它指向了错误的地址而引起的。当你给一个错误指针赋值时，你就向错误的内存存储单元中写进了数据，这叫作内存冲突。有时内存冲突会导致灾难性的后果:有时它会改变程序另一部分的计算结果;有时它会使你的程序意外地跳过某个子程序;而有时它则什么也没有做。这最后一种情况相当于埋下一颗定时炸弹。当你在向最重要的客户演示程序的前几分钟，它才会突然爆炸，把你的程序搞得一塌糊涂。总之，指针错误的现象看起来似乎是与指针错误无关的。因此，改正指针错误最困难的是找出错误。

需要采取两个步骤来防止指针错误。首先应防止引入指针错误。应采取一切可能的手段来防止引入指针错误，因为它一旦产生便很难改正，其次应尽可能早地发现指针错误。最好是一产生指针编码错误便发现它。因为指针错误的表现往往是令人难以琢磨的，因此应采用各种办法来争取尽早发现它。下面是怎样才能达到这两个目的的一些方法:

把指针操作独立在子程序中。假设在程序中有几处使用了链表，不如用诸如 Nextlink()、Previouslink(),Insertlink()和Deletelink()之类的存取子程序来存取它们.通过尽量减少对指针的存取,可以减少犯指针错误的机会,你也就不必费尽心机地去寻找这些错误了。同时由于子程序中的代码相对其余部分是独立的，以后也可以方便地重新使用它们。编写分配指针的函数也是一种对数据进行集中控制的方法。

在使用指针之前对它进行检查。在程序的关键部分中使用指针之前，一定要确认它所指向

的内存存储单元是合理的。例如，你希望内存存储单元介于 StartData 和 EndData 之间，那么应该检查一下 StartData 之前和 EndData 之后，看是否把指针指向了这里。同时，你还将决定 StartData 和 EndData 中的值是否属于你的环境。而如果你使用存取子程序的话，便可以让这些工作自动进行了。

在使用变量之前应先检查一下这一变量。有时需要对指针指向的变量值进行合理性检查，比如，你认为指针将指向一个介于 0 到 1000 之间的整数时，应该检查一下这个值是否超过了 1000。如果指向的是一个 C 语言中的字符串，就应检查一下长度超过 100 的字符串。如果你使用存取子程序的话，这一工作也可以自动进行。

使用标记字段来查找错误内存。“标记字段”是仅仅出于查错的目的而加入某一结构的字段。当分配变量时把应该保持不变的值放入标记字段中，当使用这个结构时，检查一下这个标记字段的值。如果标记字段的值与预期不符的话，说明数据有误。

当然，不必每次使用这一变量时都检查一下标记变量。你可以只在丢弃这个变量之前检查一下标记字段的值，如果其值有误则说明在变量生存期内它的内容有错误，不过，越是频繁地检查标记安全，你也就越容易找到引起错误的原因。

使用显示冗余技术。一个替代标记字段的方法是使用某一字段两次。如果在冗余字段中的数据不匹配，则说明内存有误。如果你直接操作指针的话，这样作需要进行许多内存操作；而如果你把指针操作孤立在子程序中的话，则只需在几处添加几个重复代码。

释放指针后，把它设为 NULL 或 NIL。一种常见的指针错误是“悬挂指针”，即误用了已经被 Dispose( )或 Free( )的指针。指针错误难以发现的原因之一便是有些指针错误并不产生任何现象。在释放指针之后把它们置于 NULL，你并没有改变读到了由悬挂格针指向的数据这一事实，但可以保证在向悬挂指针写数据时产生错误。这个错误可能是非常丑陋、肮脏或是灾难性的，但可以使你知道发生了错误。与其它许多操作一样，你也可以通过存取子程序来自动进行这项工作。

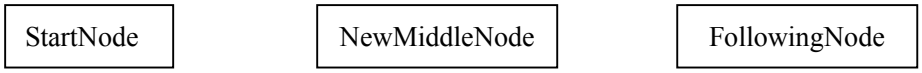
使用额外的指针变量以增加清晰性。不管怎样，都不要吝于供给指针变量。这一原则在别的地方也被说成某一变量应只具有一个功用。这一点对指针变量尤其重要。如果不知道为什么反复使用变量 GenericLink 和 Pointer.Next,Last.Next 指向的是什么的话,是很难弄清正在对链表进行什么操作的。考虑一下下面的这个 C 语言结点插入程序:

```
void insert_link
  (
  NODE CrntNode，
  NODE InsertNode
  )
  {
  /* insert "InsertNode" after "CrntNode" */
    InsertNode.Next = CrntNode.Next;
    InsertNode.Previos = CrntNode;
    If( CrntNode.Next != NULL)
      {
```

```
            CrntNod.Next.Previous = InsertNode;   ——这行是难以理解的
        }
         CrntNode.Next = InsertNode;
         }
```

这是一段向链表中插入结点的传统程序，事实上它完全不必这样难以理解。插入一个新结点需要引入三个对象:当前结点、当前结点的下一个结点、等待被插入的结点。而上面的程序只是清楚地表示出了两个结点——InsertNode 和 CrntNode(插入结点与当前结点)。这迫使你要看出并且记住 CrntNode.Next 结点也包括在这一插入过程中，如果不用当前结点之后的结点而要表示出正在发生了什么的话，那么你作出的图应该是这样的:

而更清楚的应该是像下图这样同时表示出三个结点的:



下面是明确表示出引入了三个对象的 C 语言程序:

```
 void insert_link
      (
      NODE    StartNode，
      NODE    NewMiddleNode
      )
      {
      NODE    FollowingNode;
      /* insert “NewMiddleNode” between “StartNode” and “FollowingNode” */
      FollowingNode              = StartNode.Next;
      NewMiddleNode.Next     = FollowingNode;
      NewMiddleNode.Previous=StartNode;
      If ( FollowingNode != NULL)
         {
         FollowingNode.Previous = NewMiddleNode;
         }
      StartNode.Next = NewMiddleNode;
      }
```

这一段程序加了一行代码，但是没有了前一段例程中的 CrntNote.Next.Previous，从而变得容易理解了。

简化复杂的指针表达式。复杂的指针表达式是非常难读的。试想一下，有几个人会读懂含有像 P.q.r.s.^.data 或 p->q->r->S.data 之类东西的表达式呢？下面是一个尤其难读的 C 语言代码段:

```
for(RateIdx = 0; RateIdx < Num; RateIdx++)
 {
 NetRate[RateIdx]=
    BaseRate[RateIdx]*Rates>Discounts->Factors->Net;
 }
```

对像上例中这样复杂的指针表达式，恐怕只能“破译”而没法读。如果程序中含有复杂的指针表达式的话，可以通过使用清楚命名的变量来使其意图更清楚些。下面是对上面程序改进后的程序:

```
QuantityDiscount = Rates->Discount->Factors->Net;
for ( i=0; i < Num; i++)
    {
    NetRate[i] = BaseRate[i] * QuantityDiscount;
    }
```

通过上述简化，不仅改善了程序的可读性，同时可能也提高了程序的性能，因为循环内的指针操作也被简化了。

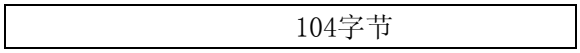
编写跟踪指针存储单元的子程序。通常,对程序危害最大的错误便是使用free()或dispose()去再释放已经被释放的指针。不幸的是，几乎没有什么语言能查找这类错误，不过，如果语言不支持我们的话，我们可以自己支持自己。保留一张含有全部已经分配的指针表，将释放一个指针之前检查一下指针是否在这个表上。例如，在C中，可以使用如下两个子程序:

- safe_calloc()。这个子程序与C中的calloc()接受同样的参数。它调用calloc()来分配指针，把新的指针加到已分配指针表上，然后把新指针返回到调用子程序中。这样作的另一个好处是你只需要在这一个地方检查由calloc()返回的是否是NULL，从而简化了程序其余部分的错误处理工作。
- safe_free()。这个子程序与C中的free()子程序接受同样的参数。它会检查传给它的指针是否已在分配指针表中。如果在的话，safe_free()会调用C中的free()子程序来释放这个指针并将它从表上去掉;如果不存在，safe_free()会打印出诊断信息并中止程序。

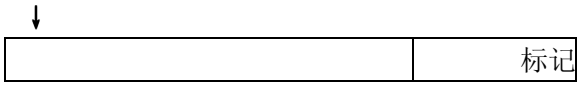
可以很容易对它们作出改动以把它们移到别的语言中。

在calloc()中分配一些冗余字节，并把这些字节作为标记字段。当使用safe_free()释放指针时，检查一下标记字段看是否有错误。在检查之后，抹掉标记字段，以便当你错误地试图再次释放同一指针时可以发现这一错误。例如，你要分配100字节。

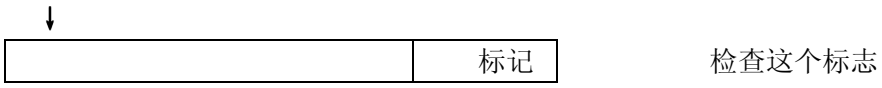
1.用Calloc()分配104字节，4个字节是冗余的



2.把最后4个赋值成标记字段然后向已分配的内存中返回一个指针;

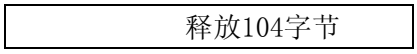


3. 当用 safe_free( )　释放指针时，检查一下标记字段



4. 如果标识字段正常的话，将其赋为 0 或其它你的程序认为是正确的标记值。因为你不会愿意在内存已经被释放后仍在其中保留一个正确的标记值。出于同样原因，应把这段存储单元中的数据改成某特定值而不是随机值。

5. 最后，释放这个指针



你可把这种方法与前面提过的合理性检查方法联合使用。为了检查指针所指的是不是合理的存储单元，应检查这一指针是否在已分配指针表上，而不是检查可能的内存范围。

画图。对指针的代码表示往往是令人困惑的，画图往往更有帮助。例如，图11-2中便表示出了前面提到过的链表插入问题。可以把它与程序对照一下:

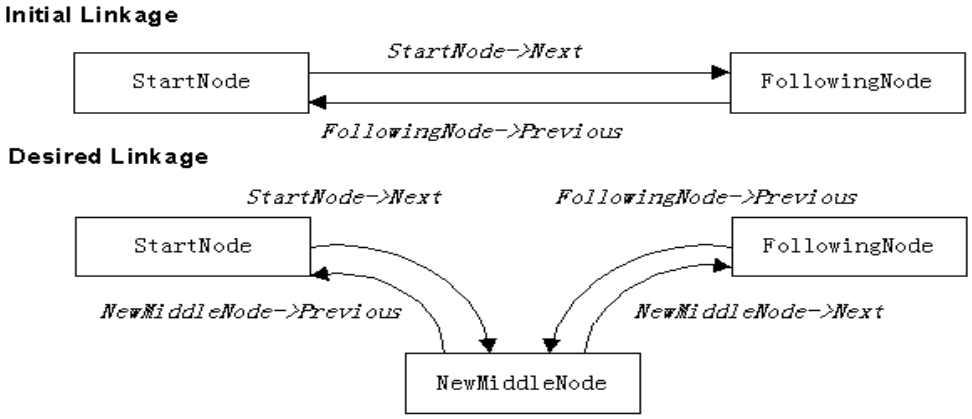


图11-2　指针再链接

按正确顺序释放链表中的指针。 在处理动态分配链表时，一个常见的问题是：在释放了链表中的第一个指针后，无法到达链表中的下一个指针。为避免这个问题，在释放当前指针之前，应保证有指针指向链表中的下一个元素。

编写输出指针地址的子程序。如果你所用的是具有分段式结构的 80X86 系列处理器，那么你所用的语言很可能不支持指针地址的格式化输出。这使得诊断信息的打印变得非常困难。一个比较容易的办法是编写一个子程序，它把指针作为变元并返回一个字符串“03af:bf8a”或类似的东西，当你调用语言支持的 Print、writeln()、Printf()等标准输出子程序时，可以调用这个输出指针地址的子程序。

在内在中划分出一段空间作为“降落伞”。如果你的程序是用动态内存，那么当突然出现内存溢出错误时，程序应该能够避免把用户的辛苦和数据扔在 RAM 中。解决这一问题的方案之一便是制作一个内存”降落伞”。首先确定你的程序为完成存储数据，退出等工作所需要的内存，然后在程序开始运行时，分配出相应的内存并把它保留起来。当内存溢出时，就可以靠这个内存“降落伞”来拯救你的辛苦数据了。

使用非指针技术。指针非常难以理解，很容易产生错误，而且往往对硬件有依赖性从而影响可移植性。因此，如果有其它方法能胜任指针工作的话，应该尽量采用这种方法来代替指针。

11.9.3　C 中的指针

以下是针对在 C 中使用指针的一些指导方针:

应使用显式指针类型而不是缺省类型。C 中对任何类型的变量都允许使用 char 或 void 指针，只要有指针在那儿就行，程序是不会关心它指向的是什么的，而如果你使用的是显式类型指针的话，编译程序则会对不匹配的指针类型等进行警告，但若你不用的话，编译程序是不会警告的。因此应该尽量使用特定的指式类型。

对这一原则的推论是，当你不得不进行类型转换时，应使用显式强制类型转换。比如，在下面的程序，一个 NODE_PTR 类型的变量正在被分配这一点就很清楚:

```
NodePtr=(NODE_PTR)calloc(1，sizeof(NODE));
```

避免强制类型转换。强制类型转换指的是把一种类型的变量强行送入另一种类型变量的空间中。强制类型转换会使你的编译程序失去对类型不匹配进行警告的能力，从而使你所采用的防错性编程技术产生漏洞，一个需要进行许多强制类型转换的程序往往说明其结构设计有问题。因此如果可能的话，应重新进行设计;如果做不到这一点，那么你应该尽量避免强制类型转换。

遵守参数传递的星号规则。在 C 中，只有当赋值语句中的变元前面带有“*”号时，才能从子程序中把这个变元传递回来。许多 C 语言程序员在面临这个问题时都感到很难作出决定。事实上，这一点是很容易记住的。只要在你给参数赋值时它的前面带有一星号，那么这个值就是被传回到调用程序的。不管你在说明中堆积了多少个星号，如果你想让某一值传回的话，那么在赋值语句中至少要有一个星号。例如，在下面的这个程序段中，赋给 parameter 的值就不是被传回调用程序的.因为在赋值语句中一个星号也没有:

```
void TryToPassBackAValue( int * parameter )
    {
    parameter= SOME_VALUE;
    }
```

而在下面的这个程序段中，赋给parameter的值就是被传回来的，因为在给parameter赋值的语句中，parameter 前面带有一个星号:

```
void TryToPassBackAValue(int * parameter)
    {
    *parameter = SOME_VALUE;
    }
```

使用 sizeof()来确定内存存储单元中变量的规模。使用 sizeof( )来查找变量规模要比在手册中查找变量规模容易，而且 sizeof()可以处理自己建立的结构，而这种结构在手册中是没有的，使用 sizeof()不会影响性能，因为它的计算是在编译过程中进行的，sizeof()也是可以移植的——在另一种环境下编译会自动改变由 sizeof( )计算出来的值。它所需要的维护工作也是很少的，因为你可以改变你已经定义的类型，并且分配工作将是自动进行的。

11.9.4　检查表

基本数据

常数

- 代码中是否避免了"奇异数"(常数？)
- 程序中是否采取了措施来防止出现被“0”除错误？
- 类型转换是显式进行的吗？
- 如果在同一个表达式中出现了两种不同类型的变量，是否对表达式按照你的意愿进行求值？
- 程序中是否避免了混合类型比较？
- 在编译过程中是没有警告的吗？

整型数

- 使用整型数相除表达式的结果是否与预期的一致？
- 整型数表达式中是否避免了整型数溢出问题？

浮点数

- 是否避免了数量级相差过大的数之间加减运算？
- 程序中是否系统地采取措施来防止舍入误差问题？
- 程序中是否避免了对浮点数进行相等比较？

字符和字符串

- 程序中是否避免了常数型字符和字符串？
- 对字符串的引用是否避免了边界错误？
- 若是用 c 写成的程序，是否是把字符数组和字符串指针区别对待的？
- 若程序是用 C 写成的，是否遵守了把字符串长度说明为 CONSTANT+1这一约定？
- 是否用字符数组代替指针？
- 在 C 语言程序中，是否把字符由初始化成了 NULL 以避免出现无限长字符串？
- 在C语言程序中,是否用strncpy()代替了strcpy()？ 并且用了strncat()和strncmp()？

逻辑变量

- 程序中是否使用了附加的逻辑变量来说明条件判断？
- 程序中是否使用了附加的逻辑变量来简化条件判断？

枚举类型

- 程序中是否用枚举类型代替了命名常量来改善可读性、可靠性和易改动性？
- 是否用了枚举类型代替逻辑变量以改进可读性和灵活性？
- 在使用了枚举类型的判断中是否检查了无效值？
- 枚举类型的第一个入口是否是保留为无效的？

命名常量

- 在数据说明中使用的是命名常量吗？
- 是否一致地使用了命名常量，而不是一会儿使用命名常量，一会儿使用数值？

数组

- 是否所有的下标都在数组界限之内？

- 是否对数组所有的引用都没有发生越界错误？
- 多维数组的下标排列顺序正确吗？
- 在嵌套循环中，作为循环变量的数组下标是正确的吗？是否出现了交叉错误？

指针

- 是否把指针操作独立在函数中？
- 指针引用是有效的吗？是否误用了悬挂指针？
- 程序中在使用指针之前，是否对它进行了检查；
- 在使用由指针指向的变量之前，是否对其有效性进行了检查？
- 在释放指针之后，是否把它们的值赋成了 NULL 或 NIL？
- 为了提高可读性，程序中是否使用了所有需要用的指针变量？
- 链表中的指针是否是按正确的顺序释放的？
- 程序中是否分配了备用内存空间以作为内存溢出时拯救数据和工作努力的降落伞？
- 是否是在万不得已时才使用指针的？

11.10　小　结

使用各种特定的数据类型意味着需要记住许多种规则。因此要用上面的检查表来确认你已考虑过了所有常见问题。

第十二章　复杂数据类型

目录

相关章节

基本数据类型：见第 11 章
信息隐蔽：见 6.2 节

本章将要讨论的是自己建立的数据类型。在第十一章所叙述过的基本数据类型是必不可 少的。当以它们为基础来建立高层次结构时，才打开了通往有效使用数据的大门。

如果你对高级数据结构熟悉的话，你可能会对本章的某些内容已经知道了，你可以跳过 12.1 节。浏览一下“灵活信息格式化举例”和 12.3 节，在 12.3 节你将发现在其它数据结构课本中所没有的观点。

12.1　记录与结构

“结构化数据”指在其它类型基础上建立起来的数据。由于数组是其中一个特例，所以它在第十一章讨论。本书主要讨论由用户定义的结构化数据——Pascal 和 Ada 中的“记录”和 C 中的“结构”。以下是使用结构化数据的几点原因：

使用结构化数据来表明数据间的关系。结构化数据把同类的所有数据都集中在一起。有时,读懂一个程序最大的障碍是找出哪一个数据是与另外一个数据相联的。这就像到一个小镇上去问一个人都有谁与他有关系,最终你会发现每个人都与其它人有些关系．但却又都不很确定，所以你永远也得不出确切的答案。

如果数据是经过仔细结构化的,那么找出数据间的联系就容易得多了。下面是一个使用没有结构化的、容易会人误会的数据的 Pascal 程序的例子：

```
Name        :=  InputName;
Address     :=  InputAddress;
Phone       :=  InputPhone;
Title       :=  InputTitle;
Department  :=  InputDepartment;
Bonus       :=  InputBonus;
```

由于数据是没有结构化的，看起来似乎所有的赋值语句都是属于一类。而事实上 Names、Address 和 Phone 是雇员的变量,而 Title、Department、Bonus 则是与监工有关系的变量。而从上

面的程序段中根本找不出使用了两种变量的暗示。在下面的程序段中，由于使用了结构化程序，使得数据间的关系清楚多了：

```
Employee.Name          :=   InputName;
Empployee.Address      :=   InputAddress;
Employee.Phone         :=   InputPhone;

Supervisor.Title       :=   InputTitle;
Supervisor.Department  :=   InputDepartment;
Supervisor.Bonus       :=   InputBonus;
```

由于在代码中使用了结构化数据，因此某些数据是与雇员有关，而另一些数据是与监工有关这一点就变得十分清楚了。

使用结构化数据来简化对成块数据的操作。你可以把相关的数据组合进一个数据结构中，然后对这一结构进行操作。对数据结构进行操作要比对每一个元素进行同样操作容易得多。而且可读性更好、代码行数相对也要少些。

假设有一组相关的数据，例如人事数据库中关于雇员的数据。如果不把这些数据组织成结构，那么即使简单地拷贝这些数据也要许多行代码，请看下面这个用 Basic 写成的例子：

```
NewName        =   OldName
NewAddress     =   OldAddress
NewPhone       =   OldPhone
NewSSN         =   OldSSN
NewSex         =   OldSex
NewSalary      =   OldSalary
```

每次要传送关于某一雇员的信息，都不得不使用上面整个一组语句，如果想加入一条新的雇员信息，比如，NumWithholdings——你都将不得不找出每个使用这一级语句的地方并在其中加一条赋值语句：

```
NewNumWithholdings=   OldNumWithholdings
```

你能想象出两个雇员之间的交换数据有多么可怕吗？请看下面：

```
' swap new and old employee data
PrevOldName        =   OldName
PrevOldAddress     =   OldAddress
PrevOldPhone       =   OldPhoe
PrevOldSSN         =   OldSSN
PrevOldSex         =   OldSex
PrevOldsalary      =   OldSalary

OldName            =   NewName
OldAddress         =   NewAddress
OldPhone           =   NewPhone
OldSSN             =   NewSSN
```

```
OldSex          =   NewSex
OldSalary       =   NewSalary

NewName             =   PrevOldName
NewAddress          =   PrevOldAddress
NewPhone            =   PrevOldPhone
NewSNN              =   PrevOldSSN
NewSex              =   PrevOldsex
NewSalary           =   PrevOldSalary
```

处理这个问题比较简单的方法是说明一个结构化变量。下面是一个用 Microsoft Quick Basic 写成的采用这种方法的例子。其中使用了非标准 Basic 的特性、TYPE…ENDTYPE 语句。

```
TYPE  tEmployee
    Name        AS  STRING
    Address     AS  STRING
    Phone       AS  STRING
    SSN         AS  STRING
    Sex             AS  STRING
    Salary      AS  STRING
END  TYPE

DIM     NewEmployee         AS  tEmployee
DIM     OldEmployee         AS  tEmployee
DIM     PrevOldEmployee     AS  tEmployee
```

现在，你只要用三个语句就可以在新旧雇员记录之间进行数据交换：

```
PtevOldEmployee     =   OldEmploydee
OldEmployee             =    NewEmployee
NewEmployee         =   PtevOldEmployee
```

如果你想在其中加入一个域，如 NumWithholdings，那你只要把它加人类型说明就可以了。而不对程序其余部分作任何改动。所有的标准 Pascal、各 C 语言都有类似能力。

使用结构化数据来简化参数表。可以通过使用结构化数据来简化子程序参数表。这一技术与刚才展示的技术是基本类似的。你可以把相联系的数据组成一个数据结构，然后把整个数据结构进行传递，从而省去了一个个地传递每一个要传递元素的麻烦，下面是用 Basic 写成的未使用数据结构调用子程序的例子：

```
CALL  HardWayRoutine(  Name,  Address,  Phone,  SSN,  Sex,Salary  )
```

下面是用数据结构简化了参数表的子程序调用例子；

```
CALL  EasyWayRoutine(  Employee  )
```

如果想在第一种调用中(未使用数据结构)加入 NumWithholdings，你就不得不在整个程序中找出并修改每一个对 HardWayRoutine()的调用，而如果在第二种调用中想在 EmoloveeRec 中加入 NumWithholdings，则只需改动类型说明而根本不必改动 EasyWayRoutine()。

你也可以用极端的方法来使用这技术，把程序中的所有变量都放入一个巨大的、臃肿的结

构中，然后把它在子程序间进行传递。细心的程序员都会避免超出逻辑需要地把数据捆在一起。而且，当结构中只有一或两个域被用到时，往往是传递这一两个用到特定的域而不是传递整个结构。这也是信息隐蔽的一种形式；某些信息被隐含在子程序中，而有些则要对子程序是隐含的。信息是在需要知道的基础上进行传递的。

使用结构化数据来降低维护工作量。因为在使用数据结构类型时你把相关数据都组织在一起，因此在改变数据结构时只需在程序中作出少数改动就可以了。对于与被改动的数据结构没有逻辑联系的代码段来说更是这样。由于改动往往容易产生错误，因此较少的改动就意味着更少的错误。如果你的 EmployeeRec 结构中有一个 Title 域要被删掉，那么你不必改动任何用到整个记录的参数表或赋值语句。当然，你将不得不改动特别处理 Title 的代码，因为它们与被删掉的 Title 域有概念上的联系。因而不能被忽略掉。

由使用结构化数据带来的优势主要体现在与 Title 域没有逻辑联系的代码段中，有时，程序中含有指向数据集合全体而不是其中某一部分的代码段。在这种情况下，个别结构要素如 Title 之所以被引用，仅仅是因为它们是数据集合的一部分，这些代码段与 Title 域没有任何特别的逻辑联系，因此，可以在变动 Title 时被忽略掉(当使用数据结构时)，因为它是把数据结构当作整体而不是一个个的元素来对待的。

12.2　表驱动方法

表是几乎所有数据结构课本都要讨论的非常有用的数据结构。表驱动方法出于特定的目的来使用表，下面将对此进行讨论。

程序员们经常谈到“表驱动”方法，但是课本中却从未提到过什么是“表驱动”方法。表驱动方法是一种使你可以在表中查找信息，而不必用逻辑语句(if 或 case)来把它们找出来的方法。事实上，任何信息都可以通过表来挑选。在简单的情况下，逻辑语句往往更简单而且更直接。但随着逻辑链的复杂，表就变得越来越富于吸引力了。

例如，如果你想把字符排序成字母、逗号和数字，将很可能采用如下所示的复杂逻辑链(用 Pascal 写成)：

```
if (( 'a' <= InputChar ) and ( InputChar <= 'z' )) or
    (( 'A' <= InputChar ) and( InputChar <= 'Z' )) then
    begin
    CharType := Letter
end
else if ( InputChar = " ) or ( InputChar = ',' ) or
      ( InputChar = '.' ) or ( InputChar = '!' ) or ( InputChar = '(' ) or
      ( InputChar = ')' ) or ( InputChar = ':' ) or ( InputChar = ';' ) or
      ( InputChar = '?') or ( InputChar = '-' ) then
begin
    CharType := Punctuation
end
else if ( '0' <= InputChar and InputChar <= '9' ) then
begin
    CharType :=  Digit
End;
```

而如果你使用的是查询表的话，就可以把每个字符类型都存储在由字符类型来存储的数组中，那么上例中复杂的逻辑链就可以简化成下面这个样子：

CharType:=CharTypeTable[InputChar]；

在这个程序段中，假定 CharTypeTable 已经被建立了,把程序的内容建立在表而不是 if 判断的基础上。如果使用是适当(如上例)的话,表方法要比复杂的逻辑简单得多，而且也更易于改动，效率也更高。

12.2.1　表驱动方法的通常问题

使用表驱动方法时，将不得不说明两个问题：

首先，你不得不说明如何寻找表中的入口,你可以使用某些数据来直接存取表。比如，你需要按月份把数据进行排序，那么进入月份表是非常容易的，你可以使用以 1 到 12 为下标的数组来实现它。

而若用其它数据来直接查找表的入口则显得有些力不从心了,比如需要把社会保险号码进行排序时,就不能用社会保险号来直接进入表,除非你把 999,999,999 个入口全部存在表中,这时你将被迫采用比较复杂的方法。下面是查找表的人口的几种方法：

- 直接存取
- 变址存取
- 阶梯存取

后面将对上述每种方法都进行详细的论述。

在使用表驱动方法时需要说明的另一个问题是，你将在表中存储些什么。在某些情况下，表查寻的结果是数据。如果是这种情况，你可以把数据存储在表中；在其它情况下，表查寻的结果是动作。在这种情况下,你可以把描述这一动作的代码存储在表中。在某些语言中，也可以把实现这一动作的子程序的调用存储在表中。但不论是哪种情况，表都已经变得很复杂了。

12.2.2　直接存取

与其它查寻表一样,直接存取表是用来代替比它更复杂的逻辑控制结构的,之所以称其为“直接存取”是因为用这种方法时,你不必为了找到你想要的信息而在表中绕来绕去。正如图 12-1 所表示的那样，你可以直接找出你想要的入口。

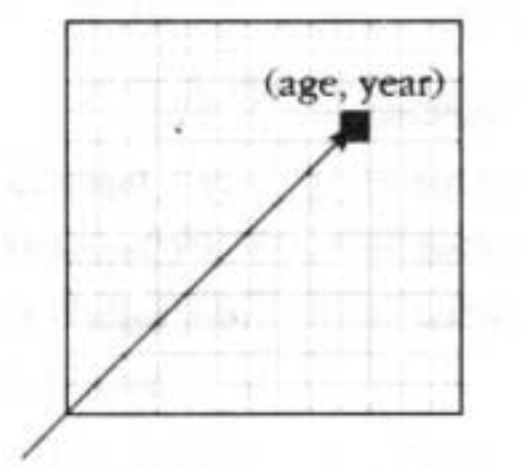


图 12－1　直接存取

示例 1：一个月中的天数

假设你需要一个可以返回每个月中天数的函数(为简单起见不考虑闰年)，一个比较笨的

方法是写一个大的 if 语句：

```
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
ENDIF
```

更简单,效率更高也更容易改动的方法是,把这些数据放在一个表中。在 Basic 中,必须首先建立表：

```
' INITIALIZE  TABLE  OF  "Days  Per  Month"  DATA
'
DATA 31，28，31，30，31，30，31，31，30，31，30，31
    DIM  DaysPerMonth(I)
    FOR  I=1  TO  12
      READ  DaysPerMonth(I)
    NEXT  I
```

现在，有了这个表，就可以用一个简单的数组存取来代替上面那段复杂而又臃肿的 if 语句了。

```
Days=DaysPerMonth(Month)
```

即使现在你想把闰年也考虑进来，程序仍然是非常简单的:

```
Days=DaysPerMonth(Month，IsLeapYear)
```

显然，如果再用 if 语句的程序中计算闰年的话，那么程序将不知有多么复杂。

确定一个月中的天数是一个比较简单的例子，因为你可以用变量 Month 来查寻表的入口。一般来说，可以采用控制着一大串 if 语句的数据来直接存取一个表。

示例 2：保险费用

假设你要编写一个计算医疗保险费用的程序,其中保险费用是随着性别、年龄、婚姻状况和是否吸烟而变化的。如果你用逻辑控制结构作这些工作的话,它应该是与下面这个 Pascal 程序段类似的：

```
if ( Sex = Female ) then begin
    if ( MaritalStatus = Single ) then begin
        if ( SmokingStatus = NonSmoking ) then begin
            if ( Age < 18 ) then
                Rate = 40.00
```

```
        else if ( Age = 18 ) then
             Rate = 42.50
        else  if ( Age = 19 ) then
             Rate = 45.00
        …
        else if ( Age > 65) then
              Rate = 150.00
    end
    else begin { SmokingStatus = Smoking}
        if ( Age < 18 ) then
             Rate = 44.00
        else if ( Age = 18 ) then
             Rate = 47.00
        else if ( Age = 19 ) then
             Rate = 50.00
        …
        else if ( Age > 65 ) then
              Rate = 200.00
    end
    else { Marital Status = Married }
  …
end; { if Sex …}
```

在上例中,只考虑了是否吸烟和性别、年龄而没有考虑婚姻状况,也没有考虑 18 岁到 65 岁之间的大部分年龄，但是其复杂程度已经相当惊人了。你可以想象一下,如果把影响保险率的所有因素都考虑进来的话，它将有多么复杂。

你或许会问:“为什么要对每一个年龄都进行判断而不把保险费用放入年龄数组呢？”问得好，如果把保险费用放入年龄数组的话，将极大地改进上面的程序。

不过,如果把保险费用放入所有影响因素的数组而不仅仅是年龄数组的话,将会使程序更简单，以下是 Pascal 中是如何说明数组的：

```
type
    Smoking_t = (Smoking，Nonsmoking);
    Sex_t     = (Male，Female);
    Marital_t = (Single， Married);
    Age_t     = 1..100;
var
    RateTable=array[Smoking_t，Sex_t，Marital_t，Age_t];
```

在 Pascal 中使用枚举类型的一大特点是你可以用类似 smoking_t 的参数来说明数组，而编译程序会自动识别出有两种抽烟状态从而知道数组中应该有两个元素。

定义好数组之后,你就需要确定如何把数据放进去。你可以用赋值语句,从磁盘中的一个文件中读入数据计算出数据或其它任何合适的方法来做到这一点。当你建立好数据之后,便做

好了计算保险费用的一切工作。现在就可以用下面这个简单的语句来代替前面那个复杂的逻辑结构了：

Rate:=RateTable[SmokingStatus，Sex，MaritalStatus，Age]；

这种方法的好处是可以用表查寻来代替复杂的逻辑控制，而表查寻的方法往往具有更好的可读性并且修改容易，同时还具有占用空间少和可读性强的优点。

示例 3：灵活信息格式

你还可以用表来描述由于变化太快而无法用代码来描述的逻辑,通过上面的几个例子,我们已经知道某些问题是可以用 if 语句来实现的，虽然有时这种方法显得很拙劣，但毕竟是可用的。然而在某些情况下，有些非常复杂的数据是无法用 if 语句来描述的。

如果你认为对直接表存取方法已经很熟悉了的话，可以跳过下一个例子,因为它只是比前几个例子稍微复杂一些。

假设你要编写一个打印存储在某一文件中信息的子程序。文件中通常含有 500 条信息,信息共有大约 20 种。这些信息来自于一个给出自身位置和水温的浮标。

每一条信息都有几个域，其开头都是一个表示该信息种类的识别标志。图 12-2 表示了信息的存储方式：

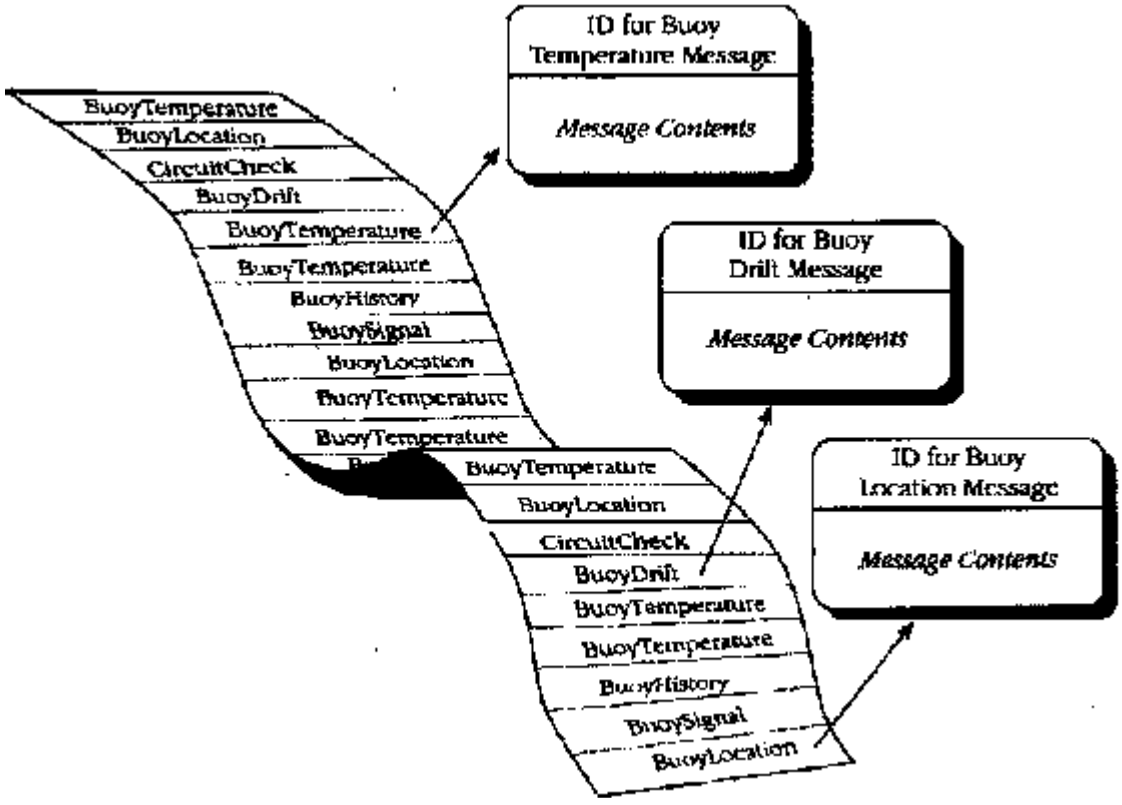


图 12-2　无特定次序存储的信息，其中每个信息有一个信息识别标志

由于信息格式是由用户决定的，因而是反复无常的，你也不能寄希望于用户来把格式稳定下来。图 12-3 表示出了几条详细的信息。

如果你用的是逻辑控制方法，那么你就不得不读取每一条信息，检查它的识别标志,然后调用相应予程序来读取，解释和打印该种信息。如果共有 20 种信息的话,那么你就需要设计

20 种子程序，同时还要有许多低层次子程序来支持它们。比如，你将不得不用 PrintBuoyTemperatureMessage()子程序来专门打印浮标温度信息。

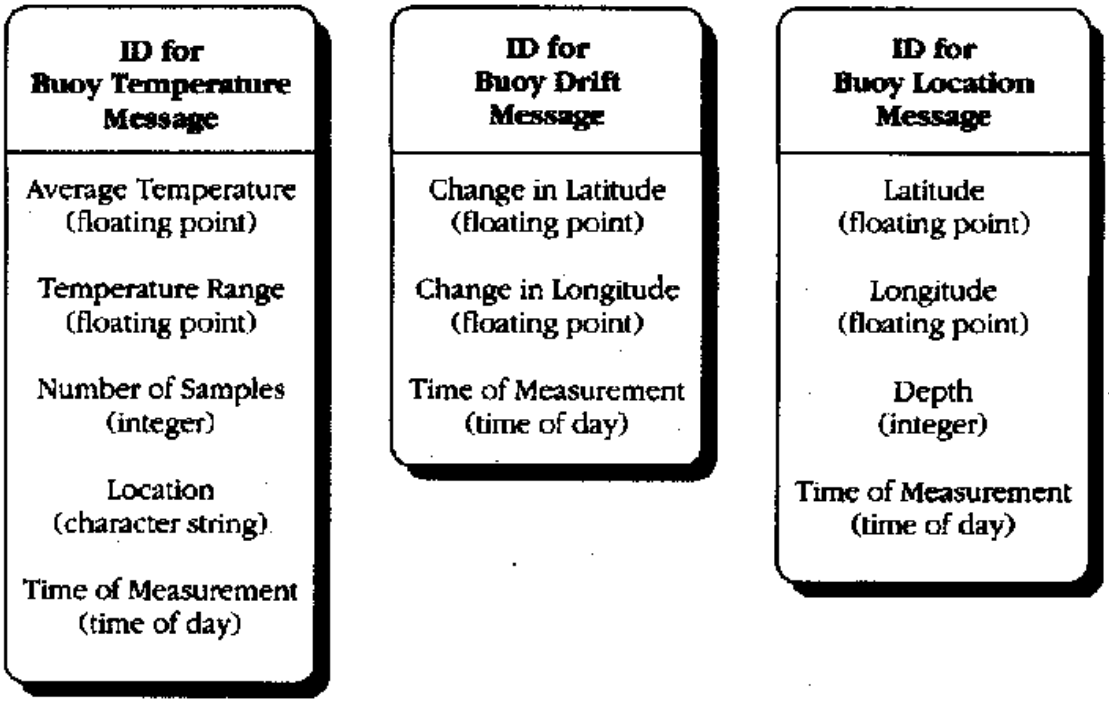


图 12-3　各种信息格式

当任何信息格式变动时,你都不得不改动打印该种信息子程序的内部逻辑。在上图表示的信息细节中，如果平均温度域从浮点型变成其它类型，那么你就不得不改动 PrintBuoyTemperatureMessage()中的逻辑结构。

而如果用表驱动方法的话，就可以把每种信息的格式放在一个表中而不必把它们硬性编码在程序逻辑结构中,这使得编程和修改都变得很容易，而且使用的代码也要少得多。要使用这种方法,你首先必须列出信息的种类和域的类型。在 Pascal 中,可以像下面这样来定义域的类型:

```
var
    FiledTypes=(FloatingPoint，Integer，CharString，
                TimeOfDay，SingleFlag，BitField);
```

这样你可以用曲指可数的几个打印基本数据类型(整型、浮点型、字符串等)的子程序来代替专门打印某种信息的 20 个子程序。你可以在表中描述每种信息的内容(包括每个域的名称)，然后根据表中的描述来解释每一条信息。用 Pascal 编写的描述某种信息表入口可能是这样的:

```
Message[Type1].NumFields        := 5 ;
Message[Type1].MessageName      := 'Buoy Temperature Message' ;
Message[Type1].FieldType[1]     := FloatingPoint;
Message[Type1].FieldLabel[l]    := 'Average Temperature' ;
Message[Type1].FieldType[2]     := FloatingPoint ;
```

```
Message[Type1].FieldLable[2]        := 'Temperature Range' ;

Message[Type1].FieldType[3]         := Integer ;
Message[Type1].FieldLabel[3]        := 'Number of Samples' ;
Message[Type1].FieldType[4]         := CharString ;
Message[Type1].FieldLabel[4]        := 'Location' ;
Message[Type1].FieldType[5]         := TimeOfDay ;
Message[Type1].FieldLabel[5]        := 'Time of Measurement' ;
```

现在，你已经用存储在数据中的信息取代了存储在程序逻辑结构中的信息,而数据要比逻辑结构灵活得多,当一个信息格式变化时，很容易改动相应的数据来适应它。如果不得不在其中加入一种新信息，那你只需在 Messaae 数组中添加一个元素就可以了。

这时读取信息的代码也会变得简单多了。在以逻辑为基础的方法中，信息读取子程序要用一个循环来读取每条信息，再根据信息识别标志判别出它的种类，然后再从 20 个打印子程序中调用相应的子程序来打印它。下面是以逻辑为基础方法的伪代码：

```
While more message to read
    Read a message header
      Decode the message ID from the message header
      If the message header is type 1 then
           Print a type 1 message
      Else if the message header is type 2 then
           Print a type 2 message
           ……
      Else if the message header is type 19 then
           Print a type 19 message
      Else if the message header is type 20 then
           Print a type 20 message
```

上段伪代码事实上是省略的，因为其中只表示了 20 种信息中的几种。在低于这个层次的逻辑中，20 种信息中的每一处都要求专门的子程序来打印它们。这些子程序也可以用伪代码来表示。下面是表示浮标温度打印子程序的伪代码：

```
Print ' Buoy Temperature Message'

Read a floating-point value
Print ' Average Temperature'
Print the floating-point value

Read a floating-point value
Print ' Temperature Range'
Print the floating-point value
```

```
Read an integer value
Print ′ Number of Samples′
Print the integer value

Read a character string
Print ′ Location′
Print the character string

Read a time of day
Print ′ Time of Measurement′
Print the time of day
```

而表驱动方法则要比这简单得多。信息读取子程序首先利用循环来读取每条信息的开头，再利用识别标志判定它的种类,然后在 Message 数组中查寻关于该信息的描述，然后调用同一个（而不是 20 个中的某一个）子程序来解释和打印信息。下面是表驱动方法中的高层次伪代码：

```
While more message to read
Read a message header

Decode the message ID from the message header
Look up the message descript in the message-description table
Read the message fields and print them based on the message description
```

前三行与以逻辑为基础的相同

与以逻辑为基础方法不同的是，上面的伪代码并没有省略，因为现在的逻辑关系是非常简单的。在低于这个层次的逻辑上，你将会发现只用一个子程序就可以同时解释和打印所有信息。这个子程序的通用性要比以逻辑为基础方法中的打印子程序通用性强得多。但却并不比后者复杂多少。而且只用这一个子程序便可代替 20 个子程序，下面是这个子程序的伪代码。

```
While more fields to print
  Get the field type from the message description
  Depending on the type of the field
        case of floating point  =>
              read a floating-point value
              print the field label
              print the floating-point value

        case of integer =>
              read a Integer value
              print the field label
              print the integer value
        case of character string =>
              read a character string
              print the field label
              print the character string
```

```
case of time of day =>
      read a time of day
      print the field label
      print the time of day

case of single flag =>
      read a single flag
      print the field label
      print the single flag

case of bit field =>
      read a Integer value
      print the field label
      print the bit field
```

当然,这个子程序要比前面单个的浮标温度打印子程序长一些。但这却是你在打印时所需要的唯一一个子程序，从而节省了 19 个子程序。这个子程序可以处理六种域类型并可以打印和解释所有种类信息。

不过,这个子程序采用的是表查寻中最复杂的一种方法,因为它采用了 6 个 case 语句。许多语言支持像存储数据一样把对子程序的调用存储在表中。如果你所用的正是这种语言，那么就不必再用 case 语句了。你可以把子程序存储在表中并根据域的类型来调用它们。

下面是一个如何在 Pascal 中建立过程表的例子,首先,你需要建立一个 Pascal 的“过程类型”,即一种可以在其中存放对过程调用的变量。以下是这种类型的一个示例：

```
type
      HandleFieldProc= procedure
     (
          FieldDescription：  String
          var FileStatus:      FileStatusType
      );
```

上面这段代码说明了一个过程类型，这种过程可以带有字符串参数和 FieldType 参数。第二步工作是说明一个存放过程调用的数组。这个数组就是查寻表,下面就是这个查询表的示例:

```
var
   ReadAndPrintFieldByType: array[FieldTypes] of HandleFileProc;
```

最后,要建立一个把某一特定过程的名称赋给 ReadAndPrintFieldByType 数组的表。下面就是用 Pascal 写成的这个表：

```
ReadAndPrintFieldByType[FloatingPoint]:=ReadAndPrintFloatingPoint;
ReadAndPrintFieldByType[Integer]       :=ReadAndPrintInteger;
ReadAndPrintFieldByType[CharString]    :=ReadAndPrintCharString;
ReadAndPrintFleldByType[TimeofDay]     :=ReadAndPirntTimeofDay;
```

```
ReadAndPrintFieldByType[SingleFlag]:=ReadAndPrintSingleFlag；
ReadAndPrintFieldByType[BitField]   :=ReadAndPrintBitField；
```

在上段代码中假定 ReadAndPrintFloatingPoint 和其它等号右边的标识符都是 HandleFieldProc 类型的过程名称。把过程名称赋给由过程变量组成的数组中的元素,意味着你可以通过引用数组元素来调用过程，而不必直接使用过程名称来调用它们。
当过程表建立好之后，你只需访问过程表并调用表中的某个过程便可以处理信息中一个域。具体代码是这样的；

```
MessageIdx  = 1
While(MessageIdx<=NumFieldsInMessage) and( FileStatus=OK) do
    begin
    FieldType:=FieldDescription[MessageIdx].FieldType；
    FieldName:=FieldDescription[MessageIdx].FieldName；
    ReadAndPrintFieldByType［FieldType］(FieldName，FileStatus)
    end；
```

还记得那段使用了 case 语句的有 27 行的伪代码子程序吗？如果你用一个子程序表来代替 case 语句的话，上面便是你为实现同样功能而所需要的全部代码，显然它要比使用 case 语句的子程序短得多。在其它类似支持过程变量的语言中如 C，你也可以使用类似的方法。这种方法很难产生错误，非常容易维护同时效率也非常高。

建立查寻标志。在上面的三个例子中，你都可以用数据作为标识符来直接进入表。比如，你可以用 MessageID 而不必作任何变动就可以把它作为进入表的标志。你可能总想使用直接存取表方法，因为这种方法比较简单，而且速度比较快。可是有时数据却不允许你这样作。比如在计算保险费用的示例 2 中，Age 就不能用来作为直接存取标志，因为对 18 岁以下的人和 65 岁以上的人都各只有一个保险费用。这便产生了如何建立查寻标志的问题。

复制信息以建立直接存取标志。Age 成为直接存取标志的一个简单办法是为 0 到 17 岁和 66 岁以上的每个年龄都规定一个保险费用，当然，这两个年龄段中每个年龄段保险费用都分别是相同的，这相当于把这两个年龄段的保险率进行了复制。这种方法的好处是表结构和表存取方式都是直接的。当要对 0~17 岁中的某一个年龄规定一个特殊的保险费用时，你只要改动一下表就可以了。其缺点是大量的冗余信息会浪费空间并且很容易使表中存在错误，因为表中的数据变多了。

变换数据使它成为直接存取标志符。把 Age 变成直接存取标志的另一个办法是用一个函数作用于 Age。在这种情况下，这个函数将把 0 到 17 岁之间的所有年龄都变成一个标志，而把 65 岁以上的所有年龄变成另外一个标志。用 min()和 max()函数就可以完成这项工作，比如可以用如下的表达式：

max(min(66，Age)，17)

使用转换函数要求，你对将被用作标志的数据模式有清楚的了解，而且这种方法也不总是如上例那样简单用 min()和 max()就可以完成的。假设上例中的保险费用是每 5 岁为一个段而不是以每一岁为一个段的，除非你把所有的保险费用都复制 5 次，否则的话你必须找到一个可以使年龄被 5 整除并且可以调用 min()和 max()的函数才能完成转换，而这却不是件容易的事。

把标志转换独立在函数中。当你使用变换数据的方法来建立直接存取标志时，应把对数据的转换操作放在函数中，函数的使用消除了在不同地方使用不同转换方法的可能性，当需要对变换作改动时也非常容易。同时，要恰当地命名转换函数以清楚地说明使用函数的目的，比如GetkeyFromAge()就是一个很好的名称。

还要把转换函数放入含有表定义和其它与表有关子程序的模块中。当转换函数与数据表距离比较近时，如果要改动数据表的活，也比较容易想起改动转换函数。

12.2.3　变址存取

有时依靠简单的数字变换还不足把数据转换为直接存取标志。这时可以用变址存取方法解决某些问题。

当使用变址时，首先用基本数据在变址表中找到一个标志，然后利用这个标志查找你感兴趣的主要数据。

假设你在考虑一个存货问题，并且其中有一张岂有 100 项的清单，每一项都含有一个在 0000 到 9999 之间的四位数。在这种情况厂，如果你想用四位数作为标志来直接进入一个描述了某一项某些方面的表，就要先建立一个有 10,000 个入口的变址数组。这一个数组中除了那些与上述 100 个 4 位数相应的入口外，其余入口都是空的。如图 12-4 所表示的那样，这些入口指向一个描述了清单中的项，但入口数要远远小于 10,000 的表。

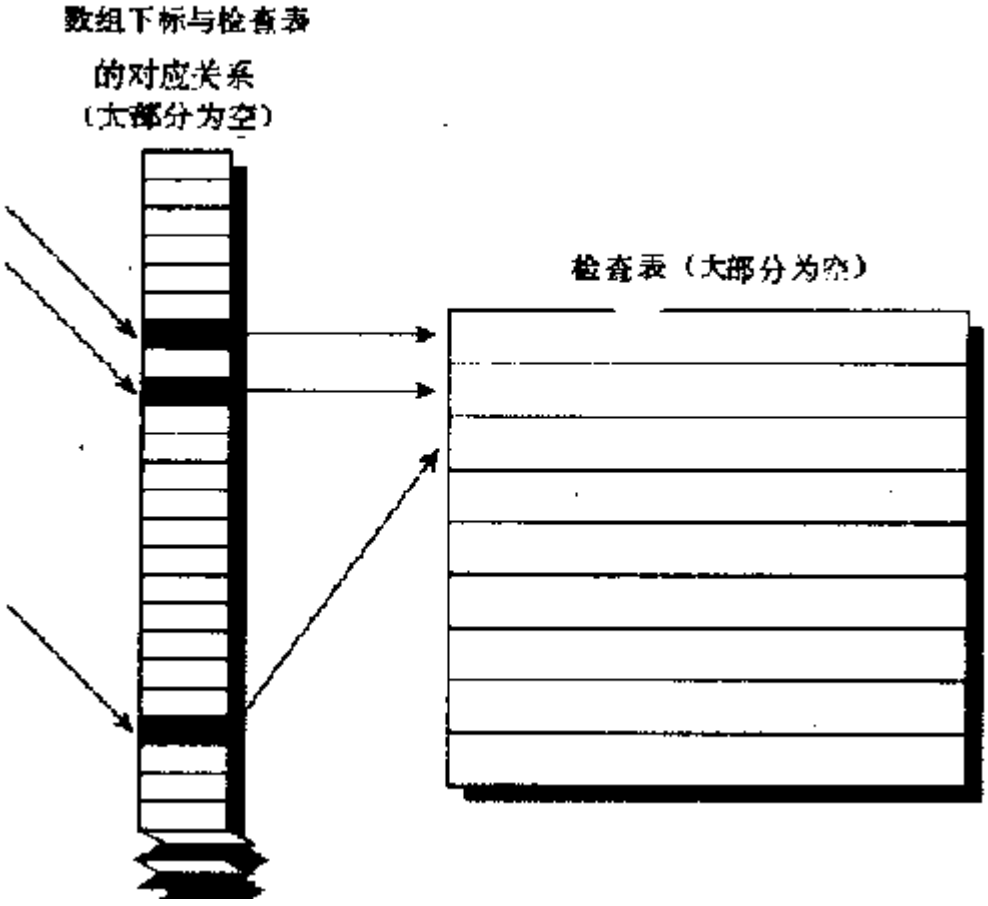


图 12-4　　变址存取

变址存取方法主要有三个优点。首先，如果主查寻表中每一个入口占用的空间都很大的话，那么建立一个有许多空入口的变址表所浪费的空间要比建立有许多空入口的主查寻表浪费的空间少得多。比如上例中如果主查寻表每个入口占用 10 个字书，而变址表每个入口占用 5 个字节的话，就可以节约(10-5)＊ 9900=49500 个字节。

第二个优点是即使使用变址表没有节约空间，对变址表中的入口进行操作也要比对主查

寻表口进行操作容易得多。比如，假设有一个含有雇员姓名、雇佣日期和工资等信息的表，你可以建立一个通过姓名来查寻主表的变址表，还可以建立一个工资查寻主表的变址表，而且你也可以建立通过雇佣日期查寻主表的第三个变址表。

使用变址存取的第三个优点是表驱动方法所共有的良好维护性。存在表中的数据维护起来要比淹没在代码中的数据容易得多。为了增强灵活性，可以把变址存取代码放在子程序中，当你需要由数据得到一个直接存取标志时再调用这个子程序，当需要改动表时，对子程序中的变址存取代码进行修改要比对散布在程序中的同样代码进行修改容易得多。

12.2.4　阶梯存取

另一种表存取方法是阶梯法。这种存取方法不如变址法直接，但要比变址法节约空间。

阶梯法的主要思想是如图 12－5 所示的阶梯结构，其核心是表中的入口对数据范围而不是对不同数据点有效的。例如你正在编写一个评分程序，"B" 入口的取值范围可以是从 77.5%到 90%。下面是一些你可能遇到的评分范围：

≥90%	A
< 90.0%	B
< 77.5%	C
< 65.0%	D
< 50.0%	F

这是一个很难用表来查寻的问题，因为你无法用一个简单的转换函数来得到从 A 到 F 的直接存取标志。这对变址存取法来说也是比较困难的，因为上面的数都是浮点型的。或许你会考虑把浮点型转化为整型，就这个问题而言，这样作是可行的。不过为了说明阶梯法，我们把解决方案限制为只能用浮点型数。

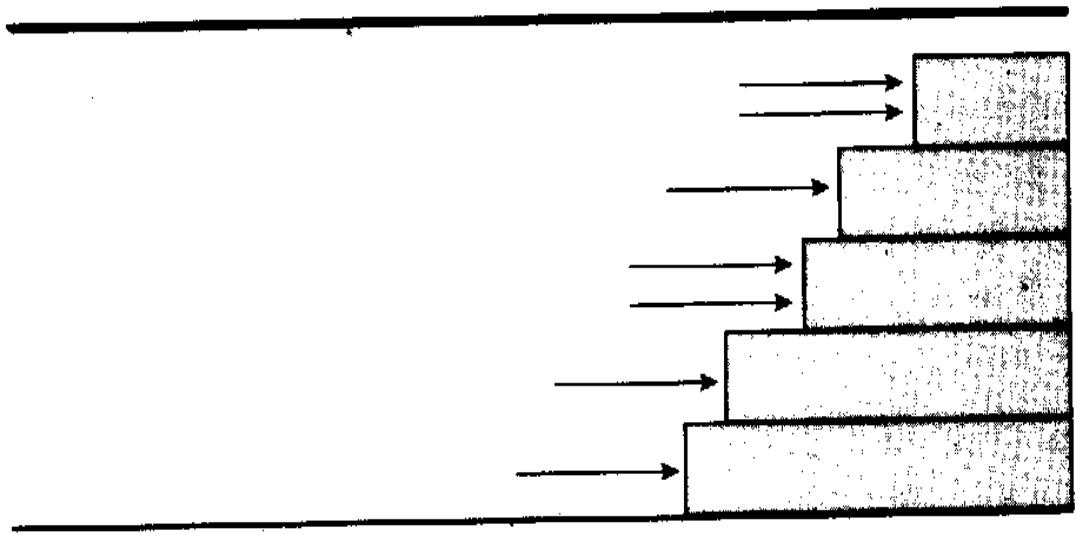


图 12－5　阶梯活确定入口

使用阶梯法时，首先把每个范围的上界放入表中，然后用一个循环来查找超过每一个范围上界的分数。当你找到某一分数第一次超过某一范围上界的地方时，也就知道了该分数应属于哪一级。使用这一技术时，必须仔细而且恰当地处理每一个范围的边界。下面是一段给一组学生的分数分级的 Basic 程序；

```
'set up data for grading table
```

```
        RangelLimit(1) =   50.0
        Grade(1) = 'F'
        RangleLimit(2) = 65.0
        Grage(2) = 'D'
        RansleLimit(3) = 77.5
        Grage(3) = 'C'
        RangleLimit(4)=   90.0
        Grage(4)= 'B'
        RangleLimit(5)= 100.0——这是范围边界
        Grage(5) = 'A'
        MaxGradeLevel = 5
        ...
    'assign a grad to a student based on the student's score
         GradeLevel= 1
        StudentGrade = ”A"
        while( StudentGrade = "A" and GradeLevel < MaxGradeLevel )
          if( StudentScore < RangeLimit( GradeLevel ) ) then
                StudentGrade = Grade ( GradeLevel)
          end if
          GradeLeveli = GradeLevel + 1
    end
```

虽然这是一个非常简单的例子，你可以很容易地把它推广到多个学生、多等级方法(例如,不同水平的分数对应不同的级)中。

阶梯方法优于其它表驱动方法之处，在于它比较善于处理不规则的数据。上面这个评分例子的简单之处在于虽然不同等级之间间隔是不规则的，但是数据已经经过四舍五入了，其结尾部是 0 或 5。阶梯法对于不是以 0 或 5 结尾的数也是同样有效的。你可以用阶梯法来处理统计学中概率分布问题，比如下面的数据：

```
0，458747          $0.00
0 547651           $254.32
0.627764           $514.77
0.778883           $747.82
0.893211           $1,042.65
0.957665           $5,887.55
0.976544           $12,836.98
0.987889           $27,234.12
```

像这样不规则的数据是很难用函数把它们转换为直接存取标志的，但是用阶梯法却很容易解决这个问题。

同时这种方法也具有表驱动方法的共同优点。它也是非常灵活而且容易更改的。如果上例

中的分级范围要进行变动的话，只要改动 RangeLimit 数组中的入口就可以了。你可以很容易地改动程序中等级赋值部分，以使得它可以接受不同级和相应的分数范围的表。在程序中把分数转换成了百分数，事实上你也可以直接使用自然分数而不必将其转换为百分数。

以下是在使用阶梯法时应该注意的几个问题:

注意边界。要保证已经把每一台阶范围的上界值都考虑在内了。使用阶梯搜寻法来找出所有不在最高一级范围内的值，然后把剩下的值全部归入最高一级中。有时候需要人为地为最高一级范围添加一个上界。

同时应小心不要错误地用“<”来代替“<=”。要保证循环在找出属于最高一级范围内的值后恰当地绍束，同时也要保证恰当地处理了范围边界。

应使用交叉搜寻而不是顺序性搜寻。在上面的评分示例中，循环顺序地按照分级界限来进行分级工作。如果表稍微大一些的话，进行顺序搜寻的代价将是非常巨大的。如果出现了这种情况，你可以用“准交叉”搜寻来代替顺序搜寻，之所以称其为准交叉搜寻是因为绝大多数交叉搜寻的目的都是为了寻找某个值。而在这种情况下，你想找到的是值属于的类别而不是值本身。在交叉搜寻算法中，必须正确地决定值的归宿，同时，也要特别注意考虑边界问题。

考虑使用变址存取方法来代替阶梯存取法。有时使用变址存取法来代替阶梯存取法会更好。阶梯法中所需要的搜寻是时间累积的，如果运算速度是你考虑的重点的话，可以考虑用变址法来代替阶梯法，即以牺牲空间为代价来换取速度。

显然，并不是在所有情况下这种替换都是可行的。在评分的例子中或许这是可行的。如果只有一百个待分级的分数，并且这些分数也比较规则的话，建立一个变址表并不会占用太大的内存，因而是可能的。但是如果分数是很不规则的，则无法使用变址法。因为你无法用这种方法进入存有 0.458747 或 0.513392 这类数的入口。

在很多情况下往往几种方法都是可行的。这时设计的要与是从几种方法中选出一种适用的，而不是过多地考虑从中选择出一种最好的。找到一种较好的方法，并避免灾难性的后果，往往比竭力找出最好的方法要实用，也合适得多。

最后一点应注意的是把阶梯表放入它自己的子程序中。

12.2.5　其它的表查寻的例子

在本书的其它章节中出现了一些表查寻的例子。在那些地方使用它们的目的是为了强调其它观点而不是为了强调表查寻本身，下面是它们所在的章节：

- 在保险表中查寻保险费用：见 15.3 节
- 在表查寻中查找的代价：见 28.3 节
- 逻辑变量的值(A 或 B 或 C)：见 29.2 节
- 贷款偿还表中的计算：见 29.4 节

12.3　抽象数据类型(ADTS)

抽象数据类型是由数据及对数据的操作组成的。这些操作既向程序的其余部分描述了数据，也允许程序其余部分改变它所描述的数据。在“抽象数据类型”中的数据指的是广义的数据。抽象数据类型可能是图形窗口及影响该窗口的操作，也可能是一个文件及对文件的操作。

甚至还可能是一个保险费用表及对该表的操作。

抽象数据类型通常是作为模块来实现的。模块和抽象数据类型都是数据以及对数据进行的操作的组合。要想详细了解模块的问题，请参阅第六章。

通常关于编程的课本在讲到抽象数据类型时往往会让人感到味同嚼蜡。他们总是喜欢这样说："你可以把抽象数据类型当作一个有许多操作来定义的数学模型。"接着，便是一些令人厌烦的关于如何编写存取堆栈、队列或表的子程序的内容。这些书总是使人感到自己水远也不可能方便地使用抽象数据类型。

这些关于抽象数据类型的干巴巴的解释根本没有说到点子上。抽象数据类型是应该令人激动的，因为你用它们来解决的是客观世界中的实体问题，而不是计算机中的实体问题，你可以向表格中加入一个空格，可以向窗口类型表中加入一个新的类型，也可以在特快列车上加挂一节车厢而不是向链表结构中加入一个结点。不要低估了应在问题域而不是在计算机域中工作这一准则的威力。这并不是说你不应该用抽象数据类型来实现堆栈、队列和表。你应该这样作。但是抽象数据在客观世界问题中的威力要比在课本中讲的要强大得多。

12.3.1　需要用到抽象数据类型的例子

下面是一些需要用到抽象数据类型的情形。我们将在首先讨论这些具体情况之后再来看一下抽象数据类型（ADT）的具体理论。

假设你正在编写一个用一系列打印页、点阵规模和字体来控制向屏幕输出的程序，程序中的一部分要对字体进行操作。如果你使用抽象数据类型（ADT）的话，那么就需要把一组字体子程序和它们作用的数据——打印页名称、点阵大小和字体属性组成一个集合。这个由字体子程序及其数据组成的集合，就是一个抽象数据类型（ADT）。

如果不用抽象数据类型（ADT）的话，你就不得不用麻烦得多的方法来操作字体。比如，你需要把字体改为 12 点阵的，那么你很可能要用如下的代码：

```
CurrentFont.Size = 12
```

如果要在程序中的几个地方改变点阵规模，那么在程序中就会有许多类似的代码散布其中。

如果你要把字体置为黑体的话，可能要使用如下的语句：

```
CurrentFont.Attribute = currentFont.Attribute or 02h
```

如果你很幸运的话，实际代码行可能会比上面的简单一些，但最好也只能是下面这个样子：

```
CurrentFont.Bold =True
```

采用这种方法来编程的话，那么在程序中就会在许多地方出现非常类似的代码行，而且如果你要设置打印页名称的话，你将使用如下的代码：

```
CurrentFont.Typeface = TIMES_ROMAN
```

12.3.2　使用抽象数据类型的好处

这并不是说上面的编程方法是不好的。但是我们可以用更好的编程方法来代替它，从而获得很多收益，何乐而不为呢？

下面就是使用抽象数据类型的一些优点：

可以隐含实现细节。对字体数据的信息隐蔽意味着当需要变动字体时，你只需在程序中的某一处作出改动而不会影响程序的其它地方。比如你如果不用抽象数据类型的话，那么在需要改动字体时，你就不得不在程序中每一个设置了字体的地方进行修改，而不是只在一处进行修改。如果进行了信息隐蔽，那么即使你想把数据从存在内存改为存放在外部文件中，或是想用其它语言重写全部字体子程序，也不会对程序其余部分产生什么影响。

把改动的影响限制在局部。如果你需要更加丰富的字体，并且要支持更多的操作时，只需在一处作出改动就可以了，程序的其余部分将不受影响。

更容易改进性能。如果你需要改进字体性能的话，只要修改几个已经明确定义的子程序就可以了，而不必在整个程序中到处进行修改。

减少修改时犯错误的可能性。通过使用抽象数据类型，你可以用对子程序 SetCurrentFont ToBold()简单修改，来代替对 CurrentFont.Attribute= CurrentFont.Attribute or 02h 的语句的复杂修改，从而减少了犯错误的地方有:使用了错误的结构名称、使用了错误的域名称或者使用了错误的逻辑操作(比如误用了“and ”来代替“or”)，还有可能使用了错误的属性值(误用了 20h 而不是 02h)；而在修改了程序时，你可能犯的唯一错误便是调用了错误的子程序,而这个错误又是很容易发现的。

使程序成为自说明的。在上面的设置字体的语句中，你用 SetCurrentFontToBold()干程序来代替原来语句所获得的可读性是不可同日而语的。

WoodField 和 Dumsmore 和 Shen 曾于 1981 年进行过如下实验：他们用两个内容相同的程序----一个被分成 8 个按功能划分的子程序，而另一个被分成 8 个抽象数据类型子程序，对计算机专业研究生和高年级本科生进行测试，结果接受抽象数据类型程序测试学生对程序的理解程度要比另一组高出 30％。

避免了在程序中四处传递数据的麻烦。上前面的设置字体的例子中，如果不使用数据的话，那么你将不得不直接改变 CurrentFont 的值，或者将其在每一个处理它的子程序中传递。而通过使用抽象数据类型，你既不必把 CurrentFont 在程序中四处传递也不必使用全局数据。在抽象数据类型中将有一个含有 CurrentFont 数据的结构，对这个结构只有在 ADT 内部的子程序才能进行直接存取，而 ADT 之外的子程序则无需关心这个结构。

你可以直接处理客观世界中的实体而不是计算机专业的结构。通过使用抽象数据类型，你可以对处理字体的操作进行定义，从而可以使程序的其余部分是以字体的名义来进行操作,而不是以数组存取、记录定义或者逻辑变量的名义来进行操作。

在这种情况下，为了定义抽象数据类型，你将首先定义几个控制字体的子程序，很可能是如下几个子程序：

```
SetCurrentFonSize(size)
SetCurrentFontToBold()
SetCurrentFontToItalic()
SetCurrentFontToRegular()
SetCurrentFontToTypeFace(FaceName)
```

在这些子程序内部的代码，可能是非常短的，而且很可能与你所见到的不使用抽象数据类型时的代码差不多。但它们的区别在于现在你可以把对字体的操作独立在一组子程序中，从而为程序其余与字体有关的部分提供了较高的抽象程度，而且也避免了在改动字体操作时影响

程序的其余部分。把对数据的操作隐含在存取子程序后面相当于在宇宙飞船中加装的减压舱，宇航员可以通过减压能进入太空，也可以再从太空通过减压舱回飞船。但飞船内的空气却不会泄漏进太空。

12.3.3　关于抽象数据类型的其它例子

下面是关于抽象数据类型更多的例子：

假设你正在编写一个控制核反应堆冷却系统的软件，那么你可以通过定义下述操作而把冷却系统处理成抽象数据类型：

GetReactoTemperature()
SetCirculationRate(Rate)
OpenValue(ValueNumber)
CloseValue(ValueNumber)

特定的环境将决定实现这些操作的具体代码，程序的其余部分将通过这些函数来处理这个冷却系统，而不必担心数据结构实现细节、数据结构限制、改动等问题。

下面是关于抽象数据类型及对它的可能操作的例子：

表：
初始化表
在表中加入项
从表中去掉项
从表中读下一项

灯：
开灯
关灯

文件：
打开文件
读取文件
写文件
设置当前位置
关闭文件

堆栈：
初始化堆栈
把项压入堆栈
从堆钱中弹出项
读取栈顶

通过研究上述这些例子，我们可以得出几条准则：

把典型的计算机专业数据结构建为抽象数据类型。绝大多数关于抽象数据类型的讨论都集中在把典型的计算机专业数据结构建为抽象数据类型。正如你从上面的例子中发现的，你可以用抽象数据类型来代表堆栈、表等数据结构。

你需要提出疑问的是这些堆栈、表所代表的到底是什么？比如，如果堆栈所代表的是一组雇员，那么应把抽象数据类型当作雇员而不是堆栈来处理。又比如假设表所代表的是一组帐单，那么应把 ADT 作为帐单而不是作为表来对待。总之，应从尽可能高的抽象水平上来处理问题。

把常见的目标如文件等处理为抽象时据类型(ADT)。绝大多数语言中都含有一些你可能非常熟悉的抽象数据类型，只不过你没有意识到罢了。文件操作就是一个很好的例子。当向磁盘写文件时，操作系统会把读/写磁头放置在一个特定的物理地址上，而在你放弃一个旧文件时，又会分配一个新的磁盘扇区并查找交叉代码。操作系统提供了一个低层次的抽象，并提供了这个层次的抽象数据类型，高级语言将可能提供稍高一些的抽象出乎并可以提供更高层次

的抽象数据类型。高级语言可以使你避免与复杂而繁琐的操作系统调用和数据缓冲区操作细节打交道。这种能力可以使你把一段磁盘空间当作“文件”来处理。

与此类似，你也可能对抽象数据类型进行分层。如果你在数据结构操作层次(如压入或弹出堆钱)上使用了抽象数据类型(ADT)，那很好。接着，你可以在比它更高的层次上使用处理客观世界真实问题的 ADT。

即使是简单的问题也应考虑使用抽象数据类型(ADT)。不要在碰到复杂得可怕的数据结构时才考虑使用抽象数据类型。在使用 ADT 的示例表中有一个非常简单的例子——只有开灯和关灯两个操作的灯控制问题。你可能会认为“开”和“关”两个操作过于简单了，不值得为它们分别编写专门的子程序，但事实上，即使是简单的问题也可以因为使用 ADT 而受益。把对灯的操作放入 ADT 中，可以提高代码的自我说明程度并区改善其易改动性，并把改动的影响限制在 Turn_Light_On()和 Turn_Light_Off()两个子程序中，同时也降低了数据传递的工作量。

可以提供一对互补的操作。绝大多数操作都需要有与其相对应的、效果相反的操作。如果有一个打开灯的操作，那么就需要一个关灯的操作。如果有向表中添加项的操作，那么就要有从表中去除项的操作。因此，当你设计抽象数据类型时，应检查每一个功能以确定是否有与其互补的功能。不要根据直觉，而是根据需要来决定是否设计互补功能。

应相对 ADT 所存储的介质独立地引用它。假设有一张非常大的保险费用表，你不得不把它存储在磁盘中，你可能会把它作为一个“保险费用文件”，并且用诸如 ReadRateFile()之类的存取子程序来访问它。但是，如果把它作为文件来处理的话，你就暴露了超过实际需要的信息。如果你对程序作出改动，把保险费用表由存放在磁盘中改为存放在内存中，那么把它作为文件来引用的代码就会变得不正确而且是令人困惑的了。应努力使存取于程序的名字与数据存取方式是相互独立的，而且应该引用抽象数据类型如保险费用表。你应该用 ReadRateTable()来代替 ReadRateFile()作为存取子程序的名称。

12.3.4　用 ADT 来处理多事件数据

如果你正在为 ADT 定义基本操作，可能会发现需要处理一种类型的几条而不是一条数据。例如，在处理字体时，你可能需要跟踪许多字体。用 ADT 的术语来说，你所需要跟踪的每一种字体都是一个“事件”。一种办法是建立几个分立的 ADT 来处理你要跟踪的每一个字体事件。但更好的办法是用一个 ADT 来处理这个多事件字体数据。这通常意味着设计建立和取消事件及其它功能，以便用它们来处理多事件数据。

字体 ADT 原来可能会包括如下功能：

```
SetCurrentFontsize(size)
SetCurrentFontToBold()
SetCurrentFontToItalic()
SetCurrentFontToRegular()
SetCurrentFontToTypeFace(Face_Name)
```

如果你想一次不止处理一个事件，那么，需要加入如下的建立和取消字体事件的几个功能：

```
CreateFont(FontID)
DeleteFont(FontID)
```

SetCurrentFont(FontID)

在其中引入了 FontID 这一符号，用它来作为建立和取消字体事件时的跟踪标志。至于其它操作，你可以从处理 ADT 接口的三种方法中任选一种：

隐蔽地使用字体事件。设计一个新的函数来调用某一设置当前特定字体事件的函数——比如 SetCurrentFont(FontID)。设置当前字体使其它函数在被调用时可能使用当前字体。当你用这种方法时，可以不必把 FontID 作为其它函数的参数。

每次使用 ADT 函数时，显示识别事件。在某种情况下，你没有用来表示“当前字体”的符号。使用 ADT 函数的子程序不必跟踪字体数据。但它们必须使用字体识别标志。这需要在每个子程序中都加入 FontID 来作为参数。

显示提供 ADT 函数要用到的数据。用这种方法时，需要在每一个用到 ADT 函数的子程序内部说明 ADT 所用数据。换句话说，你要建立一个传递到每一个 ADT 功能子程序中的 Font 数据结构。同时，要把 ADT 功能子程序设计成每当它们被调用时，它们就会使用传入其中的 Font 数据。这时，你不在需要使用字体识别标志，因为你跟踪的就是字体数据本身(虽然这种数据可以直接从 Font 数据结构中得到，但你还是应该通过 ADT 功能子程序来存取它，这种方法称为把记录保持为“封闭”的。将在后面详细讨论)。

这种方法的优点是 ADT 功能子程序不必根据字体识别标志来寻找字体信息。其缺点是这种方法会使程序其余部分面临危险。因为数据是可以被直接存取的，所以它很容易被错误改动。

在抽象数据类型(ADT)内部，有许多种处理多事件数据的方法可供选择，但在 ADT 外部则只有上述三种方法可供选择。

12.3.5　混合抽象级别(要避免！)

如果你只在 ADT 功能子程序内部才对数据结构进行存取，而且在程序其余各部分只通过 ADT 功能子程序来存取数据，那么你就保持了一致的抽象级别。如果不这样作，那么就产生了混合抽象级别问题，这足以抵消由使用 ADT 而带来的所有好处。其直接后果是在修改程序时非常容易犯错误。因为这时你会错误地以为修改了存取子程序便等于修改了程序中所有存取方式。

我们继续使用前面的飞船减压舱隐喻来说明问题：如果说 ADT 中的功能子程序相当于飞船上减压船的话，那么对功能子程序的不一致使用就相当于在减压舱门上钻了个孔。这个孔可能不至于使飞船中的空气一下了漏光，但是只要时间足够，它便足以毁掉整个飞船。在程序中如果你使用了混合抽象级别，那么当你对程序进行修改时，程序便会变得越来越令人难以理解，其功能会逐渐退化直至最后它成为完全无法维护的。

开放和封闭的记录

随着混合抽象级别而来的是“封闭”和“开放”记录的思想。当一个子程序直接用到了在其中的记录或结构的任一个域时，便称这个域或结构是开放的：而如果记录或结构的任一个域都没有被直接使用时，就称它是“封闭”的。向前面所提到的那样，你可以把使用封闭记录作为处理多事件数据的一种方法。一个数据是可直接存取的这一事实，并不意味着你必须开放它。除非是受到 ADT 功能了程序的限制，否则就该把记录保持为封闭的。

当你在用处理多事件数据的三种方法进行选择时，应选择 FontID 法而不是 FontRecord

法。

12.3.6　抽象数据类型与信息隐蔽、模块和目标

抽象数据类型和信息隐蔽是互相联系的概念。当你使用 ADT 时，就已经隐蔽了它的实现细节。这也正是从抽象数据类型通往信息隐蔽之路。当你使用信息隐蔽时，首先要寻找可以隐蔽的内容。最明显的需要隐蔽的内容就是抽象数据类型(ADT)的实现细节。这也是从信息隐蔽通往抽象数据类型的道路。

抽象数据类型这一概念与模块的思想也是相互联系的。在直接支持模块语言中，你可以在模块中实现抽象数据类型。模块与抽象数据类型一样，都是由数据以及作用在数据上的操作组成的。

抽象数据类型这一概念与目标的概念也是相互联系的。“目标”是一个比较广义的概念，但它通常指的是数据及使用在数据上的操作。从这个观点来说，所有的目标都是抽象数据类型。“目标”的思想事实上也利用了继承性和多形性的概念。把目标当作抽象数据类型的想法事实上便是把继承性和多形性加到了一起。抽象数据类型只是目标思想的一部分而不是全部。

12.3.7　抽象数据类型所需要的语言支持

有些语言对 ADT 的支持要比其它语言更强有力。Ada 语言对抽象数据类型的支持近乎完美的。比如在字体的例子中，Ada 允许把所有的字体功能子程序都放入一个“包”中。你可以把几个子程序声明为开放的，而其条子程序则只在包中才是可用的。你可以限制 Font 的定义，从而禁止 ADT 功能的子程序对 Font 的内部进行操作。这些子程序可以声明 Font 数据。但是它们不能对任一属于 Font 的域进行操作，也不知道 Font 的内部结构。

在其它语言，如 C、Fortran、Basic 和 Pascal 中，你可以把 ADT 放入单一的一个源文件中，开放某些子程序而把其余子程序保持为专用的 C 对 ADT 的支持也是比较有力的，因为它支持多重源文件(模块)和专用于程序。而其它语言对 ADT 的支持能力则取决于特定的实现细节。换句话说，在其它语言中，含有 ADT 的程序不一定是可移植的。

在任一种给定的语言中，无论你怎样有效地隐含子程序、隐含数据都是一个棘手的问题。仍以字体程序为例：如果你在功能子程序外说明了一个 Font 变量，就可以在任意能够引用这个变量的地方对其内部进行操作。这时它总是一个开放的记录，这意味着它很可能成为一个错误的记录。只有通过规定，编程标准和你痛苦的失败经历才能使其保持为封闭的。

12.4　小　结

- 恰当地对数据进行结构化，可以使程序更简单、更容易理解也更容易维护。
- 可以用表来代替复杂的逻辑结构。当你被程序的复杂逻辑迷惑时，应考虑是否可用查寻表来简化程序。
- 抽象数据类型是降低复杂性的有力武器。它使你可以分层编写程序，而且是从问题域而不是程序语言细节来编写顶层的程序。

第十三章　顺序程序语句

目录

相关章节

常见的几个控制方面问题：见第 17 章
有条件的代码：见第 14 章
循环代码：见第 15 章

从这章起，论点已由程序的数据衷心论转移到控制中心论方面。这章介绍一种最简单的控制流－－按顺序放置程序语句和程序块。

虽然组织顺序式程序代码（straight line Code）是件相对比较简单的事情，但是组织形式的技巧却影响到代码的质量、正确性、可读性与可维护性。

13.1　必须有明确顺序的程序语句

必须按先后顺序往下执行的语句，是最简单的一类有顺序关系的语句，举例如下：

按顺序执行的 Pascal 程序例子：

```
    RecordData ( Data );
CalculateResultFromData( Data, Results );
PrintResults (Results);
```

除非代码段中发生了什么不可思议的事情，否则程序将按语句的先后顺序往下执行，即要得到计算结果必须先读取数据，而只是计算出结果后才能有结果输出。

这个例子中默认的一个概念是语句间的依赖性。第三个程序语句依赖于第二个，而第二个则依赖于第一个。这个例子中一个语句对另一个的依赖关系，由子程序名就看得很清楚。在下面的代码段中，这种依赖关系就不是很明确。

虽不明显但仍有依赖关系的 C 程序：

```
ComputeMonthlyRevenues( Revenues);
ComputeQuarterlyRevenues( Revenues);
ComputeAnnualRevenues( Revenues);
```

这个例子中，计算季度收入时是假设月度收入已经算出了的。如果对会计学很了解或有一些常识，我们应当知道在算出季度收入之后，才能算出年度收入。这里存在着依赖关系，但仅从代码本身却不很明显。下面这个代码中，程序语句间的依赖关系更不明显,这种依赖关系从字

面上是看不出来的。

依赖关系隐藏的 Basic 程序：

```
CALL ComputeMarketingExpenses
CALL ComputeMISExpenses
CALL ComputeAccountingExpenses
CALL PrintExpenseSummary
```

由于各子程序的调用没有参数，你或许能猜出各子程序是通过模块数据或全局变量等方式获得数据的。假设是在 ComputeMarketingExpenses()于程序中来初始化全局变量，其它子程序也就得到了各自需要的数据。在这种情况下，这个子程序必须在其它各子程序前被调用才行，但在这个代码中你无法从通读程序得知。

当程序语句间有依赖关系时，你需要把它们按一定的先后顺序组织起来而使这种依赖性更明显。下面提供一些这方面的指导。

组织代码使它们间的依赖关系明显。在上面的 Basic 程序中，ComputeMarketingExpense()没有必要初始化全局变量。这个子程序的名字表明它计算的是市场数据，而不是 Mix 或 Mis 会计数据。在 ComputeMarketingExpense()中初始化全局变量是个不太好的习惯，应当改掉。为何我们要在这个子程序中初始化全局变量而不是在其它两个子程序中呢？除非你有什么正当理由，否则你必须另写一个子程序——InitializeExpense()去初始比全局变量。这个子程序的名字已经很清楚地表明它应当在其它子程序前被调用。

子程序的名字应当清轻地表明依赖关系。在上面例子中，ComputeMarketingExpense()的名字起得不好，因为它仅说明它是计算市场费用的，但同时它却初始化了全局变量。假如你不乐意另写一个初始比数据的子程序，那么你至少应该给 ComputeMarketingExPense()起一个能描述其作用名字。本例中，起个 ComputeMarketingExpenseAndInitGlobelData()名字是较为准确的。　你或许要说太可怕了，名字那么长，但这个名字却实实在在地反映了其作用，因而显得不可怕。原来的子程序名名不符实才害人呢？

使用子程序参数使依赖关系明显。在上面这个例子中，因为子程序间无参数传递，你不知道其它各子程序是否用了同一个数据。重写上代码段，使子程序间有参数传递，这样在代码中就留下一些表明执行先后顺序的线索，下面是重写的代码段。

用数据传递来显示程序语句间依赖关系的 Basic 例子：

```
CALL InitializeExpenseData(ExpenseData)
CALL ComputeMarketingExpenses(ExpenseData)
CALL ComputeMISExpense(ExpenseData)
CALL ComputeAccounting(ExpenseData)
CALL PrintExpenseSummary(ExpenseData)
```

因为所有子程序都用了 ExpenseData 这个参数，你可能得到暗示，即所有子程序都用了同一数据，因而语句的先后顺序是重要的。但反过来想也不算错，既然有数据传递，那么是否该说明执行顺序并不重要呢？下面是一个例子。

有数据传递并不表明语句间依赖关系的 Basic 例子：

```
CALL ComputeMarketingExpense(MarketingData)
```

```
CALL ComputeMISExpenses(MISData)
CALL ComputeAccountingExpenses(AccountingData)
CALL PrintExpenseSummary(MarketingData,MISData,AccouningData)
```

既然前三个程序语句中没有用到任何相同的参数，这个代码表明调用各于程序的次序并不重要，因为第四个子程序用到了前三个子程序的数据，你可能想到这个子程序必须在其它三个子程序后执行。

注明不明确的依赖关系。首先写一个没有次序依赖关系的代码，然后再写一个有明显依赖关系的代码。如果你还觉得依赖关系不够明显，那么就用注释注明这种依赖关系。注明不明确的依赖关系，是说明你代码设想的一个方面，而这对编写出易维护、易修改的代码至关重要。

在 Basic 例子中，写出这些行的注释是很有帮助的。下面程序语句的次序依赖关系很隐藏，但用注释来注明的 Basic 程序：

```
' Compute expense data．each of the routines acceses the
' global data structure ExpenseData．ComputeMarketingExpenses
' should be called first because it initlializes Expenses
' PrintExpenseSummary should be called last because it uses
' data calculatal by the other routines．
CALL ComputeMarketingExpenses
CALL ComputeMISExpenses
CALL ComputeAccountingExpenees
CALk PrintExpenseSummary
```

这个例子中，代码没有用技巧来使语句的依赖关系变得明显。最好还是用那些技巧来表明这种关系而不要用注释。但若你想让别人无论何时都能很明白你的程序，或因为某些原因你不能通过改进表明这种关系，那就只好用注释了。

13.2　与顺序无关的程序语句

可能有这种情形，即代码中某些语句或程序块的先后顺序并不重要，一个语句并不依赖于或说逻辑上从属于另一些语句，但实实在在的情况是，次序是影响可读性、性能、维护性的。而且当语句执行顺序的依赖关系不存在时，可用下面的准则来组织这些语句或程序块的顺序。指导原则是“接近原则”，使相关操作组织在一起。

13.2.1　使代码能由上读到下

作为一种基本原则，应使程序能由上读到下，而不要到处转移。专家们认为"从上到下"的次序对可读性最有帮助。简单地使控制流在运行时从上到下是不够的。如果为获得某一所需的信息而必须去通读你的全部代码，那你就该重新去组织一下你的代码。下面举例说明：

```
InitMarketingData(MarketingData)；
InitMISData(MISData)；
InitAccountingData(AccountingData);
```

```
ComputeQuarterlyAccountingData(AccountingData);
ComputeQuarterlyMISData(MISData);
ComputeQuarterlyMarketingData(MarketingData);

ComputeAnnualMISData(MISData);
ComputeAnnualMarketingData(MarketingData);
ComputeAnnua1AccountingData(AccountingData);

PrintMISData(MISData);
PrintAccountingData(AccountingData);
PrintMarketingData(MarketingData);
```

假如你想知道 MarketingData 是怎样算出来的，你必须从最后一行开始，上溯搜查所有有关 MarketingData 的语句直到第一行，其实 MarketingData 仅在另外两行出现过，但你却不能放过每一个语句而得从头看到尾。换句话说，你得看完整个代码才能知道 MarketingData 中如何算出的。下面是修改后的程序：

```
InitMarketingData(MarketingData);
ComPuteQuarterlyMarketingData(MarketingData);
ComputeAnnualMarketingData(MarketingData);
PrintMarketingData(MarketingData);

InitMISData(MISData);
ComputeQuarterlyMISData(MISData);
ComputeAnnualMISData(MISData);
PrintMISData(MISData);

InitAccountingData(AccountingData);
ComputeQuarterlyAccountingData(AccountingData);
ComputeAnnualAccountingData(AccountingData);
PrintAccountinData(AccountingData);
```

这个代码比上一个在几个方面要好。要查的每个变量都局部化了，用到同一变量的语句都集中到一起。变量赋值后紧接着就用到这些数据。代码段中每个变量集中出现在一处。或许更重要的一点是代码可能被分割成市场、MIS、会计数据等几个子程序，但前一个代码能看出这种可能性吗？

13.2.2　使同一变量局部化

有不同参数出现的代码段是一个“脆弱的窗口”。在这个窗口内，很有可能无意中加上了新的代码行，无意中改变了变量。或读着读着忘了这个变量的值是多少，因此把一个变量出现的语句集中到一起总是一个好主意，例如上面这个例子。

把变量的再现局部化到一起的观点，其优点是不言自明的，但它却受制于常规的评价方法。一个评价变量局部化方法是计算变量的跨度(span)。例子如下。

用来计算变量跨度的 Pascal 程序示例：

```
  a:=0;
  b:=0;
  c:=0;
```

```
a:=b+c;
```

在这个例子中，第一次出现 a 和第二次出现 a 的语句间隔了两行，因此 a 的跨度为 2。两次出现 b 的语句之间隔了一行，因此 b 的跨度为 1。而 C 的跨度则为 0，下面是例子。

跨度为 1 和 0 的 Pascal 程序例子：

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

在这个例子中，第一次和第二次出现 b 的语句间隔了一条语句，因此 b 的跨度为 1；而第二次与第三次之间无插入行，因此此时跨度为 0。

平均跨度取各个跨度的平均值，上例中以 b 为例，平均跨度为(1＋0)/2=0.5。当你坚持把同一变量出现的语句集中一起时，你能使读者把注意力一直都集中在一起。如果同一变量出现得很分散，那么读者就只能在你的程序中满天飞了，因此把同一变量局部化的主要优点在于提高程序的可读性。

13.2.3　使变量存活时间尽可能短

与变量跨度相关的另一概念是变量“存活时间”,即变量存活期涉及到的程序语句的总行数。一个变量的生命由第一次出现它的程序语句开始，而终止于最后一次出现它的语句。

不像变量跨度，存活时间不受存活期间变量被用过多少次的影响，假如变量第一次出现在第一行而最后一次出现在第 25 行，那么存活时间是 25 行，假如在这 25 行中这个变量仅被用了两次报(即只在第 1,25 行)，那么跨度是 23 条语句，假如变量从第一行到 25 行中行行都出现，那么平均跨度为 0,但存活时间依然为 25 条语句。下面的图显示了跨度和存活时间的具体含义：

“长的存活时间”意味着变量存活期间有许多条语句,“短的存活时间”意味着变量存活时只出现过几条语句；跨度则指变量出现的密度情况。

跟变量跨度一样，在考虑存活时间时要使它尽可能地小，使存活期间出现的语句条数最小。同样地，减小存活时间也降低了变量出现窗口的脆弱性。当你想改变变量时，若存活时间短，那么最早和最晚出现该变量的两条语句间的语句条数就小，因而减小了有意无意的修改出错的可能性。

减小存活时间的另一大好处是在编码时你能在头脑中有一幅很清晰的画面。假如一个变量在第 10 行赋值而在第 45 行才被用到，你可能认为在这期间也会用到这个变量，老想着这种可能性，头脑当然很乱了。假如在第 44 行给一个变量赋值而在第 45 行马上就用到，那么之间不可能有别的语句，你只需在这小的范围内考虑这个变量就行了。

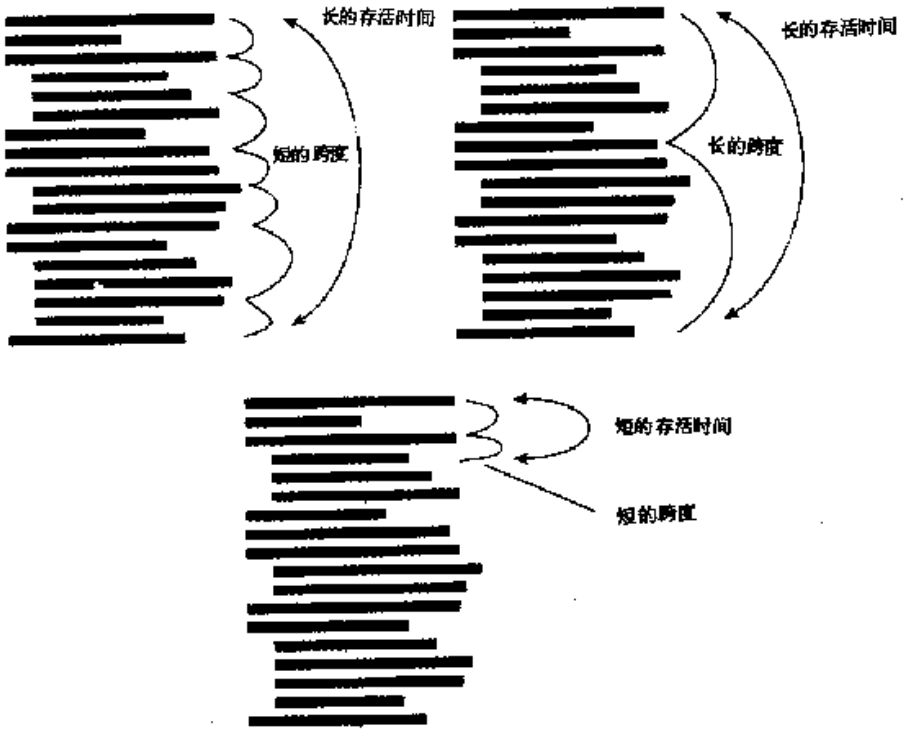


图 13-1　存活时间与跨度图

小的存活时间能减小初始化的错误，顺序式程序代码往往倾向于采用循环。若你把初始化数据的地方放到远离你循环的地方，当你修改循环时，你很可能忘了去修改初始化值。把初始化代码与循环代码紧放在一起，你就可能减小修改程序带来的初始化错误。

最后，小的存活时间使你编的代码可读性好，读者脑中装的代码愈少，你的代码便愈是易懂。同样地，存活时间愈小，当你编辑或调用代码时变量出现的范围也愈小。

计算变量的存活时间

我们规定计算变量的存活时间是这样的：第一次和最后一次出现该变量的语句间总共有的语句行数（包括两个端点）。下面是例子：

这个 Pascal 程序不好，变量存活时间长。

```
1     {Initialize all variables }
2
3     RecordIndex := 0
4     Total             := 0
5     Done              := False
  ....
27    while (RecordIndex<RecordCount) do
28    begin
```

```
29   RecordIndex := RecordIndex+1;        ——RecordIndex 的最后引用
  ....
65   while not Done
       begin
  ....
70   if (Total > ProjectedTotal)          ——Total 的最后引用
71   Done := True;                        ——Done 的最后引用
```

这里几个变量存活时间分别为：

RecordIndex　(第 29 行-第 3 行+1)=27
Total　(第 70 行-第 4 行+4)=67
Done　(第 71 行-第 5 行+1)=67
平均存活时间　(27＋67＋67)/3=54

这个平均存活时间显得太长。

把上例重新写一下使出现变量的语句比较靠拢。

这个 Pascal 程序较好，因为变量存活时间短。

```
...
26      RecordIndex:=0     ——RecordIndex 的初始化从第 3 行向下移
27      while (RecordIndex< RecordCount) do
28    begin
29      RecodIndex:=RecordIndex+1;
63      Total:=0      ——Total 和 Done 的初始化从第 4、5 行向下移
64      Done:=False
65      while not Done
          begin
70      if (Total>ProjectedTotal)
71      Done:=True;
```

在这个例子中，各变量存活时间计算如下：

Recordlndex　(第 29 行-第 26 行+1)=4
Totel　(第 70 行-第 63+1)=8
Done　(第 71 行-第 64 行+1)=8
平均存活时间　(4＋8＋8)/3=7

我们能靠直观觉得第二个例子比第一个例子好，因为初始化的语句和用到该变量的语句较靠近。计算出两个例子的平均存活时间很有意义：54 行的平均存活时间与 7 行的平均存活时间相比，正好从数据上说明了为什么我们的直观是正确的。

本章前面有几个例子,有一个含 InitMarketingData()的子程序经过重新合理编写后，使出

现同一变量的语句较为靠近，因而同时减小了平均跨度和平均存活时间，程序得以改进，读者可自行计算一下改进的程序。

那么仅从数字上能说明短的存活时间的代码就比长的存活时间的代码好吗？同样地，小跨度的代码就一定比长跨度的代码好吗？研究人员还没有一个定量的答案。但有一点是肯定的，即减小变量跨度和存活时间对设计程序来说是一个好的思想。

假如用跨度和存活时间来衡量全局变量你会发现，全局变量的跨度和存活时间都相当之大，这是为何要避免用全局变量的重要原因之一。

13.2.4　相关语句组织在一起

把相关的语句放在一起，这些语句之所以能放在一起是因为它们用一个数据，所起作用相同、或语句的执行是有先后之分的，下一句的执行依赖于上一句的结果。

检查是否把相关语句很好地组织在一起的一个简单办法是，把代码的子程序列出来并把相关的语句用框线框框起来，如果组织得很好，那么就应当如图 13-2 所示那样，各框之间无交叉。

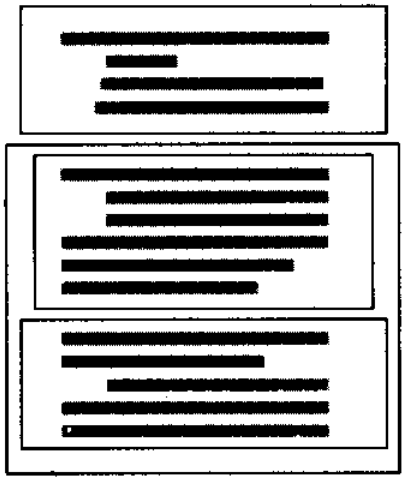


图 13－2 如果代码组织得好．画出相关语句的框与框之间无交叉部分，这些框是一个一个叠起来的

如果组织得不好，很有可能是使用 goto 引起的，那么画出图就如图 13－3 所示，框与框之间有交叉，这时应重新编写和组织代码，以便把有关的语句更好地组织在一起。

一旦你组织好了相关语句，你会发现他们内部之间联系非常紧密，但这一组相关语句与其前后的语句却无太大联系。这时你可以把这组紧密联系的程序语句写成一个子程序。

13.2.5　检查表

组织顺序式程序代码

- 把语句间的依赖关系表示得很清楚吗？
- 子程序名是否把依赖关系表示得很清楚？
- 子程序的参数是否把依赖关系表示得很清楚？
- 若代码的依赖关系不清楚，用注释注明了吗？
- 代码能从上读到下吗？
- 变量的出现靠得很近吗？ ——从跨度和存活时间来考虑。

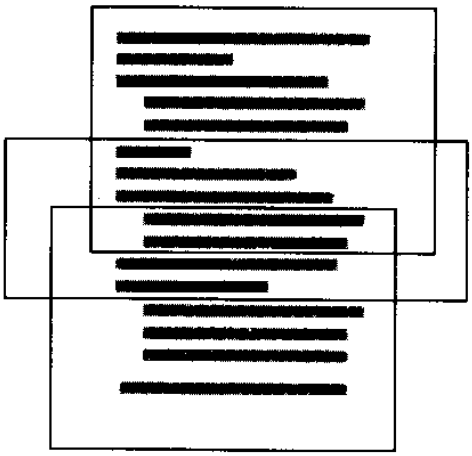


图 13－3 如果代码组织得不好，画出的相关语句的框与框之间有交叉，这种情形很可能是用 goto 引起的

- 是否把相关语句组织在一起？
- 是否把相对独立的各相关语句写成子程序了？

13.3 小　结

- 组织顺序式代码最好的原则是整理出依赖关系。
- 用合适的子程序名、参数表、注释来标明依赖关系。
- 如果代码没有明显依赖关系，把相关语句组织在一起，特别是使用同一参数的那些语句。

第十四章　条件语句

目录

相关章节

经常遇到的有关控制语句的几个问题：见第 17 章
顺序式程序代码：见第 13 章
使用循环的代码：见第 15 章

条件语句是控制别的语句是否执行的语句，有些语句如 if，else，case，switch 能控制别的程序语句是否执行。虽然循环控制语句如 while 和 for 通常也认为是条件语句，但习惯上一般把它们单独讨论。在第 15 章的循环语句中，将讲到 while 和 for 的用法。

14.1　if 语句

你所用语言可能不同，但你总会用几种 if 语句。最简单的是常用的 if 和 if－then 语句，if-then－else 稍复杂一点，长链形式的 if－then－else 更复杂。
假如你所使用的语言不支持结构形式的 if－then－else 语句，可以用第 17.6 节中用到的技巧来模仿它，即把 goto 语句模仿结构化结构。

14.1.1　简单 if-then 语句

编写 if 语句时请参考以下几点：

在代码中，先按正常情况的路径往下编写，然后再写异常清况。在编写代码时，一定要使得正常情况的路径在代码中显得很清晰。记住，异常情况一定不要使正常情况路径产生模糊。这对程序的可读性及功能是很重要的。

在出现等号情况时，一定要弄清程序的流向。当读取数据或计算循环的控制变量时，用'＜'替代'＜='或用'＜'替代'＜='都同样会产生边界错误（所谓边界错误指在控制循环或重复时，由于控制变量的某一边界值出错或超出范围而使整个程序无法继续下去的错误）。在循环时，一定要弄清结束点，以避免产生边界错误，在条件语句中，一定要弄清等号时的情形，以避免产生边界错误。

把正常情况放在 if 后面而不是 else 后面。你当然希望正常的情况在程序中先处理，这样按层层递推的方法，按先正常后不正常情况，按直线式往下排。下面这个例子犯了好多随意安排正常不正常情况的毛病：

```
OpenFile (InputFile,Status)
```

```
if Status=Error then
    ErrorType=FileOpenError                              ——错误情况
else
        ReadFile(InputFile,FileData,Status)                  ——正常情况
    if Status= Success then
        SummarizeFileData(FileData,SummaryData,Status)   ——正常情况
        if Status=Error then
            ErrorType=DataSummaryError                   ——错误情况
        else
          PrinSummary(SummaryData)
          SaveSummaryData(SummaryData,Status)
          if Status=Error then
            ErrorType=SummarySaveError                   ——错误情况
          else
            UpdateAllAccounts                            ——正常情况
            EraseUndoFile
            ErrorType=None
          end if
        end if
    else
        ErrorType=FileReadError                          ——错误情况
    end if
end if
```

这段代码中是很难让人苟同，因为正常情况和异常情况都混杂在一起。在代码中很难看出整个程序是按正常情况的路径贯穿的。另外，因为异常情形有时也出现在 if 语句后而非 else 之后，因此很难相信程序是按正常情况的线索贯穿的。下面重新写了这段代码，先集中编写了正常情形，然后才编写异常情形。这样寻找和读起来，都很容易发现哪些是正常情况。

```
OpenFile(InputFile,Status)
if Status<>Etror then
    ReadFile(InputFile,FileData,Status)                  ——正常情况
    if Status=Success then
        SummaryFileData(FileData,SummaryData,Status)     ——正常情况
        if Status<>Error then
            PrintSummary(SummarData)                     ——正常情况
              SaveSummaryData(SummaryData,Status)
            if Status<> Error then
                UpdateAllAccounts                        ——正常情况
              EraseUndoFile
              ErrorType=None
            else
              ErrorType=SummarySaveError      /* 错误情况 */
            end if
```

```
    else
       ErrorType = DataSummaryError    /* 错误情况 */
    end if
    else
       ErrorTyre = FileReadError          /* 错误情况 */
    end if
 else
    ErrorType=FileOpenError             /* 错误情况 */
 end if
```

在以上例子中，可以顺着主程序流的 if 语句去发现哪些是正常情况，修改后的代码使人集中阅读主程序流而不是费力地去寻找异常情况，整个程序读起来轻松明快。把异常情况都放在程序的后部，就是很好地处理了异常情况的代码。

if 语句后跟上一个有意义的语句。有时你可能碰到下面这个例子，if 语句后什么也没有。

```
If(SomeTest)
 ;
else
{
 /* do something*/
 …
}
```

若你是一个有经验的编程员，绝对不会这样去编程，这样显得很呆板。改进的方法是把 if 的条件逻辑取反，把 else 后的可执行语句移到 if 后，然后去掉 else 语句，改正如下：

去掉了 if 语句后空语句情况的 C 程序例子：

```
if(!SomeTest)
      {
      / * do something * /
      }
```

考虑是否需要 else 语句。如果你真的想用一个简单 if 语句，考虑到底是否真的就需用一个 if－then－else 语句的形式。通用动力公司（General Motors）在分析了用 PL / I 编写的代码且发现仅 17％的 if 语句后跟 else 语句，而这用了 else 语句的情况中，也有 50～80％的仅用一个 else 语句（Elshoff 1976）。

一种观点认为编写 else 语句（如果需要，跟上一个空语句），是要表明 else 情况已经考虑过了。为了表明考虑过了 else 情况而编一个空 else 语句，可能显得多余，但至少说明你并没有忽略 else 情况。如果你的 if 语句后无 else，除非原因很明显，否则用注释来标明为什么 else 语句不需要。例子如下：

这是一个 Pascal 程序例子，用注释说明为什么不要 else 语句是很有帮助。

```
{ if color is valid }
```

```
if (Min.Color<=Color and Color<=Max.Color)
    begin
    { do some thing }
    …
    end;
    { else color is invalid }
    { Screen not written to -- safely ignore command }
```

检查 else 语句的正确性。检查代码时，像 if 语句这样的主要语句是应当检查到的。但也应当检查 else 语句，以确保其正确。

检查 if 语句和 else 语句是否弄反了。一个常见的错误是在编写 if-then 语句时，把该跟在 if 后的代码与该跟在 else 后的代码正好弄反了，或者使 if 中的判断逻辑正好弄反了。检查你的代码避免这种错误。

14.1.2　if-then-else 语句

假如你所使用的语言不支持 case 语句，或者只是部分支持，那么你会发现，你得经常编写 if-the-else 语句。如下例子中，代码要把输入的字符归类，因而用到如下语句：

用 if-then-else 语句排序字符的 C 语言例子。

```
if(InputChar<SPACE)
  CharType=ControlChar;
else if(InputChar==' ' || InputChar==',' || InputChar=='.' ||
        InputChar=='!' || InputChar=='(' || InputChar==')' ||
        InPutChar==':' || InputChar==';' || InputChar=='?' ||
        InPutChar=='-')
  CharType=Punctuation;
else if('0' <= InputChar && InputChar <= '9')
   CharType=Digit;
else if('a' <= InputChar && InputChar <= 'z') ||
        ('A' <= InputChar && InputChar <= 'Z')
   CharType=Letter;
```

以下几条是在编写 if-then-else 时要遵循的：

用布尔函数调用（boolean function 亦称逻辑函数）简化程序。上面代码不好读的一个原因是把字符进行排序的条件过于复杂。为了提高可读性，可以用布尔函数来代替这些判断条件。

用布尔函数调用的 if-then-else 的 C 语言程序例子。

```
if(IsControl(InPutChar))
   CharType=ControlChar;
else if(IsPunctuation(InPutChar))
   CharType=Punctuation;
else if(IsDigit(InputChar))
   CharType=Digit;
else if(IsLetter(InputChar))
```

```
CharType=Letter;
```

把最常见的情形放在最开始。把最常见的情形放在最开始，你就可以少读许多处理异常情况的代码，而直接读常见情况的代码。这样就提高了寻找常见情况的效率。在上述例子中，字母该是最常见的情况，但检查是否为标点代码却放在最先。把检查字母的代码放在最开始修改如下：

这个 C 语言的例子中，把处理最常见情况的代码放在最先：

```
if(IsLetter(InputChar))      ——这个判断最常见，放在第一位
  CharType=Letter;
else if(IsPunctuation(InputChar))
  CharType=Punctuation;
else if(IsDigit(InPutChar))
  CharType=Digit;
else if(IsControl(InputChar)) ——这个判断最少见，放在最后
  CharType=ControlChar;
```

保证覆盖全部情况。最后用一个 else 语句处理那些你未曾想到的错误信息。这个错误信息很有可能是由你而非用户引起的，所以一定要正确处理这种情况，下面例子中，增加了处理其它情况的代码。

这个 C 语言程序例子，用缺省情况应付可能出现的其它情况。

```
if(IsLetter(InputChar))
   CharType=Letter;
else if(IsPunctuation(InputChar))
   CharType=Punctuation;
else if(IsDigit(InputChar))
   CharType=Digit;
else if(IsControl(InputChar))
   CharType=ControlChar;
else
   PrintMsg("Internal Error: Unexpectecd type of character detected.");
```

假如你所使用的语言支持别的结构能代替 If-then-else，替换掉 if-then-else 形式。少数几种语言，如 Ada 支持 case 语句，能很方便地替换大多数的 if-then-else，那就用 case 语句，case 语句易写易读，较 if-then-else 结构为好！下面用 Ada 的 Case 语句来归类输入字符：

这个 Ada 语言程序用 Case 语句代替了 if-then-else。

```
case InputChar is
   when 'a'..'z' | 'A'..'Z' =>
      CharType := Letter;
   when ' ' | ',' | '.' | '!' | '(' | ')' | ':' | ';' | '?' | '-' =>
      CharType := Punctuation;
   when 'o'..'9' =>
      CharType:＝Digit;
   when nul..us=>
```

```
    CharType:=ControlChar;
  others=>
   PUT_LINE("internal Error: Unexpected type of character detected.");
end case;
```

14.2　case 语句

case 语句和 switch 语句因语言不同其结构也变化很大。许多版本的 Basic 语言根本就不支持 case 语句。C 语言支持的 case 语句，其分支变量每次只能对应一个值。Pascal 中的 case 语句也只对应一定范围内的数据。Ada 支持 case 语句，并且它用表示值的范围和组合的能力很强。

在 Apple Macintosh 和 Microsoft Windows 环境中编程，大型的 case 语句经常用到。随着交互式事件驱动程序的日益普及，case 语句的使用频率越来越高。下面部分指出了怎样有效地使用 case 语句。

14.2.1　选择最有效的方法来组织各种情况

在 case 语句有许多方法可以用于组织各种情况。假如 case 语句很小，仅有三种选择和相应的三条响应的语句，那么你怎样组织这个 case 语句都行。如果你的 case 语句很大，如在事件驱动程序中用到的 case 语句，把各种情况组织成一个合理的顺序是很重要的。下面是可能的组织方法。

把各种情况按字母或数字顺序组织。如果各种情况是同等重要的，按 A—B—C 顺序安排，以提高可读性。每个事件都可很容易从整体中挑出来。

把正常情况的事件放在最开始。如果是一个正常情况及几个异常情况，把正常情况放在最先，并用注释标明哪些是正常情况哪些是异常情况。

按出现频率组织情况。把最经常执行的情况放在最先，而最不可能的情况放在最后。这种方法有两大好处。首先，读者很容易找出最普遍的情况。读者为寻找某个具体的情况而浏览整个程序，极有可能就是找最常见的那种情况。把常见的情况放在代码的最开始，读者能很快找到它。第二，机器执行起来也快。每一种情况都在代码中有相应的执行语句，机器执行都要花时间搜索，如果有 12 种情况而最后一个情况是要执行的。那么机器要搜索 12 条 if 语句，直到最后发现相应的情况。若把最普遍情况放在最先，你就可以减少机器的搜索区间，这样就可提高代码的效率。

14.2.2　用 case 语句该注意的几点

下面是用 case 语句时要注意的几点：

使每种情况对应的执行语句最简单。每种情况对应执行代码应当短些。短的执行代码结构显得清楚。如果某个情况对应的执行代码显得很复杂，应当把它写成一个子程序然后对应这种情况调用于程序，而不是在这种情况之后直接跟上复杂的执行代码。

不要为了用 case 语句而去定义伪交量。一个 case 语句应当用在易被归类的简单数据上。假如数据不简单，用 if-then-else 代替 case。伪变量显得很乱，应当避免使用。下面是几个反例：

这个 Pascal 程序编得不好，因为它定义了一个 case 伪变量

```
Action:=UserCommand[1];
case Action of
   'c': Copy;
   'd': DeleteCharacter;
   'f': Format;
   'h': HelP;
else PrintErrorMsg('Error: Invalid command.');
end；{case}
```

控制 case 语句的变量是 Action。在本例中，变量 Action 是通过取 UserCommand 字符串的第一个字母形成的，字符串由用户自己输入。

这种勉强的编码方法非常不可取，也产生了许多问题。在 case 语句中，你自己定义了变量，但真实的数据可能并不如你所期望那样产生预期动作。比如在上例中，读者把 copy 的第一个字母 C 定义替代 copy，并且能正确调用 copy()子程序。但另一方面，如果用户想在 case 语句中定义"cement overshoes"，"clambake"，"Cellalite"抽出第一个字母来代替这个变量，那么也为 C，调用的却是 Copy()子程序。case 语句中的 else 语句也可能出毛病，因为它只管错误命令的第一个字母而不管这个命令本身。

若不用定义伪变量的方法，则可用 if-then－else 来达到检查整个命令串的目的。以下是重新编写的代码：

这个 Pascal 程序较好，用 if-then-else 代替 case 中的伪变量。

```
if(UserCommand='Copy') then
   Copy
else if(UserCommand＝'delete') then
   DeleteCharacter
else if(UserCommand='format') then
   Format
else if(UserCommand='help') then
   Help
else
   PrintErrorMsg('Error: Invalid command.');
```

若用缺省语句只用合法的缺省。有时还有一种情况，你就想把这种情况编写成缺省语句。这种做法有时虽很诱人，但却不足取。要这样你就没有利用 case 语句的好处，并且失去利用缺省语句检查错误的能力。

这样使用 case 语句不便于修改。如果用合法的缺省，增加一种新情况是很容易的事情——你只需增加一种情况并编写相应的执行代码就行了。假如用了缺省，修改起来很困难。若你要增加一种新情况则要先编一个新的缺省情况，然后把以前用作缺省的情况改为一般情况，才能使整个 case 语句写成合法代码。在编写程序时从一开始时就该用合法缺省。

用缺省语句检查错误。如果缺省语句不用作处理一些操作，也不支持某些情况，就把诊断信息放在里面。下面是这样的例子：

这个 Pascal 程序较好，用缺省情况发现错误。

```
case Letter of
  'a': PrintArchives;
  'p': {no action required,but case was considered}
    ;
  'q': PrinrQuarterlyReport;
  's': PrintSummary;
  else PrintErrorMsg('Internal Error 905:Call Customer assistance.');
end; {case}
```

缺省语句中的信息对调试和编写代码都是很有用的，用户最喜欢的信息是在系统运行失败时给出“内部错误，请调用客户支持”之类的信息；或者最坏的情形，但结果是错误的却好像是正确的，直到检查到了为止。如果缺省语句不仅仅用作发现错误，那么就意味着对每一种情况的选择都应当正确，应当检查以确保进入 case 语句的值都合法。如果有进入 case 语句的值不合法的，修改对应情况的语句，以便使缺省能真正检查错误。

在 C 语言中，每个情况对应的代码段都应当有一个结束语句。C 语言是一种通用的高级语言，在 case 语句中执行每种情况都并不自动跳出，因而你得给它一个明确的结束命令，如果你不给每种情况一个结束的语句，执行完这种情况的代码后接着转入执行下一个情况的代码。如果真是这样，那就犯了编程中的大忌。下面这个例子正是这样。

这个 C 语言程序中 case 语句没用好

```
switch(Inputvar)
  {
  case('A'): if(test)
               {
                /* statement l*/
  case('B'):   /* sratement 2*/
               /* statement 3*/
               /* statement 4*/
               } /* if(test) */
             break;
  }
```

这个例子编得不好，因为整个控制结构混在一起了，这种嵌套结构很难理解，要修改例子中的’A’情况和’B’情况比做脑部手术更难。若你真的要修改这两种情况，倒还不如推翻了重写。你可能一次就写出正确的代码。总的说来，在用 case 语句时，不要忘了在每种情况中安排一个结束代码。

在 C 语言中，case 语句的最后都应当准确无误地标明结束。假如你有意地在最后不标明结束，那么用注释说明你为什么要这样。

14.2.3　检查表

条件语句

if-then 语句

- 正常情况路径在代码中流向是否很分明？
- if-then 语句在出现等号时流向是否正确？
- else 语句是否有必要？
- else 语句正确吗？
- if 语句和 else 语句正确吗？它们是否弄反了？
- 正常情况是否跟在 if 后而非 else 后？
- if-then-else
- 复杂的条件是否封装成布尔函数调用了？
- 最常见情况放在前面吗？
- 全部情况都覆盖住了吗？
- if-then-else 语句是最好的选择吗？——用 case 语句代替是否更好？

case 语句

- 各情况的安排次序有含义吗？
- 每种情况对应的操作简单吗？——如需要调用别的子程序。
- case 语句中的变量有实际意义吗？它是为了用 case 语句而单纯地定义出来的伪变量吗？
- 缺省语句的用法是否合法（规范）？
- 用缺省语句检查和报告异常情况吗？
- 在 C 语言中，每一情况的结尾用了 break 了吗？

14.3　小　结

- 注意 if 和 else 的顺序，特别是在处理好多异常情况时，务必使正常情况流向清晰。
- 组织好 if-then-else 和 case 语句中的几种情况，使可读性最好。
- 在 case 语句中用缺省值，在 if-then-else 中的最后一个 else 中获取意外错误。

各种控制结构并不都同样有用，在编码时选用最合适的控制结构。

第十五章　循环语句

目录

相关章节

一般控制语句应当注意的几个问题：见第 17 章
直接式程序代码：见第 13 章
条件语句代码：见第 14 章

所谓循环指任何一种类型的重复性控制结构。这种结构让代码的某一块被重复执行。几种常见的的循环类型有 Basic 中的 FOR NEXT 语句、Fortran 中的 DO 语句、Pascal 中的 while-do 和 for 语句、C 语言中的 while 和 for 语句以及 Ada 中的 loop 语句等，使用循环是用程序化语言编程时最复杂的情形之一，知道如何及何时用哪种循环是编写高质量软件的决定性因素。

15.1　选择循环类型

在大多数语言中，你总可以用上几种循环类型。

- **计数循环**要执行给定的循环次数．也可能就只循环一次。
- **条件循环**先并不知道要执行多少次循环，而要在每次重复之前检查是否满足循环条件。只要满足循环条件，循环不断继续下去，除非用户退出循环或出错。
- **死循环**只要开始便无限执行下去。在心脏起搏器、微波炉、巡航控制等系统中，把死循环置入其中。

以上几种循环的不同首先在于其灵活性——循环前检验一下是否满足退出循环的条件。循环还在于把控制循环的语句放置在何处。你可以把控制循环的语句放在循环体的开始，中间或结尾。这种特性告诉你循环是否至少要执行一次。如果控制循环的语句放在开始那么循环体可能不被执行。而若放在结尾，循环体至少要执行一次。若放在中间，有一部分循环体至少要执行一次，而另一部分（控制循环语句之后）则可能一次也不执行。

在选用循环结构时，灵活性和控制循环语句的位置是关键。表 15-1 列出了几种语言的循环类型和控制循环语句的位置。

表 15-1 循环类型

语言	循环类型	特性	放置位置
Ada	For	计数循	开始
	While	条件循	开始
	loop-with-exit	条件循环	通常在中间
	Loop	n / a	无（用于死循环）
Basic	FOR-NEXT	计数循	开始
	WHILE-WEND	条件循	开始
	DO LOOP	条件循环	开始或结尾
C	do-while	条件循环	开始
	While	条件循环	开始
Fortran	Do	计数循环	开始
Pascal	FOR	计数循环	开始
	repeat-until	条件循	结尾
	While	条件循环	开始

15.1.1 while 循环

初学者有时认为 while 循环是要不断地作判断。当 while 的条件为假时，循环立即停止；而不问在循环中哪些语句被执行了。虽然 while 循环不全是条件循环，但一般把它看成是条件循环。假如事先你并不知道循环的次数，那就用 while 循环。与初学者想法正好相反，使循环中止的语句在每次循环时只执行一次，而主要要考虑的事情是确定把控制循环的语句放在开始还是结尾。

把控制循环的语句放在开头的循环

把控制循环语句放在开始的循环在 C、Basic、Pascal、Ada 中是 while 循环，在 Fortran、汇编或其它语言中，可模仿 while 循环。

若想把控制循环的语句放在开始，最好选 while 循环。有些研究人员建议除非不能用，否则一定要作这种循环类型。若认同这种评论，那么读者在读你的代码时所需理解的循环结构种类就减少了。如果你只用一种基本的循环类型，读者就会慢慢适应你的这种编程风格，如果你用几种循环类型，你就使读者很烦——他们要熟悉这几种循环类型，而你自己也要精通这几种类型。

把控制循环的语句放在末尾的循环

有时你想用条件循环，但又想使循环至少执行一次，那么用 while 循环并把控制循环的语句放在循环体的末尾。当然你也可以在 C 中用 do-while 循环，在 Pascal 中用 repeat-while 循环、在 Basic 中用 DO-LOOP-WHILE 循环、或在 Ada 中用 loop-with-exit 循环结构。也可在 Fortran、汇编或其它语言中模仿这种把控制循环的语句放在结尾的循环。

C 语言的一大好处便是 while 和 do-while 这两种循环结构所用控制循环和语句相同，两者唯一的区别在于 while 把控制语句放在开始而 do－while 把控制语句放在结尾。Pascal 的 while

循环和 repeat-until 循环有两点不同：跟 c 语言一样，while 控制循环语句放在开头，而 repeat-until 则放在末尾。而另一奇怪的不同之处在于 repeat-until 的循环条件在逻辑上与对应的 while 循环的条件正好相反。如果 while 的条件是“当…不循环”，则 repeat-until 的条件是“当…循环”。这种区别比控制循环语句的放置位置更让人容易糊涂，因此一般建议选用把控制循环语句放在开头的 while 循环。

15.1.2　Loop-with-exit 循环

Loop-with-exit 循环是把终止条件放在循环体中间而不是开头或结尾的循环。Ada 支持 Loo-with-exit 循环，你也以在 C 中用 while 和 break，或在其他语言中用 goto 来模仿这种循环结构。

正常的 loop-with-exit 循环

正常的 loop-with-exit 循环包含循环头、循环体（含结束条件）循环尾，如下面 Ada 程序例子：

一个常见的 Ada 的 Loop-with-exit 例子程序：

```
loop
   …
   exit when(some exit condition);
   …
end loop;
```

用到 loop-with-exit 循环的情况是，在循环体退出前至少执行循环体中部分情况一次。

下面的 C 语言子程序用到了 loop-with-exit 循环。

```
/* Compute scores and ratings */

score=0;
GetNextRating(&RatingIncrement);
Rating=Rating + RatingIncrement;          ┐
                                          ├── 这些行出现在这里
while(Scroe < TargetScore && RatingIncrement!=0)
   {
   GetNextScore(&ScoreIncrement);
   Scroe=Score + ScoreIncrement;
   GetNextRating(&RatingIncrement);       ┐
   Rating=Rating + RatingIncrement;       ├── 这些行出现在这里
   }
```

上例中，循环体中的最后两行重复了循环体前的两行。在修改程序时，你可能忘了同时修改这两处，而其它的编程人员或许根本就没有意识到这两处是一样的，而需要一起修改，这样因为改得不彻底程序会出错。下面告诉你怎样修改这个程序。

这个 C 程序用了 loop-with-exit 循环，易于修改。

```
/*Compute scores and ratings. The loop uses a FOREVER macro
```

```
            and a break statement to    emulate a loop－with－exit loop. */

        Score = 0;
        FOREVER
            {
        GetNextRating(  ＆RatingIncrement );
        Rating = Rating + RatingIncrement );

        If (!( Score  ＜  TargetScore && RatingIncrement != 0 ))    -- 这是循环退出条件
            break;

        GetNextScore( &SoreIncrement);
        Score  ＝  Score  ＋   ScoreIncrement;
        }
```

下面用 Ada 写了一段代码；

这个 Ada 程序用到了 loop－with－exit 循 环

```
    - Compute scores and ratings

    Score := 0;

    Loop
        GetNextRating( RatingIncrement);
        Rating := Rating + RatingIncrement;

        exit when ( not ( Score < TargetScore and RatingIncrement!=0 ));

        GetNextscore ( ScoreIncrement ) ;
        Score := Score + ScoreIncrement;
    end loop;
```

在用 loop-with－exit 循环时，请遵循以下几点 ：

• 把所有的终止循环语句放在一起。分散放置终止语句可能使一个或几个其它的终止　语句在调试、修改、测试时被忽视。

• 用注释阐明。如果一种语言不支持 loop－with－exit 循环结构（实际上，Ads 之外的任何语言都不支持），那么用注释标明哪些地方你用了 loop-with-exit 循环技术。

loop－with－exit 循环是单进单出的结构化控制结构，而且是 Ada 循环控制中较为常用的一 种。它较之其他种类的循环显得简单易懂。有人把 loop－with－exit 在中间结束循环和其它在开始或末尾结束循环的循环作比较发现，初学者中有高达 25％人用 loop－with－exit 循环。研究人员从这个研究中得出结论认为，loop-with-exit 循环结构更接近于人对循环控制的理解。

但是 loop－with－exit 循环用得并不普遍。问题在于对这种循环是否是一个好的方法的无穷无尽的争论。除非最后确实证明，否则我要说 loop－with－exit 循环是一个很好的技巧，你尽可能地使用。

不正常的 loop-with-exit 循环

为避免循环无法作用在边界情况上，用到另一种 loop－with－exit 循环。

这个 C 语言用 goto 直接进入循环体内部，显得很不好：

```
goto Start;
while（expression）
{
 / *   do something * /
  ...
  Start;
   / * do something else * /
  ...
}
```

乍一看，上面这个程序与以前的 loop-with-exit 例子相似。这种编程方法用在循环体的前半部分 do－something 第一次要跳过而不执行，但后部分 / *do something else* / 第一次却需执行的情况。这也是种单进单出的结构：唯一的进口在循环体的上边的 goto 语句，唯一的出口是 while 语句。这种方法有两个问题：用了 goto 语句并且显得很乱。

在 C 中，不用 goto 语句同样能达到效果，如下面有这个子程序。如果一种语言不支持 break 或 leave，可用 goto 去模仿。

这个 C 程序较好，没有用 goto 却达到同样效果

```
 FOREVER
     {
      / * do something else * /
     ...
     if (!( expression ))
          break;

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

15.1.3　for 循环

当你想使程序块循环给定次数时，for 循环是一个好的选择。在 C、Pascal、Basic、Ada 中用 for 循环，在 Fortran 中用 DO 语句。

用 for 循环做简单动作时，不需设内部循环控制变量。你所需做的是在循环之前设一个控制变量就可以不管了。你不需在循环内部设任何控制。若在某些条件下必须跳出循环，选用 while 循环更好。

当然，不要强行修改控制标志值使循环终止，这时可用 while 循环替代。for 循环仅用于简单情形，大多数复杂的循环还需要很大 while 循环。

15.2　控制循环（Controlling The Loop）

在编写循环时会出哪些错？当然包括以下错误：忽略了与循环有关的变量或累加器初始化、不正确的嵌套、不正确的循环中断、忘记给循环变量一个增量或给错了增量、用不正确的循环指标访问数组元素。

你可以先来看两个例子以对上述问题有一个感性认识。首先，应减少影响循环的因素，简化、简化、再简化。第二，把循环体内部当作一个子程序看一一把控制语句尽可能地放在循环体外，以明确循环体执行的条件。不要让读者看了循环主体本身后才弄清循环控制。可以把循环体着作一个黑盒子：周围的程序只判断控制条件而不知里面的内容。

这个 C 程序把循环体作为一个黑盒子；

```
while（！ eof（InputFile）&& MoreDataAvailable）
{
[redacted]
}
```

循环到底在什么条件下终止？在这个程序中，你只知道当 eof（InPutFile）为真而 MoreDataAvailable 为假时才退出循环。

15.2.1　进入循环

下面几条告诉你如何进入循环：

仅从一个入口进入循环。许多循环控制结构允许你从开头中间或末尾进入循环。你从循环头部进入循环，大可不必多口进入。

把初始化循环的代码紧放在循环前头。最近原则提倡把相关语句放在一起。如果把相关语句分散在程序的各处。修改时可能顾及不全而产生修改错误。如把相关语句放在一起。可避免在修改时出错。

把与循环体相关的初始化循环的语句放在一起。如果不这样，当你扩大循环体或修改时很可能忘了修改初始化循环的代码。当你把一个循环体拷贝到子程序里时，你就可能因为忘了一起拷贝初始化部分而出错。把初始化部分放在远离循环的地方（在数据定义区域）或内务处理区就可能产生初始化循环出错的麻烦。

在 C 中，用 FOREVER 宏编写死循环或事件循环。有时你想写一个无终止的循环，如埋植在心脏起搏器中或微波炉中的循环，有时你也可能写一个循环，在响应某一事件时才终止，这就是事件循环。你可能有许多方法编制循环，但下面的宏定义在 c 中是标准的：

这个 c 语言程序用了一个死循环：

```
＃define FOREVER for（；；）
...
```

```
FOREVER——这里是无限循环
{
 ...
}
```

在 Pascal 中，用 while（真）结构编写死循环或事件循环。如果希望循环有条件终止，在 Pascal 中用 goto 或 exit 离开循环。在这种情况下，goto 或 exit 是一个保护性的结构性编程结构，保证了循环的单口退出，保证了单入单出的结构性编程需要，下面的 Pascal 程序是标准的建立死循环的方法；

这个 Pascal 程序编写了一个死循环：

```
while(True)
    begin
    {infinite loop}
    ...
    end;
```

以上是用 Pascal 和 C 编写死循环和事件循环的标准方法。伪死循环像 for i:=-i to 999 使得循环的替换性很差。因为试图用循环次数不会超出极限以达到死循环目的——可能 9999 就是一个合法值（没超出范围）。这个伪死循环在修改时也可能出错。

在 C 中，只要允许就用 for 循环：C 的 for 循环是这种语言强有力的结构之一。它不仅灵活性强，而且把循环控制代码封装在一起，增加了其可读性。程序员在修改软件时易犯的错误是；修改了循环前面的初始化循环的代码，但却忘了修改其后面的有关代码。在 C 的 for 循环中，所有相关代码集中在循环的顶部，修改起来很容易。如果在 C 中能用 for 循环替代别的类型的循环，尽量这样做。

当 while 循环更合适时，别用 for 循环。在 C 中乱用 for 循环的一例是在 for 的条件中用了 while 循环的条件，下例便是这种情形：

这个 C 中程序虽是 for 循环却用了 while 循环的条件头：

```
/* read all the records from a file */

for(rewind(InFile).RecCount = 0; !feof(InFile); RecCount++)
{
fgets(InputRec[RecCount], MAX_CHARS, InFile);
}
```

C 语言的 for 循环比其它语言的 for 循环优点在于，它的初始化和结束条件很灵活，而这种灵活性带来的固有缺点是把控制条件放在了循环头，因而对循环体就无法控制了。

把控制循环的语句放到 for 循环头如初始化循环变量、终止循环或转向终止的表达式。上例中，fgets（）语句使循环转向中止，但 RecCount 语句却没起到这个作用，它是内部语句，没有起到控制循环的作用。把 RecCount 语句放在循环头而把 fgets（）语句放在循环体中是一个错误，它使人误解为是 RecCount 在控制循环。

在这种情形下，若想用 for 循环而不用 while 循环，那就把控制循环的语句放在循环头，而把不起这个作用的其他语句放到循环体中。下面这个程序用了正确的循环头：

这个 c 语言的例子，循环头很不一般：

```
RecCount = 0;
for(rewind(InFile);
    !feof(InFile);
    fgets(InputRec[RecCount], MAX_CHARS, InFile));
    {
    RecCount++;
    }
```

这个程序的循环头中的内容都与循环的控制有关，rewind（）语句初始比循环，feof（）语句检查是否该终止循环，fgets（）语句转向终止操作。修改 RecCount 语句并不使循环转向终止，因而不把它放在循环头，在这种情况下用 while 循环或许是最合适的，但要用得好，仍要用 for 循环头的形式。在这里给出了用 while 循环写的程序。

这个 C 程序较好地用了 while 循环：

```
/* read all the records from a file */

rewind(InFile);
RecCount = 0;
while(!feof(InFile))
    {
    fgets(InputRec[RecCount], MAX_CHARS, InFile);
    RecCount++;
    }
```

15.2.2　处理好循环体

以下几点对处理好循环体会有帮助：

用 begin 和 end 或｛和｝把循环体括起来。如果你容易忘记从哪到哪是循环体，就用括号把它们括起来。括号不占任何空间和运算时间，也不破坏可读性，相反，括号能保证你不出错，因此尽量用就是了。

尽量避免用空循环。在 C 中有可能出现空循环。要检查某一工作是否完成时，把检查作为循环条件就可能出现了只有一句的空循环。如下例：

这个 c 程序是一个空循环：

```
  while((InputChar＝ getch())！＝'\n')
    ;
```

此例中，循环为空循环，因为它只做了两件事，循环动作中——InputChar=getch（）和检查循环是否该终止——InputChar!='\n'下面是修改后的程序，循环所做的工作对读者来说相当清楚，程序也显得明朗。

把空循环改为有操作语句的循环的 c 程序：

```
    do
    InputChar= getch();
```

```
while(InputChar!='\n');
```

这个程序较好，因为在三行中完成一个操作总比在一行外加一个分号完成一个操作好。

把循环的“内务处理工作放在循环的开头或结尾。“内务处理”工作指像 i：=i＋1 这样的表达式，它的作用不是循环要做的事情，而是去控制循环。下面的例子中，内务处理工作在末尾处完成。

这个 Pascal 程序把完成内务处理的语句放在末尾；

```
StringIdx:=1;
TtlLength:=0;
while not eof(InputFile) do
    begin

    {do the work of the loop}

    ReadString(InputFile, StringIdx, Str);
    ...

    {prepare for next pass through the loop--housekeeping}

    StringIex:=StringIdx+1;
    TtLength:=TtLength + Length(Str)        <-- 这里是内务处理语句

    end;
```

一般来说，你所初始化的循环变量往往是在内务处理部分要修改的变量。

使每个循环仅执行一项功能。一个循环一次做两件事并不能说这两件事是同时完成的。循环应当像子程序一样一个只完成一件事且应做好。如果两个循环的效率比用一个循环的效率低，那先用两循环编码并注明它们能合成一个循环以提高效率，然后用标准测试程序来对这部分进行测试，如有必要最后才合并成一个循环。

15.2.3　退出循环

以下几点对如何退出循环会有帮助：

确保循环能终止。先要在脑中模仿，确信在各种条件下循环都能终止。这一点很重要。仔细考虑正常与异常情形下循环的终止问题。

使循环的终止条件明显。如果用 for 循环且不用 goto 或 break 语句来退出循环，终止条件应该很明显。同样，若用 while 或 repeat－until 循环，并把控制条件放在 while 或 repeat－until 句子中，循环中止条件也会很明显。关键是要把控制条件放在一处。

循环中不要强制改变控制循环变量从而达到提前终止循环的目的。下面这个程序就是这样的：

这个 Pascal 程序强行改变循环控制变量：

```
  for i ＝1 to 100 do
```

```
begin
{ some Code }
...
if(...) then
    i = 101    ◀—  这里做得不好
{more code}
...
end;
```

这个程序的本意是在某些条件下，改变控制变量的值为 101，这样大于 i 的范围 1～100，循环终止。一个优秀的程序员是避免这样做的。一旦写好了 for 循环，控制变量就在你的控制之下。若实在想在某些条件下终止，用 While 循环就能满足对终止条件的需求。

尽量避免直接用到循环控制变量的终值。这样用显得很不好。循环控制变量的终值随语言和用法的不同而不同，在正常和异常情况下退出循环也会不同。即使你知道终值是什么，但别人在读程序时却要考虑才知道。在循环体中的适当位置先把终值赋给一个变量，这种方法好些。下面这个程序就乱用了终值。

这个 C 程序乱用了循环控制变量的终值：

```
for (i=0; i<MaxRecords; i++)
{
    if(Entry[i] == TestValue)
    {
        break;
    }
}
if( i< MaxRecords)
    return(TRUE);
else
    return(FALSE);
```

在这个程序段中，好像是若发现某一输入 Entrg[]等于 Testvalue，程序返回 TRUE，因而循环检查了整个输入；否则输出为 FALSE。但是记住，控制条件中控制变量是先加 1 再执行循环体的，若最后一次发现了输入等 Testvalue 但变量却大于 MaxRecords，那还是输出 FAIHE，这就产生了边界错误，最好把程序改写一下，使后面的代码不依赖于控制变量的终值。修改如下：

这个 c 语言不依赖于控制变量的终值：

```
Found = FALSE;
for(i<0; i<MaxRecords; i++)
{
if(Entry[i] == TestValue)
    {
        Found = TRUE;
        break;
```

```
            }
        }
    return(Found);
```

这个代码段定义了一个额外的变量，这个布尔变量的使用使得结果很清楚。有时一个循环接着另一个循环，用循环变量终值会产生好多问题。所以 Ada 中规定在 for 循环外用控制变量无效，编译时算错。

考虑用安全计数器。如果你程序的某一错误会产生灾难性后果，可以使用安全计数器保证所有的循环结束，下面的这个程序就可以受益于用安全计数器。

这个 Pascal 程序的循环用了安全计数器：

```
SafetyCounter := 0;
repeat
   Node := Node^Next;
   ...
   safetyCounter := SafetyCOunter+1;
   if(SafetyCounter>=SAFETY_LIMIT)
   begin
   PrintErrorMsg("Internal Error : Safety-Counter Violation.");
   exit(Error)
   end;
   ...

until(Node^.Next = Nil);
```

这里是安全计数器代码

这个 Pascal 程序引进一个安全计数器，可能会导致额外的错误。如果安全计数器不是每个循环都用，当你修改用到安全计数器的循环代码时，可能忘了去修改安全计数器。如果你用安全计数器相当熟练，那么它就不会比别的语句更容易出错。

15.2.4　用 break 和 continue 语句

break 语句是一种辅助控制结构，它能使循环提前退出，break 使循环从正常出口退出，程序继续执行接在循环后的程序。这里说的 break 是指属同一类的语句，在 C 中是 break，在 Pascal 中是 exit 或类似的结构，包括不支持 break 的语言中有相似功能的 goto 语句。

continue 语句和 break 一样同是辅助循环控制语句，但正好相反，continue 使程序又回到控制体的开始并执行下一个循环。continue 语句是 if－then 语句的简写形式，而后者可能使循环的余下部分不被执行。

用 break 和 continue 一定要谨慎。用 break 语句就不可能再把循环体看作一个黑盒子，把控制循环退出的条件都写在一个语句里，这是简化循环的有力手段。用了 break 语句就要使人从头到尾看你的循环体内容才能理解循环控制，这样就使得循环更难读了。

有选择性地用 break 语句。有些计算机科学家认为这种结构是合法的技巧，而另一些则认为不然，既然你不知用 continue 和 break 好还是不好，那么你用它时应当小心可能用错了。其实情形很简单，若非一定要用，尽可不用。

在 while 循环中，若要用布尔量标志时就应考虑用 break 语句。在 while 循环中有时要增

加一个逻辑标志来模拟循环体的出口，这就使得程序难读。有时为了好看，在有几个 if 语句时，下一个比上一个要缩进几列，这时可用 break，把 break 放在独立的语句段之后，使它靠近它所属的代码段能减少嵌套，使程序易读。

要特别小心一个循环中许多 break 语句分散出现的情形。一个循环中包含了许多 break，则表明对循环的结构或者对周围代码的作用还考虑得不清楚。若 break 语句出现多了，就表明该循环若用许多小循环替代可能更好，因为一个大循环会有许多出口。循环中有多个 break 并不说明一定有错误，但起码给出一个信号；这种用法不太好！

如果用 goto 来模拟 break，转向执行的语句应紧接在循环的后面。有时你很想让循环的出口指向的不是紧跟在循环后的第一条语句而是更远，这时你就可能离用 goto 来做的危险尝试不远了，或者你就处于那种要用 goto 来结束循环的境地。不要因为用了 goto 而把问题弄复杂了。

把 continue 放在循环的头前检查。continue 语句的一种很好的用法是把它放在循环的头部检查，若满足某一条件才执行循环体。如下例程序，循环要做的是先读人记录，然后看是否满足一定条件，若不满足，就放弃这个记录；若满足，就处理这个记录。这时把 continue 放在循环的头部：

用 continue 比较安全的伪代码程序：

```
while(not eof(File)) do

    read(Record, File)
    if(Record.type<>TargetType)then
            continue

        {process record of TargetType}
    ...

end while
```

这种用 continue 的方法可使你避免在用 if 语句时，需把循环体往后缩几列的做法。但是若 continue 出现在循环体的中间或结尾，那最好还是用 if-then 替代。

15.2.5　检查循环边界

一个简单的循环通常要注意三种情况：初始情况、中间情况、结束情况。当你要编写一个循环时，你头脑中要想清楚：在初始、中间、结尾情况下都不会出现边界错误。如果初始或结尾情形可能出问题，仔细检查一下，如果循环包含了复杂的计算，拿出计算器核算一下。

进行这种检查是高效率与低效率程序员关键不同之处，高效率的程序员在脑中或用手算一下，因为他们知道：这样做可以帮助发现问题。

低效率的程序员总是随意地编写，反复修改到最后发现应该是怎样做。如果循环不按设想的那样去做，低效率的程序员总要把憭'<'改成'<='。如果这样还行不通，他们就要改变循环控制变量了，或者加或者减 1。到最后，他们用这种方法幸运时，撞对了，但很可能把简单的错误改得更隐晦。即使这种试凑方法最后把循环改对了，程序员也不知道为什么就改对了。

在脑中先想想或先算算有几个好处。如，在编程阶段减少错误，在调试时能很快发现错误，

且能更好地理解整个程序。先想想能使你知道代码是如何运行的，而试凑者则不然。

15.2.6　使用循环控制变量

以下几点对如何用循环变量有帮助；

在循环和数组中，只能用整数。一般说来，循环计数器应当是整数值，浮点数不好使。如，你把 1.0 加到 42,897.0 上去得到的是 42,897.0 而非 42,898.0，若这种情形发生在循环计数器上，就发生死循环了。

要用有意义的变量名使得循环嵌套易读。循环变量通常就用作数组下标。如果数组是一维的，你可以用 i,j 或 k 作下标去标识它。但是若数组是二维或多维的，你就要用有意义的下标去标明你在做什么了。有意义的数组下标名能清楚说明每一层循环的目的和要处理的数组元素。

下面这段代码没有遵循上述规则，用无意义的变量名 i、j、k：

这个 Pascal 程序用了无意义的变量名：

```
for i:=1 to NumPayCodes do
    for j:=1 to 12 do
        for k:=1 to NumDivisions do
            Sum := Sum + Transaction[j,i,k];
```

你明白 Transaction 下标的意义吗？i、j、k 告诉你 Transaction 的含义了吗？一旦这样定义 Transaction，在循环中你就无法确定下标的顺序是否对了。

这个 Pascal 程序的循环变量意义明朗:

```
for PayCodeIdx:=1 to NumPayCodes do
    for Month:=1 to 12 do
        for DivisionIdx:=1 to NumDivisions do
            Sum:= Sum + Transaction[Month, PayCodeIdx, DivisionIdx];
```

这回你知道 Transction 数组下标的含义了吗？在本例中,答案很清楚:变量 Paycodelndex、Month、DivisionIdx 很清楚地给出了各自代表意思，而 i、j、k 则不能。计算机在读下标时同样简单，但人读起来却觉得第二个比第一个简单多了。记住，你的最基本读者是人而非计算机。

用有意义的变量名以避免循环变量用重复了。若习惯都用 i、j、k 作变量可能引起冲突--一个循环中不同地方用了同一个循环变量名，如下例子：

这个 c 程序中循环变量冲突：

```
for(i=0；i<NumPayCodes; i++)     —— i 在这里使用
{
    /* lots of code */
    ...
    for(j=0; j<12; j++)
        {
        /* lots of code */
        ...
        for(i=0; i<NumDivisions; i++)          —— i 又在这里使用
            {
```

```
                Sum += Transaction[j][i][k];
            }
        }
    }
```

因为用惯了 i，所以在同一嵌套结构中两处用 i 因为第二个 for 循环包含于第一个，其中的 i 就与第一层循环中的 i 冲突了。若用有含义的变量名则可避免出此错误，一般说来，循环不止几行，而且可能还要往下写，若本来就是几层嵌套的循环，这时应避免用 i、J、k。

15.2.7　一个循环该有多长

循环的长度可以用行数或嵌套深度来衡量。注意以下几点：

使循环尽可能短，能一目了然。如果把程序打印在纸上，纸一页能打印 66 行，那么循环的长度不要超过 66 行；若在屏幕上，一页显示 25 行，那么 25 行就是你循环长度的极限；如果你习惯于编简单代码，那么你写的循环一般不超过 15～20 行。

限制嵌套超过 3 层。研究表明，程序员理解循环的能力随循环超过 3 层后明显下降，假如你的程序循环超过三层，可以通过把部分循环写成子程序或简化控制结构的方法来缩短循环。

使长循环显得很清楚。越长越复杂。如果你的循环较短，则可以大胆地用 break 和 continue 之类的控制结构及多重出口、复杂控制终止条件等等；若你的循环较长，从考虑读者的方便起见，你的循环应当是单出口，且终止条件应当是明白无误的。

15.3　编写循环的简单方法——从里到外

有时你要编一个复杂的循环而显得很麻烦，这时你可以用下面提供的简单技巧去做。

下面是一般过程，从一种情况开始，编码时先用一些文字，然后把整个代码往后退几格，套上循环；再把文字用循环，替代掉能替代的文字，如此下去，直到完成为止。整个过程完成以后，加上必需的初始化条件。因为你是从最简单情形开始的，从里向外逐级编写，你编程时也就相当于从里到外。

假设你在为一家保险公司编程序。人寿保险费用随年龄和性别不同而不同，你的任务是写一个程序来计算一群人的人寿保险金。在程序中要用到一个循环来根据年龄不同从一个列表中取每个人的人寿保险费用，并把算得的人寿保险金加到总额中去。下面是过程：

首先，把要完成的任务写成循环体。若不考虑语法、循环控制变量、数组下标细节，这个步骤是很好写的（写成注释形式）。

从里到外写循环：第一步

```
{get rate from table}
{  add rate to total}
```

其次，把写成注释形式循环尽可能多地转换成代码。在这里是从表中取每一个人寿保险费用并加到总和上去。尽可能写成具体明确的数据而非未知数。

从里到外写循环：第二步

```
Rate := Table [ ];
Ttl := TtlRate＋Rate;
```

例子中假设 table 为数组，对应于保险费用，你不必开始就关心数组的下标，Rate 变量直接从保险费用表中取出来的保险费用。同样地，TtlRate 是计算总保险费用的变量。

第三步，给 Table 数组明确下标，

从里向外写循环：第三步

```
Rae := Table［CenCus.Age，Census.Sex］;
TtlRate := TtlRate＋Rate;
```

给出年龄、性别，就可得到相应数组元素。这里，Census．Age 和 Densus．sex 是具体的数组下标。Census 是一个结构变量，它包含每个人对应的保险费用。

第四点是在已经写出来的程序外加上循环。既然循环是从每个人的保险费计算总的保险费，那么循环控制变量就用 Person 表示。

从里向外写循环：第四步

```
for Person :=     FirstPerson to LasttPerson do
    begin
      Rate := Table[Census.Age,CenSus.Sex];
      TtlRate := TtlRate + Rate;
    End;
```

至此你给已写出的程序加上了一个循环，把循环体部分向后退了几格，并在循环中加上了一个 begin－end 时，最后发现，循环控制变量已经确定，那么依赖于这个变量的数组下标就可确定了。Census 的下标是 Person，所以把 Person 代入：

从里向外写循环：第五步

```
for Person := FirstPerson to   LastPerson   do
    begin
      Rate := Table[Census[Person].Age,Census[Person].Sex];
      TtlRate := TtlRate +Rate;
    End;
```

最后，变量初始化。这里，TtlRae 变量需要初始化。下面是总的程序：

从里向外写循环：第六步

```
TtlRate := 0;
for Person := FirstPerson to LastPerson do
    begin
      Rate := Table[Census[Person].Age,Census[Person].Sex];
      TtlRate：＝  TtlRate＋  Rate
    end;
```

如果你还要在 Person 循环外再加一个循环，照此写下去。你也不必生搬硬套这套方法。基本思想是从一个具体事值入手，一次只管一件事，由这种简单情形建立起循环。当你编写通用和太复杂的循环时，所走的步骤每次要小、要易改。这样，编代码你每次要注意的语句较少，也就减小了出错的机会。

15.4　循环与数组的关系

数组和循环经常联系在一起。许多情形下，循环就是为了处理数组而编，循环计数器和数组下标一一对应。例如，下例中的 Pascal 的 for 循环变量就和数组下标一致。

这个 Pascal 程序计算数组相乘：

```
for    Row := 1 to MaXRows    do
    for Column := 1 to MaxCols do
        Product［Row,Column］ ：= a［Row，Column］*b［Row，Column］;
```

在 Pascall 中，计算数组时要用到循环。但值得注意的是，循环结构和数组之间并无内在联系。有些语言如 APL 和 Fortran 90 提供了强有力数组计算方法，因而消除了像上面那样对循环的需要。下面是 APL 程序段

APL 程序，计算数组乘法

```
Product   <  —  a   X   b
```

这个 APL 程序简单且不会出错，它仅用了 3 种操作，以前的 Pascal 程序段则用了 15 个。APL 程序中，没有用到循环、数组下标及易使编程出错的控制结构。

以上例子说明一点，即你编写程序去解决一个问题时，有时用某种语言较简单，所用语言不同会影响你对问题的求解。

15.4.1　检查表

循环

- 循环是从顶部进入的吗？
- 循环的初始化是靠近着循环顶部的吗？
- 循环是死循环还是事件驱动循环？它的结构比诸如 for I := 1 to 9999 之类的更清楚吗？
- 是 C 的 for 循环吗？循环头包含了全部的循环控制条件了吗？
- 循环体用 begin 和 end 或类似的结构去标明以免在修改时出错了吗？
- 空循环还是非空循环？
- 把循环内务处理归结到一起了吗？放在头部还是放在结尾了？
- 循环是完成一个且仅完成一个功能吗？（像一个子程序一样）
- 循环在所有可能情况下都能退出吗？
- 循环的中止条件明显吗？
- 如果是 for 循环，在循环体内有没有改变控制变量而使循环强行退出？
- 循环体内部用一个变量保留重要循环控制变量的值，而不在循环体外引用控制变量的终值吗？
- 循环用了安全计数器了吗？
- 循环控制变量是整数类型吗？
- 循环控制变量是否有一个有含义的名字？
- 避免了控制变量的冲突没有？
- 循环短到可一目了然地步了吗？

- 循环嵌套限制在三层以内没有？
- 若循环很长，能保证它特别清晰吗?

15.5　小　结

- 循环很复杂，使其简化有利于阅读。
- 简化循环的技巧有：避免使用怪样子循环、使循环次数最小、使进出口清楚、把内务代码放在一个地方。
- 循环控制变量不可滥用，应给它起一个有含义的名字并让它只起一个用途。
- 仔细考虑一下整个循环，保证循环在各种情况和终止条件下都能照常运行。

第 16 章　少见的控制结构

目录

相关章节

经常碰到的有关控制的几个问题：见第 17 章
条件编码：见第 14 章
循环的代码：见第 15 章

在结构化程序中，有几个控制结构比较特别，它们不是典型的结构化结构，但也不是无结构可言。这种情况不是任何语言都有。

16.1　goto 语句

计算机科学家对他们的观点都热心，有许多共同点，但讨论一旦转向 goto 语句，他们之间就表现出针锋相对的两种意见了。

若一种语言不支持结构化控制，那么用 goto 来模仿起来结构化的作用时，是无人有意见的。问题出在那些支持结构化程序结构的语言中，这些语言中，理论上 goto 是不需要的，因此用了就引起很大争论。下面是两派意见的总结。

16.1.1　反对用 goto 的意见

反对用 goto 的一方总的观点是不用 goto 的代码质量高。最早提出这争论的 Edsger Dijkstra 的一句名言是:“考虑 goto 语句是有害的”。Dijkstra 经观察后认为，程序语句的质量与所用 goto 数目成反比。因此他认为不用 goto 语句的程序更容易检查错误。

含 goto 的代码不利于格式化。在有逻辑结构的程序中，往往用到了缩排形式，而 goto 则影响结构化设计。想用缩格的结构去表示含 goto 的程序很难，几乎不可能。

用 goto 影响了编译程序的优化。无条件的 goto 语句使程序流很难分析，且降低编译程序对代码的优化能力。因此，即使 goto 语句使源程序显得有效率，但在编译程序优化时却费事了。

Goto 的拥护者总是认为，goto 语句使他们在编码时既快且省。但含 goto 的程序几乎很难有最快和最短的可能。Donald Knath 在有名的文章《Structured Programming with go to statement》中给出了几个用 goto 传代码效率低，目标大的例子(1974)。

实际上，goto 的使用常破坏了结构化程序的原则，即使很仔细地用 goto 且不引起混乱，

一旦用了 goto 语句，整个程序的质量都受到影响，因此最好一个 goto 也别用。

16.1.2　支持用 goto 的意见

支持用 goto 的人认为：在某些特殊环境下使用 goto 是有好处的。大多数反对 goto 的人反对普遍用它。对 goto 的争论发生在 Fortran 成为最盛行的语言时。Fortran 没有提供很好的循环结构。若没有注意到含 goto 的循环结构性不好的话，程序员写出来的程序流就到处转移。这种代码无疑是质量很低的，但仅想着怎样用好 goto，又能有什么改进呢？因为 goto 与程序结构化能力还有那么一截的距离呢？

用 goto 语句恰到好处，可减少对同一程序段的多次复制，复制代码段引起各种麻烦：当要同时修改这些复制部时易出错，复制代码增加了源文件和执行文件的长度。在这种情况下，goto 就显得比复制代码好了。

若一个程序要求光分配资源，对资源进行处理，然后重新分配资源，这时 goto 就很有用了。用 goto 你可以在一个代码段中进行修改。当你在每个地方查错时，用 goto 你忘记重新分配资源的可能性减少了。

有时，用 goto 可使编程速度快且程序小。Kunth　1974 年的文章中就举了几个用 goto 编得很成功的例子。

编程水平高并不意味一定消除 goto。在方法上对控制结构进行分解、提炼、选择，大多数情况下会自动消除用 goto 的可能。编写无 goto 的代码不是目的，而是结果，因而着意不用 goto 是有害无益的。在一篇文章中，B.A.shell 得出结论：在现在测试条件不成熟，分析数据不充分，研究结果无说服力的情况下，还不能证明 shneiderman 和其他人所说的程序质量与所作出的结论，同样说用 goto 就是一个好方法，与反对意见一样都理由不充分。

最后要说明，Ada 语言支持 goto，它是历史上用得最仔细的工程程序语言。Ada 语言是在 goto 的争论后进行开发的，但经过仔细分析，Ada 决定保留 goto。

16.1.3　肤浅的 goto 争论

关于 goto 基本特征的争论，是较肤浅的。认为用 goto 是罪过的人，通常找到一些有 goto 的小程序段，并说明若不用 goto 而重新编写程序是多么简单容易。但这样就使人认为好像用 goto 只能编写一些小程序似的。

而认为“我不用 goto 就无法活”的人则找些例子，说明去掉 goto 导致要多写好多复制程序段。这好像证明用 goto 是为了减少程序长度似的，在今天的计算机技术中，这些已不重要了。

大多数参考书都没有什么高见。它们也只不过找了一些用 goto 的小程序然后修改成不用 goto 的代码，好像这就是问题的全部。下面是一本书上的例子：

这个 Pascal 程序被用来说明不用 goto 能写得很容易：

```
repeat
    GetData( InputFile , Data );
    If   eof( InputFile )   then
        goto      LOOP_EXIT;
         DoSomething ( Data );
until ( Data = -1 );
```

```
   LOOP_EXIT;
```

在书上随后就写出了不用 goto 替代上面程序的例子：

这个 Pascal 程序与上面那个一样，但不用 goto:

```
GetData( InputFile , Data );
While (not eof ( InputFile ) and ( Data < > -1 ) do
   Begin
   DoSomething( Data );
   GetData (InputFile , Data );
   End;
```

这个所谓的一般例子包含了一个错误。当 Data 等于－1 时，条件部分检查到－1 就退出循环，这时，DoSomething()不可能执行。而用 goto 的程序在检查到-1 前已经执行过了 DoSomething()。写有这个程序的书，本想用它来说明用结构化写程序显得多么容易。但没想到转换却带来了错误。但该书作者也不用感到害臊，因为别的书也有类似的错误。

下面这个程序没用 goto 但是正确。

这个 Pascal 程序与有 goto 的程序是同一目的，但这里无 goto 且程序正确。

```
Repeat
   GetData (InputFile , Data );
   If ( not eof( InputFile ) )   then
      DoSomething ( Data );
Until (Data = -1 or eof( InputFile ) );
```

即使程序的转换正确，但这个例子还是显得很虚假，因为它们把 goto 的用法看得太繁琐了。上述情况并不是那种思想的程序员愿用 goto 的情形。下面这种情形是常见的，即使极不愿用 goto 的程序员有时为了增强程序的可读性与可维护性，却选用了 goto。

下面几节给出了一些情况，在这种情形下，是否用 goto，在有经验的程序员那是有争议的，讨论用和不用 goto 的一些代码，最后得到一个折衷答案。

16.1.4　出错处理和 goto 语句

写交互式程序代码要做的几件事是，要特别注意出错处理和出错时要清除资源。下面这个代码要清除一组文件。程序首先读入要清除的文件组，然后找到每一个文件，覆盖掉并清除它，程序每一步都要检错。

这个 Pascal 程序用 goto 来处理出错和清除资源：

```
PROCEDURE   PurgeFiles( var   ErrorState : ERROR_CODE );
{This   routine   purges   a   group   of   files.}
var
    FileIndex :      Integer;
    FileHandle :     FILEHANDLE_T;
    FileList :       FILELIST_T;
```

```
    NumFileToPurge :    Integer;

Label
    END_PROC;

Begin
     MakePurgeFileList ( FileList ,NumFilesToPurge );
     ErrorState := Success;
     FileIndex := 0;
     While ( FileIndex < NumFileToPurge )    do
          Begin
          FileIndex := FileIndex + 1 ;
          If not FindFile ( FileList[ FileIndex ], FileHandle )    then
               Begin
               ErrorState := FileFindError;
               Goto    END_PROC                              ——这里是一个 goto
               End;
          If    not    OpenFile(FileHandle)    then
                    Begin
               ErrorState:=FileOpenError;
               Goto    END_PROC                              ——这里是一个 goto
                    End;
        If    not    OverWriteFile(FileHandle)    then
                    Begin
               ErrorState:=FileOverWriteError;
               Goto    END_PROC                              ——这里是一个 goto
               End;
          If    Erase( FileHandle )    then
               Begin
               ErrorState := FileEraseError;
               Goto    END_PROC                              ——这里是一个 goto
               End
          End;{while}
    END_PROC;                                                ——这里是 goto 的标号

    DeletePurgeFileList( FileList ,NumFilesToPurge )
```

```
End;  {PurgeFiles}
```

这个程序是那种有经验程序员肯定要用 goto 的情形。同样的，当程序要分配和清除资源时（像内存、或处理字形、窗口、打印机），也要用 goto。这种情形下用 goto 通常是为了复制代码或清除资源。若遇到这种情况，程序员就要掂量是 goto 的缺点令人讨厌呢？还是复制代码那令人头痛的维护更讨厌呢？最后还是认为 goto 的缺点更可忍受。

可以用许多方法重新编写上述程序而不用 goto，把两种比较一下。下面是不用 goto 的方法：

用 if 嵌套重新写程序。用嵌套的 if 语句重新写程序时，嵌套 if 语句是为了在上一个条件满足后才进到这一层嵌套的。这是不用 goto 的标准的、书本式的结构化编程方法。下面用标准的方法重新编程。

这 Pascal 代码用 if 嵌套来避免用 goto:

```
PROCEDURE PurgeFiles( var ErrorState : ERROR_CODE );
{This routine pruges a group of files.}
var
   FileIndex :        Ingeter;
   FileHandle :       FILEHANDLE_T;
   FileList :         FILELIST_T;
   NumFilesToPurges:  Integer;
begin
   MakePurgeFileList( FileList , NumFilesToPurge );
   ErrorState   := Success;
   FileIndex    := 0;
   While ( FileIndex < NumFilesToPurge   and   ErrorState = Success ) do
      begin
      FileIndex   :=   FileIndex + 1 ;
            If   FindFile ( FileList[ FileIndex ] , FileHandle ) then
                 Begin
                 If OpenFile ( FileHandle ) then
                      begin
                      If OverWriteFile ( FileHandle ) then
                           Begin
                           If not Erase ( FileHandle ) then
                                Begin
                                ErrorState := FileEraseError
                                End
```

这个 While 已经改为增加测试错误状态

```
                    end
                else {couldn't overwritefile}
                    begin
                    ErrorState := FileOverWriteError
                    end
            end
                else {couldn't open file}
                    begin
                    ErrorState := FileOpenError
                    end
        end
        else   {couldn't find file}
            begin
            ErrorState := FileFindError      ——这行与调用它的语句距离有 23 行
            end
    end;   {While}
    DeletePurgeFileList( FileList ,NumFilesToPurge )
end;   {PurgeFiles}
```

习惯于编程不用 goto 的人对这段代码可能看得很清楚。如果你把程序写成了上面例子这样，那么在读时你就不必担心由 goto 带来很大跳跃了。

用 if 嵌套的主要弊病是嵌套层次太多、太深。为了读懂代码，你得同时把所有嵌套的 if 都记在脑中。而且处理出错的代码距引起它的代码距离太远：如在上例中，ErrorState 到 FileFindError 的距离是 23 行。

用 goto 的程序，发现错误的语句离处理出错的语句都不超过四行，而且你无需同时把整个结构都放在脑中，你尽可把不成立的条件置之不理而集中精力于下一步操作。由此看来，在这种情况下 goto 倒是更可读与易维护了。

重新编程时可调一个状态变量。定义一个状态变量以指示程序是否处于错误状态。上例中已经用到了 ErrorState 这个状态变量，因而下面还用它：

这个 Pascal 代码设置状态变量以免用 goto:

```
PROCEDURE PurgeFiles( var ErrorState : Error_CODE );
{This routine pruge a group of files.}
var
    FileIndex :          Integer;
    FileHandle :         FILEHANDLE_T;
    FileList :           FILELIST_T;
    NumFilesToPurge :    Integer;
```

```
begin
   MakePurgeFileList ( FileList , NumFilesToPurge );
   ErrorState    := Success ;
   FileIndex     := 0;
   While ( FileIndex < NumFilesToPurge ) and ( ErrorState = Success ) do
   begin
         FileIndex := FileIndex + 1;
         if not FindFile( FileList[ FileIndex ] , FileHandle ) then
            begin
            ErrorState := FileFindError;
            end;
        if ( ErrorState = Success ) then                    ——    测试这个状态变量
            begin
            if not OpenFile( FileHandle )    then
                  begin
                  ErrorState := FileOpenError
                  end
            end;
            if ( ErrorState = Success ) then                ——    测试这个状态变量
                  begin
                  if    not OverwriteFile( FileHandle ) then
                        begin
                        ErrorState := FileOverwriteError
                        end
                  end;
            if ( ErrorState = Success ) then                ——    测试这个状态变量
                  begin
                  if not Erase( FileHandle ) then
                        begin
                        ErrorState := FileEraseError
                        end
                  end
            end;   {while}
            DeletePurgeFileList( FileList , NumFilesToPurge )
end; {PurgeFiles}
```

这个 While 测试已经增加了一个 ErrorState

设置状态变量的好处是避免 if-then-else 结构的嵌套层次太深，且易于理解。这种方法也使跟在 if-then-else 条件后的实际操作语句离测试条件更近，且完全不用 else 语句。

为理解嵌套 if 语句是需动一番脑筋的，而设置状态变量的方法则易于理解，因为它接近于人的思维方式。你要先找到文件，如果无错，就打开文件；如果还不出错，覆盖文件；如果还不出错...。

这种方法的不足之处在于状态变量并不如所想象的那么好，使用状态变量时你要表示清楚，否则读者不能理解变量是什么意思。在上例中，用了有含义的整数类型的状态变量相当有帮助。

各方法比较。三种方法各有优点。Goto 方法避免嵌套太深和不必要的条件测试，但同时也有 goto 固有的弊病；嵌套的 if 方法避免用 goto 但嵌套深且增大了程序的复杂性。设置状态变量法避免用 goto 且不会使嵌套太深，但增加了额外的条件判断。

状态变量法相对前两个好些。因为它使程序易读且使问题简化，但它不可能在所有情况下都好用。从整体上来讲，这三种方法在编程时都能用得很好，这时就需要全盘考虑，权衡利弊，选择最好的方法。

16.1.5　goto 和 else 语句中的共用代码

一种挑战性的情况是，若程序有两个条件测试语句和一个 else 语句，而你又只想执行一个条件语句中的代码和 else 中的部分代码，这时就得用 goto 语句，下面这个例子是迫使你用 goto 的情况；

这个 C 语言程序用 goto 转向执行 else 中的共用代码：

```
if ( StatusOK )
{
    if ( DataAvail )
    {
        ImportantVar = x;
        Goto MID_LOOP;
     }
}
else
    {
          ImportantVar = GetVal( );
          MID_LOOP ;
      /* lots of code */
      ……
   }
```

这个程序的高明之处在于它用了一个逻辑的迂回－它可不像你所见的那么好读，但若不用 goto 你很难再写出另外的样子出来。如果你认为你能很容易地不用 goto 就写出新程序来。那么让别人检查看看。好些有经验的程序员都写错了。

改写的方法有几种，你可以复制代码、把共用的代码放在一个子程序里面，且从两个地方调用它、或重新写条件测试语句。在多数语言中，改写并不比写原程序快，虽然应当是一样快的。除非循环在程序中多次使用，否则编写时不用考虑效率。

改写最好的方法莫过于把/*lots of code*/部分放在一个子程序里。你可以在代码原来出现的地方和 goto 语句转向的地方调用它而保留原结构的条件语句。下面是程序：

这个 C 程序把 else 中的共同程序部分写成一个子程序

```
if ( StatusOK )
{
    if ( DataAvail )
        {
        ImportantVar = x;
        DoLotOfCode( ImportantVar );
        }
}
else
   {
    ImportantVar = GetVal( );
    DoLotsOfCode( ImportantVar );
    }
```

通常，写一个新子程序（C 中可用宏定义）是最好的方法。但有时把代码放进一个子程序中实际是不可能的。这时只能修改原结构中的条件部分而无需把共用部分放到一个子程序中。

这个 C 程序用 else 中的共用代码替换 goto:

```
if ( ( StatusOK && DataAvai ) || !StatusOK )
    {
    if ( StatusOK && DataAvail )
        ImportantVar = x;
    Else
        ImportantVar = GetVal( );
    /* lots of code */
    ……
    }
```

这种转换方法不会出错但很机械。程序意义虽然一样，但却多出来的两个地方测试 StatusOK 和一个地方测试 DataAvail。使得条件测试显得很麻烦。注意：第一个 if 条件中 StatusOK 值不必测试两次，你也可以把第二个 if 的条件中对 DataAvail 的测试减少。

16.1.6　使用 goto 方法的总结

用 goto 是一个个人爱好的问题。我的意见是，十个 goto 中有九个可以用相应的结构化结构来替换。在那些简单情形下，你可以完全替换 goto,在复杂情况下，十个中也有九个可以不用：你可以把部分代码写成一个小的子程序调用；用嵌套 if 语句；用状态变量代替；或者重新

设计控制条件的结构，消除 goto 是很难的，但它却是很好的脑力活动，前面讨论过的方法会对你有所帮助。

如果 100 个用 goto 的情形中有一个靠 goto 很好地解决问题的方法，这时你要把它用得好些。只要问题能解决，我们是不约束用不用 goto 的，但应当注意，最好还是少用或不用 goto 编程，因为有些问题你可能还没弄清楚。

下面对用 goto 的方法作一总结：

- 若语言不支持结构化语句，那就用 goto 去模仿结构化控制结构，但一定要写得正确。不要把 goto 灵活性用得出了格。
- 能用结构化结构的地方尽量不用 goto.
- 评价 goto 的性能的方法是看其是否提高效率，在多数情况下，你可以把 goto 替换而增加可读性且不失效率。在少数例外情况下，用 goto 确实有效且不能用别的方法来代替。
- 每个程序至多用一个 goto 语句，除非你是用它来模仿结构化的结构。
- 使 goto 语句转向循环前面而不要往后转移，除非是在模仿结构化结构中。
- 保证 goto 转向的语句标号都要用上，若有没有用到的，则表明掉了部分代码，或该转向那部分代码没有标上标号。所有给出的标号都要用上，若没用上，就去掉。
- 应保证 goto 语句不会产生执行不到的代码。
- 用不用 goto 应全面考虑，程序员对用不用 goto 考虑后，选择了 goto，那么这个 goto 可能是好的选择。

16.2　Return 语句

return 和 exit 语句都属控制结构语句，它们使程序从一个子程序中退出。它们使子程序从正常出口退出到调用它的程序中去。这里 return 泛指有类似作用的一类词：return、exit 和相似的结构，下面说明如何使用 return 语句。

减少每个程序中的 return 语句。如果你在看一个子程序的后部分，而又不清楚在前面是否有 return，那么就很难读懂这个程序。

用 return 增强可读性。有些子程序中，一旦你知道了答案，就想退回到调用它的程序中去，如果子程序不需要清除，那么不立即退出意味着要执行别的代码。

下面这段程序是个较好的例子。它从子程序中多个地方退出，满足多种情况。

这个 C 程序是较好地从子程序中多个地方退出的例子：

```
int Compare
    (
        int   Value1,
        int   Value2
    )
{
 if ( Value1 < Value2 )
        return( LessThan );
```

```
    else if ( Value1 > Value2 );
            return( GreaterThan );
    else
            return( Equal );
}
```

16.3　递归调用

在一个递归调用中，子程序本身只解决很小一部分问题，而把问题分成许多小片段，然后调用自己，来解决比本身更小的片段。递归调用一般在这种情形下，即问题的一小部分，但整个问题很复杂。

递归调用不常用，但若用好了，它能解决其他方法不好解决的大问题。下面是递归解决排序算法的例子。

这个 Pascal 程序用递归解决排序算法：

```
Procedure QuickSort
    (
    FirstIdx :      Integer;
    LastIdx :       integer;
    Names :         NAME_ARRAY
    );
var
   MidPoint :   integer;
begin
    if ( LastIdx > FirstIdx ) then
        begin
        Partition( FirstIdx , LastIdx , Names , MidPoint );
        QuickSort( FirstIdx , MidPoint – 1 , Names );
        QuickSort( MidPoint + 1 , LastIdx , Names );
        end
end;
```

这里是递归调用

这个程序中，排序算法把一个数组分成两部分，然后调用自己再去排序那两个半个的数组，如此下去，直到不能再分为止。

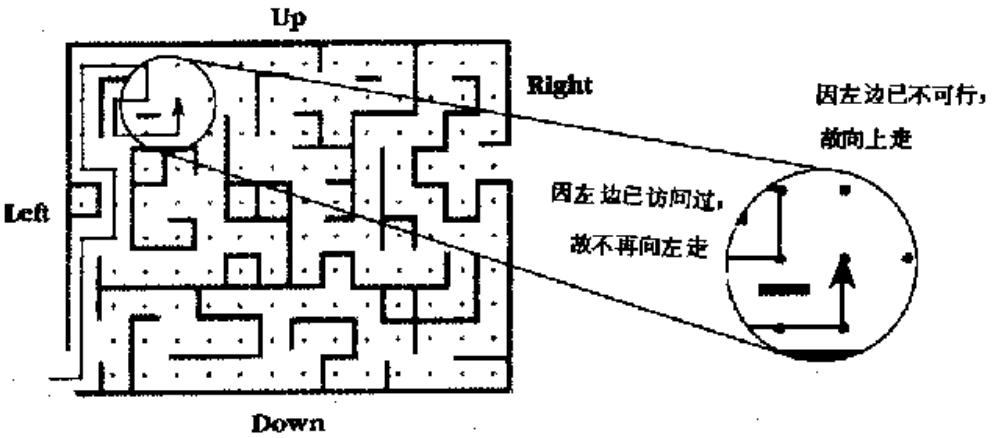
一般说来，递归调用代码较短，执行速度慢，占用堆栈空间大。若问题较小，递归调用可得到简单、巧妙的解；对稍大的问题，它们产生简单、巧妙、难于理解的解。对大多数问题，递归调用得到很大的复杂解-在这种情形下，简单的循环或许更好理解。用递归要有选择。

16.3.1　递归调用的例子

假设你用一个数据结构代表一个迷宫。一个迷宫就是一个网格，在网格的每一点上你可以向左转，也可向右转，可以向上，亦可向下。在每一点上，你有不止一种选择。

那么你如何写一个程序去解决这个问题即通过迷宫呢？若用递归调用，问题相当直观。从

起始点开始，试走各种可能的路径，直到最后走出迷宫。第一次你走到一个点，试着往左：若不能往左，就试着向上或向下；若都不行，那就往右，你不用怕迷路，因为每过一点，你作下标记，因此你不会从同一个地方走过两次。



下面是用递归写的代码：

用递归写过迷宫的 C 程序

```
BOOLEAN function FindPathThroughMaze
      (
      MAZE_T MAZE ,
      POINT Position
      )
{
/* if the position has already been tried,don't try it again */
if (alreadyTried(Maze,Position))
      return(FALSE);

/* if this position is the exit,declare success */
if (ThisIsTheExit(maze,Position))
      return(TRUE);

/* remember that this position has been tried */
RememberPosition(Maze,Position);

/* check the paths to the left,up,down,and to the right;
   if any path is successful,stop looking */
if ( MoveLeft( Maze , Position , &NewPosition ) )
     if ( FindPathThroughMaze ( Maze , NewPosition ) )
        return( TRUE );

if ( MoveUp( Maze , Position , &NewPosition ) )
```

```
        if ( FindPathThroughMaze ( Maze , NewPosition ) )
            return (TRUE);


    if ( MoveDown ( Maze , Position , &NewPosition ) )
        if ( FindPathThroughMaze ( Maze , NewPosition ) )
            return (TRUE );


    if ( MoveRight ( Maze , Position , &NewPosition ) )
       if ( FindPathThroughMaze ( Maze , NewPosition ) )
          return ( TRUE);


    return( FALSE );
    }
```

代码的第一行检查是否已经走过那点。写递归程序要防止无限递归调用。在上例中，若不检查是否走过，那可能产生无限递归。

程序第二段检查这一点是否是通向迷宫出口的。如果 ThisIsTheExit()返回 TRUE(真)，那么子程序也返回真。

第三句记录下这一点已经走过。这样防止走入循环路径而导致无限递归调用。

下面各行找出各左、上、下、右的路径。在找出通过迷宫的路径以后，程序返回 TRUE。

这个程序的思想很直观。许多人觉得递归调用自己而感到不舒服，但在本例中，其它方法肯定都复杂而递归调用是很好的。

16.3.2　怎样用递归

下面是用递归的几点建议：

要保证递归能终止。检查程序确保含有一个不再用递归调用的路径，这通常表明程序已经满足条件不再需要递归了。在迷宫例子中，条件测试 AlreadyTried()和 ThisIsTheExit()保证递归调用停止。

设置安全计数器防止无限递归。如果不许用简单的条件测试，如上例所示，要防止无限递归，那就应设安全计数器。安全计数器应当是每次递归时不能改动的变量，可用一个全局变量或把安全计数器作为程序的参数。

下面是例子：

用安全变量防止无限递归的 Pascal 程序：

```
Procedure RecursiveProc( var SafetyCounter : integer );
/*这个递归程序必须可以改变 SafetyCounter 的值，它在其中是一个 Var 参数*/
begin
    if ( SafetyCounter > SAFETY_LIMIT ) then
        exit;
    SafetyCounter : = SafetyCounter + 1;
    ……
end;
```

在上例中，如果超出安全计数器的范围，则停止递归调用。

把递归调用限制在一个子程序里。循环递归调用(A 调用 B，B 调用 C，C 调用 A)是很危险的，因为程序很难检查。在一个程序内的递归调用就够麻烦的，要理解程序间的递归调用太难。如果你的程序用了循环递归，那就修改代码，使递归在一个子程序中。如果你不能修改且认为这种递归是最好的方法，那就一定要设置安全计数器。

要注意堆栈。写递归调用无法计算程序要用多少堆栈空间，而且也无法事先预测程序是如何运行的。你可采取几个步骤来控制程序运行时的行为。

首先，若用了安全计数器，在设置安全计数器的最大值时，要考虑到可能分配给你多大的堆栈空间。安全计数器的最大值要限制在不使堆栈溢出的范围内。

其次，你可估算一下，在运行递归程序时要用多少内存。在运行之前，用可辨识的值充满相应内存；0 和 0xCC 是一个很好的值，把程序编译一下，然后用 DEBUG 去看对应的内存，看看程序占用了多少堆栈空间。查查 0 和 0xCC 没有被改变的点。然后用这点来估计你程序占用的堆栈。

不要用递归去计算阶乘或菲波那契(fibounacci)数。计算机教科书上经常出现的一个问题是给出了许多不太好的递归调用例子。典型例子是计算阶乘或计算 fibounacci 数列。递归调用是一个强有力的工具，但用在这里都显得很笨。假如我请人给我编一个计算阶乘的程序，而他用递归调用算法，那我宁愿换个人。下面是用递归调用计算阶乘的程序：

这个 Pascal 程序用递归调用算阶乘不太好：

```
Function Factorial(Number:integer):integer;
begin
    If ( Number = 1 ) then
        Factorial := 1
    Else
        Factorial := Number * Factorial ( Number – 1 );
end;
```

除了速度慢，计算机用到的内存难以估计外，递归调用在这个程序中要比用循环难懂多了。下面用循环编程序：

这个 Pascal 程序用循环来计算阶乘比较好：

```
Function Factorial( Number : integer ) : integer;
var
  IntermediateResult :   integer;
  Factor :               integer;
begin
    IntermediateResult := -1;
    For Factor := 2 to Number do
        IntermediateResult := InterMediateResult * Factor ;
  Factorial := IntermediateResult
end;
```

你可以从这件事得到三点教训。第一，计算机教科书的这些所谓的例子对于说明递归的作

用没有一点好处；第二点也就是最重要的一点，递归调用的功能要比用来计算阶乘和 fibonacci 数列强大得多。有时你可用堆栈加循环做递归调用能做的同样的事情。有时用这种方法好，有时另一种更合适，你得仔细权衡选用一种。

16.3.3　检查表

少见的控制结构

goto

- goto 是最后的选择吗？用 goto 使程序更好读更好维护吗？
- 用 goto 是为效率的目的吗？用 goto 达到此目的了吗？
- 一个程序是否只用一个 goto 呢？
- Goto 只转向前面的程序段而不是转向其后面的程序段吗？(后面指已执行过程序)
- Goto 所转向的标号都有了吗？

return

- 每个子程序的 return 数目是否最少？
- Return 增强了可读性了吗？

递归调用

- 用递归调用的代码含使递归结束的语句吗？
- 程序设置了安全计数器来保证递归调用终止了吗？
- 是否只在一个程序中用递归调用？
- 递归调用的深度是否限制在程序堆栈容量可满足的条件下。
- 递归调用是实现程序的最优途径吗？它比循环更简单吗？

16.4　小　结

- 有些情况下，goto 是编出易读易维护程序的最好方法。
- 多重 return 有时增强了程序的可读性与可维护性，并且防止多重嵌套逻辑，但没必要只想到怎样用好 return。
- 在问题较简单时，递归调用能把问题很巧妙解决。要慎用递归调用。

第十七章　常见的控制问题

目录

相关章节

条件代码；见第 14 章
循环的代码：见第五 5 章
少见的控制结构：见第 16 章

不讨论在编写控制结构时碰到的几个问题，那么关于控制的任何讨论都是不完全的。这一章所讲的东西是细节性的、实用性很强的。若你想看有关控制结构的理论，那就清集中精力看 17.5 节关于“结构化编程的作用”和 17. 7 节中关于“对控制结构和复杂性之间关系”的研究好了。

17.1　布尔表达式

除了那些按顺序往下计算的最简单控制结构，几乎所有的结构都依赖于对布尔表达式的计算。

用 True 和 False 作为布尔变量

用 True 和 False(真和假)作为布尔表达式结果的标识符，而不要用 0 和 1。像 Pascal 之类的语言有布尔型变量来支持定义 True 和 False 作为标识符。要弄清楚，对布尔型变量，你还只能用 True 和 False 来给它赋值而不能用其它的，对那些没有布尔型变量的语言，你需用一些规则来使布尔表达式易读。如下例子；

这个 Basic 程序任意定义布尔变量的值：

```
1200 IF PRINTERROR = 0 GOSUB 2000      ' initialize printer
1210 IF PRINTERROR = 1 GOSUB 3000      ' notify user of error
1230 '
1240 IF REPORTSELECTED = 1 GOSUB 4000    ' print report
```

```
1250 IF SUMMARYSELEEED ＝　I GOSUB 5000　'print summary
1260 '
1270 lF PRINTERROR = 0 G0SUB 6000　　'clean up successful printing
```

如果类似 0 和 1 这样的标志很普遍的话，那会有什么错呢？问题是当测试条件为真还是假的时候程序应当执行 GOSUB 吗？不清楚。在程序段中没有什么规则来说明是“1”代表真，“0”代表假。正好相反，甚至“1”和“0”是否表示“真”和“假”的意思都弄不清楚。比如在 IF REPORTSELECTED = 1 这一行，“1”很容易代表第一个记录，“2”代表第二个，“3”代表第三个，在这个代码中，并没有什么标准说明”1”表示的是真还是假，同样，对“0”也是如此。

用 True 和 False 之类的词来表示布尔表达式的结果，如果所用语言不支持这种类型，用预定义宏或全局变量的方法来创造一个。下面的例子用全局变量 True，False 来重新编写：

这个 Basic 程序用全局变量（True 和 False）来表示布尔变量的值：

```
110 TRUE = 1
110 FALSE =0
 ……
1200 IF PRINTERROR = FALSE GOSUB 2000　　' initialize printer
1210 IF PRINTERROR = TRUE GOSUB 3000　　' notify user of error
1230 '
1240 IF ERPORTSELECTED = FALSE GOSUB 4000　' print report
1250 IF SUMMARYSELECTED = TURE GOSUB 5000　' print Sllttifi13Yy
1260 '
1270 IF PORINTERROR = FALSE GOSUB 6000　　' clean up successful printing
```

用 True 和 False 作变量名使得用意很清楚。你不用记“1”和“0”表示什么意思，也不用偶尔回头检查。而且在重新编写了以后发现，原程序中的那些“1”和“0”并不作为布尔量的标志。IF REPORTSELECTED = 1 这一行根本不是一个布尔测试表达式，它只是检查第一个记录被选中了没有。

这种方法可告诉读者这一行是用作布尔测试目的。你不太可能用 TRUE 来表示 FALSE（假）的意思，但把“1”当作“0”的意思却有可能，而且用 True 和 False 还可避免那令人眼花缭乱的“0”和“1”在程序中到处出现的。下面几点意见对在布尔测试条件定义 True 和 False 很有帮助。

在 C 中用 1 == 1 的形式定义 TRUE 和 FALSE。在 C 中，有时很难记住是否 TRUE 等于 1 和 FALSE 等于 0 或者正好相反，你得记住测试 FALSE 和测试空终止符或其它零值一样。否则用下面定义 TRUE 和 FALSE 的方法来避免出现这个问题。

这个 C 程序用易于记住的方法定义布尔量:

```
# define　TRUE　（1 ==1）
# define　FALSE（!TRUE）
```

隐含地把布尔空里与 False 比较。如果所用语言支持布尔变量，把测试表达式当作布尔表达式来看待显得清楚。比如：

```
while（ not Done）…
while（ a = b）…
```

用上面的式比用下面的清楚：

```
while（ Done＝ False）…
while ( (a = b) = True ) …
```

用隐含的比较方式可减少测试条件语句中词的个数，读程序时可少记好多东西，因此表达式读起来就简单些。

如果使用的语言不支持布尔变量，你就得去模仿。你也可能有时不能用这种技巧，因为在有些语句像 while（not Done）之类语句，不能模仿 True 和 False 用来作检测。

使复杂的表达式简单些

采用以下几步来简化表达式：

把复杂的测试条件用中间的布尔交量变成几个部分。宁愿定义几个奇怪的中间变量并给它们赋值，这样可编写简单的测试条件。

把复杂的表达式写成一个布尔型函数。如果测试条件要经常重复用到或很分散，把这部分代码写成函数。如下例，测试条件很复杂。

这个 PaSCal 程序的测试条件很复杂：

```
if ( ( eof ( InputFile ) and ( Not InputError ) ) and
        ( ( MIN_ACCEPTABLE_ELEMENTS_C＜CountElementsRead )   and
                （CountElementsRead＜= MAX_ELEMENTS_ C)   ) or
        （ Not ErrorProcessing）
        ）then


        { do something or other }
        …
```

如果你对测试条件部分不感兴趣的话，那要你读这个程序真是一件可怕的事情。把这部分写成一个函数，就能把条件部分独立起来，除非读者感兴趣，否则可以忽略这部分。下面这个程序给出了怎样把 if 的条件部分写成一个函数：

这个 Pascal 程序把测试条件部分写成一个布尔型函数：

```
    Function FileReadError
    （
      var   FILE        InputFile;
            Boolean     InputError;
            Integer     CountElementsRead
    ): Boolean;
  begin
  if ( ( eof ( InputFile ) and ( Not InputError ) ) and
    ( ( MIN_ACCEPTABLE_ELEMENTS_C＜CountElementsRead ＝ and
      ( CountElementsRead＜ = MAX_ ELEMENTS _C＝ ＝ or
      ( Not ErrorPocessing )
    ）then
    FileReadError := False
```

```
else
   FileReadError:=True;
end;
```

上例中把 Error_Processing 定义为一个标志现在过程状态的布尔型函数。现在，当你读整个主程序时，可不必去管复杂的测试条件部分：

这个 Pascal 的主程序没有复杂的测试条件：

```
if（ not FileReadError（ InPutFile， InPutError， CountElementsRead） then
    { do something of other }
    …
```

如果测试条件仅用一次，你可能认为没有把这部分写成一个子程序的必要。但把这部分编成一个子程序并给出一个合适的名字，那么不仅能增大可读性，而且让你一看就明白这部分代码的作用，因而很有必要这样做。

用决策表代替复杂的条件。有时复杂的测试条件涉及几个变量。若用决策表去编这个测试条件部分则显得比用 if 或 case 好。一个决策表使得开始编程时很容易，因为它仅需几行代码而且不需什么复杂的控制结构。这种减小复杂性的方法会降低出错的机会。若要修改数据，仅需修改决策表而无需修改程序本身，仅需更改数据结构的内容。

17.1.3　编写肯定形式的布尔型表达式

不少人对较长的否定形式的表达式理解起来很困难。也就是说大多数人对否定太多的句子来困难。为避免出现复杂的否定形式布尔型表达式，你可依从以下几点：

在 if 语句中，把条件从否定形式转化为肯定形式，再把 if 和 else 语句后跟着的代码对换。如下例所示：

这个 Pascal 程序乱用否定形式的测试条件

```
if（ not Status0K ） begin
   {do something}
   …
end
else begin
    {do something else}
     …
end;
```

你可以把程序改为肯定形式表达式：

这个 Pascal 程序用肯定形式的布尔型测试条件显得很直观：

```
if（ StatusOK ） begin      ——这个测试条件转换为肯定形式
   {do somthing else}       ——这个模块中的代码已经转换
   …
end;
else      begin
```

```
{do something}
    …
end;
```

这一段代码与前一段代码实际上是一回事，但比前一个好读，因为把否定形式的表达式转换成了肯定形式的表达式。

当然，你可以选用不同的变量名，但其中一个要与真值的意思相反。在上例中，可用 ErrorDetected 替换 StatusOK，它在 StatusOK 错误时表示正确的意思。

用 DeMorgen 定律去简化否定形式的布尔型测试条件。DeMorgen 定律揭示了在取反时一个表达式与另一个表达式之间的关系。比如下面这个代码段；

否定形式测试条件的 Pascal 程序：

```
if（ not Display0K　or　not Printer0K ）　then …
```

在逻辑上它与下面这段代码相等

这个 Pascal 程序应用了 DeMorgen 定律简化：

```
if（（ not Display0K　and　Printer0K ））　then…
```

这里你无需对调 if 和 else 语句后的可执行代码。上述两个表达式在逻辑上是一致的。把 DeMorgen 定律应用于逻辑运算 and 或 or 或其它运算时，你把每个运算取反，对换 and 和 or，然后整个表达式取反，表 17－1 归纳了 DeMorgen 定律各种可能的转换。

表 17－1　应用 DeMorgen 定律转换逻辑表达式

初始表达式	相应表达式
not A and not B	not(A or B)
not A and B	not(A or not B)
A and not B	not(not A or B)
A and B	not (not A and not B)
Not A or not B	not(A and B)
Not A or B *	not(A and not B)
A or not B	not (not A and B)
A or B	not (not A and not B)

*例子中已用到

17.1.4　用括号使布尔型表达式清晰

如果布尔型表达式较复杂，用括号使表达式意思更明晰，而不能仅依靠语言运算的顺序。读者并不了解语言是怎样去运算布尔型表达式的，因此用括号可解决这个问题，读者无需知道内部细节。如果你是一个聪明的程序员，你不必依赖自己和读者来记运算优先级，特别是几种语言混用时，用括号不像打电报，你无需为多写的字符付出代价。

下面这个例子没有适当多用括号：

这个 C 程序表达式中少许多括号。

```
if　( a< b = =　 c = =d ) …
```

这个程序一开始就被那个条件表达式弄得昏头转向，但更令人难以捉摸的是不清楚条件表达式的意思是 (a< b)= =(c = =d) 呢还是((a<b) = = c)　= = d 下面的表达式虽然还是不太

清楚，但加上括号则好理解多了。

这个 C 程序用括号比较好：

```
if（(a<b）= =（c= =d)）…
```

这个例子中，括号提高可读性和程序的正确性——因为在前一个例子中，编译程序并非像这个程序这样来解释表达式。若不太清楚，加上括号。

用下述技巧来使括号数平衡。如果括号太多你一时无法知道正括号和反括号是否一样多，用下面的技巧来解决。开始时置“0”。沿着表达式从左到右，遇到一个开括号置“1”，每次遇到一个开括号把数加 1，每遇到一个闭括号把数减 1，如果最后数目回到“0”，则括号数保持平衡。

括号数保持平衡的 Pascal 例子；

读括号：if (((A＜B＝＝（c＝d)) and not done）…　？

```
if   (((A＜B) ＝ (C＝D)) and not done）…
     |||    |    |    ||              |
    0123    2    3    21              0
```

在这个例子中，结果为 0，因此正反括号数持平。在下例中，正反括号数不一样多：

这个 Pascal 例子中正反括号数不一样多：

读括号: if ((A＜B) ＝ (C＝D)) and not Done）…

```
    if ((A＜B) ＝ (C＝D)) and not Done）…
       ||    |    |    ||              |
     0 12    1    2    10             -1
```

若“0”在一行的中间出现，它就暗示前面掉了括号。也就是说，不到末尾它不能出现。

17.1.5　了解布尔型变量是怎样运算的

许多语言隐含着一种怎样运算布尔型变量的规则，有些语言中，编译程序先计算各分量的值，然后合起计算整个式子的值，另一些语言中编译程序采用“短路”或“懒惰”算法，即只计算所需部分的值。这在有时只需计算第一个测试时就可得出结果的情形下很有用，因为第二个测试已不需再算了。例如为了检查一个数组的元素，如下编程：

一个查错的伪程序：

```
while（ i<= MaxElements and item[i]<> 0）  do …
```

假如算了整个式子，在最后一次循环时就会出错，当变量 i 等于 MaxElements＋1 时，表达式 item 目等于 item [MaxElements＋1]，这个数组下标有错误（超出维数）。或许你要说你仅看数组的值，不改变它，也就无所谓。这种说法在 Macintosh 和 MS－DOS 操作系统中是正确的，但若操作系统是一个保护模式，像 Microsoft Windows 或 OS / 2，你可能就改变了保护性。在 Pascal、basic 中也是这样。

可重写测试条件，避免出错。

这个 Pascal 程序测试条件正确：

```
while（ i<= MaxElements）  do
      if（ item[i] <> 0）  then
          …
```

这个程序正确是因为除非 i 小于或等于 MaxElements，否则不计算 item[i]。

许多高级的语言提供在前部分防止这类错误的规则。比如，C 和 Pascal 用短路法计算，如果第一个运算 and 是假，那么第二个运算不再执行，因为整个式就已经是假了。也就是说，在 C 和 Pascal 中。

if something False and somecondition

被计算的部分是 if Something False。当 Something 被证明为假时，整个计算停止。

短路计算类似于 or 操作。在 C 和 Pascal 中，

if something True or somecondition

假如 if somethingTrue 是真，那么就仅执行这一部分，整个计算停止。正是考虑到这种方法，下面的语句较好而合法的。

用短路法计算测试条件的 Pascal 例子：

if （(Denominator 〈〉0） and （Item / Denominator＞MinVal)）do…

当 Demominator 等于 0 时，整个表达式计算会出现被 0 除错误。但既然当第一部分为真（不等 0）时第二部分才被计算，因而不会出现 Denominator 被零除的情形，也就不会出被零除的错误。

另外，既然 and 操作是从左到右运算的，下面的语句可能出错；

这个 Pascal 程序不能用短路法避免错误；

if（(Item / Denominator<MinVa) and （ Denminator<>0 ） do …

在这个例子中，Item / Denominator 在判断 Denominator〈〉0 之前运算，所以程序会出现被零除错误。

不同的语言采用不同的运算方法，而语言设计者又倾向于擅自采用所爱好的表达算法。因此要查你所用语言是何种算法，就得查阅相应版本的手册。但既然读者对这方面理解得不如你那么深，那就用括号标明你的意图，而不仅仅依赖于运算顺序和短路运算法。

17.1.6　按数轴上的顺序编写数字算式

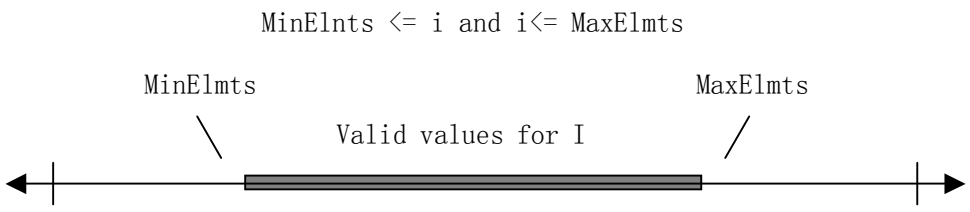
按数轴的顺序组织数字运算。一般说来，仔细组织程序的数字测试条件，以便于比较，如下

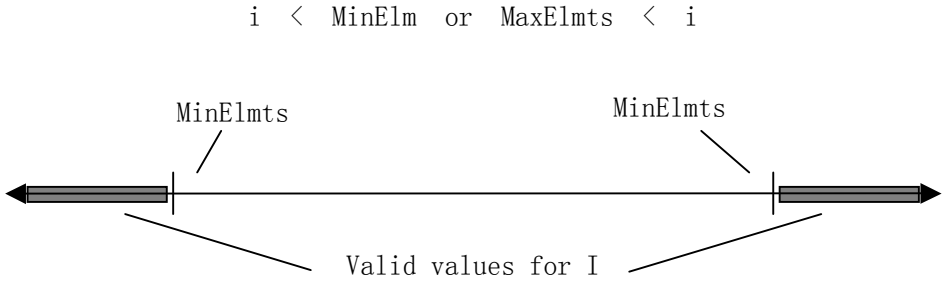
例：

MinElmts <= i and i<= MaxElmts

I < MinElmts or MaxElmts < i

这种思想是按从小到大的顺序从左到右安排各部分。上例中，第一行 MinElmts 和 MaxElmts 是两个端点，因此把它放在这句的两端。变量 i 假设处于这两者之间，因此放在句子中间。第二句的目的是检查 i 是否超出范围，因此 i 放在句子的两端而 MinElmts 和 MaxElmts 放在里面。这种方法一下子就勾画出你作比较的意图。





如果仅把 i 与 MinElmts 比较，i 的位置随条件的不同而变化，如果 i 被认为是小些，你得这样写比较语句：

while（i<inElmts）…

但如果 i 被假设为要大些，得这样写：

while（minElmts<i）…

这种写法比下面的写法清楚：

i>MinElmts and I<MaxElmts

这种写法不能给读者任何启示。

17.1.7　C 语言中与 0 比较的用法

C 语言中 0 有几个用途。它是一个数字量；在字符串中它是一个结束符（’ \0’），它是地址指针所允许的最小值；在逻辑表达式中它表示假，因为它有这么多的用途，你在编程时要清楚地表明是哪种。

隐含地和逻律变量比较。正如前面提到，这样写逻辑表达式是正确的。

while（！ Done）

这个表达式隐含在和 0 作比较是正确的，因为这个比较是在一个逻辑表达式里。

数与 0 比较。虽然可隐含地把逻辑表达式与 0 比较，你却不可以把数字表达式隐含地与 0 比较。对于数字，编程：

while（Balance!=0）…

而不可写成

while（Balance）…

把字符与结束符（’\0’）字一样，字符也不是逻辑表达式。因此对字符编程：

while（* CharPtr!=’0’）…

而不能写成:

while (* CharPtr) …

这种说法与常用的 c 语言在处理字符数据时的习惯可能有出入，但它强化了这种观点，即表达式是在处理字符数据，而非逻辑数据。C 语言的有些用法并不是基于可读性和或维护而设计的，比如上例即是。

把指针与 Null(空指针)比较。对指针编程：

while（BufferPtr!=Null）…

而不写成：

while（BufferPtr）…

与字符的情形一样，这也不是 C 的常用用法，但却增强可读性。

17.1.8　布尔型表达式中的几个常见问题

在用 C 语言写布尔型表达式时常会碰到几个问题。在 C 中，互换位操作与逻辑操作是一个常用的做法。用'|'替代'||'和用'&'替代'&&'好用些。如果碰到这种情形，用宏来定义布尔型的 and 和 or，而用 AND 和 OR 代替&&和||。类似地，用'＝ '表示'＝＝'的意思也是个常见的错误。如果常有这种错误，用宏来定义赋值等号或逻辑等号，通常用 EQUALS 宏来代替逻辑等号（＝＝）是行之有效的。

17.2　复合语句（块）

一个“复合语句”或“块”是指几个语句组成的集合，但仅当作单个语句看待。这一般在控制程序流的情况下用。复合语句通常是在一组语句的前后写 begin 和 end 或{}组成。有时也由命令语句标明，如在 basic 中的 FOR 和 NEXT 或 Ada 中类似的结构。下面两点对怎样用好复合语句有帮助。

成对用大括号或 begin－and－end。在块的开始和结尾标志之间写入内容。人们总埋怨平衡括号对和 begin－and－end 对很难，因此用它们完全没必要。但若你遵循下面的做法，你就不致为平衡括号对之类的东西发愁了：

```
首先写:    for i=0 to MaxLine do
接着写:    for i=0 to MaxLines do
                    begin
                    end
最后写:    for i= 0 to MaxLines do
                    begin
                    {whatever goes in here}
                    ……..
                    End
```

这种方法可应用到所有的分块结构上去，在 Basic 中是 FOR-NEXT、 WHILE-WEND、IF-THEN-ELSE 形式，而在 Fortran 中则为 do 循环和相应的语句。

用括号对或 begin-and-end 对把条件标明。在 if 的条件测试中，若不标明紧跟在 if 后要执行的语句是哪些，那么这个程序也是很难读的。如果代码是任意编写的，就用块语句标明你的意图。

17.3　空语句

在 C 中可能用到空语句。空语句是指一句中仅有一个分号。下面是这样的例子。

含空语句的 C 程序：

```
while ((c=getch ())! ='\n'
    ;
```

C 中 while 语句后必须跟一条语句，但也可以是空语句。一行中仅有一个分号就是一个空

代码语句。下面几点可指导如何处理 C 中的空语句。

注意 C 中的空语句。空语句并不平常，因此应写清楚它。写空语句的一个方法就是在一个行中仅写上分号本身而无其它，并往后退几格就跟写其它语句一样，上面这个程序就用这个方法写的。当然，你也可以用一对空的大括号以示强调这是一空语句。如下例：

这个 c 语言例子用括号强调空语句：

```
while（(c = getch（)）！ ＝ ’\n’｛｝；——这是表示空语句的方法之一
        while（(c = getch（)）！ ＝’\n’
            {
              ；—一这是另一种空语句表示方法
             }
```

给空语句预定义一个 NUll（）宏。这个语句并不做任何事情，仅仅是清楚地说明不做任何事情这一事实。这类似于把文件的空白页写上“这页是空白页”。事实上有了这几个字，这页并不真的是空白的了，但它却说明这一页本来就不准备写东西的。

下面这段程序定义了空语句的宏：

```
#define null（）
…
while（(c＝getch（)）！ ＝’\n’
 null ();
```

除了用 Null（）宏作为 while 和 for 的空循环，你还可以把它用到不重要的 switch 语句分支中。用 Null（）语句表明你已经考虑了这样一种情况，即它不做任何事情。这些方法对其它语言同样适用，如 Pascal 中，可用一个子程序来模仿 Null（），而在 Ada 中本身就提供一个 Null().

注意到 Null（）宏和传统的预处理程序中的宏 NULL 是不一样的，后者指空指针。空指针 NULL 的值依赖于硬件，但通常是 0 或 OL 或类似的值。但它并不如这里所定义的 Null（）宏一样，它并非真表示什么都没有。

17.4　防止危险的深层嵌套

过深的嵌套已经困扰计算机界达15年之久了,但现在依然是使代码出乱子的罪魁祸首之一。Noam Chomsky 和 Gerald Weinlberg 研究表明，很少有人理解嵌套超过三层的 if 语句，许多研究者也认为应当避免嵌套超过 3 到 4 层。

避免用深的嵌套并不难。如果嵌套太深，可用 if 和 else 语句重新编程或把代码拆成简单子程序。后面提供几种减少嵌套方法。

通过重新编写部分测试条件来简化嵌套的 If 语句。如果嵌套太深，可重新组合一个 if 的检验条件以减少嵌套层次，下面这个程序嵌套太深，需要重新设计：

这个 c 程序不太好，嵌套太深：

```
if（Error==None）
        {
      / *lots of code* /
```

```
…
if（  PrinterRoutine!=  NULL）
{
  /*lots of code*/
  if（SetupPage()）
      {
        /*lots of code*/
        …
        if（AllocMem（&PrintData））
            {
            /*lots of code*/
            …
            }/*  if AllocMem（）*/
      } /* if SetupPage()*/
  } /* if PrinterRountine*/
} /*  if Error *  /
```

写这个程序的目的是显示一下较深的嵌套。/*lots of code*/部分的意思是这部分代码很　　长很长，可能要分几个屏幕显示，或打印时一页打不下。下面是修改以后（用重新组织测试条件的方法）的程序；

这个C程序重新组织了测试条件，减少了嵌套：

```
if（ Error==None）
  {
   /*lots of code*/
   …
   if（ PrinterRoutine!= NULL）
      {
       /*lots of code*/
       …
       }
   }
if（Error== None && PrinterRoutine!= NULL && SetupPage（））
   {
    /*lots of code*/
    …
    if（AllocMem（&PrintData））
       {
        /*lots of code*/
        …
        }
   }
```

这个程序显得很实在，因为你很难再往下减少嵌套了。如果还要减少嵌套层次，那么就要写出很复杂的测试条件了。把嵌套次数从 4 减少为 2 是一个了不起的改进，能提高可读性，但要考虑代价是否值得。

把一个嵌套的 if 语句改为一对 if－then－else 语句，如果仔细考虑一下嵌套的 if 测试条件，你就会发现用 if 一 then－else 重新组织程序要比嵌套的 if 好。假设你有一个很深的决策树如下：

这个 Pascal 程序是一个很深的决策树：

```
if（ Qty > 10） then
if（ Qty > 100） then
    if（ Qty > 1000） then
        Discount :=0.10
    else
        Discount := 0.05
  else
    Discount := 0.025
else
    Discount := 0.0;
```

这个程序的测试条件太乱太散，这有几个原因。一个原因是测试条件显得多余。当你测试 Qty 是否比 1000 大时，你就无需测试它是否比 100 大了，更不用说 10，考虑到这一点，重新写代码：这个 Pascal 程序把 if 嵌套转化为 if-then－else 的形式：

```
if（ Qty > 1000） then
    Discount := 0．10
else if（ Qty > 100） then
    Discount >= 0. 05
else if（ Qty >10） then
    Discount := 0.025
else
    Discount := 0;
```

这个程序就比上一个好多了，因为测试条件中比较的数值是速增的。如果数值不是那么有规律地增长，你可替换肤套的 if 语句如下：

这个 Pascal 程序把 if 嵌套转化为 if－then－else，在这里数值是无规律的：

```
if（Qty > 1000） then
  Discount：=0.10
else if（ Qty > 100 and Qty <= 1000） then
  Discount：=0.05
else if（ Qty > 10 and Qty <= 100） then
  Discount ＝0.025
else if（ Qty <= 10）
  Discount：=0
```

这个程序与前一程序的主要区别在于 else－if 的测试表达式不依赖于前一测试结果，这

段代码也可不要 else 语句，且测试条件可按任意顺序放置。代码可包含 4 个 if 语句而无 else。用 else 语句的原因是可避免不必要的重复测试。

把 If 嵌套改成 case 语句。你可以把某些类型的 if 语句测试条件（特别是用到整数）用 case 语句来代，而不用 if－else 来代替。但这种技巧并非所有语言都能用。若能用，这种技巧的效果很好。下面这个 Pascal 程序用了这种技巧：

这个 Pascal 程序把 if 嵌套转换成 case 语句：

```
case Qty of
    0..10:          Discount := 0. 0;
    11..100：       Discount := 0. 025;
    101..1000:      Discount := 0. 05;
    else            Discount := 0. 10;
end; {case}
```

这个例子读起来很轻松，把它与前几页的多次缩排程序例子比较，显得相当清楚。

提取深层锻茗的代码写成一个子程序。如果深层嵌套出现在一个循环中，你可把循环的内部写成一个子程序。这种技巧当嵌套是条件控制和重复时显得特别有效。把 if-then-else 分支留在主程序里以显示清楚决策分支，而把各分支里的代码写成一个子程序。下面这个程序就需用上述方法改造：

这个 C 程序的嵌套代码需分别写成子程序：

```
while（ ! feof（ TransFile ））
    {
    / * read transaction record * /
    ReadTransRec（TransFile，TransRec）;
  / * process transaction depending on type of transaction * /
    if（ TransRec．TransType = = Deposit ）
      {
      / * process a deposit * /
      if（TransRecs.AccountType = = Checking）
            {
      if（ TransRec.AcctSubType ＝＝ Business）
          MakeBusinessCheckDep（ TransRec. AcctNum , TransRec. Amount ）;
      else If（TransRec． AcctSubType = = personal ）
          MakePersonalCheckDep( TransRec.AccNum,TransRec.Amount );
      else if（TransRec. AcctSubType ＝＝ School ）
          MakeSchoolCheckDep（ TransRec.AcctNum，TransRec.Amount ）;
      }
    else if (TransRec.AccountType = = Savings )
```

```
                    MakeSavingsDep（TransRec．AcctNum，TransRec.Amount ）；
                else if（TransRec．AccountType = = DebitCard ）
                    MakeDebitCardDep（TransRec.AcctNum , TransRec.Amount ）；
                else if（TransRec．AccountType==MoneyMarket）
                    MakeMoneyMarketDep（TransRec.AcctNum,TransRec.Amount ）；
                Else if（TransRec.AccountType==CD）
                    MakeCDDep（TransRec.AcctNum ,TransRec.Amount）；
                }
            else if (TransRec.TransType==Withdrawal)
                {
                / * process a withdrawal * /
                if（ TransRec．AccountType ＝＝ Checking）
                    MakeCheckingWithdrawal( TransRec.AcctNum ,TransRec.Amount )；
                else if（TransRec．AccountType = = Savings ）
                    MakeSavingsWithdrawal(TransRec.AcctNum , TransRec.Amount );
                Else if (TransRec.AccountType = = DebitCard )
            MakeDebitCardWithdrawal（ TransRec．AccyNum， TransRec．Amount）；
        }
    else if（ TransRec.TransType = = Transfer）
        {
        MakeFundsTransfer（TransRec.SrcAcctType,TransRec.TgAcctType，
                    TransRec．AcctNum，TransReC.Amount）；
        }
    else
        {
        / * process unknown kind of transaction * /

        LogTransError(”Unknown Transaction Type”，TransRec);
        }
    }
```

虽然很复杂，但这个程序还不是你见到的最次的一个。它的嵌套仅有四层，并且写得很清楚，有编排形式，功能分解得也很完整，特别是对 TransRec。的处理。尽管有这些好处，但还是应该把内层 if 的内容写成子程序。

这个 C 程序较好，把嵌套内的内容写成子程序：

```
While(！feof( TransFile） ）
    {
    / *  read transaction record * /
    ReadTransRec（TransFile，TransRec）；

/ * process transaction depending on type of transaction * /
```

```
If（ TransRec．TransType＝＝　Deposit）
    {
    ProcessDeposit（TransRec．AccountType, TransRec.AcctsubType）;
          TransRec.AcctNum，TransRec．Amount）;
     }
else if ( TransRec.TransType==Withdrawal)
    {
    ProcessWithdrawal( TransRec.AccountType,TransRec.AcctNum,
          TransRec.Amount ）;
    }
else if（ TransRec TransType＝＝　Transfer）
    {
     MakeFundsTransfer（ TransRec．SrcAcctType， TransRec．TgtAcctType，
       TransRec.AcctNum，TransRec．Amount）;
     }
  else
   （
    / * process unknown transaction type * /
  LogTransError（”Unknown Transaction Type”，TransRec）;
}
```

新子程序部分的代码只是简单地从原程序中提取出来并形成子程序（这里没有写出来）。新的程序有几个优点。第一，仅有两层嵌套，使得结构显得简单、易懂；第二，你可在显示器的一屏上阅读、修改、调试这个较短的 while 循环，不会因为需几幕显示，几页打印而限制你的眼界；第三，把 processDeposit（）和 processWithdrawal（）功能写成子程序具有了易于修改的一切优点;第四，现在可容易看出代码能写成 switch－case 语句形式，那样就显得更易读了。如下例子：这个 C 程序较好，嵌套内的内容写成于程序且用了 switch－case 语句：

```
while（！ feof（ TransFile ））
        {
       / * read transaction record * /

        ReadTransRec( TransFile ， TransRec );
       / * process transaction depending on type of transaction * /

      switch(TransRec.TransType）
          {
          case（ Deposit ）;
               ProcessDeposit（TransRec.AccountType，TransRec AcctSubType，
               TransRec.AcctNum , TransRec.Amount ）;
```

```
            break;

        case（Withdrawal ）;
            ProcessWithdrawal（ TransRec．AccountType， TransRec.AcctNum,
            TransRec.Amount）;
            break;

        case（ Transfer ）;
            MakeFundsTransfer（TransRec SrcAcctType，TransRec．TgtAcctType，
                TransRec. AcctNum，TransRec．Amount）;
            break;
        default:
          / * process unknown transaction type * /
          LogTransError（”Unknown Transaction Type”，TransRec）;
            break;
          }
    }
```

重新设计深层嵌套代码。一般说来，复杂代码表明还没有完全理解你的程序，应使其更简单些。深层嵌套提示需把某些部分写成一个子程序或需重新设计复杂的那部分程序。这虽然并不意味着就需要去修改程序，但若无充分理由，还是要修改。

17.5　结构化编程的作用

结构化编程是什么意思？结构化编程的核心基于这样一种简单思想，即程序总是单入单出的结构，即程序只能从一个地方开始且也只能从一个地方退出的代码块，没有其它的进口与出口。

17.5.1　结构化编程的好处

为何本书还要在整整一章去讨论一个20年的老话题——结构化编程呢？每个人都用它吗？

否。并非每个人都用结构化编程，且许多人认为他们不用。Gerald Weingerg 调查了大约100家软件公司并访问了数千名程序员。他报告说，大约仅有5%的代码是完全结构化的；20%的程序结构化程度很高（这比 1986 年是一个提高）；50%的程序显示了用结构化编程的努力，但并不成功，而还有25%的程序则显示丝毫没有被过去20年的结构化思想影响。

有些程序员不相信结构化编程的作用。他们错误地认为结构化编程是多此一举与浪费时间，且编程效率低，打击编程员的积极性和降低其编程效率，有些报告证实了有这种对结构化编程抵制的行为。

尽管有这些否定意见，结构化编程的有效性是证据确凿的，观察实验数据发现结构化编码使总编程效率增加，再考虑到限制条件和复杂性，产量增量显得更大，从200%～600%不等。

通用动力公司（General Motors）一项研究表明结构化技巧可节省时间和培训近六个月（Elsoff 1977）。

除了提高产量，结构化编程还可提高可读性。在一项试验中，36 个专业程序员要理解结构化和非结构化的 Fortran 程序，结果显示，对结构化程序的理解得分为 56 分（100 分制），而非结构化程序的则仅有 42 分。这表明更多结构化编程比非结构化编程要提高 33 个百分点（Sheppardetal 1978）。

结构化编程的执行进程是一个有序的、有规律的过程，而不是不可预测地到处转移。程序可以从上往下读下来，执行大抵也按这个顺序。源程序无规律性会在机器中产生一些无意义的不好读的程序执行路径。而低可读性则意味着不好理解，程序质量差。

17.5.2　结构化编程设计的三个组成部分

“结构化”这个词是这许多年来使用频率极高的调，被应用到软件开发的各个领域，包括结构化分析、结构化设计、结构化实现。不同的结构化方法并没有统一起来，它们都是同时出现的，结构化是它们好处的标志。

许多人错误地理解结构化编程，把这个词在几个方面被乱用。首先，结构化编程不是缩排编写的方法，这种方法对程序的结构毫无益处；第二，结构化编程不是自上而下的设计，这只是编程的一些细节问题；第三，结构化编程不是一种节约时间的技巧。但以上几点在面向对象程序设计时是正确的。

下面几节讨论构成结构化编程的三个方面。

顺序编程

一个顺序程序指一组按顺序执行的语句。典型的顺序语句包括赋值和子程序调用。如下两例：

这个 Pascal 程序包含顺序代码：

```
{a sequence of assignment statements}

a := 1;
b := 2;
c := 3;

{a sequence of calls to routines}

Writeln（a）;
Writeln（b）;
Writeln（c）;
```

选择

选择是一种控制结构。这种结构使语句有选择地被执行。If-then－else 就是一个普通的例子。或者 If－then 语句，或者 else 语句被执行，但不会两者都不执行，总有一个被选择执行。

case 语句是另一种选择控制的例子。Pascal 和 Ada 中的 case 语句及 C 中的 switch 语句

都属 case 类的例子。在每一种语句中，都有几种情况被选择去执行。一般说来 case 语句和 if 语句在概念上是相似的。如果一种语言不支持 case 语句，可用 if 去模仿它。下有两例：

有选择句子的 Pascal 程序：

```
{ selection in an if statement }

    if（ a = 0） then
            { do something }
    else
            {do something else}

    {selection in a case statement}

    case（ Shape ） of
            Square:         DrawSquare;
            Circle:         DrawCircle;
            Pentagon:       DrawPentagon;
            DoubleHelix:    DrawDoubleHelix;
    end;
```

重复

重复也是一种控制结构，它使一组语句被多次执行。重复常指“loop”（循环），常见的几种重复有 basic 中的 FOR—NEXT、Fortran 中的 Do、Pascal 和 C 中的 while 和 for 等语句。如果不用 goto，那么重复将是控制结构的技巧。下面是用 Ada 编的重复例子：

Ada 编的重复的例子：

```
一 example of iteration using a for loop

for INDEX in FIRST..LAST loop
    DO_SOMETHING（INDEX）;
end loop

--example of iteration using a while loop

INDEX ： = FIRST;
while（ INDEX ＜= LAST ） loop
    DO_SOMETHING（ INDEX ）;
    INDEX := INDEX ＋ l;
End loop;

一 example of iteration using a loop-with exit loop

INDEX:=FIRST;
```

```
loop
    exit when   INDEX ＞ LAST;
    DO_SOMETHING ( INDEX );
    INDEX ：= INDEX + 1;
end loop;
```

17.6　用 goto 模拟结构化结构

汇编、Fortran、一般 Basic 或其它有些语言不支持结构化控制结构，那你可能要问：我怎么写结构化程序呢？正如 goto 的批评者所指出，你可以用三种结构化结构中的一种或几种来取代每一控制结构，也可用一个或几个 goto 来取代三种结构化结构。如果你的语言没有提供其它控制程序流的方法，那么避免用 goto 的结论就没什么意义了。这种情况下，你可用 goto 去模拟三种结构化编程结构，只是 goto 要尽量少用。

17.6.1　模拟 if－then－else 语句

如果你想用 goto 来模拟 if－then－else 语句，首先用注释行的形式写出测试条件和 if－then－else 语句，如下例所示，这里用 Fortran 写，但你也可很容易地采用汇编、Ihac 或其它语言写：

用注释行写出 if－then－else 的测试条件的程序；

```
C           if (A < B ) then
C           else
C           endif
```

其次，在注释行间写代码。作为一般的方法，为了转向分支，你对注释行中的条件取反，对应于重写的条件是真——即原条件为假时转向分支。为了对原条件取反，把条件两边用括号括上，并在其前面加上．NOT．运算，用一个 goto 语句从 if 部分转到 else 部分去，下面是程序：

用 Fortran 填充了 if－then－else 条件的程序：

```
C       if（ A＜ B＝ then
         IF（.NOT.（A. LT. B））GOTO 1100
              CODE TO PERFORM IF THE ORIGINAL CONDITION IN COMMENTS IS TRUE
              …
              GOTO 1200
C       else
        1100  CONTINUE
              CODE TO PERFORM IF THE ORIGINAL CONDITION IN COMMENTS IS FALSE
               …
        1200 CONTINUE
C       endif
```

注意到注释行中的 if、else、endif 语句往后退了几格与正式程序对齐，以免代码的逻辑结构受到注释行的影响。

下面是相应的 Basic 程序段：

用 goto 模拟 if-then－else。测试条件的 Basic 程序：

```
1000  'if  (A<B)  then
1010  IF（NOT（A＜B＝THEN GOT0 1100
1020  CODE TO PERFORM IF THE ORIGINAL CONDITION IN COMMENTS IS TRUE
          …
1090  GOTO 1200
1100  'else
1110  CODE TO PERFORM IF THE ORIGINAL CONDITIONIN COMMENTS IS FALSE
          …
1200  'endif
```

对其它语言你也可用同样的技巧用 goto 去模拟结构化结构。

17.6.2　模拟 case 语句

从概念上来讲，当你用 goto 模拟 case 语句时，你得写一大长串的 if－then－else。语句，每一个语句通常都从最后一个语句的终点退出。像上述 if－then 语句一样，你最好还是先把注释行写出来，然后往中间写入程序。下面是用 Fortran 改造的 case 语句：

```
C          case（ LETTER ）
C               'A':
                 IF（ LETTER .NE. 'A'）GOTO 1100
                        DO SOMETHING FOR CASE A
                    GOTO 1900
C               'E':
1110            IF（ LETTER .NE. 'E'） GOTO 1200
                    DO SOMETHING FOR CASE   'E'
                    GOTO 1900
C               'F' "
1200            IF（ LETTER .NE. 'P'）GOTO 1300
                    DO SOMETHING FOR CASE 'F'
                    GOTO 1900
C                 else
1300          CONTINUE
                    CALL ERROR（"Internal error:Call customer assistance."）
1900          CONTINUE
C           end case
```

程序中，每一种情况都用 if 来检查。若不满足条件（结果为假），用 goto 转向下一个 if 去测试。else 语句则用 CONTINUE 语句来代替，其前面的标号是 goto 要转向的标号。每一种情况下的代码段的最后一句用 goto 转向 case 结构的结尾（这里是 1900 行）。这样，仅有一种情况被执行。同样的技巧可应用到 BasiC 和汇编中去。

17.6.3　模拟 while 循环

如果你编写结构化程序，while 循环可能是最普遍的循环结构，如果你的 while 循环在开

始进行条件测试，像下面这样改：

这个 Fortran 程序用 goto 来模拟 While 循环：

```
C       loop initialization
              I=0
C        while ( I<MAXI and X >0.5)    do
1000     IF    (.NOT.    ( I .LT. MAXI .AND.X .GT. 0.5 ) )    GOTO 1010
C                DO SOMETHING
C                …
                 GOTO 1000
1010     CONTINUE
C        end while
```

这里要注意的地方跟上面讨论 case 语句一样，可以参考前述，注意的是循环初始化是明显给出的，作为提醒，和前面的 for 的条件一样，这里 While 的条件在被取代后要取反。

17.6.4　模拟控制结构概述

用 goto 来模拟结构化控制结构可帮你用某种语言编程。正如，DavidGriex 指出，选择何种语言并不重要，你要确定的是你该怎样编程。理解把程序编成某种程序语言的程序与用某种语言编程之间的区别是本书的关键。许多编程原则并不依赖于某种语言，而仅给你提供了一种方法。

如果所用语言不支持结构化结构或易于产生其它问题，努力弥补这种缺陷，形成一套你自己的编程风格、标准、库子程序或其它观点。

通过模拟结构化编程的结构，那些汇编、一般 Basic、Fortran 等语言对结构化控制结构的限制也就不复存在了。最后你能把程序从一种语言转化成另一种你碰巧要用的语言。

17.7　控制结构和复杂性

注意控制结构的一个原因是，它们对于克服程序的复杂性有很大贡献。不用控制结构，会增加程序的复杂性。用得好则能降低复杂性。

标量一个程序复杂性的方法是，若要理解程序你得一次连续在脑中记住多少目标程序语句，结构的处理是编程中最困难的方面，也是为什么需要特别注意结构性的原因，这也是程序在处理“快速中断”（quick interruption）时手忙脚乱的原因，这里快速中断相当于要求一个杂技员手上拿着东西的同时要不断向空中抛三个球做杂耍的情形差不多。

程序的复杂性很大程度上决定了要理解一个程序要花多大努力。T 哦 TomMcCabe 在一本书中认为一个程序的复杂性就决定于它的控制流。别的研究人员在此之外还确认了别的影响复杂性的因素，但都承认，控制流若不是最大的，起码也是影响复杂性的最大原因。

17.7.1 程序复杂性的重要性

计算机科学起码在一十年前就注意到了程序复杂性的重要了。二十年前，Edager Dijkstra 就意识到复杂性的危险，他说：“一个聪明的程序员总是清楚地知道自己的脑力容量有限，因此

他得十分小心谨慎地完成编程任务”（1972）。这并不意味着为了处理很复杂性问题你得增大你的脑力，而是说你得想办法尽可能降低复杂性。

控制流的复杂性很重要，因为它和低的可读性和频繁的出错紧密联系在一起（McCabe 1976，Shen at al 1985）.William T. ward 在 Hewlett-Packard 公司用 McCabe 的复杂性度量标准来研究软件的可读性问题时，得到一个很有意义的结果（1989）。把 McCabe 的复杂性度量原理用于确定一个 77，000 行程序的出错范围。这个程序的最后错误率为每一千行 0.31 个错，而另一个 125，000 行程序最后出错率为每一千行 0.02 个错。ward 发现，由于这两个程序的复杂性较低，因此这两个程序的出错数比 Hewlett－Packard 的其它程序都低。

17.7.2　减少复杂性的常用方法

下面两种方法可以帮助降低程序复杂性。首先，你可以做一些动脑筋练习来提高在脑中打底稿的能力。但大多数程序都很大，而人同时考虑的问题一般都不能超过 5～9 个，因此靠提高脑子的容量来帮助降低复杂性能力有限，第二，要降低程序的复杂性就要求你彻底理解你所要解决的问题。

怎样度量复杂性

你可能要问怎样从直觉上判断哪些因素使程序变得复杂了还是简单了呢？研究人员已经把他们的直觉归纳出来并形成了几条度量复杂性的方法，或许最有影响的数字技巧是 Tom McCabe 的方法，这种方法通过计算程序的“决定点”（decision point）的数目来度量复杂性，见表 17－2。

表 17－2 计算程序决定点的技巧

1．从 1 开始一直往下通过程序。 2．遇到下列关键词或其同类的词加 1。 if while repeat for and or 3．case 语句中每一种情况都加 1，如果 case 语句没有缺省情况再加 1

如下例：

```
if   (( ( Status = Success ) and   Done ) or
        (not Done and ( NumLines >= MaxLines)) ) then…
```

在这个程序中，从 1 算起，遇到“if”得 2，遇到“and”得 3，遇到“or”得 4，又遇到一个“and”得 5，这样程序总共含有 5 个决定点。

有了复杂性度量数日该怎样判断程序的复杂性

当你已经算出决定点的数目时，你可用算得的数目分析程序的复杂性。如果数目是：

0～5	程序可能很好
6～10	得想办法简化程序
10 以上	得把部分代码写成子程序并在原程序调用

把部分程序写成子程序并不能减少整个程序的复杂性，它仅仅是把决定点转移到别的地方，但它却降低了一次涉及到的复杂性。既然你的目的是降低一次要在你脑中考虑的问题数目，因此编成子程序降低复杂性的方法是有帮助的。

决定点的最大数目为 10 并不是一个绝对的极限，而仅用这个数目作为一种提醒标志，来告诉你程序需重新设计一下，不要死套这个规则，比如一个 case 语句有许多种情况，因而决定点数目会比 10 大得多，但你却不能把它分解成子程序。这种 case 语句在事件驱动程序中用得很多，如 Microsoft Windows 和 Apple Macintosh 中的许多程序。在这些程序中一个长的 Case 语句可能是降低程序复杂性的最好方法。

17.7.3　度量程序复杂性的其它方法

McCabe 的度量程序复杂性的方法并不是唯一的方法，但它却是用得最多的方法，特别当考虑控制流问题时。其它的方法包括用到数据的次数、控制结构中嵌套层次、代码的行数、变量连续出现的程序行行数及输入输出点的数目。另有一些研究人员已经开发了以上各方法的综合方法。

17.7.4　检查表

控制结构方面

- 表达式用 True 和 False 而非 1 和 0?
- 布尔型表达式的值是否隐含地与 False 比较?
- 是否通过定义中间布尔型变量和布尔型函数及用决策表的方法来简化表达式?
- 布尔型表达式是用肯定形式写出来的吗?
- 在 C 中，数值、字符，指针是显式与 0 比较的吗?
- begin 和 end 能保持平衡吗?
- 为了使程序看起清楚，需要的地方用 begin 和 end 对标明了吗?
- 空语句看起来清楚吗?
- 通过诸如重新组合测试条件、转化为 if-then－else 或 case 语句或把嵌套内代码写成子程序的方法来简化嵌套语句了吗?
- 如果程序的决定点数超过 10，有什么正常理由不重新设计它吗?

17.8　小　结

- 使布尔型表达式简单可读性高对代码的质量很有好处。
- 深层嵌套使程序难懂，不过可用相对简单方法避免这样做。
- 结构化编程是一个简化程序的思想，用顺序编程、选择或循环中的一种或几种方法的组合可编出任何程序。
- 作这种简化程序的思想可提高程序的产量和质量。
- 如果所用语言不支持结构化结构，你能模仿它们。你应该把程序编成某种语言的程序而不是用某种语言编程的。
- 降低复杂性是编写高质量的代码的关键。

第十八章　布局和风格

目录

相关章节

文档代码：见第 19 章

从这一章开始转向计算机编程的美学问题——源程序代码的规划布局问题，组织得很好的代码无论从直观上还是从内心里都产生一种愉悦的感觉，这恐怕是非程序员很少能达到的境界。使那些对自己工作很有自豪感的程序员能从他们代码的优美结构得到极大的艺术满足。

这一章所讨论的技巧并不影响速度、内存的使用及其它程序外观方面的问题，所影响的仅仅是怎样很容易地理解代码、检查代码、日后很容易地修改代码等。它也影响到别人如何很轻松地去读、去理解、当你不在时去修改代码。

这一章尽是些人们在谈到要注意细节时涉及到的小细节问题。在整个编程过程中，注意这些细节问题对程序最后质量和最后维护性都产生很大影响。对编码过程来说这些细节是必不可少，以致到最后无法改变。如果你是和一个小组一起工作，在编程前要和全组的人一起看看本章的内容并形成一个全组统一的风格。

你可能不太同意这一章的所有问题，但与其说是要你同意本章的观点还不如说是要让你考虑涉及到格式化风格的许多问题，如果你有高血压，请翻过本章，这里的观点都是要引起争论的。

18.1　基本原则

本节介绍布局原理，其它各节介绍实例。

18.1.1　布局的几个例予

考虑表 18-1 列出的程序。

表 18-1 Pascal 布局的例子

```
procedure InsertionSort ( Var Data : SortArray_t ; FirstElmt : Integer
LastElmt : Integer ) ;   { use the insertion sort technique to sort the
“Data” array in ascending order. This routine assumes that Data
[ FirstElmt ] is not the FirstElmt element in Data and that Data
[ FirstElmt - 1 ] can be accessed. }Const   SortMin = ’’ ; Var SortBoundary :
Integer ; { upper end of sorted range } InsertPos: Integer ; {position
to insert element } InsertVal : SortElmt_t ; {value to insert}
LowerBoundary :   SortElmt_t ; {first value below range to sort} begin
{ Replace element at lower boundary with an element guaranteed to be first
  in a sorted list } LowerBoundary : = Data[ FirstElmt_t ] ; Data
[FirstElmt-1] : = SortMin ; {The elements in positions FirstElmt through
SortBoundary-1 are always sorted. In each pass through the loop,
SortBoundary is increased, and the element at the position of new
SortBoundary Probably isn’t in its sorted place in the array,so it’s
Inserted into the proper place somewhere between FirstElmt and
SortBoundary.}for SortBoundary : = FirstElmt + 1 to stElmt do begin
InsertVal : = Data[SortBoundary] ; InsertPos : = SortBoundary ;while
InsertVal <= Data[ InsertPos – 1 ] do begin Data[InsertPos]
: = Data[ InsertPos – 1 ] ; InsertPos : = InsertPos – 1 ; end; Data
[ InsertPos ] : = InsertVal ;end;{Replace orginal lower-boundary
element }Data[ FirstElmt – 1 ] : = LowBoundary ; End; { InsertionSort}
```

这个程序从语法上来说是正确的。它用注释说明得很清楚, 而且变量名也有含义、逻辑思路也很清楚。如果不信，读完这段程序看看哪出错了。这段程序所缺省的是一个好的布局，这是一个极端的例子。若用好坏标准数轴来表示的话，这个例子的布局是要处于负无穷大方向的。表 18－2 的例子稍好些：

表 18-2 Pascal 程序布局的例子

```
procedure InsertionSort(   Var Data : SortArray_t ; FirstElmt:Integer ;
LastElmt : Integer   ) ;
{ use the insertion sort technique to sort the “Data” array in ascending
order. This routine assumes that Data[ FirstElmt ] is not the
first element in Data and that Data[ FirstElmt – 1 ] can be accessed. }
Const
SortMin = ’’ ;
Var
SortBoundary : Integer ; { upper end of sorted range }
InsertPos : Integer ; {position to insert element }
InsertVal : SortElmt_t ; {value to insert}
LowerBoundary :SortElmt_t ; {first value below range to sort}
begin
{ Replace element at lower boundary with an element
guaranteed to be first in a sorted list }
```

```
LowerBoundary : = Data[ FirstElmt_t ] ;
Data[ FirstElmt –1 ] : =    SortMin ;
{The elements in positions FirstElmt through SortBoundary-1 are
always sorted. In each pass through the loop, SortBoundary
is increased, and the element at the position of the
new SortBoundary Probably isn't in its sorted place in the
array, so it's inserted into the proper place somewhere
between FirstElmt and SortBoundary.}
for SortBoundary : = FirstElmt +1    to    LastElmt    to do
begin
InsertVal : = Data[ SortBoundary ];
InsertPos : = SortBoundary;
while InsertVal < Data[ InsertPos - 1 ] do
begin
Data[ InsertPos ] := Data[ InsertPos-1 ] ;
InsertPos := InsertPos –1 ;
end;
Data[ InsertPos ] := InsertVal ;
end; { Replace orginal lower-boundary element }
Data[ FirstElmt-1 ] : = LowBoundary ;
end; { InsertionSort }
```

这段程序与表 18-1 的一样。虽然大多数人要说这段代码的布局要比前一段的要好得多，但它还是显得不太好读。这段代码的布局还是显得拥挤而且无法看出程序的逻辑结构。它处于好坏标准数轴的 0 位置。第一个例子是一个分行的过程，而第二个则什么也没有。我见过一些程序有上千行那么长，其结构布局跟这个例子一样差劲，既无文件说明，也无好的的变量名，读起来跟这个例子一样难受。第二个例子是为计算机格式化的，而无丝毫迹象表明，编程者想让人读这一段程序。表 18－3 则是一个改进：

表 18-3 Pascal 程序布局例子

```
procedure InsertionSort
(
Var Data :          SortArray_t;
    FirstElmt:      Integer;
    LastElmt :      Integer
    );

{ use the insertion sort technique to sort the "Data" array in ascending order.
This routine assumes that Data[FirstElmt] is not the FirstElmt element in Data and that
Data[ FirstElmt-1 ] can be accessed. }

Const
        SortMin = ' ';
Var
```

```
        SortBoundary :    Integer;        { upper end of sorted range }
        InsertPos :       Integer;        {positionto insert element }
        InsertVal :       SortElmt_t;     {value to insert}
        LowerBoundary :   SortElmt_t;     {first value below range to sort}

    begin
        { Replace element at lower boundary with an element
        guaranteed to be first in a sorted list }
        LowerBoundary     := Data[FirstElmt_t];
        Data[FirstElmt-1]         := SortMin;
        { The elements in positions FirstElmt through SortBoundary-1 are
          always sorted. In each pass through the loop, SortBoundary
          is increased, and the element at the position of the
          new SortBoundary Probably isn't in its sorted place in the
          array, so it's Inserted into the proper place somewhere
          between FirstElmt and SortBoundary. }
    for   SortBoundary   :=   FirstElmt   +   1 to   LastElmt   do
        begin
        InsertVal   := Data[ SortBoundary ];
        InsertPos   := SortBoundary;
        while InsertVal< Data[InsertPos-1] do
            begin
            Data[InsertPos]     :=   Data[ InsertPos-1 ] ;
            InsertPos           :=   InsertPos-1;
            end;
        Data[ InsertPos ] := InsertVal;
        end;

       { Replace orginal lower-boundaryelement }
      Data[ FirstElmt-1 ] :=   LowBoundary;
    end; { InsertionSort }
```

这个程序的布局在好坏标准数轴的绝对正方向上。这个程序的布局完全按本章讲的原则来设计。这个程序易读得多，而注释和好的变量名都显而易见，变量名与第一个程序一样好，但第一个程序的布局太差，显示不出这种好处来。

这段程序与前两个的唯一差别只在有空格——其实代码和注释都一模一样。加空格只是有利于人的阅读，计算机可认为这三段程序是一样的。你和计算机对程序的感觉不一样是当然的事情。

另一种格式化的例子见图 18-1。这种方法是基于源代码格式的，由 Ronald M.Baecker 和 Aaron Marcus 创建。这种方法中,除了用空格外,这种方法还用到了阴影、不同字体及别的排版技巧，Baecker 和 Marcus 创造了一种能按类似图 18-1 所示的方法打印出通常源代码的工具。虽然这种工具还没有商业化，但由于它支持源代码的布局设计，因而不出几年就会普及。

用排序技巧把“Data”数组按递增的顺序排序。程序中的 data 数组的第一个元素并不是 Data[FirstElmt]而是 Data[FirstElmt-1]。

```
procedure
InsertionSort (
    Var     Data:       SORTARRAY_T;
            FirstElmt:  Integer;
            LastElmt:   Integer )
const
    SORTMIN = '';
var
    SortBoundary:   Integer;      upper end of sorted range
    InsertPos:      Integer;      position at which to insert
    InsertVal:      SORTELMT_T;   value to insert
    LowerBoundary:  SORTELMT_T;   first value below range to sort
begin
    Replace element at lower boundary with an element
    guaranteed to be first in a sorted list.

    LowerBoundary       := Data[ FirstElmt - 1 ];
    Data[ FirstElmt - 1 ]  := SORTMIN;

    The elements in positions FirstElmt to SortBoundary-1 are
    always sorted. In each pass through the loop, SortBoundary
    is increased, and the element at the position of the new
    SortBoundary probably isn't in its sorted place in the array,
    so it's inserted into the proper place somewhere between
    FirstElmt and SortBoundary.

    for SortBoundary := FirstElmt + 1 to LastElmt do begin
        InsertVal := Data[ SortBoundary ];
        InsertPos := SortBoundary;
        while InsertVal < Data[ InsertPos - 1 ] do begin
            Data[ InsertPos ]  := Data[ InsertPos - 1 ];
            InsertPos          := InsertPos - 1;
        end;
        Data[ InsertPos ] := InsertVal;
    end;

    Replace original lower-boundary element.

    Data[ FirstElmt - 1 ] := LowerBoundary;
end;
```

图 18-1 用排版技巧来格式源代码

18.1.2　格式化的基本原理

格式化的基本原理是用直观的布局显示程序的逻辑结构。

使程序看起来显得漂亮，目的是为显示程序的结构。如果一种技巧使结构看起来更好而另一种技巧也是这样，那就选两种技巧中最好的一个。本章提供了许多格式形式很好但扰乱了逻辑结构的例子。实际上优化考虑结构并不会使程序难看——除非代码的逻辑结构本身就很别扭。那种使好代码显得更好，坏代码显得更差的技巧要比使所有代码都显得好看技巧更有用。

18.1.3　人和计算机对程序的解释

对程序的结构来说，布局是一个有用的线索，虽然计算机并不理睬 begin 和 end 之类的词，但人却易于从这些看得见的标志中得到线索，考虑表 18-4 中的代码段，其中用缩排的方法使人看起觉得每次循环时三条语句都被执行了一次。

如果代码不用括号，那么编译程序执行第一条语句 MAX_ELMTS 次，而第二条、第三条语句仅执行一次。这种缩排的方式使你我都很清楚地觉得编程员是希望这三条语句一起执行，而应该放在一个大括号中的，但编译程序却并不这么看。

表 18-4 这个 C 语言程序的布局对人和计算机来说理解是不一样的。

表 18-4

```
 /* swap left and right elements for while array */

for (i=0; I<MAX_ELMTS; i++)
    LeftElmt   =   Left[i];
    Left[i]      =   Right[i];
   Right[i]    =   LeftElmt;
```

表 18-5　人和计算机理解不一样的 C 程序

```
    X   =   3+4   *   2+7
```

当一个人读这一句时往往倾向于把这句程序理解为 x 被赋值为（3+4）*（2+7），等于 63，但计算机并不理会空格而只遵从运算化先级的原则得出 X = 3 + (4 * 2) + 7 = 18。好的布局能从直观上就看出程序的逻辑结构，使人与计算机的理解一致。

18.1.4　好的布局价值多大

我们的研究表明，编程计划和编程的规则对于程序的理解影响很大。在《The Elements of Programming style》(《影响程序形式的因素》) 这本书中，kernighan 和 plauger 也证实了我们所说的实施规则，我们的经验是要特别注意这些规则；把程序写成一种恃殊的形式，并不仅仅是个美学问题，而是一般编程的哲学思想。一般的程序员都希望别的程序员也能跟从他的实施规则。如果规则打乱了，一个程序员辛辛苦苦积累的一些有用的规则就作废了。因此无论是新手还是专业程序员都很支持本书的观点（Elliot Soloway 和 Kate Ehrilich)。

对于布局来说，不像编程的其它方面，计算机和人对程序的不同理解起了作用。编程过程中所做的工作仅有一小部分，是为使计算机能读懂程序而做，而大部分工作则为了能使人读懂程序。

在那篇很有名的论文“Perception in chess”中，Chase 和 Simon 报道了对新手和专业棋手记忆残局能力比较的研究，当把可能在比赛中出现的残局摆在棋盘上时，专业棋手水平远高于新手。而当棋子随便放时两者水平差不多，传统的解释是，专业棋手并不比新手记性好，只

不过是专业棋手的知识结构帮助他们记住某些特殊情况的信息。当新的信息与他们的知识结构相符时——这里是有意识地摆残局——专业棋手当然很容易记住它。但新信息不与他们的知识结构相符时——这里是指随意摆残局——专业棋手的记性就并不比新手好。

几年后，Ben Shneiderman 在研究计算机编程方面得到了与 Chase 和 Simon 相同的结论。在他们论文“Exploratory Experiments in Programmer Behavior”中，Shneiderman 发现当把程序语句有规则安排时，专业编程员对程序的记忆比新手好。但当随意安排各语句时专业程序员的优势下降了。别的研究也证实了 Shneiderman 的发现，这些发现在别的领域诸如围棋、电子、音乐、体育等方面都得到证实。

上述研究表明结构化可帮助有经验的程序员想象、理解和记住程序的重要特征。许多程序员都坚持用自己的风格。虽然所用编程的结构化风格不一样，但有一点是可以肯定的，即毕竟都用了结构化的方法。

18.1.5　把布局作为一种信仰

一味强调对程序理解的重要性和要把程序结构化成相似的方式，已使一些研究者在考虑，若一个程序的风格与一个专业程序员的风格不一样时，是否会影响他对程序的理解能力。认为程序的布局不仅是一个逻辑问题而且也是一个美学问题使得对程序结构化的争论看起来更像一场宗教战争而不是一个哲学战争。

一般说来，有些布局的形式是比另一些好的。本章前面的一些布局用得较好的程序证明了这一点。本书并不想进一步指出哪些布局形式较好，是因为这些都是有争议的。好的程序员一定能正确看待他们自己所用的布局风格，并能接收那些被证明是更好的布局形式，即使在接收新的布局风格时，开始有所不适应也在所不惜。

18.1.6　好布局的任务

许多有关布局细节的结论是一个主观美学上的问题。通常你有许多方法完成一个任务。如果你明确了自己的喜好标准，那你发现我们争论的是一些主观的观点，而非客观的东西。一个好的布局应当是很明确的。

正确表达出程序的逻辑结构。这是结构化的基本原理，是编写好的布局的最直接目的，程序员通常用缩排或空行来标明程序的逻辑结构。

自始至终地体现代码的逻辑结构。有些布局形式的规则有许多特殊规定，这恐怕很难一直沿用下去。好的风格应当对大多数情况适用。

提高可读性。标明逻辑结构的编排方法若使程序难读是没有用的。一个好的布局模式应使代码易读。

易于修改。好的布局模式在修改代码时也能保持得很好。修改一行代码不需要涉及其它语句。

除了这些原则，有时用简单语句或块来减少代码的行数也被用到。

怎样使用评价布局标准

你可用一些评价布局好坏的标准去衡量布局，这样你认为一种风格比另一种风格更好的一些主观原因就有了依据。

衡量标准的不同可能得到不同的结论。比如你认为能把代码行数减少到能在一屏上显示最重要——或许是因为你的计算机屏幕很小，你可能要批评那种把程序的参数行拉到比两个屏幕行还长的布局形式了。

18.2　布局技巧

本节讨论用不同的方法得到较好的布局。

18.2.1　空格

用空格可提高可读性。空格包括空格、制表符、断开行及空行，它们是显示程序结构的最主要工具，不可想象一本书词之间无空格及段之间无间断，又不分章节，该是什么样的。这种书一页一页翻过还可能，但却无法从一页中找到你需要的行或段落。或者更重要的是作者希望通过书的布局使读者看出他组织内容的目的。作者通过组织布局就能给读者一个怎样理解该题目逻辑结构的很重要的线索。

把一本书分成章、段、句子本身就给读者显示出作者怎样在脑中组织这个题目。如果这种组织不明显，那读者就得自己来想象这种组织结构，因而极大的增加了读者的负担，或者读者自始至终都没有弄懂你的组织形式。

程序段所包含的信息要比书的段包含的信息多，你可能一两分钟内读懂一本书，但大多数程序员却很难在那个速度上读懂一大段的程序。因此一个程序应当包含着更多的从组织结构上给出的信息，而不能更少。

分组。从另一角度看，空格是一种组织形式；它使得相关的语句看起来很好地组织在一起。

写文章时可把不同的想法分成段，一个写得很好的程序仅包含了有关的句子。而不可包含许多无用的语句。类似的，一个代码段应该也只能包括有关的、为完成某一任务而组织在一起的语句。

空格。正如把相关语句组织在一起很重要一样，把不相关的语句从其它语句中分离开来也很重要。在英语的段落开始都用缩排或空格的方式表示出来。在程序段的开头也应当加空格标明。

空格表明了你是怎样组织程序的。用这种方法可把相关的语句划分成段，把一个子程序从别的区别开来及突出注释部分。虽然很难用数据来统计，但 Gorla，Benader 和 Benander 的研究发现，一个程序中空行数目最好是占 8%～16%，超过 16%调试时间就明显增加了。

对齐。把同属性的元素对齐。例如把同一类的一组语句的等号排成一条直线下来，直观上对齐使人一看就知道这些句子是同属性的。如果不是同一类型的，不要这样排列整齐。

缩排。利用缩排显示程序的逻辑结构，作为一种规定，当一个句子从属于上一个句子时（在逻辑上），这个句子就比上一行退几格。

缩排的方法被证明是增强对程序理解的行之有效的方法。在“Program Indentation and Comprehensibility”这篇文章中有几个关于缩排与增强理解度关系的研究结果。研究表明，当程序编排是 2～4 个空格时，程序的理解度要比无缩排的程序高出 20%～30%。

同样的研究发现既要强调程序的逻辑结构也不要过分强调其逻辑结构。对程序理解度最低的是根本不用编排的程序，第二低的却是缩排退了 6 格的程序，因此最佳的缩排退空格是 2～

4，有兴趣的是许多人主观上认为退 6 格的缩排比少退格好用，尽管这时程序的理解度低。这也难怪，退 6 格看起相当舒服。但不管看起来怎样漂亮，退 6 格的缩排可读性下降了。这是美学与可读性之间的冲突。

18.2.2　括号

要尽量多地用括号。当一个表达式超过两个运算部分时要用括号来使它更清楚。有时可能有些括号是不必要的，但它们却增加了式子的清晰度而且并不要求计算机多做什么。下面的例子中你能算出结果吗？

C 语言：　　12+4%3*7/8
Pascal 语言：　12+4 mod 3*7 div 8
basic 语言：　12+4 mod 3*7/8

关键问题是你得考虑式子是按什么顺序计算的。你能不查有关参考资料而确信你的答案吗？

即使是有经验的程序员也无充分把握，这就是为什么当你在计算一个表达式时只要有不清楚的地方就加话号的原因。

18.3　布局风格

用布局的观点来看，控制语句下的一组语句要区分成一组。这些组用关键字括住：在 Pascal 中为 begin 和 end，在 c 中为｛和｝，在支持结构化的 Basic 中为 if-then-else，为方便起见，下面的讨论集中用 begin 和 end 来代表。下面分几点来讲一般的布局格式：

- 纯块结构
- 行尾布局
- 模仿纯块结构
- 把 begin 和 end 作块边界

18.3.1　纯块结构

许多关于布局的争论源于那些常用语言内在的不足，一个精心设计出来的语言其本身有很清楚的块结构可供自然的缩排形式用。在 Ada 中，每一控制结构都有其相应的结束符，你不能用别的控制结束符，因此分块是很自然的事，表 18-6、18-7、18-8 列出几个 Ada 的例子：

表 18-8 Ada 语言的纯 if 块结构

```
if    PixelColor = RedColor then
      statement1;
      statement2;
      …
end if;
```

表 18-7 Ada 语言的 while 纯块结构

```
while PixelColor = RedColor loop
      statement1;
```

```
        statement2;
        …
        endloop;
```

表 18-8　Ada 语言的纯 case 块结构

```
    case   PixelColor   of
      when RedColor =>
         statement1 ;
         statement2 ;
         …
      when GreenColor =>
         statement1 ;
         statsment2  ;
         …
      when Others =>
         statement1 ;
         statement2 ;
         …
  end case;
```

Ada 语言中一个控制结构总有一个起始语句——if-then，while-loop，case-or 等，并且都有一个对应的结束语句。控制结构中间的缩排却是公认的，但选用别的关键词却有些限制。表 18-9 抽象地表示了这种格式化形式：

表 18-9　纯块结构布局形式的抽象例子

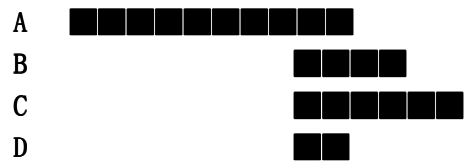
```
A   ■■■■■■■■■■■■
B       ■■■■■■■
C       ■■■■■■■■■
D   ■■■■
```

上例中，A 语句是控制结构的开始，而 D 语句则是结束。这两语句间的其它语句对齐、缩排。有关格式化控制结构的争论，部分起源于有些语言根本不需块结构。有时 if-then 语句之后仅限一条语句而不是通常所说的语句块，这时你却得加 begin-end 对或开拓号”｛”和闭括号”}”来制造出一个完整块而不是直接在控制结构后跟一条语句了事。程序中的 begin 和 end 不成双，这就产生了在哪去放置 begin 和 end 的问题。通常缩排的问题仅仅是因为你得补偿语言在设计结构上的不足。下面讨论不同的补偿方法。

18.3.2　行尾布局

如果你所用的不是 Ada 语言，你就不能用纯块结构布局。你得真正用 begin 和 end 关键词来实现块结构。其中一种方法是“行尾布局”，指一大组语句几乎要退格到行尾的布局方法。这种行尾缩排方法使下面的一组语句与开始这个控制结构的关键词对齐，而下面的参数行与第一个参数行对齐排下来。表 18-10 是一个抽象的例子：

表 18-10　行尾布局形式的抽象例子



这个例子中，A 语句是控制结构的开始，D 则是结束，B、C、D 语句与 A 语句的块结构关键词对齐。B、C、D 统一缩排说明它们是一组的，表 18-11 是一个用这种格式的 Pascal 例子。

表 18-11　行尾布局 Pascal 例子

```
while   PixelColor = RedColor   do    begin
                                        statement1 ;
                                       statement2 ;
                                       …
                                       end ;
```

这个例子中，begin 放在第一行末尾但不作为相应的关键词，有些人愿意把 begin 作为关键词，但在这两点间选一个作关键词仅仅是这种风格的一个细节问题。行尾布局风格有几种可接受的变样，表 18-12 是一个例子：

表 18-12　这种形式的 Pascal 程序虽少见，但这种行尾布局形式也是可行的

```
if（ SoldCount > 1000 ）then    begin
                                  Markdown   :=   0.10 ;
                                  Profit   :=   0.05 ;
                                  end
                          else   Markdown   :=   0.05 ;
```

这个例子中程序块与 then 后的 begin 对齐，但在最后一行 else 与 then 关键词对齐。这使得逻辑结构很清楚。

如果你认为前面的 case 语句不怎么样，你可能想要打破这种形式。如果条件表达式变得复杂了，那么这种布局形式对于理解逻辑结构是无用的或说是有误的。表 18-13 的例子给出了用复杂控制表达式打破上面例子的布局形式：

表 18-13　这个 Pascal 程序更典型，其中的行尾布局已被破坏

```
if ( SoldCount > 10 and PrevMonthSales > 10 ) then
        if ( SoldCount > 100 and PrevMonthSales > 10 ) then
                if ( SoldCount > 1000 ) then begin
                                                Markdown := 0.10;
                                                 Profit := 0.05
                                             end
                                        else   Markdown   :=   0.05
                                                        else   Markdown   :=   0.025
                                    else   Markdown   :=   0.0 ;
```

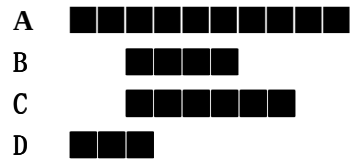
这个例子末尾的 else 语句怎样？它们也都与其相应的关键词对齐了，但却不能说这种编排形式使逻辑结构更清楚了。如修改第一行使其长度变化，那么按行尾布局的要求所有相应语句的缩排格数都要跟着变，这就在修改程序时产生别的布局形式不会产生的问题。

简言之，不用行尾布局是因为其不精确。它很难有连续性和维修性。本章中你到处可能见到行尾布局所产生的问题。

18.3.3　模拟纯块结构

若使用的语言不支持纯块结构，那么替代行尾布局的一个较好的选择是，使用语言去模仿 Ada 的纯块结构，表 18-14 是你要模仿的纯块结构抽象表示：

表 18-14　纯块结构布局形式的抽象例子:



在这种布局形式中，A 句开始块结构，而 D 语句则结束块结构，这表明 begin 需在 A 语句的结尾而 end 在 D 语句中，要模仿纯块结构，其抽象过程如表 18－15 所示:

表 18-15　模仿纯块结构布局的抽象例子

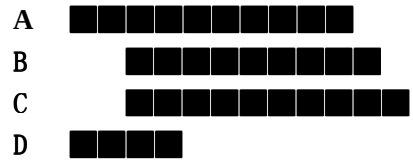


表 18-16、18-17、18-18 是以上类型的具体的 Pascal 例子：

表 18-16　模仿纯 if 块结构的 Pascal 例子

```
If    PixelColor = RedColor then begin
      statement1 ;
      statement2 ;
      …
end;
```

表 18-17 模仿纯 While 块结构的 Pascal 例子

```
while    PixelColor = RedColor do begin
      statement1 ;
      statement2 ;
      …
end;
```

表 18-18 模仿纯 case 块结构的 Pascal 例子

```
case    PixelColor of
        RedCelor : begin
                   statement1;
                   statement2;
                        …
                    end;
        GreenColor : begin
```

```
                statement1;
            statement2;
            …
        end
else        begin
            statement1;
            statement2;
            …
        end
end;
```

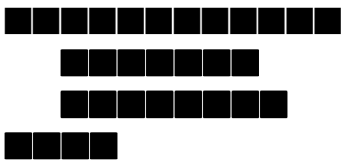
这种控制语句内句子对齐的形式显得很好，你能连续地应用这种结构形式，它的维护性也很好。这种形式符合格式化的基本原理且能显示代码的逻辑结构。因此这种形式充分可行。要注意的是这种形式在 Pascal 中不常用，但在 C 中用得很多。

18.3.4　用 begin 和 end 作块的边界

取代纯块结构的一个替换方法是使用 begin 和 end 作为块的边界而成对出现。用这种方法时，begin 和 end 作为控制语句下的一个语句而非作为控制语句的一个部分。

模仿纯块结构的抽象结构如表 18-19。

表 18-19　模仿纯块结构布局形式的抽象例子



在把 begin-end 作块边界的方法中，我们是把 begin 和 end 用为块结构本身的一部分而不是作为控制语句的一部分。你得把 begin 置于块的开始（而非放在控制语句的末尾），而把 end 放在块的结尾(而非作为控制结构的结束符)。作为一种抽象的表示方法，可把这种结构如 18-20 表示出来：

表 18-20　用 begin 和 end 作为块边界的抽象例子

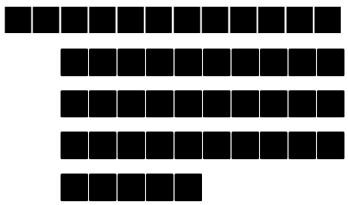


表 18-21、18-22、18-23 分别给出这种形式的具体 Pascal 例子。

表 18-21　在 if 块中用 begin 和 end 作块边界的 Pascal 例子

```
if   PixelColor = RedColor   then   begin
        statement1 ;
        statement2 ;
        …
    end;
```

表 18-22　在 while 块中用 begin 和 end 作块边界的 Pascal 例子

```
while PixelColor＝ RedColor do
    begin
    statement1;
    statement2;
    …
    end;
```

表 18-23 在 case 块中用 begin 和 end 作块边界的 Pascal 例子

```
case PixelColor of
    RedColor,
        begin
        statement1;
        statement2;
        end;
    GreenColor:
        begin
        statement1;
        statement2;
        end
    else
        begin
        statement1;
        statement2;
        end
end;
```

这种对齐块中语句的方法显得很好，它满足了格式化的基本原理又充分体现了逻辑结构。在所有情形下都能连续使用且维护性很好。

18.3.5　哪种形式是好

很容易回答哪种形式最次。行尾布局是最次的，这种形式无连续性且难维修。

如果用 Ada 编程，用纯块结构的缩排方法。

如果用 C 或 Pascal 编程，选择使用模仿纯块方法或 begin-end 作块边界的方法。这两种方法基本上没有什么区别，选择你所喜欢的。

以上各种形式都不是绝对的，都需要考虑各种偶然的因素，并采取综合兼顾的方法。你可能觉得这种或那种更美观。本书都用到了多种程序方法，通过看例子你可领略这些风格的异同。一旦选定某种形式，你应连续地应用去发现好布局的优点。

18.4　控制结构布局

对于一些程序来讲，布局基本上是一个美学问题，但是控制结构的布局却影响到程序的可读性

和理解性，因而是个需要考虑的问题。

18.4.1　关于格式化控制结构块的几点好意见

涉及到控制结构块的布局时需注意到几点细节，以下提供一些指导。

begin-end 对应当退格。在表 18-24 中 begin-end 对与控制结构语句对齐了，但 begin-end 对内的语句相对 begin 退了两格。

表 18-24　begin-end 对内语句缩排的 Pascal 例子

```
for i := 1 to Maxlines do
begin
    Readline(i);
    PrecessLine(i);
end
```

begin 与 for 对齐

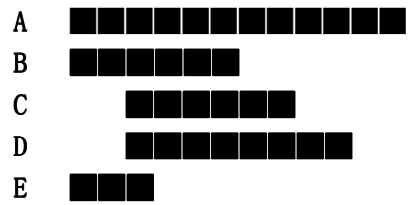
这两个语句缩进两格

end 与 for 对齐

这种方法看起来很好，但它却违反了格式化基本原理。它并没有显示出代码的逻辑结构。这种方法中，begin 和 end 好像既不是控制结构的一部分，也不属于它们之间语句组的一部分。

表 18-25 是这种方法的示意图。

表 18-25　引起错误导向的缩排方法的抽象例子



这个例子中，语句 B 属于 A 吗？它看起来不像是 A 语句的一部分。如果你已用了这种方法，把它改成前面讲过的两种布局形式，这样你的格式化会更清楚些。

用了 begin-end 对的代码不要进行两次缩排。反对 begin-end 对无缩排的对立面，反对 begin-end 对中再次缩排，这种方法如 18-26 所示，begin 和 end 相对控制语句退后几格，而它们之间包含的语句又退后了几格。

表 18-26　用 begin-end 对缩排两次的 Pascal 例子

```
for i := 1 to MaxLines do
    begin
        ReadLine(i);
        ProcessLine(i);
    end
```

这是另外一种看起来很好但却违反了格式化基本原理的另一布局形式。研究表明，一次缩排与两次缩排在理解上是差不多的，但却不能正确反映代码的逻辑结构。ReadLine()和 ProcessLine()看起来好像从属于 begin-end 对，但事实上却不是。

这种方法增加了程序逻辑结构的复杂性，表 18-27 和表 18-28 中哪一个看起来更复杂？

表 18-27　抽象结构 1

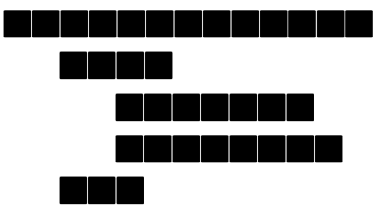
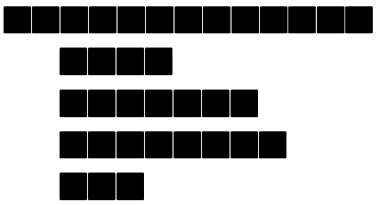


表 18-28　抽象结构 2



上述两个抽象结构都代表了 for 循环的结构。虽然两者形式表示同一代码，但抽象结构 1 却显得比抽象结构 2 更复杂。如果你的嵌套有两至三层，那么这种两次缩排的形式会使你的程序产生四到六次退格。这种布局会使代码显得比原来的更复杂。避免的方法是用纯块结构模仿法，或用 begin 和 end 作块边界并与其内部语句对齐的方法。

18.4.2　其它方法

虽然块的缩排是格式化控制结构的主要方法，但也有另外几种可供选用的方法。以下提供几点参考。

段之间用空行。有些代码的块不能用 begin-end 对来划分。一个逻辑块——同一类的一组语句——应当像写英语文章一样一起形成一段。把这样的逻辑块(段)之间用空行隔开。表 18-29 的例子便是用空格隔开的例子。

表 18-29 应当组织成块并隔开的 C 语言例子

```
Cursor.Start              = StartingScanLine;
Cursor.End                    = EndingScanLine;
Window.Title              = EditWindow.Title;
Window.Dimensions             = EditWindow.Dimensions;
Window.ForegroundColor    = UserPreferences.ForegroundColor;
Window.BlinkRate          = UserPreference.ForegroundColor;
Windows.BackgroundColor   = UserPreferences.BackgroundColor;
SaveCursor(Cursor);
SetCursor(Cursor);
```

这段代码是正确的，但有两条理由应当用空行。第一，当一组语句与执行顺序无关时，你可以如上例随意混在一起，无需为计算机作进一步的排序。但是作为读者都希望从你的特定程序顺序中获得某些线索，以判定哪些句子该属于同一组。在一段程序中加上空行使你很清楚地知道哪些语句是属于一起的。修改程序如表 18－30 所示。

修改后的代码显示出程序要做两件事，但：

表 18-30　这个 C 程序把语句适当分组后分开

```
Windows.Dimensions            = EditWindow.Dimensions;
Window.Title                  = EditWindow.Title;
Window.BackgroundColor        = UserPreferences.BackgroundColor;
Windows.ForgroundColor        = UserPreferences.ForegroundColor;

Cursor.Start              = StartingScanLine;
Cursor.End                = EndingScanLine;
```

```
Cursor.BlinkRate        = EditMode.BlinkRate;
SaveCursor(cursor);
SetCursor(Cursor);
```

上一个例子不分组也无空行，使人觉得好像这些语句都有联系似的。

第二个加空行分段后，若想给每块写上注释，那这里相当于自然地留下了空间。本例中若要加上注释则更提高了布局的透明性。

单条语句的程序块也要坚持格式化。单语句程序块是一个控制结构仅跟一条语句，比如 if 测试条件后仅跟一条语句。这种情形中，begin 和 end 对于保证正确的编译是不必要的了。这种情况有如表 18-31 所示的三种可选格式。

表 18-31 这个 Pascal 例子提供三种 for 的单语句块的选择模式

```
if( expression ) then                    ——格式 1
    one-statement;

if( expression ) then begin              ——格式 2a
    one-statement;
    end;

if( expression ) then                        ——格式 2b
    begin
    one-statement;
    end;

if( expression ) then one-statement;     ——格式 3
```

这三种方法各有千秋。格式 1 是单条语句退格作为一个块方式，它显得很协调。格式 2（2a 或 2b）也很协调，它在 if 测试条件后增加了语句，完了再加上 begin 和 end 对，因而减少了出错的机会。这种错误常很隐含，因为你所增加的句子都是退格写的，不注意是看不出来的；但是计算机却不理会这种退格。格式 3 是这样一种形式，当把它拷回到另的地方去时，作为一个整体拷贝出错的机会很少。这种格式的一个不好之处便是在一个面向行调试器中，调试器把这整行当作一行看待且不显示在 if 测试条件后的行是否被执行。

我曾用过格式 1，并且常成为修改时出错的受害者。我也不喜欢缩排方式的变异情况格式，因而总是避免用它，我比较喜欢用格式 2 的两种情况，因为它们看起来协调且易修改。不管你用哪种格式，你要一直都用这种格式。

对于复杂的表达式，把单独的条件列成单独的一行。把复杂条件的每一部分写成自己单独的一行。表 18-32 显示了一个没注意可读性的例子。

这个例子是为计算机格式化的而不是为读者。

表 18-32　这个 Pascal 例子的表达式可读性极差

```
if ( ( '0' <= InChar and InChar <= '9' ) or ( 'a'    <= InChar and
        InChar <= 'z' ) or ( 'A' <= InChar and InChar <= 'Z' ))    then
```

…

把条件分成几行，如表 18-3 所示，能增强可读性。

表 18-33　这个 Pascal 例子条件虽然复杂却可读

```
if ( ( '0' <= InChar and InChar <= '9') or
     ( 'a' <= InChar and InChar <= 'z' ) or
     ( 'A' <= InChar and InChar <= 'Z' )) then
```

这个程序段中用了几个格式化的技巧——对齐、退格，使各行明显独立——使得条件表式可读性好。而更重要的是这个测试条件的用意很明显。如果表达式包含了一个小错误，如把“Z”写成了“z”。那么上面程序中能很清楚地表现出来。

避免用 goto 语句。避免用 goto 语句的最基本原因是，它使得程序很难被证明是正确的。这点被所有希望自己程序是正确的人所接受。而更急迫的问题用了 goto 则很难把程序格式化。goto 语句及它要转向的标号之间的语句都要往后退格吗？假如有几个百 goto 语句转向同一个标号你又该怎么退格？下一个 goto 相对前一个 goto 语句退格吗？以下几点对格式化 goto 会有帮助。

- 避免用 goto，这就不存在再把程序格式化的问题了。
- 给 goto 所要转向的标号起个名字，这使得标号明显。
- 把含 goto 的语句单独列成一行，这使得 goto 更突出。
- 把 goto 所要转向的标号单独列成一行，且前后都加上空行。这使得标号明显。把含标号的行与周围的行退同样的格数，以使得程序的逻辑结构看起来紧凑。

表 18-34 显示了 goto 语句的例子。

表 18-34　这个 Pascal 例子在用 goto 的情况下使程序显得很好

```
    PROCEDURE PurgeFiles (    var ErrorCode :    ERROR_CODE    );
  var
        FileldX:            Integer;
        FlleHandle:         FILEHANDLE_T;
        FileList:           FILLIST_T;
        NumFllesToPurge:        Integer;
  label
        END_PROC;

  begin
        GrabScreen;
        GrabHelp ( PURGE FILES ) ;
        MakePurgeFileList ( FileList, NumFilesToPurge );

        ErrorCode := Success;
        FileldX := 0;
        while ( Fileldx < NumFilesToPurge ) do begin
              Fileldx    :=    Fileldx + 1 ;
```

```
                if Not FindFile( FileList [FileIdx], FileHandle ) then begin
                        ErrorCode :=    FileFindError;
                        goto END_PROC;          ——这里有一个 goto
                end;
                if not OpenFile ( FileHandle ) then begin
                    ErrorCode := FileOpenError ;
                        goto END_PROC;          ——这里有一个 goto
                end;
                if not OverwriteFile ( FileHandle ) then begin
                        ErrorCode := FileOverwriteError;
                        goto END_PROC;          ——这里有一个 goto
                end;
                if not Erase ( FileHandle ) then begin
                        ElrorCode := FileEraseError;
                        goto END_PROC;          ——这里有一个 goto
                end;

        end ; { while }
        END_PROC                                ——这里是 goto 的标号
        DeletePurgeFileList (    FileList , NumFileToPurge    );
         ReleaseHelp;
         ReleaseScreen;
end; { PurgeFiles }
```

这个 Pascal 例子较长，在这种情况下一个专业编程员可能认为 goto 就是最好的选择。这种情况下，表 18-34 是你所能做到格式化的最好地步。

不要用行尾布局结构特别是 case 语句。用行尾布局方法时 case 语句在修改时会出现严重问题，一个常用的格式化 Case 语句的方法是把每一种情况的执行语句退格到这种情况的描述语句之右（后），如表 18-35 所示。但在修改时这种形式却产生很大问题。

表 18-35　把行尾结构用于 case 语句很难修改的 Paseal 例子

```
case BullColor of
    Blue :              Rollout( Ball );
    Orange :            SpinOnFinger( Ball );
    FluorescentGreen :   Spike( Ball );
    White ：            KnockCoverOff( Ball );
    WhiteAndBlue :      begin
                        if( MainColor := White ) then
                          begin
```

```
                KnockCoverOff( Ball );
                end
            else if（ MainColor := Blue ） then
                begin
                Rollout( Ball );
                end
            end
    else    FatalError( "Unrecognized kind of ball.")
end;    { case }
```

如果你增加一种情况而它的描述名字比所有已存在的名字长，你就要把所有情况后的执行代码整个右移，而开始时很多的退格已经使得逻辑结构不能再往右移，如上例中 WhiteAndBlue 情况，解决的办法是，把你在每一种情况下的退格数确定下来而把执行语句移到描述语句的下一行。如果在循环中描述语句退三格，则在每一种情况下，执行语句在下一句都退后三格，如表 18-36 所示：

表 18-36　按标准数退格的 Pascal 例子

```
case BallColor of
    Blue:
        Rollout( Ball );
    Orange:
        SpinOnFinger( Ball );
    FluorescentGreen：
        Spike( Ball );
    White:
        KnockCoverOff( Ball );
    WhiteAndBlue:
        begin
        if( MainColor = White ) then
            begin
            KnockCoverOff( Ball );
            end
        else if( MainColor =    Blue ) then
            begin
            Rollout( Ball );
            end
        end
    else
FatalError（"Unrecognized Kind of Ball．"）
end;｛case｝
```

本例子是大多数人愿意看到的样子。它轻易地使程序在有长句子时易修改、具有连贯性、可维护性等。

如果你的 case 语句的各情况基本平行且执行语句很短，你可考虑把情况的描述语句与执

行语句放在同一行，然而在多数情况下你不要作这种指望。因为格式化过程是一个变动很大的修改过程，且很难保证长的执行语句与短执行语句平行。

18.5　单条语句布局

本节给出了在程序中如何安排好单条语句的方法。

18.5.1　语句长度

一个常规是限制一条语句不超过 80 十字符，以下是原因：

- 超过 80 个字符的语句很难读。
- 80 个字符的限制也防止了深层嵌套。
- 超过 80 个字符的行不能在 8．5”X 11”的打印纸上打印。
- 超过 8.5” X 11”的打印纸不好用。

现在已有了宽显示器、字体窄打印机（一行打印超过 80 字符）、激光打印机、前景模式显示器，那么对 80 个字符的限制已不如以前那么有效了。把一句写成 90 字符一行单句远比为了避免超出 80 字符而分成两句写为好。现在的水平允许偶尔让一行超过 80 字符。

18.5.2　用空格使语句显得清楚

用空格加在语句中有时可增强可读性。

用空格使逻辑表达式可读。表达式

```
while（pathName[startpath+pos ] <> ';' ）and
        (( startpath + pos ) <= length ( Pathname )) do
```

就显得很难懂。

作为一种规定，你可用空格把标识符分开。用这种规定，上述 while 语句如下所示：

```
while（pathName[ startpath+pos   ] <> ':' ）and
        ((startpath + pos ) <= length ( Pathname )) do
```

有些软件专家可能在上述表达式加更多条的空格以强调其逻辑结构。如下:

```
while（  PathName [ startpath 十 pos ] < ':'  ）   and
          (  ( startpath + pos ) < = length（ Pathname   ）  ) do
```

这就显得很好，空格有效地增强了可读性。多加的空格不造成什么资源浪费，所以尽可用上。

用空格使数组下标更好读。表达式：

```
GrossRate [ Census[ GroupID ].Sex, Census[GroupID].AgeGroup]
```

这个程序跟前面的挤在一起的 while 表达式一样难读。在数组下标前后加上空格使其更读。如果用上述规则，表达式如下：

```
GrossRate [ Census [ GroupID ].Sex，Census [ GroupsID ].AgeGroup ]
```

用空格使子程序参数更好读。下面子程序有四个参数，但各是什么呢？看不太清。

```
ReadEmployeeData（MaxEmps，EmpData，InputFile，EmpCount，InputError）;
```

经加空格后能看清楚吗？

GetCensus (InPutFile, EmpCount, EmpData, MaxEmps, InputError);

上面两个哪个更清楚？这是一个现实的有意义的问题，因为所有的语言都是涉及到子程序参数，它的位置也很重要。通常是在上半屏幕定义子程序的参数表，而下半屏幕就是调用它的地方，两者参数正好一一对应地作比较。

18.5.3　把相关的赋值语句对齐

若几个赋值语句是相关的，则应把等号对齐。表 18-37 却是没有对齐的例子：

表 18-37　这个 Basic 的赋值语句等号未对齐

```
EmployeeName = InputName
EmployeeSalary = InputSalary
EmployeeBirthdate = InputBirthdate
```

表 18-38 则做得较好。

表 18-38　Basic 的等号对齐,较好看

```
EmployeeName      = InputName
EmployeeSalary    = InputSalary
EmployeeBirthdate = InputBirthdate
```

第二个程序段除了看起来整齐外，其格式化也较第一个好，但如何看待等号前的空格呢？（多占存储空间）。又如何看待为对齐等号而多做的工作呢？

这种做法主要的考虑是因那些语句是同类的面对齐等号正好从直观上反映了这一点。如果这几句不是相关的，最好别这样做。表 18-39 是一个引起误解的对齐：

表 10-39　引起误解的对齐的 Basic 例子

```
EmployeeName      = InputName
EmployeeAddress   = InputAddress
EmployeePhone     = InputPhone
BossTitle         = Title
BossDept          = Department
```

以上例子使人误以为是同类操作，但实际上却做了两件事：一个是有关雇员的数据，另一是老板的数据。格式化这段程序的目的是要区分这两件事，而改的方法就是分别把各自的等号对齐并在中间加一空行，如表 18-40 所示。

表 18-40　是正确的对齐等号的 Basic 例子

```
EmployeeName    = InputName
EmployeeAddress = InputAddress
EmployeePhone   = InputPhone

BossTitle       = Title
BossDept        = Department
```

18.5.4　格式化续行

程序布局一个伤脑筋的事情是如何安排好续行。对于一行写不下，而在下一行继续的语句

行，你能按标准格数退后吗？要把它与关键字对齐吗？对赋值语句又怎样续行？

下面的方法是一个有用的、协调的方法，特别是对 Pascal，C，C++，Ada 及其它支持写长变量名的语言更有用。

使续行明显。有时必须要把一个语句拆成两句写，原因可能是一个语句太长而一个标准行内无法装下，或把什么都放在一行里显得很不合理。这种情况下，放在第一行中的那部分要清清楚楚地表明它仅是一个语句的一部分。断句最好的方法是若第一行部分独立出来则它有明显的语法错误。表 18-41 是一些例子：

表 18-41 这些 Pascal 例子不完全部分很明显

```
while（ PathName[StartPath + Pos ] <> ';' ） and              ——and 表示这个语句不完整
            ( (   StartPath + Pos   )   <= length （PathName ）)do
…
TotalBill   := TotalBill + CustomerPurchases [ CustomerID ]+      ——表示这个语句不完整
            SalesTax( CustomerPurchases[ CustomerID ] ) ;
…
DrawLine(Window.North , Window.South , Window.East , Window.west, ——表示这个语句不完整
          CurrentWidth，CurrentAttribute   ) ;
…
```

除了能告诉读者这个第一行部分不是一个完整的句子外，这种断句的方法也可避免在修改时出错，如果你把续行部分去掉了，那么第一行看起来不仅仅是一个忘了括号或分号的问题，它是缺成份。

把紧密关联的元素放在一起。当你断开一个句子时，把相关的事物放在一起，如数组下标，子程序参量等。表 18-42 是一个不太好的例子：

表 18-42　断句不大好的例子

```
CustomerBill :=   PrevBalance（ PayMentHistry[ CustomerID ] ）+ LateCharge（
                        PaymentHistory[ CustomerID ] ）  ;
```

不可否认，上例中的断句法确实使分开的两个部分明显不能独立，但却无谓地增加了不可读性。你可能发现在有些情况下这种断法可以，但本例却没必要这么断。把数组下标与数组名放在一起是必要的。表 18-43 是较好的断句法：

表 18-43　这个 Pascal 程序断句较好

```
CustomerBill   :=   PrevBalance( PaymentHistory[ CustomerID ] ) +
                        LateCharge( PaymentHistory[ CustomerID ] ) ;
```

子程序调用的续行可退后标准格数。假如你的循环或条件语句缩排三个空格，那么子程序调用语句的续行也后退三个空格。表 18-44 是这个例子：

表 18-44　这个 Pascal 例子的子程序调用的续行用了标准退格

```
DrawLine ( Window.North,   Window.South,   Window.East,   Window.West,
            CurrentWidth, CurrentAttribute );
SetFontAttributes(   Font.FaceName,   Font.Size,   Font.Bold,   Font.Italic,
```

```
    Font.SyntheticAttribute[ FontID ].Underline,
    Fout.SyntheticAttribute[ FontID ].Strikeout)
```

另一可选用的办法是把续行起始处放在上行的第一个参数处，如表 18-45 所示；

表 18-44　这个 Pascal 程序把续行放在第一个参数下以示强调子程序名

```
DrawLine( Window.North , Window.South , Window.East , Window.West ,
          CurrentWidth , CurrentAttribute ) ;
SetFontAttributes( Font.FaceName , Font.Size , Font.Bold , Font.Italic ,
                   Font.SyntheticAttribute[ FontID ].Underline,
                   Font.SyntheticAttribute[ FontID ].Strikeout  ) ;
```

从美学观点来看，这个程序看起来有点参差不齐，但它却突出了子程序名，因而你选用它是一个个人爱好问题。

使续行的结尾易于发现。上面几例有一个问题便是不容易找到每行的结尾，一种可选择方法是把每个参数放在自己单独的一行，最后用一个闭括号括住以示结束，表 18-46 是这种例子。

表 18-46　这个 Pascal 例子，格式化子程序调用续行时把一个参数名放一行

```
DrawLine
    (
    Window.North ,
    Window.South ,
    Window.East ,
    Window.West ,
    CurrentWidth ,
    CurrentAttribute
    ) ;
SetFontAttributes
    (
    Font.FaceName ,
    Font.Size ,
    Font.Bold ,
    Font.Italic ,
    Font.SyntheticAttribute[ FontID ].Underline ,
    Font.SyntheticAttribute[ FontID ].Strikeout
    );
```

最后使每个调用句子的结束显得很清楚。实际上，仅有很少的子程序调用句子需分成几行来写。以上提供的三种处理子程序调用续行的方法都较好，但你得用得一致。

控制语句的续行应编排标准格数。假如你一行写不下 for、while 循环语句或 if 语句，那么其续行退后的格数与循环或 if 语句后的语句退后格数一样。表 18-47 是这样的两个例子。

表 18-47 这个 Pascal 例子处理好了控制语句续行的编排

```
while( PathName[ StartPath + Pos ] <>  ';' )  and
      ( ( StartPath + Pos) <= length ( PathName ) ) do        ——这个续行缩进标准格数
```

```
    begin
    …
    end;

for RecNum := ( Employee.Rec.Start + Employee.Rec.Offset )    to
( Employee.Rec.Start + Employee.Rec.Offset + Employee.NumRecs ) do
        begin
…
        end;
```

因为 C 的格式化有点不一样，那么在 C 中类似的写法如表 18-48 所示。

表 18-48　C 中处理控制语句续行的例子

```
while( PathName[ StartPath 十 Pos ] != ';') &&
    ( ( StartPath + Pos )<= length ( PathName ) )
    {
     …
    }
for( RecNum = Employee.Rec.Start + Employee.Rec.Offset;
    RecNum <= Employee.Rec.Start + Employee.Rec.Offset + Employee.NumRecs;
    RecNum ++ ）
    {
    …
    }
```

这种写法正好满足了本章早些时候提出的原则，语句的续行部分处理得很合乎道理——总是在它所对应的句子下退格。这种缩排可连续做下去，只不过到最后比最开始的句子多退几格罢了，这种方法跟别的好方法一样可读易修改。有时你可能通过加空格或空行的方法来增强可读性，但要记住，当你想着怎样提高可读性时不要忘了维护性。

赋值语句的统行要写在赋值号以后。受上面处理续行方法的影响，你可能也要在赋值语句行时想到缩排标准空格，但千万别这样做。这时若后退标准空格会严重扰乱了赋位语句组的直观。如表 18-49 所示：

表 18-49　这个 Pascal 例子是个不好的实例，因在赋值语句的续行时也后退标准空格

```
CustomerPurchases    :=    CustomerPurchases + CustomerSales（CustomerID ）;
CustomerBill         :=    CustomerBill + CustomerPurchases ;
TotalCustomerBill    :=    CustomerBill + PreviousBalance（ CustomerID ）+
    LateCharge ( CustomerID ) ;
CustomerRating       :=    Rating ( CustomerID, TotalCustomerBill ) ;
```

本程序的目的是想通过对齐赋值号，来表明这是一组相关例子。但续行 LateCharge（customerID）因为仅后退标准格数而影响了这种直观性，这种情况下后退标准格数却没有获在别的地方所应有的可读性。这时用行尾布局法却很好。表 18-50 表明了这种情况该怎样格式化代码:

表 18-50　这个 Pascal 例子显示了如何用行尾布局来格式化赋值语句续行

```
CustomerPurchases   :=   CustomerPurchases + CustomerSales ( CustomerID );
CustomerBill        :=   CustomerBill + CustomerPurchases;
TotalCustomerBill   :=   CustomerBill + PreviousBalance ( CustomerID ) +
                         LateCharge ( CustomerID );
CustomerRating      :=   Rating ( CustomerID, TotalCustomerBill    );
```

18.5.5　每行仅写一条语句

较高级的几种语言如 Pascal，C，Ada 允许一行写多条语句。Fortran 要求注释行由第一列开始写起，而实际语句则由第 7 列或以后开始写起，但自由格式化对这一要求是一个很大的提高。自由格式有多种好处，但却有把多条语句放在一行的不好之处。

```
i=0，j=0，k=0；DestroyBadLoopNames(i，j，k);
```

这一行包含了几条语句，而这几条语句实际可以各自放一行的。

把几条语句放在一行的支持意见认为，这样所占屏幕空间或打印纸空间小，因而能同时看到更多的代码，而且这也是把相关语句组织在一起的方法，有些程序员甚至称这是给编译程序提供线索的方法。

这些原因都很好，但要求你一行写一条语句的理由更充分：

- 一行写一句准确地反映了程序的复杂性。它不因为把几句写成一行而隐藏了程序的复杂性因而显得琐碎。语句是复杂就是复杂，语句是简单就该如所见的那么简单。
- 一行写几条语句也不会给现代的编译程序提供什么优化线索，现在的组合编译程序并不依赖格式化的线索去作出它们的选择。这一点后面详述。
- 一行写一句使程序能从上往下读，而不是从上往下之中又有从左至右。当你要搜索某一代码行时，你的眼睛只需停留在代码的左边缘而不用担心因为一行含两句而要往右看。
- 一行写一句容易寻找语法错误，因为编译程序仅提供出错的行号。如果你的一行里有多句，这个行号能告诉你到底是哪句出错了吗？
- 一行写一句容易用面向行调试程序来设计代码。如果一行写几句，调试程序一次调试这几句，你就不得不切换成单条语句的汇编方法。
- 一行写一句容易编辑单条语句——去掉一行或把一行转变成注释。如果一行写多句，你就还得编辑其它语句。

C 中应避免产生副作用。副作用是一条语句的附加作用，而不是主要作用。在 C 中，++运算符在一行含有别的运算时就是一种有副作用的运算，同样，在条件语句中用左侧赋值就是一种副作用。

副作用使代码难读，比如在表 18-51 中，如果 n 等 4，那么输出结果是什么呢？

表 18-51 一个意义不明的有副作用的 C 程序

```
Prinf(”%d. %d\n”,++n, n+2);
```

那么是 4 和 6 呢?还是或 5 和 7 呢?还是 5 和 6 呢?以上都不对。第一项++n 结果为 5。但是 C 语言却并没有定义运算的次序，因而编译程序既可计算第二项 n+2 在第一项之前，也可在第一项之后，结果可以是 6 或 7，这得看不同的编译程序。表 18-52 是如何修改以看得更清楚：

表 18-52　避免产生意义不清副作用的 C 语言

```
++n;
printf(”％d％d\n”，n，N+2);
```

如果你还没有想清为何把产生副作用的运算单独放在一行的话，那请指出表 18-53 中程序的都做了什么。

表 18-53　这个 C 程序一行中运算太多

```
strcpy(char * t，　char * s）
  {
   while (* ++ t = * ++ s )
  }
```

许多有经验的程序员可能不觉得有何复杂，因为它很熟悉，一看可能会说它是 strcpy()函数。但这个程序却不是 strcpy()，它有错误。这就是那种你一看就认识但却没有仔细读它因而忽略了错误的情形。表 18-54 修改后显得可读：

表 18-54　这个 C 程序把各操作放在各自的行中增强可读性

```
strcpy( char * t，char * s )
    {
    do
        {
         ++ t;
         ++ s;
         * t = * s;
        }
        while ( * t !=　‘\n’ )
    }
```

这个程序中的错误非常明显。很明显,t 和 s 在把*S 赋给 *t 之前已各自加 1，因而漏过了第一个字符的复制。

提高程序的性能不能由把多个运算放在同一行来判定。因为以上两个 strcpy()程序逻辑上是等价的，那你可能认为编译程序应当生成相同的代码。但当你用两个程序复制 50000 个字符的字符率时，你会发现第一个程序用 3.83 秒而第二个只用 3.34 秒。

即使你读有副作用语句时非常轻松，不要指望别人也跟你一样，大多数程序员读这种程序时要读两次才能理解。宁可花些脑筋去理解你程序中可能出现的许多问题，也不要把一些语法问题接合在一个特殊的语句中。

18.5.6　数据类型定义布局

注意数据类型定义的对齐。对齐数据类型定义的主要原因和对齐赋值语句是不同的。你已知道所有的数据类型定义都是有联系的，因此这种对齐的好处是显得整齐且能在右列很快浏览下来。定义表的右侧的内容各不相同，有些语言的右列含的是变量类型，而另一些则合的是变量名。如果所用语言为 Pascal，那么你必须如表 18-55 一样把数据类型写在表的右侧:

表 18-55　怎样对齐数类型走定义的 Pascal 例子

```
SortBoundary:Integer;
InsertPos:          Imeger;
InsertVal:          SORT_STRING;
LowerBoundary:   SORT_STRING;
```

这个例中，你或许要说着定义表的右侧部分根本看不出什么，除非你要经常看看 Integer(整型）的而非 SORT_STRING 型的。

但在 C 语言中，数据名是放在右边的，如表 18-56 所示：

表 18-56　C 语言中对齐数据类型定义的例子

```
int              SortBoundary;
int         InsertPos;
SORT_STRING InsertVal;
SORT_STRING LowerBoundary;
```

这个例子中，浏览列表的右侧部分是有用的，因为这部分是变量名。

每行只定义一个数据。如以上两例所示，你一行只能定义一个数据。若一行仅含一个数据定义，那在其后加注释是轻而易举的事情，同时修改起来也很方便；要找出某个变量名很容易，不需从左到右读完一行；在出错时容易发现和修改语法错误。

相反地，在表 18-57 的数据定义中，你能说出 CurrentBottom 是哪一类型的变量吗？

表 18-57　C 程序把几个交量定义都放在一行了

```
int   Rowldx，ColIdX；COLOR PreviousScreen，CurrentScreen，NextScreen;
      POINT PrevlousTop, PreviousBottom，CurrentTop，CurrentBottom，NextTop,
      NextBottom；  FONT PreviousFace，  CurrentFace, NextFace;
      COLOR Choices [ NUM_COLORS ];
```

这种定义变量的方法并不常见，但关键是因为所有的定义都挤在一起，你很难找出某个指定的变量。变量类型也很难一下子找出。上例就是那种可读性差的例子。

表 18-58 几个变量定义挤在一行的例子

```
RowIdx，ColIdx:          Integer;

CurrentScreen,
NextScreen,
PreviousScreen：      COLOR

CurrentBottom,CurrentTop,
NextBottom，NextTop,
PreviousBottom,
PreviousTop:          POINT;

CurrentFace，  NextFace,
PreviousFace：            FONT;
```

```
Choices :          array[ 1..NUM_COLORS ] of COLOR;
```

这也是种常用的形式，它试图把相应的项对齐，但是每一个仅用一种类型名、一行中放了几个变量名，无论从美学观点还是从可读性来看，这种形式并不比上一例好多少。

那么表 18-59 中的 NextScreen 的类型是什么呢？这种形式中一行仅定义一个变量，每行都有一个变量类型，相当于一个完整的定义，这种对齐看起来相当美观。

表 18-59　一行说明一个变量的例子

```
RowIdx:              Integer;
ColIdx:         Integer;

CurrentBottom:  POINT;
CurrnetRop:     PONIT;
NextBottom:     POINT;
NextTop:        POINT
PreviousBottom:  POINT;
PreviousTOP:     POINT;
CurrentScreen:  COLOR;
NextScreen:     COLOR;
PreviousScreen: COLOR;
Choices:        array[ 1..NUM_COLORS ] of COLOR ;

CurrentFace:    FONT;
NextFace:       FONT;
PreviousFace:        FONT;
```

当然，这种类型也多用了一大堆空格，并不能说明形式增强了可理解性。但是如果 Sally Programmer, Jr 要我去检查她的代码数据定义如第一种形式，我会说：“太不好了，简直无法读。”如果是第二种形式，我会说：“嗯，可能我还不如去看第一种。”若是第三种，我会说：“当然，好极了。”

有意识地安排定义顺序。在第三种形式中，各定义按类型组织在一起。按类型把定义组织在一起比较合理，因为同一类型的变量用在相关运算中的可能性较大。另一种情况是，你可能会按变量名开头字母的先后顺序来安排定义表。虽然按字母顺序排的原因很多，但我的感觉是没必要这样做，如果你的变量表很长，这时按字母排会对你有帮助的话，那么这时程序也肯定相当长，那我还是建议你把它分成几个小的子程序，每一个程序也就仅含几个变量了。

C 语言中，在定义指针变量时，把星号（* ）放在紧靠类型名之后。在指针变量的定义中，常见的是把星号（* ）靠近变量名而非类型名，如表 18-60 所示：

表 18-60　　这个 C 程序中，指针走义容易引起误解

```
EMP_LIST       *Employees;
FILE           *InputFile;
```

虽然这种写法相当普遍,但却容易引起误解。事实上,若一个变量的指针变量,那么星号属于类型的一部分，若星号靠近变量，它显得好像星号是变量名的一部分，引起误解的原因是这

个变量使用中可以用也可以不用星号。把星号放在靠近类型名的后面则不会有这种误解。如表 18-61 所示：

表 18-61　这个 C 语官的指针变量定义写法正确

```
EMP_LIST *    Employees;
FILE *        InputFile
```

这种写法的一个问题是若在一行定义多个变量，星号到底属于谁呢？实际上星号仅属于第一个变量。此时若几个变量都是指针类型，那么先定义一个指针类型名，然后用这个类型名来定义变量，表 18-62 就是这样的例子：

表 18-62　这个 C 程序用指针类型名来定义指针变量，很好

```
EMP_LIST_PTR    Employees;
File_PTR        InputFile;
```

18.6　注释布局

一个好的注释能极大提高程序的可读性。若注释不成功，则会帮倒忙，而能否安排好注释对于是增强还是损坏可读性关系甚大。

注释行与相应的代码同样缩排。一个好的缩排方式能有助于理解程序的逻辑结构，好的注释行不应当破坏这种美观的缩排方式。表 18-63 中你能看出其逻辑结构吗？

表 18-63　　这个 Basic 程序注释行的缩排不正确

```
for TransactionID = 1 TO MaxRecords

' get transaction data
      read TransactionType
      read TransactionAmount

'procass transaction based on transaction type
      if TransactionType = CustomerSale then
            AcceptCoustomerSale(TransactionAmount)

      elseif TransactionType＝  CustomerReturn then

'either process return automatcally or get manager approval,
'if required
      if TransactionAmount >= MgrApprovalRequired then

' try to got manager approval and then accept or reject return
' based on whether approval is granted
            GetMgrApproval(APProval)
            if Approval = True then
                  AcceptCustomerReturn(TransatiopnAmount)
```

```
                else
                    RejectCustomerReturn(TransactionAmount)
                end if
        else
    'manager approval not required, so accept return
                AcceptCustomerReturn(TransactionAmount)
            end if
        end if
    next TransactionlD
```

从这个例子中你得不到多少程序逻辑结构的线索，因为注释行不正确的退格完全掩盖了代码的可观性。你可能很难相信还有人如此糊涂地用这样一种缩排方式，但事实上我确实见过，甚至在教料书上。

表 18-64 程序与 18-63 一模一样，所不同者是注释行的退格方式。

表 18-64 这个 Basic 程序正确地缩排了注释行

```
for TransactionID = 1 To MaxRecords

'read Transaction data
    read TramsactionType
    read TransacionAmount

'process transaction based on transaction type
    if TransactionType = Customersalc then
        AcceptCustomerSale(TransactionAmount)

    elseif TransactionType = CustomerReturn then

    ' either process return automatically or get manager approval,
    'if required
    if TransactionAmoun t>=    MgrApprovalRequired then

            'try toget manager approval and then accept or reject the return
            'basedo on wherther approval is granted
            GetMgrApproval(Approval)
            if Approval = True then
                    AcceptCustomerReturn(TransactionAmount)
            else
                RejectCustomerReturn(TransactionAmount)
            end if
    else

    'manager approval not required, so accept return
```

```
            AcceptCustomerRetun(TransactionAmount)
        end if
    end if
next TransactionID
```

表 18-64 中程序的逻辑结构更明显、研究表明，注释对程序的可读性并不都有帮助，因为注释行安排不当常破坏了程序直观性。从以上这些例子你不是已有所感触了吗？

把注释行至少用一个空行隔开。如果想很快地理解一下程序，那么最有效的方法是只读注释不读代码。把注释行用空行隔开有利于读者浏览代码。表 18-65 是这样的例子。

表 18-65 这个 Pascal 例子用空将把注释行分开

```
{ comment zero }
CodeeStatermentZero ;
CodeStatementOn ;

{ comment one }
CodeStatementTWO ;
CodeStatementThree ;
```

当然也有人在注释行前后都加空行。两行空行可能要占用较多屏幕空间，但有人可能主张这样代码更好读，如表 18-66 所示：

表 18-86　用两行隔开注释行的 Pascal 例子

```
{ comment zero }

CodeStatementZero;
CodeStatmentOne;

{ comment one }

CodeStatmentTWO;
CodeStatmentThree;
```

除非屏幕空间是个要优先考虑的因素，否则这种写法相当美观。记住一种规定的存在比其细节更重要。

18.7　子程序布局

于程序由单条的语句、数据、控制结构、注释组成，即包含本章所讨论到的所有部分。本节提供一些如何安排好子程序的指导。

用空行把予程序备田分分开。在子程序的头、数据和常量名定义及程序体之间加空行。

对子程序的参数用标准缩排。跟别的情况一样，子程序头的安排可选用方法是：任意布局、用行尾布局或标准缩排。大多数情况下，标准缩排更准确、连贯、可读、易维护。

表 18-67 是两个没注意子程序头布局的例子。

表 18-67　这 C 子程序没注意子程序头的布局

```
B00LEAN ReadEmployeeDate( int MaxEmployees, EMP_LIST * Employees,
    FILE * InputFile ,  int * EmployeeCount ,  BOOLEAN * IsInputError  )
    ……

void InsertSort( SORT _ARRAY Data, int FirstElmt, int LastElmt )
```

这种子程序头纯属实用主义的东西。计算机肯定能读，但对人呢？没注意到这一点使读者吃了苦头，还有比这更糟糕的吗？

第二种可选用方法是用行尾布局，这种方法一般都显得较好。

表 18-68　这个 C 程序用行尾布局法格式化程序头

```
B00LEAN ReadEmployeeData (   int              Max Emlpoyees,
                             EMP_LIST *       Employees,
                             FILE *           InputFile,
                             int   *          EmployeeCount,
                             BOOLEAN *        IsInputError    )
…
void   InsertSort(   SORT_ARRAY     Data,
                     int            FirstElmt,
                     int            LastElmt   )
```

行尾布局法显得整齐而美观。但主要问题是修改时要花好多功夫，即维修性不好。比如函数名由 ReadEmployeeData()改成 ReadNewEmployeeData()，它就使第一行往后了而不能与下面四行对齐，因而需要重新格式化其余四行，即加空格。

表 18-69 的例子是用标准缩排方式格式化子程序头，一样地美观，但维护性好。

这种形式在修改时也能保持美观性。

表 18-69　这个 C 程序用标准缩排方式格式化程序头，可读，易维护

```
 B00LEAN ReadEmployeeData
    (
      int              MaxEmployees;
      EMP_LIST *       Employess;
      FILE *           InputFile;
      Int *            EmployeeCount;
      BOOEAN *         IslnputError
      )
    …
 void InsertionSort
    (
        SORT_ARRAY       Data,
        int              FirstElmt,
        int              LastElmt
    )
```

要改子程序名，这种改动也不会影响参数的对齐。如果要增删一个参数，仅需修改一行（外加一个分号），而其直观外形依然存在。这种布局形式中，你能很快得到你所需要的信息，而不必到处搜寻。

在 Pascal 中这种形式可直接应用上去，但却不能用到 Fortran 上去，因为 Fortran 的参数定义是在于程序定义以后才进行。

在 C 中用新的子程序定义方式。ANSI　C 标准中的子程序头定义方式与原始 C 定义方式不一样，但大多数编译程序和少数的程序员仍支持旧的方式。不幸的是，旧方式已不太好用了。表 18-70 是新旧两种方法的比较：

表 18-70　C 程序的新旧两种子程序头

```
void InsertionSort( Data, FirstElmt, LastElmt)          ——旧的方式
        SORT_ARRAY      Data,
        int             FirstElmt,
        int             LastElmt,
        {
        …
        }

void InsertionSort                                      ——新的方式
    (
    SORT_ARRAY      Data,
    int             FirstElmt,
    int             LastElmt
    )
{
…
}
```

旧方式定义子程序头时，先到了一次变量，然后列表再定义了一次各自的类型，这样在一开始就要再次提到同一变量名，如果要修改它，你还得记住要在两处修改。在第二版的《The C Programming Language》（《C 程序语言》）中，Kernigham 和 Ritchie 强烈要求用新的定义方法。

在 Fortran 中需单独定义参微，这时最好按程序参数表中的顺序依次定义。Fortran 的一大不幸是你得先在子程序头中有一参数列表，然后再定义一次，正好与旧形式的 C 一样，Fortran 也通常把参数与局部变量放在一起定义．形成一个很宠杂的列表。表 18-71 即是此例；

表 18-71　这个 Fortan 的例子参数顺序混乱

```
SUBROUTINE READEE( MAXEE, INFILE, INERR )
INTEGER         I, J
LOGICAL         INERR
INTEGER         MAXEE
CHARACTER       INFILE*8
INTEGER         EMPID( MAXEE )
```

这个程序段初看很整齐，因为几个相应项各自上下对齐了。但整齐并不能解决主要问题。

当我还小的时候，我不允许我盘中的食物接触放在我盘里的他人的食物；但我老了时，却喜欢把几样食品混杂起来吃。但我还是不太喜欢把参数与局部变量混杂起来定义。表 18-72 显示了该怎样在 Fortran 子程序中定义变量或参数：

表 18-72　这个 Fortran 子程序注意到了参数顺序

```
SUBROUTINE READEE(MAXEE,    INFILE, INERR)     ——参数以参数表的顺序先义
    SINTEGER          SMAXEE
    SCHARACTER    INFILE*8
    SLOGICAL          INERR

* local variables                    ——再定只局部变量，与参数定义分开

INTEGER    I
INTEGER    J
INTEGER    EMPID( MAXEE )
```

这个程序遵从了本章的几个观点，即把子程序变量放在子程序下定义且缩排几格、每行只定义一个变量、对齐变量的各相应、把注释与相应代码对齐、用空行把注释行与其它代码分开等。

18.8　文件、模块和程序布局

格式化技巧除了应用于子程序外还有更大应用的空间，如怎样组织一个文件内的各子程序？一个文件中哪个子程序该放在第一？

把一个板块放在一个文件里。一个文件不仅仅只放了一大堆代码。如果你所用语言允许，一个文件里应该放且只放那些支持某一目的的子程序集合。一个文件应该是把一些相关的子程序包装成一个模块。

一个文件中的所有子程序构成一个模块。一个模块才是程序中体现你设计的那一部分（或许仅仅是一个逻辑分支）。模块是一个语义学上的概念，文件则是一个物理操作系统上的概念。两者之间的关系是并存关系。未来的环境可能倾向于强调模块而不强调文件，但现在两者是有联系的，所以应当等同相待。

把一个文件内的子程序区分清楚。要把一个子程序与别的子程序区分开来至少要用两个空行。空行的作用与星号行或虚线行意义差不多，但空行更易维修，用两或三空行来区别表明这些空行是用来分隔于程序的而不是子程序内部的。一个例子如表 18-73：

表 18-73　这个 Basic 例子用多个空行分隔字程序

```
Function Max!(    Arg! ,    Arg2! )
                                    ——这是一个空行，子程序中空行的典型使用
'find the arithmetic maxinmum of Argl and Arg2
if( Arg1! > Arg2! )    then
        Max! = Arg1!
else
        Max! = Arg2!
```

```
endif
end Function


Function Min!(Arg1!, Arg2!)

'find the arithmetic minimum of Arg1 and arg2
if ( Arg1!< Arg2!) then
    Max! = Arg1!
else
    Max! = Arg2!
endif
end Function
```

至少两个空行分开子程序

空行比其它任何分隔符都好输入，但效果却是一样好。用三个空行来区分使得程序内部的空行与用来分隔子程序的空行的差别更明显。

如果一个文件里有多个俱块，要把这些模块区分得清清楚楚。相关的子程序组织成一个模块。当读者测览你的代码时，他应当很容易地知道到哪算一个模块。模块之间应用更多的空行来区分。一个模块犹如你书中的一章。在一本书中，每一章都是从一个新页开始，开始的章节名用大写字。可用同样方法来突出每一个模块，表 18-74 是分隔模块的例子。

表 18-74　注意这个 Basic 程序如何分隔模块

```
Function ConvertBlanks$( Mixedstring$ )

'Create a String identical to NixedSttring$ except that the
'blanks are replaced with underscores.

dim i%
dim StringLength%
dim workString$

WorkString $ = ""
StringLength% = Len(MixedString$)
for i% = 1 to StringLength%
    if (mid$(MixedString$,i%,1)="" then
        Workstring$ = WorkString$ + "_"
    else
        Workstring$ = WorkString$ + mid$(MixedString$,i%,1)
    endif
 next i%
ConvertBlanks$ = WorkString$
end Function
```

这是一个模块中的最后一个子程序

```
'-----------------------------------------------------
'  MATHEMATICAL   FUNCTIONS
'
' This module contains the program's mathematical functions
'-----------------------------------------------------
Function Max! (Arg1!, Arg2!)

'find the arithemetic maximum of Arg1 and Arg2

if (Arg1! > Arg2!) then
        Max! = Arg1!
else
        Max! = Arg2!
end if
end Function


Function Min(Arg1!, Arg2!)

' find the arithemetic minimum of Arg1 and Arg2

if (Arg1! < Arg2!) then
        Max! = Arg1!
else
        Max! = Arg2!
end if
end Function
```

新模块开始用几个空行和模块名标记

这是一个新模块的第一个子程序

这个子程序仅用空行与前面的子程序分开

避免过分突出模块内部的注释，如果你模块内部的注释或程序间的间隔用星号替代空行，那么你实际上已经很难用别的符号或方法来强调突出模块间的间隔了。表 18－75 是这样的例子。

表 18-75　这个 Basic 程序过分突模块了

```
'************************************************************
'************************************************************
'  MATHEMATICAL FUNCTIONS
'  This module contains the program's mathematical functions
'************************************************************
'************************************************************


'************************************************************
Function Max! (Arg1!, Arg2!)
```

```
'*************************************************************
' find the arithmetic maximum of Argl and arg2
'*************************************************************
 if ( Arg1! > Arg2! )    then
       Max! = Arg1!
 else
       Max! =    Arg2!
 endif
end Function

'*************************************************************
Function Min!(Arg1!, Arg2!)
'*************************************************************
' find the arithmetic maximum of Arg1 and Arg2
'*************************************************************
if( Arg1!<arg2! ) then
       Max!=    Arg1!
else
       Max!= Arg2!
endif
end Function
```

这个例子中，那么多地方都用星号来强调，到最后实际上什么也没强调。这个程序成了一个星号密布的森林似的。虽然有些美观、好看，但从格式化角度来看，什么也没有。

如果你真想为分隔程序而用一长行的特殊符号，那么用别的字符就不能仅依靠星号，例如，用星号区别分开模块而虚线分隔子程序．空行分隔注释。不要把两行星号或虚线分在一起。表18-76即是这样的例子。

表 18-76　这个 Basic 例子是一个很好的格式化例子，两种不同字符不放在一起，而是中间有空格，很清楚

```
'*************************************************************
'         MATHEMATICAL FUNCTIONS
'
' This module contains the Program's mathematical function.
'*************************************************************
'-------------------------------------------------------
Function Max!( Arg1! ,    Arg2! )
'-------------------------------------------------------
'find the arithmetic maximum of Arg1 and Arg2

if( Arg1! < arg2! ) then
          Max! =    Arg1!
else
   Max! = Arg2!
   endif
   end Function

'-------------------------------------------------------
```

```
Function Min! (Arg1!,Arg2!)
'-------------------------------------------------------

'find the arithmetic minimum of Arg1 and Arg2

if(Arg1!>Arg2!) then
        Max!=Arg1!
else
        Max!=Arg2!
endif
end Function
```

以上的建议仅适用于这种情形：即你的语言限制在一个程序中所含的文件个数，这时你不得不在一个文件中放几个模块。如果你用 C 或某一版本的 Pascal、Fortran、Basic 等，它们支持多个源文件，这时你最好是一个文件放一个模块。即使是在一个模块里，你也要用上述方法来仔细分隔各子程序。

把各子程序按字母顺序排列。在一个文件中把相关于程序组织起来的一个方法是按子程序名开头字母的先后顺序．如果不能把一个程序分解成为模块或编译程序寻找子程序不是很快，那么按字母顺序能减小搜寻时间。

在 C 中，细心地组织源文件。以下是 C 语言中所含源文件的标准顺序：

- 关于文件描述的注释
- 包含文件
- 常量定义
- 宏函数定义
- 类型定义
- 全局变量及输入函数
- 全局变量及输出菌教
- 文件内部变量及函数

18.8.1　检查表

布局

简述

- 格式化的本意是要显示代码的逻辑结构吗？
- 格式化的形式能始终一致吗？
- 格式化后使代码易于维护吗？
- 格式化后改进了可读性吗？

控制结构

- begin－end 对中代码避免再次缩排了吗？
- 一系列的块结构用空行相互分隔了吗？
- 复杂的表达式格式化后可读性增强了吗？
- 单条语句块始终一致地格式化了吗？
- Case 语句的格式化与其它控制结构格式化相协调吗？
- goto 语句格式化后自己显得更清楚了吗？

单条语句

- 把单条语句分成几行，每行都明显地不能作独立行看待了吗？
- 续行有意识地退格了吗？
- 相关语句组对齐了吗？
- 不相关语句组不应对齐，你是这样的吗？
- 每行至多含一条语句吗？
- 每个语句避免副作用了吗？
- 数据定义时相应项对齐了吗？
- 每行至多定义一个数据，是吗？

注释

- 注释行与它所对应的代码退同样格数了吗？
- 注释行的形式易修改吗？

子程序

- 子程序的参量格式化后各参数易读、易修改、易加注释吗？
- 在 C 中是否用新子程序定义方法呢？
- Fortran 中，参数定义是否和局部变量定义分开？

文件、模块和程序

- 若语言允许有多个源文件，每个源文件仅含一个模块是吗？
- 一个文件内的各子程序是否用空行清楚隔开？
- 如果一个文件含几个模块，那么每个模块中的子程序是否被组织到被清楚隔开？
- 各子程序是否按字母顺序排列？

18.9　小　结

- 布局首先考虑的是去显示程序的逻辑结构。评价这种考虑是否达到目的标准有：准确性、连续性、可读性、易维护性。好看是第二条标准——比较弱的标准。如果以上几条标准都达到了而且程序也较好看，那么布局一般就成功了。
- 在 C、Pascal、Basic 中用纯块结构模仿及 begin－end 作块边界，这两种布局形式行之有效。在 Ada 中用纯块结构。
- 结构化代码是有其自身目的的，你最好还是用一些约定俗成的布局形式而少来创新，以保持与别人协调一致。若你的布局形式与约定的不一样，那么很有可能影响你程序

的可读性。

- 有关部局的好多观点纯属一个信仰或者说个人喜欢问题，努力把客观需要和主观喜好分开。遵从一些明显地规划，以选择你所喜欢的布局形式。

第十九章　文　档

目录

相关章节

布局：见第 18 章
PDL——代码流程：见第 4 章
高质量子程序：见第 5 章
在交流中编程：见 31.5 节和 32.3 节

假如程序标准合理的话，大多数编程人都喜欢写文档。就像布局那样，好的文档是编程人员投身编程中自豪的标志。软件文档可以采取很多形式，在描述了文档后，本章将介绍文档的“补充”即“注释”。

19.1　外部文档

软件工程中的文档既包含原码表的内部信息，又包含源码表的外部信息。通常的形式有单独的文件或者综合资料。大体上，正式软件工程中大多数文档都位于源码外部。实际上，大约一项大工程全部力量的三分之二放在了创建文档上，而不是源代码上。这和输出相似。外部结构文档要在高层上同编码相联系，在低层上和编制前的阶段文件相联系。

综合资料。一个综合资料，或者软件开发资料是一个非正式文档，它包含着供开发者在编程中使用的记录。“综合”广义地讲，通常指是常规的或是特殊的资料。综合资料的主要目的是提供一套其它地方没有描述的设计规则。很多软件工程中，有指定最少资料的标准。例如：相关需求的备份、开发标准的备份等。当前编码和综合资料的设计规则，通常都仅用于内部。

详细设计文档。详细设计文档是低层设计的文档。它描述模块层或程序层的决定，所考虑的选择对象、所选用办法的原因。有时这些信息包含在一个正式文档中。这种情况下，详细设计和结构是不同的，有时它主要包含收集在“资料”中开发者的记录。有时——常常——它仅存在于编码本身当中。

19.2　编程风格作文档

和外部文档相比，内部文档可见于程序内部。它是最详细的一种文档。由于内部文档和编码联系最密切，故也是当编码被修正后最可能保持正确的那种文档。

对编码层文档的主要贡献不是注释，而是好的程序风格。风格包含好的程序结构，直接使用和易于理解的方法、好的变量名、好的子程序名、命名常量、清晰的布局和最小的控制流及灵活的数据结构。

这里有一风格不好的代码段：

不好的编程风格导致差的文档的例子：

```
for i＝1 to Num do
 MeetsCriteria[i]:=True;
 for i:=2 to Num/2 do
 j:=j＋1;
 while（j<＝Num） do begin
 meetsCriteria[i]:=False;
 j:=j＋I;
 end;
 for i=1 to Num do
 if Meetcriteria[i] then
 writeln(i，"meetsCriteria.");
```

你知道这段子程序做什么吗？它并非必须保密。它是较差的文档，不是因为它的语言描述而是因为它缺乏好的编程风格。变量名是非正规的，布局是粗略的。下面是相同的，但改进了编程风格就使得它的意思很清晰了。

好的编程风格文档例子：

```
for PrjmeCandidate := 1 to Num do
    IsPrime[ PimeCandidate ] := True;


for   Factor :=   2 to Num / 2 do
    FactorableNumber := Factor ＋ Factor;
    while( FactorableNumber <= Num ) do
        begin
        IsPrimes[ FactorableNumber ] := False;
        FactorableNumber := FactorableNumber ＋ Factor;
        end;
  for PrimeCandidate := 1 to Num do
  if IsPrime[ PrimeCandidate ] then
          writeln ( PrimeCandidate , 'is prime.' );
```

不像第一段代码那样，这段第一眼就会让你知道它和基本数字有关。第二眼便可反映基本数字在 1 和 Num 之间。对于第一段代码，至少读两遍才能找出循环结束的位置。

两段代码的区别和注释无关。然而第二段更具可读性，达到了清晰明了的境地：这种编码依靠好的程序风格来承担大部分的文档任务。在好的编码中，注释可称得上是“锦上添花”。

19.2.1　检查表

子程序

- 每一个子程序名都确切地描述了要做什么事吗？
- 每一个子程序详细定义任务吗？
- 程序会从它们的子程序中获益吗？
- 每个子程序的接口处明确吗？

数据名称

- 类型名的描述足以帮助文件数据说明吗？
- 变量名好吗？
- 变量仅用于命名这个目的吗？
- 循环计算变量能给出更多的信息吗？
- 用枚举类型变量来代替标记或逻辑变量了吗？
- 命名常量没有用来代替数字或字串吗？
- 类型名、枚举类型名、命名常量、局部变量、模块变量和全局变量中的命名规则不同吗？

数据组织

- 附加变量在需要时要清零吗？
- 变量的引用彼此间很接近吗？
- 数据结构简化会导致降低其灵活性吗？
- 复杂的数据存取是通过子程序来完成的吗？

控制

- 正常编码路径清晰吗？
- 相关语句分成一组了吗？
- 相对独立的语句都组成子程序了吗？
- 正常情况跟在 IF 后，而不是 ELSE 后吗？
- 控制结构简化会降低灵活性吗？
- 像一个定义完好的子程序那样，每个循环执行一个且仅一个功能吗？
- 嵌套层次是最少吗？
- 逻辑表达式用附加的逻辑变量、逻辑函数和功能表简化了吗？

布局

- 程序布局显示出它的逻辑结构吗？

设计

- 代码直观吗？它的编写巧妙吗？
- 实现细节可能隐去了吗？
- 程序编写是立足于问题域而不是计算机科学或语言结构域吗？

19.3　注释还是不注释

注释写得差要比好容易些，注释有帮助但更有破坏性。关于注释的热烈讨论常常听起来像道德观点的哲学争论，这使我想到，假如 Socrates 是一个计算机编程者，他和他的学生可能会有下面的讨论。

注释讨论

人物：

THRASYMACHUS：固执，相信他读的每件事。

CALLICLES：从古老学校中经实践磨练出来的老手——一个真正编程人员。

GLAUCON：一个年轻的，自信的，热情洋溢的计算机迷。

ISMENE：一个年长的编程员，讨厌说大话，热衷于一些实际工作。

SOCRATES: 聪明的老编程员

地点:每周小组讨论会上

“我想建议一个我们工程的注释标准，”Thrasymachus 说，“我们的一些编程员仅仅注释他们的代码，每个人都知道没有注释的代码不可读。”

Callicles 说道:“你一定是学校中的新潮流，并且超过我所想象的。注释是学术上的补救方法，但做任何编程工作的人员都知道，注释使得代码更难读而不是容易。英语不比 C 或 Pascal 语言准确，这就造成了许多累赘。编程语言说明简练且符合要点。假如你不能使代码清楚，你怎么会使注释清晰呢？此外，注释也跟不上代码的变化。若你相信过时的注释，你将会失败!”

“我同意这个观点，”Glaucon 插话道，“注释的代码更难读，是因为这意味着更多的东西要读。我已经是不得不读这些代码，为何我还要必须去读这些注释呢？”

“等一会儿”，Ismene 把她的咖啡杯放入两块糖后说:“我知道注释能被乱用，但好的注释是很可贵的，我不得不保留有注释的代码和无注释的代码，但我更喜欢保留有注释的代码。我认为我们不应有一个标准指出应该每多少行代码有一行注释，但我们应鼓励每一个人去注释。”

Socrates 问道:“假如注释浪费时间，Callicles 你回答我，为何每个人都使用他们？”

“或因为他们被要求那样做，或因为他们读到那里时注释是有用的。但没有人认为注释总是有用的。”

“Ismene 认为注释有用，她在这里已三年了，保留你的无注释代码和那些有注释的代码，并且她更喜欢有注释的代码。你是怎么实现的？”

“因为注释是用更罗嗦的方式重复代码，故此他们没有用处。”

“这儿等一下，Thrasymachus 打断道，“好的注释不是重复代码或解释它，而是使代码更清楚。注释在高于代码的抽象水平上解释代码要做什么事。”

“对”，Ismene 说道，“我浏览注释会发现，在这一部分我应改变什么或应集中精力干什么，你说注释重复代码根本没有帮助也对，因为代码已把每件事都说了，当我们需要注释时，我是想让它像读书中的题目或目录表一样。注释能帮助我们发现正确的部分，然后再读代码。在一段程序语言中读一句英文要比分析二十行代码快得多了。”Ismene 给自己又倒了一杯咖啡。

“我认为人们拒绝使用注释是因为：①它们的代码已相当清楚；②认为其它编程人员对他们的代码极感兴趣；③认为其它程序员比自己更聪明；④懒惰；⑤怕其它人推算出他们的代码

是如何工作的。”

“注释对代码检查有很大帮助”，Ismene 接着说道，“假如某人声称他们不需要写注释，那么检查中一定会出问题——几个有疑问的人开始说：“这段代码中你们试图要做什么？接着他们开始增加注释。假如他们自己不那样做，最终他们的老板也会强迫他们做的。”

“我不责怪你懒惰或担心别人会发现你的代码的工作原理，Callicles，我已研究过你的代码相信你是公司中最好的编程员之一，但要当心啊!若你使用注释，你的代码对我的研究来说会更容易些。”

“但是，注释是资源的浪费，"Callicles 反驳道：“一名好程序员的代码应是自我解释的，你想知道的每件事都在代码中。”

Thrasymachus 从椅子中站起来说道：“不可能！编译程序知道任何事都在代码中！你可能也会争论你想知道的每件事在二进制可执行文件中！假如你去读懂它若不在代码中这意味着发生什么呢？”

Thrasymachus 意识到自己站着便坐下了。“Socrates 是可笑的、为何你对注释有无价值进行争论呢？我读到的每件事说明他们是有价值的，并且应当合理地使用，我们在浪费时间。”

“冷静些，Thrasymachus 问一下 Callides 他从事编程多久了!”

“多久了，Callicles? "

“好吧，大约十五年前我开始 Acropolis IV 的编写，我猜想我已经看了大约一打的主要系统了。两个这样的系统就超过了五百行代码，因此我知道我在谈论什么。注释是极其没用的。”

Socrates 看着这个年轻的程序员，“就像 Callicles 所说；注释有一些实际的问题，你没意识到那并不需要更多的经验。假如注释错了，将会更加糟糕。”

“即便是没有错，也是没用的，”Callicles 说道，“注释并不比程序语言更准确，我更希望根本就不要注释。”

“Callides 和 lsmene 的观点是说降低的精确性是注释的优点——这意味着你可以用少量的话表达更多的含义。你写注释是因为相同的原因。你要使用高层语言。他们给你一种高层的抽象，我们都知道抽象水平并非是程序员最有力的工具之一。”

“我不赞同这个观点。不应集中在注释上而应集中在使代码更可读。好的程序员可以从代码中读出代码的意图，当你知道某人的代码有错误时，读他的意图你会知道他做的如何吗？”Glaucon 对自己的观点很满意。Callicles 点点头。

“你的话听起来好像你从未被迫修改其它人的代码，”Ismene 说道，Callicles 突然间好像对天花板上铅笔标记很感兴趣。“为何你不试着读你自己半年或一年前写的代码？你能够提高你的读代码能力，并且提高注释水平。你并非不得不选这个。我是必须用注释，通过注释读几百行代码便发现要改变两行。”

“好了！能创览代码会是很方便的，”Glaucon 说道。他已经看过一些 Ismene 的程序并从中受以启发。“但 Callicles 的其它观点怎么样呢？我已编程几年了，但据我所知没有人修正过他们的注释。”

Ismene 说：“好了，是和非难以定论，假如你把注释看作是神圣的，而代码看作是可疑的，你就会麻烦的。实际上，在注释和代码间找一个分歧就意味着两者都错。有些注释不好并不意味着所有注释都不好。我去厨房另取一瓶咖啡。”Ismene 离开了房间。

Callicles 说；“我对注释的反对意见是认为它是浪费资源。”

Socrates 问道：“谁能想办法把写注释花费的时间减到最少？”

“设计 PDL 的子程序，然后在程序间转化 PDL 到注释和填充代码。”Glaucon 说道。

Callicles 说：“好了，只要注释不重复代码就可以。”

“写注释使得想你的代码在做什么变得更难了，“Ismene 从厨房中回来后说道。“假如难于注释，或者代码较差或者你并不十分地理解。任何一种情况，你都要在代码花费更多的时间，所以花在注释上的时间不是浪费。”

“好了，”Socrates 说道，“我不能考虑任何更多的问题，我想 Ismene 得到了今天你谈话的精华。我们鼓励加注释，我们对它不能是无知的。我们要对代码检查以便每个人对这种有帮助的注释有一个好的看法。若我们有困难不能理解别人的代码，就让他们知道如何去改进它。”

19.4 有效注释的关键

下面的子程序做什么呢？

```
{write out sums 1..n for all n form 1 to Num}
Crnt    :=  1;
Prev    :=  0;
Sum     :=  0;
for   i  := 1 to Num do
        begin
        writeln（ Sum）;
        OldSum    :=  Sum;
        Sum       :=  Crnt + Prev;
        Prev      :=  Crnt;
        Crnt      :=  OldSum;
        end;
```

你最好猜想一下！

这个子程序计算第一个黄金分割数字 Num。它的编码风格稍优于本章开始的子程序的风格，但注释是错误的，如果你盲目地注释，你就会走错了方向。

下面这段代码如何呢？

```
{set Product to "Base"}
Product :=  Base;

{loop from 2 to "Num"}
 fot   i := 2 to Num do
 begin

 { Multiply "Base" by "Product" }
 Product   :=  Product * Base;
 end;
```

你的猜测是什么?

这段子程序把一个整数 Base 提高到 Num 整数幂次。这段子程序的注释是准确的，但缺乏更

多的信息。它们仅是代码本身的更罗嗦的方式。

最后这里有一个子程序：

```
{compute the square root of Num unsing the Newton－Raphson approximation}
 r :＝  Num / 2;
 while ( abs( r - ( Num / r ) )  <  Tolerance )  do
     r :＝  0.5 * ( r  ＋  ( Num / r ) );
```

程序的目的是什么？

这段程序计算 Num 的平方根，程序代码并不大，但注释是精确的。

哪个子程序对你正确计算更容易呢？由于没有一个子程序是完美的——由于变量名取的较差，然而简单地说，这些子程序说明了这些内部注释的优点和弱点。子程序 1 有一个不正确的注释。子程序 2 的注释仅重复了代码，故此是没用的，只有子程序 3 的注释起了作用。差的注释不如没有注释。注释 1 和 2 没有注释都比有这差的注释要好。

下面的部分描述了写有效注释的关键点。

注释的种类

注释可以分成五类：

代码的重复

重复的注释，用不同的词重申了代码的内容。它没有给读者提供代码的附加信息。

代码的解释

解释性注释，典型地用于解释复杂的，有效的和灵敏的代码段。这种情况下，他们是有用的，但常常是由于代码是易混淆的。假如代码复杂到需要解释，那么改进代码总比增加注释更好些。使代码本身清晰，然后使用总结或注释。

代码中的标记

标记注释并非是故意留在代码中的注释。它是给开发者的记录，表示工作还未做。一些开发者的标记注释为语法错误的标记（例如******），因而编译程序标记它并提醒他们要做更多的工作。其它开发者把一套特殊字符放人注释中，因而他们可以发现它们，但编译程序不能识别它们。

代码的总结

总结代码的注释做法是；它简化一些代码行成一或两句话。这样的注释比起仅重复代码而使读者比读代码更快的那种注释更有价值了。总结注释是相当有用的，特别是当其它人但不是代码的编者试图修改代码时。

代码意图的描述

意图这一层上的注释，解释了代码的目的。意图注释在问题一级上，而不是在答案一级操作。例如：

｛get current employee information｝获取当前雇员的信息

是一句意图注释，而

｛update Employee structure ｝修改雇员记录结构

是一句利用答案的总结描述。在 IBM 六个月的学习，发现编程员最常说“理解最初的编程

意图是最难的问题”（Fjelstad 和 Hamlen 1919）。意图注释和总结注释的区别不总是清楚的，但常常这不是重要的。意图注释的例子本章始终都给出了。

为全部代码接受的注释仅是意图和总结注释。

有效注释不是时间的浪费。太多注释和没有注释一样糟糕。你可采取一个合理的中间数量。

由于两种共同的原因，注释要花费许多时间去写。第一，注释可能会浪费时间或令人厌烦——脖子疼痛。假如这样，重写一个新的注释。需要许多繁重工作的注释是很令人头疼的。假如注释难于改变，他们就不必改变了；不准确和错误注释比根本没有注释更糟糕了。

第二，用语言描述程序做什么并不见得容易，所以注释会更难些。这常是你并不了解程序做什么的标志。你花在注释上的时间，应更好理解程序的真正时间，那是不管你是否注释都要花费的时间。

使用风格不应打断或妨碍修改

任何太具想象力的风格都会妨碍维护。例如，选取下面的注释部分将不便维护：

难以保存的注释类型的 Fortran 例子：

```
C           变量                    含义
C           ……                  ……
C           XPOS.....           X 坐标位置（以米为单位）
C           YPOS.....           Y 坐标位置（以十为单位）
C           NPCMP.....          计算标志（=0 若不需要计算，
C                                       =1 若需要计算）
C           PTGTTL....          合计
C           PTVLMX....          最大值
C           PSCCRMX....         最大可能的值
```

假如你说这些起头的园点（……）将难以维护，那么你就对了！他们看起来很好，但没有他们会更好些，他们将对修改注释的工作增加负担，你宁愿有准确的注释而不是好看的注释，假如有这种选择的话——常常是这样的。

下面是另一个难以维护普通风格的例子：

C 语言的难以维护的注释风格的例子：

```
/*************************************
*  模型：GIGATROM.C                  *
*  作者：Dwight K.coder              *
*  时间：2014 年 7 月 4 日           *
*                                    *
*  控制二十一世纪程序的代           *
*  码开发工具。这些程序的           *
*  入口点在这个文件的底部的         *
*  行, 程序名为 Evaluatecode()      *
*************************************/
```

这是一个好看的注释块。很清楚，整个块自成一体，并且块的开头和结尾都很明确。对这

个块还不清楚的是它变化起来是否容易。假如你必须在注释底部增加文件名，那么你对右边漂亮的星号栏的就得重新编辑。假如你想要改变这段注释，那么你就要去掉左边和右边的星号。实际上这意味着这个块不便维护，因为要做比较多的工作。假如你按一个键便可得到几列整齐的星号，那就太好了。不要使用它，他们难以维护的问题不仅是在星号上面。下面的注释看起来并不好，但它肯定便于维护：

便于维护的 C 语言的注释风格例子：

```
/*****************************
        模型：GIGATRON.C
        作者：Dwight K.Coder
   日期：2014 年 7 月 4 日
   控制二十一世纪程序的代码
   开发工具。这段程序的人口
   在文件底部，程序名为 EvaluateCode()
*****************************/
```

下面有一个极难维护的风格：

难于维护的 Basic 语言的注释风格例子：

```
   •   设置颜色枚举类型
   +————————————————————————+
         ...
   •   设置菜单枚举变量
   +————————————————————————+
     ...
```

很难知道注释里破折号行起始和结尾的加号其值是多少？但容易猜出每次注释的变化，下划线不得不调整以使得结尾的加号处于正确的位置。当一个注释行被分成两行时你将如何办呢？你将怎么安放加号？去掉注释里的文字以使得它仅占据一行吗？使两行具有相同的长度？当你试图不断应用它时，这种办法的问题会很多。

关键点是应注意如何去分配你的时间。假如你花费了大量时间增加和删除破折号以使得加号对齐，你就无法编程了，而且是在浪费时间。找一个更有效的方式。在用加号下划线的情况中，可以进行选择，使得注释没有任何下划线。假如你需要使用下划线来强调，就找别的方法，而不用加号来对注释进行强调。一种办法就是用一个标准的下划线，不管注释的长度它都一样长。这样的线不需要维护，你可在开始位置用一个文字编辑宏来定义它。

使用 PDL 编码过程来减少注释时间

假如写代码前你勾划了注释中的代码，便可通过几种方法实现。当你完成了代码，注释也就完了。在你填写低层的程序语言代码前，你可以获得高层 PDL。

在你进行过程中注释

写代码时，可以选择注释，直到工程结束再停止注释，这样做有很多的优点。在它自己的权限内，这变成了一项任务，使得看起来比每一次做一点时更有效率。以后再注释会花费更多的时间，因为你必须记住或算出代码在做什么而不是在书写你已想好的东西。因为你可能已忘记了设计中的假设和细节，所以这样也不会准确。

反对在进行编程时注释的观点认为“当你集中精力写代码时，不应当分散精力去写注释”。正确的答案是，假如你极其用心地写代码，注释会打断你的思路，你需要先设计 PDL，然后把 PDL 转化成注释。需要集中精力编代码是一个警告信号，假如你的代码很难，在你对代码和注释担忧前应简化它。若你使用 PDL 分类你的想法，编码是直接的而注释是自动的。

最佳数量的注释

工程有时采用一个标准，比如“程序必须至少每五行便有一行注释”。这个标准说明了程序员未写清晰代码的特征，且未指出原因。

若你有效地使用 PDL 编码过程，最后你会得出结论：第几行代码就要有一行注释。然而，注释的数量对过程本身来说是副作用。不能集中在注释的数量上，而是要集中是否每条注释都有效。假如明白了为什么写注释以及清楚了本章中涉及的其它法则，你就会有足够的注释了。

19.5 注释方法

注释可依照它所提供的层次：程序、文件、子程序或单独行而采取几种不同的技巧。

注释单独的行

好的代码中，注释单独代码行的需要是很少的，这里是一行代码需要一条注释的两种可能原因：

- 单独行复杂，需要一条解释
- 单独行曾有一个错误，你需要这个错误的记录

这是一些注释一行代码的准则：

避免本身无关的注释

很多年前，我听说一个故事，一个维护程序员被从床上叫起来，检查一个不正常的程序。程序的作者已离开了公司，不能来到。维护程序员开始对程序无能为力，但仔细检查了程序后，他发现只有一条注释。注释如下所示：

```
MOV AX,723h          ; R.I.P.L.V.B
```

对程序研究了一整夜后，对注释感到困惑，程序员做了一个成功的修改，然后回家睡觉。几个月后，他遇到程序作者发现注释代表的意思是：“Rest in peace. Ludwig Van Beethoven.”Beethove 死于 1827 年（十进制），也就是 723h（十六进制）。那个地方需要 723h 的事实与注释无关。

结束行注释及其问题

结束行注释是在代码行末尾出现的注释。这里有一个例子：

结束行注释的 Basic 语言例子：

```
for EmpID = l TO MaxRererds
    GetBonus( EmpID, EmpType, BonusAmt )
    if EmpType = Manager then
        PayMgrBonus( EmpID, BonusAmt )          'Pay intended, full amount
    elseif EmpType = Programmer then
```

```
        if BonusAmt >= MgrApprovalRequired then
            PayProgrBonus( EmpID, StdAmt() )    'pay company std. amount
        else
            PayProgrBonus( EmpID, BonusAmt )   'pay 1ntended, full amount
        end if
    next EmpID
```

一些情况下虽然结束行注释有用，但它还存在几个问题。注释必须位于代码的右边以便不影响代码的外形结构。假如你没有整齐地设置它们，它们会使你的表看起来好像它已经通过了洗衣机似的。

结束行注释倾向于难以安排。假如你使用很多，需要浪费时间去整理。这样的时间没有花费在学习更多的代码上，仅仅花费在按空格键或 TAB 键这样乏味的工作上。

结束行注释倾向于是隐蔽的。行的右边通常没有足够的空间，要求在一行内保留注释意味着注释要短。工作也就转到使行尽可能地短而不是尽可能的清楚。注释通常尽可能隐蔽地结束。使用 132 列的监视器和打印机，你能清除这个问题。

避免结束行注释在单独行上

除了实际问题外，结束行注释还有其它几个概念性问题。这有一套结束行注释的例子：

```
MemToInit := MemoryAvailable();          { get amount of memory available }
Pointer := GetMem( MemToInit );          { get a ptr to the available memory }
Zeromem( Pointer,    MemToInit );        { at memory to 0 }
…
FreeMem( Pointer );                      { tree memory allocated }
```

结束行注释的系统问题是难以为一行代码写一个有意义的注释。很多结束行注释只是重复代码行，其害处超过益处。

避免给多行代码结束行注释

假如结束行代码是超过一行的代码，这种格式并未显示注释是为哪些行的。这里有一个例子：

```
for RateIdx L := l to RateCount do begin          { Compute discounted rates}
    begin
    LookupRegularRate( RateIdx, ReguinrRate );
    Rate[ RateIdx ] := RegualarRate * Discount[ RateIdx ];
    end;
```

虽然这个特殊注释的内容是好的，它的布局却不好，你必须读整个注释和代码才能知道是否注释适于特定的说明或整个循环。

何时使用结束行注释

这里是三条反对使用结束行注释的异议。

使用结束行注释注解数据说明

结束行注释对注解数据说明是很有用的，因为他们就像代码上的结束行注释一样没有系统问题，条件是你有足够的宽度。对于 132 列的，你通常能在数据说明旁边写一个有含义的注释。这里有一例子。

```
Boundary:     Integer;   { upper index of sorted part or array }
InsertVal:    String;    { data elmt to insert in sorted part of array }
InsertPos:    Integer;   { position to insert elmt in sorted Part Of array }
```

为保存记录而使用结束行注释

结束行注释在初始开发后，对代码的修改很有用。这种注释典型地包含了一个数据和程序员的初始情况，或者一个错误报告数。这里有一个例子：

```
for i := 1 to MaxElmts – 1 { fixed error #A423 10/1/92 (Scm) }
```

这样的注释可由控制版本软件很好地处理，但是如果你没有这种版本控制支持的工具，你需要维护工具去注解一个单行，这是一种解决办法。

使用结束行注解标记块的结束

一个结束行注释对于标记一长段代码的结束是很有用的，例如 `while` 循环或 `if` 说明的结束，本章后面有更详细的描述。

除了两个特殊情况，结束行注释有概念性问题，而且可以使代码很复杂．它们也很难编排和维护。总的来说，最好别用他们。

代码注释段

在程序中的大多数好的注释是一或二句描述代码段的注释。这里是个例子：

```
/* swap the roots */

OldRoot =    root[ 0 ];
root[ 0 ] =    root[ 1 ];
root[ 1 ] =    Oldroot;
```

这段注释并没有重复代码。它描述了代码的意图。这样的注释是相对地易于维护的。假如方法中出现了错误，注释并不需要改变。不是在意图层次上写的注释是难以维护的。

在代码的意图层次上书写注释

描述代码块的目的后面跟随注释。这里有一个效率低的注释例子，因为它没有在意图层次上操作：

```
{ check each character in 'InputStr" until a dollar sign
is found or all character have been checked }

Done      := False;
MaxPos    := Length( InputStr );
i         := 1;
while ( ( not Done ) and ( i <= MaxLen ) )
      begin
      if ( InputStr[ i ] = '$' ) then
            Done := True
      else
            i := i + 1
      end;
```

你可以通过读代码寻找一个＄并退出循环。这段注释部分有用，它概述了问题、缺点在于它仅仅重复了代码，在代码要做什么方面并未给你任何启发，下段注释会更好一些：

{ find '$'in InputPtr }

这段注释之所以好是因为它表明了循环的目的是要找一个＄，但在循环为什么需要找一个＄方面并未给你启发——换句话说，就是循环的深层意图。下面的注释更好些：

{ find the command -- word terminator }

这段注释实际上包含了代码列中没有的信息，也就是＄结束了一个命令字。仅仅读代码段你是没办法推论出的，所以注释是很有帮助的。

在意图层次上面考虑注释另一种办法是考虑如何命名一个子程序，这个例子所做的工作和你要注释代码相同。假如你正书写几段代码，每段都有一个目的，这并不难。上面代码的注释就是一个很好的例子。FindCommandWordTerminator()会是接下来的子程序名。其它的选择，Find$InInputString() 和 CheckEachcharacterInInputStrUntilADollarsignIsFoundOrAllcharactersHaveBeenChecked()，显然是不好的名字，对一个子程序名，你尽可能地不要用缩写进行描述。这种描述很可能就是意图层次上的注释。

如果代码是另一个例程的一部分，采取步骤把代码放入自己的例程中。假如它执行一完整的功能，并且你很好地命名了这个子程序，你就增加了程序的可读性和可维护性。

代码本身一直是你应检查的首要记录描述。上面的情况下，文字必须用一个命名常量替换，并且变量应提供更多的有关要做什么的线索。假如你想扩展可读性的边界，就增加一个包含研究结果的变量。在循环过程中更要做得清晰。下面是用很好的注释和很好风格书写的代码：

```
{ find the command-word terminator }

FoundTheEnd          :=  False;
MaxCammandlength     :=  Length( InputStr );
Idx                  :=  1;
whi1e ( ( not FoundTheEnd ) and ( Idx <= MaxCommandLength ) )
    begin
    if ( InputStr[Idx] = COMMAND_WORD_TERMINATOR ) then
        begin
    FoundTheEnd      :=  True;
    EndOfCommand     :=  Idx

    end
else
        Idx := Idx + 1
end;
```

假如代码足够好，它在接近意图层次上的注释。在这点上，注释和代码可能变得有点冗余，但很少有程序出现这种问题。

把注释段集中在为什么而不是如何上

解释某某如何做的注释，通常是在程序语言水平上而不是问题水平上的。对一个注释用如何执行一个操作去解释操作的意图几乎是不可能的，而且如何做的注释也常是多余的。下面的

注释告诉你代码做什么了吗？

```
/* if allocation flag is zero */

if ( AllocFlag == 0 ) ...
```

注释告诉你的比代码本身做的并不多。下面的注释怎么样呢？

```
/ * if allocating new member * /

if ( AllocFlag == 0 ) ...
```

这段注释好些，因为它告诉你一些不能从代码本身中推论出来的事情。这段代码通过使用含义丰富的命名常量而不是 0，但仍可得到提高。这里是这种注释和代码的最好版本。

```
/ * if allocaing new member * /
if ( AllocFlag == NEW_MEMBER )...
```

用注释告诉读者准备下面要做什么

好的注释告诉人们下面的代码将干什么。读者可以只浏览注释，对代码做什么并查找哪里有特殊的动作，注释应当在代码前说明。这种概念不仅是在编程的课堂里教，它应是商业实践中的标准。

对每条注释都计数

过多地注释没有优点。太多的注释会使代码表达的意思变模糊。不要写更多的注释，把这些额外的精力放在使代码本身更可读上面。

标记有疑问的地方

假如你发现代码中有不明确的地方，应把它放到一个注释中。假如你使用一个巧妙的手法而不是直接的办法来提高代码效率，应使用注释来指出直接的办法会是什么样的，并且指出通过使用巧妙的方法提高效率的。这儿有一个例子：

```
for ( i = 0; i < ElmtCount; i++ )
    {
    /* Use right shift to divide by two. Substituting the
        right-shift operation cuts the loop time by 75%. */

    Elmt[ i ] = Em1t[ i ]>> 1;
    }
```

例于中右移的选择是有目的的。有经验的程序员中，对整数右移功能上相当于除二是普通的知识。

假如是普通的知识，为何要描述它？因为操作的目的不是执行一次右移，它是要完成除 2 的功能。代码并未直接使用技巧。另外，大多数编译程序选整数除 2 的最佳方法任何时都是右移，这意味着可以提高精确度。这种特殊情况下，编辑程序并未采用除 2 的方法，并且节省了时间。有了这些描述记录，读者读代码时就会发现使用这些技术的意图。如果没有这些注释，就会感到迟疑，认为代码没有效率上有意义的提高，也不一定“聪明”。通常这种迟疑是合理的，因此记录这些异常情况还是很重要的。

避免缩写

注释应当是明确的，不用进行缩写就应当可读，避免注释中出现缩写，除最普通的缩写外。除非你在使用结束行注释，否则，不要使用缩写，那是一种过了时的技术。

区分开主要和次要的注释

在一些情况下，你想区分不同层次的注释，表明一个详尽的注释是前面的大注释的一部分。你可用两种方法来解决。

你可以试着在主要的注释下面划线，次要的注释下面不划线来实现，就像下面这样。

```
/* copy the strng portion Of the tab1e，along the way
      omitting strings that are to be deleted */
/*    ------------------------------------------------------------ */

/* determine number of strings in the table */
…

/* mark the strings to be deleted */
…
```

这种办法的缺点是你被迫在多于你真正想划线的注释下面划线。假如在一条注释下面划了线，那就是假定后面未画线的注释比它次要。因而，当你写第一条注释，而它并不比划线的次要，那么它也必须划线，这样一直持续下去。结果是太多的下划线，或者不断地在一些位置划线而其它的不划线。

这个题目有几种变化但都有一个共同的问题，假如你把主要注释用大写表示，而次要注释用小写表示，你用太多大写注释的问题代替太多划线注释的问题。一些程序在主要注释上用大写字母起始，而次要注释上没有大写字母起始，那么次要注释则易被忽略。

一个更好的办法是在次要注释前面使用省略号。这里有一个例子；

```
/* copy the string portion of the table, along the way
      omitting strings that are to be deleted */

/* ... determine number of strings in the table */

…
/* ... mark the string to be deleted */

…
```

另一个方法也常常是最好的方法是把主要注释放进自身的例程中。逻辑上，例程应是“平行的”，在大约相同的逻辑层次上有自己的动作。假如你的代码存在一个例程中区分成主要的和次要的动作，例程就不平行了。把复杂的操作分开放进自己的例程中，使其成为两个逻辑上平行的例程，而不是一个逻辑上起伏的例程。

这种主要和次要注释的讨论不适合于循环和条件环境中的交错代码。这样的情况下，交错提供了对注释进行逻辑确认的线索。这种讨论仅适用于顺序代码段中，其中几段构成一个完整操作并且一些段从属于其它段。

错误或语言环境独特点都要加注释

假如有一错误，它可能没有记录。即使它在某处已经记录过，它也会在你的代码中再次记录。若它是个未记录的，它应在你的代码中被注释。

假定你发现库函数 WriteData(Data，NumItems，Blocksize)，除去当 Blocksize 等于 500 外，都可以正常执行。对 499 和 501 及其它你曾试过的值也很好，但你发现仅当 Blocksize 等于 500 时例程有一个缺陷。在使用 WriteData()的代码中，记录中 Blocksize 等于 500 时为何你有一个特殊的情况？下面是它的答案：

```
BolckSize := OptimalBlockSize( NumItems, SizeItem );

{ The following code is necessary to work around an error in
WriteData() that appears only when the third parameter
equals 500. ‘500’ has been replaced with a naemd constant
for clarity. }

if ( BlockSize = WRITEDATA_BROKEN_SIZE )
BlockSize := WRITEDATA_WORKAROUND_SIZE;

WriteData( File, Data, BolockSize );
```

违反好的编程风格的原因

假如你必须违反好的编程风格要解释为什么这样做。那样会阻止一个出于良好目的程序员把代码改成好的风格，那也许会打乱你的代码。这种解释会让你自己清楚在做什么，而不是由于粗心大意——给自己以信心，信心就是原因。

不要注释需要技巧的代码

一个最流行和有点冒险的编程说法就是，注释应当用在描述特别需要技巧的代码部分和比较敏感的部分。原因就是，人们应当了解他们在那些地方工作时需要小心。

这是个可怕的想法

注释需要技巧实现的代码是错误的办法。注释不应当挽救困难的代码、就像 Kernighan 和 Plauger 所强调的那样：“不要注释坏的代码编写——重新编写它”（1978）。

一项研究表明：源代码中有大量的注释也有很多的缺点，并且会耗费很多的开发精力（Lind 和 Vairavan 1989）。作者倾向于注释难代码。

当某人说，“这是真的需要技巧的代码”，而我听另一些人说，“这代码真的不好”。若某事似乎对你需要技巧，对别的人它也许不可理解。甚至某事对你似乎不需太多技巧而可能对以前来看过这种技巧的人来说是不可能解决的。假如你问自己：“这需技巧吗？”是的。你总会发现，再写时便不用技巧了，因而重写代码。使你的代码完善到不再需要注释，注释将会使它更加完善。

这种建议主要适于你首次写代码时。假如你维护一个程序，不想重写坏的代码，那么注释那些需技巧的部分是个好的练习。

注释数据说明

变量说明的注释描述变量中由变量名无法表达的部分。仔细地记录数据是很重要的；至少一家公司认为：标注数据要比标注数据使用过程更重要些。这里是一些注释数据的指导原则：

注释数字数据的单位

假如一个数字代表长度，要表明长度是否表示为英寸、英尺、米或者千米。假如是时间，要注明它是表示从 1980 年 1 月 1 日起经历的时间，还是从程序运行花费的毫秒，等等。假如它是坐标值，要注明是代表经度、纬度及高度，还是代表弧度或度。是否代表一个原点在地球中心的 xyz 坐标系统等等。不要假设单位是显然的，对一个新的编程员，他们若不知道，那么对在系统另外部分工作的人来说，他们也不知道。即使程序已经作了实质性的修改后，他们也不知道单位是什么。

注释允许数值范围

假如一个变量有一个期望值范围，注明期望的范围。假如语言支持限制范围——像 Pascal 和 Ada 那样——限制范围。假如没有限制，便使用注释标明这期望值范围。例如，若一个变量代表美元的钱数，注明你期望它在一到一百美元之间。假如一个变量表示一个电压值，表明它应处于 105 伏到 125 伏之间。

注释代码含义

假如你的语言支持数字类型，像 Pascal 上和 Ada 那样，用它们表达代码含义。若没有的话，便用注释表明每个值代表什么，使用一个命名常量而不是每个值一个文字。假如变量代表不同种类的电流，注释可采用 1 代表交流电流，2 代表直流电流，3 代表不确定的。

下面是一个记录变量说明的例子，它说明这三种建议的过程：

```
DIM CursorX% ‘horizonta1 cursor position : ranges from l..MaxCOls
DIM CursorY% ‘vertical cursor position: ranges from 1..MaxRows

DIM AnternaLength !    ‘length of antenna in meters; ranges is >= 2
DIM SignalStrength%    ‘strength of signal in kilowatts; ranges is >= 1

DIM CharCode%          ‘ASCII character code; ranges from 0..255
DIM CharAttrib%  ‘0=Pinin; 1=Italic; 2=Bold; 3=BoldItalic
DIM CharSize%          ‘size of character in Points; ranges from 4..l27
```

注释中给出了所有的范围信息。在支持多变量类型的语言中，你可以说明这些变量的范围。这儿有一个例子：

```
var
    CursorX:       l.. MaxCols;        { horizontal screen position of cursor }
    CursorY:       l.. MaXRows;        { vertical position of cursor on screen }

    AntennaLength :    Real;           { length Of antenna in meters; >= 2 }
    SignalStrength :   Integer;        { strength of signa1 in kilowatts; >= l }
```

```
CharCode:       0..255;       { ASCII character code }
CharAttrib:     Integet;      { 0＝Plain; 1＝Italic; 2＝Bold; 3＝BoldItalic }
CharSize:       4..127;       { size of character in points }
```

注释输入数据

输入数据可能来自于一个输入参数、一个文件或直接用户输入。上述应用指南也适于输入数据。要保证期望值和非期望值都被记录，在不接收某些数据的子各程序中，注释是记录唯一的方式。说明是另一种方式，假如你使用的话，代码会变得更具有自检能力。

位层次上的注记标记

假如一个变量用作一个位域，记录每一位的含义，就像下面的例子。

```
Var
    { The meanings of the bits in StatusFlage are as follows: }
        MSB     0       error detected: 1 = yes, 0 = no
                1-2     kind of error: 0 = syntax, 1 = warning, 2 = servere, 3 = fatal
                3       reserved ( should be 0 )
                4       printer status : 1 = ready,   0 = not ready
                …
                14      not used ( should be u )
        LSB     51    not used( should be 0 )}

            StatusFlags : integer;
```

假如这个例子用 C 编写，它将需要位域的语法以使得位域含义能够自我记录。

表明与有变量名的变量相关的注释

假如你有参照某一特定变量的注释，确信无论何时变量修正了，注释也会修正。一种办法可提高修正的准确性，即包含有变量名的注释。可以通过字符串搜索变量名，像寻找变量那样找到注释。

注释全局数据

若使用了全局数据，在它进行说明的地方标注好每一条数据。标注应表明数据的意图及为何它需要是全局的。命名规则应当是强调一个变量的全局状态的首要选择。假如命名规则未使用，注释应当填补这些。

注释控制结构

控制结构前面的空间，通常是放置注释的自然而然的地方。假如有一个 if 或一个 case 说明，你可以提供决定和结果的原因。这里有两个例子：

```
/* Copy Input field up to comma */
while   ( * InputStr != ','   && *InputStr != END_OF_STRING )
{
        *Filed = *InputStr;
        Field + +;
        InPutStr + +;
```

```
} /* while -- copy input field */    ——循环结束

*Filed = END_OF_STRING;

   /*    if at end of string,    all actions are complete */  ——条件的目的
   if (* InputStr != END_OF_STRING)
         {
         /* read past comma and subdsequent bkanks to get to the  ——循环目的
            next input field */

         *InputStr++;
         while (*Inputstr    == '' && *InputStr != END_OF_STRING )
               InputStr + +;
         } /* if -- at the of string */
```

这个例子给出了一些指导原则。

在每块 if、case 或循环前面加一条注释

这样的位置是注释的自然地方，这些结构常需要注释。使用注释来阐明控制结构的目的。

注释每个控制结构的结尾

使用注释说明结尾是干什么用的。例如：

```
end :     { for Clientldx —— process record for eath client }
```

在长的或嵌套的循环结尾处，注释是尤其有帮助的。在支持命名循环的语言中（例如，Ada）命名循环。在其它的语言中，使用注释来阐明循环嵌套。这是 Pascal 语言写的使用注释说明循环结构结尾的例子：

```
for Tableldx:= 1 to TableCount
   begin
   while( Recordldx < RecordCount)
      begin
      if ( not IllegalRecNum( Recordldx ))
            begin
               ...
            end; { if }
      end; { while}
   end; { for }
```

这些注释表明结束控制结构

这种注释技巧增补了由代码缺陷引起的关于逻辑结构的可见性线索。你不需要使用这种技术缩短没有嵌套的循环。然而，当嵌套很深或循环很长时这种技术才会有收效。

尽管它是有意义的，但加入注释并维护他们将是乏味的，避免这些乏味工作的最好办法是经常重写那些需要乏味记录的复杂代码。

注释子程序

例程层次上的注释是在典型计算机科学手册中的最坏建议。很多手册要求你在每个例程的

顶部有一堆信息，而不管它的大小或复杂性。这儿是个例子：

```
' * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
'
' Name: CopyString
' Purpose：              This routine copes a string from the source
'                        siring (Source$) to the target string (Target$ ).
'Algorithm:              It gets the length of Source $ and then copies each
'                        character, one at a time, into Target$. It uses
'                        the loop index as an array Index into both Source$
'                        and Target$ and increments the loop/array index
'                        after each character is Copied.
' Inputs:                Input $ the string to be copied
' Outputs：              Output $ The string to receive the copy of Inupt$
' Interface Assumptions : None
' Modification History :  None
' Author:                Dwight K. Coder
' Data Created:          10/1/92
' Phone:                 (222) 555-2255
' SSN:                   111-22-3333
' Eye Color:             Green
' Maiden Name:           None
' BloodType:             AB-
' Mother's Maiden Name: None
' * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

这是可笑的。拷贝字符串大概是一个很小的例程——大概少于五行代码。注释完全不符合例程的比例。关于例程目的和算法那一部分是受限制的，因为它很难把某事描述的像拷贝字符串例程那样简单。注释不都有用，他们仅仅是占用了列表中的空间。每个子程序和需要所有这些部分是不精确注释和维护失败的方法。它是许多的从没有收效的工作。这里有一些注释例程的指导原则；

保持注释接近于它们描述的代码

例程的序言部分不应含有大量注释的一个原因是这样的，注释会偏离它们描述的例程部分。维护过程中，偏离代码的注释不会和代码一样得到维护，注释和代码便开始产生分歧，并且突然间注释会变得没有用处。

相反，要遵循最接近原则，让注释尽可能地接近它们描述的代码。它们更可能得到维护，

也会一直都有价值。

下面描述了例程序言的几个部分，需要时应尽量包含。为了方便，创立一个常用的序言记录。不要认为每种情况都必须包括所有信息。把适用的部分用上而其它的去掉。

例程的顶部用一两句注释来描述

假如你不能用一句或两句短句来描述例程，你可能需要认真考虑。例程要做什么？创建一条简短描述是一个信号，标志着这种设计是否达到了最佳，否则重新回到画图设计桌上重新再来一次。简短的概括说明在所有实际例程中都应有。

在输入和输出变量说明时描述

假如你未使用全局变量，标记输入和输出变量最简单的办法就是紧跟着参量说明进行注释。这儿是个例子：

```
procedure InsertionSort
    (
            Var Data:       tSortArray;     { sort array elements FirstElmt.. LastElmt }
        FirstElmt:      Integer;        { index of first element to sort }
        LastElmt:       Interger;       { index of element of sort }
    );
```

这段程序是不使用结束行注释的一个很好特例，它在注释输入和输出变量时尤其有用。这种情况的注释也很好地说明了，使用标准缩写的值而不是例程参数表的结束行缩写，假如你使用结束行缩写，将没有空间给含义丰富的注释。例子中的注释，甚至是连标准缩写都受到了空间的限制，虽然本例中的代码多于八十列，但这不会是个问题。这个例子也说明了注释不是记录的唯一形式。假如你的变量名足够好，你就能够不去注释他们。最后，标记输入和输出变量，是避免用全局数据的很好的原因。你在什么地方标记呢？假定你在这庞大的序言中记录这些全局数据，那将会造成更多的工作，并且不幸地是实际上通常意味着这些全局数据并未得到记录，那真是太糟了，因为全局数据应该旬像其它事物那样得到注明。

输入和输出数据间的差别

知道哪个数据用于输入哪个用于输出是很有用的。Pascal 中相对容易地可看出，因为输出数据通过关键词 Var 进行，而输入数据不是。假如你的语言不能自动支持这些差别，就要增加注释。这里有一个 C 语言例子：

```
void StrinyCopy
    (
    char *    Target, /* out; string to copy to */
    char *    Source /* in; string to Copy from */
     );
  …
```

C 语言例程说明有些技巧，因为有些时候星号(*)表明变量是一个输出变量，更多时候它仅意味着变量作为指针类型比作为基本类型更易处理。你常不用明确区分输入和输出变量。

假如你的例程足够短，在输入和输出数据间保持了清晰的界限，注记数据的输入或输出状态可能没必要。假如例程较长，它对于帮助别人阅读例程是有帮助的。

注释界面假设

注释界面假设可能会被看作是其它汪释原则中的一部分。假如你做了变量状态的假设——合理和不合理的值、顺序分类过的数组等等——在例程序言中或在数据说明的地方注释它们。这种注释应出现在任何实际的例程中。

要保证使用的全局变量被注释，因为它有时是例程的接口，并且有时它看起来不像变量，所以它还是很危险的。

当你在写一个例程，并意识到自己在作一个接口的假设时，立即把它记下来。

记录程序变化的次数

记录一个例程自它初创建后所作的变化。变化是经常发生的，因为有错误并且错误集中在几个困难的例程上。例程中的许多错误意味着相同的例程可能有更多的错误。记录下例程测试到的错误过程，就意味着假如错误的数量达到某一点，你就知道应该重新设计和重新编写这些例程。

当你执行每个例程操作时不想看到每一个程序有错误。记录程序的变化过程可导致对例程顶部错误描述的混乱，但这种方法是可以自行排错的。假如在例程项部你得到了太多的错误，你自然的倾向将会去除错误并重写例程。很多程序员喜欢有借口来重写代码，采取他们知道的任何办法，使程序更好，那样做是很合理的。

注释例程极限

假如例程提供了数字的结果，要表明结果的准确性。假如计算在一些条件下没有定义。就记录这些条件。假如例程出现错误时有一个缺省的结果，就记录这些结果。假如例程期望仅在数组或某个大小的表上工作，就注明这些。假如你知道程序的修正，但会打断例程，就把他们记录下来。假如你在例程开发过程中陷入困境，也记录下来。

注释例程的全局效果

假如例程修正了全局数据，确切地描述全局数据做了些什么。就像在 5.4 节提到的，修正全局数据至少比仅仅阅读它会更危险些，因此修正时应谨慎行事，还要清楚地记录下来。像通常一样，假如注释变得太繁重了，重写代码会减少全局数据的使用。

注释使用的算法的来源

假如你使用了一个从一本书中或杂志中得来的算法，记录下它出处的卷号和页数。假如是你自己提出的算法，要表明在何处读者可以找到你所说的算法的介绍。

注释程序的各部分

一些程序员使用注释来标记他们程序的各部分，以便他们可以容易找到它们。C 语言中一个这样的技巧就是用下面这样的注释注记在每个例程的顶部：

```
/ * * *
This allows you to jump from routine to routine
by doing a string serach for    * * */
```

一个类似的技巧是标记不同种类的注释，要清楚他们描述的是什么。例如，Pascal 中你会用到{－X－，这里 X 是你用来表明注释种类的代码。注释{－R 一能够表明例程中描述的注释，{－I 一输入和输出数据，{－L 一表示描述本地数据等等。这种技巧允许你使用工具从你的源文件中抽取不同种类的信息。例如，你可以找{ -R- 来修正所有例程的描述。

注释文件、模块和程序

文件、模块和程序都由它们包含多个例程这个事实来表征。一个文件或模块应包含功能中所有例程。注释工作提供给文件、模块或程序的内容是相同的，所以仅参照注释“文件”，也可以假定这些指导原则也适于模块和程序。

注释的通用准则

在文件的开头，应该用一个注释块来描述文件的内容。以下是关于使用注释块的几条准则：

描述出文件的功能

如果程序中的所有子程序都在一个文件中，那么文件的功能使十分明显了。但如果你所从事的项目中用到了多个模块并且这些模块被放在多个文件中，那么就应当把某些子程序放入某一特定模块作出解释。同时说明每个文件的功能。由于在这种情况下，一个文件就是一个模块，因此有这个说明便足够了。

如果是出于模块化以外的考虑而把程序分成几个源文件的，那么对每个文件功用的清楚解释将对以后修改程序的程序员有非常大的帮助。假设某人想寻找某个执行任务的子程序，那么他能仅通过查阅文件的注释块便找出这个子程序。

把你的名字和电话号码放人注控块

在注释块中加入作者是非常重要的。它可以为继续从事这个项目的程序员提供关于程序风格的重要线索，同时，当别人需要帮助时也可以方便地找到你。

在注释块中加人版权信息

有些公司喜欢在程序中加入版权信息。如果你的公司也是这样的话，你可以在程序中加入下面的语句：

```
/*（c）Copyright 1993 Steve McConnell，Inc．All Rights Reserved. */
…
```

注释程序的注释变例

绝大多数资深的程序员都认为本书前面所论述的注释技术是非常有价值的。然而关于这方面科学的、确凿的证据则很少。但是当组合使用这些技术时，支待这一想法的证据则是非常确凿的。

在 1990 年，Paul Oman 和 Curtis Cook 发表了一组关于注释技术“Book Paradigm”的研究成果。他们想要寻找一种支持不同阅读风格的编程风格。一个目标是同时支持由下而上、自上而下和中心搜索。另一个目标是把程序分成比一长串相似代码可读性更好的大的程序块。Oman 和 Cook 想使这种风格同时提供低层次和高层次的程序组织线索。

他们发现如果把程序当作一种特殊的书并据此来组织格式就可以达到上述目标。在这本书中，代码及其注释是模拟书籍中的格式安排以便获得对程序的总体把握。

“前言”由一组在文件开头常见的注释组成。它与真正书籍中前言的作用是一样的，主要是为程序阅读者提供程序的总体信息。

“目录”中表示了文件、模块和子程序（就像书中的章）。其表示形式可能是表（如真正的书一样），也可能是图或结构字符。

“节”是子程序内部的名子部分——子程序声明、数据说明及可执行语句等等。

“参考资料”则是代码的参阅图，其中包括行数。

Oman 和 Cook 利用书籍和程序代码之间的相似性而创造的技术与第十八章和本章所论述的技术是类似的。

当 Oman 和 Cook 把用这种方法组织的程序交给一组职业程序员来维护时，发现维护这种用新方法组织的程序的时间要比维护同样内容用传统方法组织的程序的时间少 25%，而维护质量则要高 20%。同时，在 Toronto 大学（1990 ）进行的类似研究也证实了这一结果。

这种技术强调了同时对程序组织进行低层次和高层次说明的重要性。

检查表

有效的注释技术

通用部分

- 代码中是否包含了关于程序的大部分信息？
- 是否可以做到随意拿出一段代码便可以立刻理解它的意思？
- 注释是否注释了程序的意图或总结了程序的功用而不是简单地重复代码？
- 是否使用了 PDL——代码流程以减少注释时间？
- 对使人困惑的代码是否进行了重写而不是注释？
- 注释是否已经过时了？
- 注释是清楚正确的吗？
- 注释风格是否使得注释很容易修改？

语句和段落

- 是否避免了结束行注释？
- 注释的重点是“为什么”而不是“是什么”吗？
- 注释是否提示了后续代码？
- 每个注释都是合理的吗？是否删掉或改进了冗余、自相矛盾的注释？
- 是否注释了令人惊异的代码？
- 是否避免了缩写？
- 主要和次要注释间的区别明显吗？
- 用于错误处理或未说明功能的代码注释了吗？

数据说明

- 数据说明单元注释了吗？
- 数值数据的取值范围注释了吗？
- 是否注释了代码的含义？
- 对输入数据的限制注释了吗？
- 是否在位层次上对标志进行了注释？
- 是否在说明全局数据的地方对其进行了注释？
- 常数值是否被注释了？或者被用命名常量代替了吗？

控制结构

- 每一个控制语句都进行注释了吗？
- 冗长或复杂的控制结构进行注释了吗？

子程序

- 对每个子程序的功用都作出注释了吗？
- 在需要时，是否对关于子程序的其它信息进行了注释？包括输入 / 输出数据、接口假定、错误修正、算法来源、全局效果等？

文件、模块和程序

- 程序中是否有关于程序总体组织方式的简短注释？
- 对每个文件的功用都进行描述了吗？
- 注释块中有作者姓名和电话号码吗？

19.6　小　结

- 是否注释就像是立法。注释得好，是非常值得的，注释得不好，则是浪费时间而且有害。
- 源代码中应含有关于程序的绝大部分重要信息。只要程序还在运行，那么代码中的注释便不会丢失或被丢弃。把重要信息加入代码是非常重要的。
- 好的注释是在意愿层次上进行的，它们解释的是“为什么”而不是“是什么”。
- 注释应表达出代码本身表达不了的意思。好的代码应是自说明的。当你对代码进行注释时，应问一下自己“如何改进代码以使得对其注释是多余的？”，改进代码再加注释以使它更清楚。

第二十章　编程工具

目录

相关章节

版本控制工具：见 22.2 节
调试工具：见 26.5 节
测试支持工具规 25.5 节

现代编程工具减少了编程所需时间，用最先进的编程工具能提高产量达 50％以上，编程工具也能减少在编程时所需做的许多乏味的细节工作（Jones 1986，Boehm 1981）。

狗可能是人最好的朋友，但很少有哪种工具是程序员的最好朋友，正如 Barry Boehm 指出，20％工具起到了 80％工具的用途（1981）。如果错过了一个很有用途的工具，就可能失去了好好利用其许多用途的机会。本章主要介绍一下你能获得和买到的工具。

本章重点放在两个方面。首先概述结构性工具。需求排序、管理、端到端（end－to－end）进一步阅读开发工具是本章主要讲述的范围，本章结尾部分的能指导你获得软件开发方面更多的信息；第二，本章力求覆盖到各种工具而不仅仅只涉及几种特殊的分支，有几种工具应用非常普遍，我们仅提及其名字讨论一下，因为其版本、产品换代升级非常快。恐怕我们这里涉及的大部分信息已经落后了。所以你得求助于当地的代理商查问你感兴趣的工具。

如果你是一个工具专家，那么本章的内容对你没什么帮助，你可测览一个前面目录及后面 20.6 书的有关理想编程环境即可。

20.1　设计工具

现在的设计工具主要包含图形工具，这些工具主要用来绘图。设计工具有时包含在 CASE 工具中作为其一项主要功能；有些代理商干脆就认为设计工具就是 CASE 工具。

图形设计工具通常都允许你用普通的图形来表示出一种设计，如分层图，分层输入输出图、组织关系图、结构设计图、模块图等。有些图形设计工具仅支持一种图形符号，另一些则支持多种符号，大多数支持 PDL。

从某种意义上来讲，这些设计工具仅仅是想象和模拟那些画图器件。用一些简单的绘图工

具或笔和纸，你就可以做出任何绘图设计工具所能做的工作。但设计工具有提供准确数据的能力，而一般画图器件则不行。假如你画一个气泡图然后消去一个气泡，包括与该气泡连接的箭头和低层的气泡。当你增加一个气泡时，设计工具也会在内部重新安排组织。设计工具能让你在不同层次上抽象地移动，且检查你设计的连续性，有些甚至能直接从你的设计中产生代码。

20.2　源代码工具

有关帮助生成源代码的工具比设计的工具要丰富而成熟。

编辑

这种工具与编辑源代码有关

编辑程序

有些程序员估计他们在编辑源代码上的时间占全部时间的 40%，若真是这种情形，还不如多花几美元去买一种更好的编辑程序。

除了基本的文字处理功能，好的程序编辑程序常有以下特征：

- 编辑程序提供编辑和错误测试功能。
- 简述程序概貌（给出于程序名或逻辑结构而无具体内容）。
- 对所编辑语言提供交互式帮助。
- 括号或 begin－end 协调使用。
- 提供常用的语言结构（编辑程序在写入 for 以后自动完成 for 循环结构）。
- 灵活的退格方式（包括当逻辑改变时方便地改变语句的退格）。
- 相似语言间宏的可编程性。
- 搜索字符串的内存，使得公用字符串根本不需再输入。
- 规则表达式的搜索和替换。
- 在文件组中能搜索和替换。
- 同时编辑多个文件。
- 多层编辑。

考虑到原先多数人所用的原始编辑程序，你可能惊讶地发现，很少有哪种编辑程序包括以上所有这些功能。

多文件字符串转换程序

通常若能在多个文件中同时修改一字符串是很有用。比如，如果你想给一个子程序、常量、全局变量起一个更好的名字，你可能要在几个文件中同时修改。允许在几个文件中更改字符串的功能，使这项工作很容易完成，从而能很快就把一个子程序名、常量名或全局变量名修改过来，很方便。

AWK 编程语言是能方便地在多文件中搜索和替换的工具，AWK 的这种功用有时很难排序，它有时被说成是使用一“相关数组”。它的主要优点在干，擅长于处理字符串和多组文件。通常它在多组文件中修改字符串是很方便的。其它工具在处理多文件字符串修改时总有一些限制。

文件比较程序

程序常要对两个文件作比较。如果你为修改一个错误而作了几种修改尝式，最后需要把不成功的尝试消去，那么你就要把原文件与修改过的文件作比较，列出那些被修改的行。如果你和几个人合伙编一个程序，而想看看别人在你之后对代码作了何修改，可用比较程序把新版本的程序与你当时的程序作比较，找出不同之处。如果你发现了一处你不记得是不是在旧程序出现的不足，不用着急，可用比较程序比较新旧文件，看看到底修改过没有，找出问题之源。

源代码美化程序（source－code beautifiers）

源代码美化器修整你的源代码以使其看起来协调。它们标准化你的编排形式、对齐变量定义及子程序头、格式化注释使看起来协调及其它类似功能。在打印时，有些美化程序甚至使每个程序抬头从新的一页开始或进行其它格式代工作，许多美化程序让你的代码显得更漂亮。

除了使代码显得好看，优化程序在其它几个方面也很有用。如果一个程序由几个程序员编写而有几种不同的风格，那么代码美化程序能把这些风格转化成一种标准的形式。几种不同的风格转化成一种标准的形式，而无需某个人来完成此项工作。优化程序产生的缩排使人误解的可能性要比人小得多。如果用 for 循环忘了在循环体两端加上 begin－end，那么读者也许会认为你这部分都属于 for 循环，但对计算机则不这么认为。优化程序就不会有这种错误发生。当它重新格式代码中，它不会像人那样容易产生使人误解的错误。

样板(template)

如果你要开发一个简单的键盘任务，而这项任务又得经常做且要连续使用，那么样板会对你有所帮助，假设要在程序的开头写一个标准的注释性序言。那你得先写一个语法和位置都正确的、包含所有要用的序言样板，这个骨架是你要存储在文件或键盘宏中的“样板”，当你产生一个新文件时，可很方便地把这个样板插入你的源文件中，你可用这种样板技巧来建立大型的组织框架，比如模块和文件等，或程序框架，如循环。

如果你同时在完成几个工程，样板是你协调编码和组织程序风格的好方法，在整个工程开始前，整个小组应当研究出一个供全组使用的样板，这样整个小组就可能很方便地完成这项工程，而且显得一致。

浏览

这组工具可使你方便地查看源代码，编辑程序也能让你查看源代码，但浏览程序（browsers）却特别有功效。

浏览程序

有些工具专为浏览而特别编制,浏览的意思是，若你要买什么你喜欢的东西，你可以轻松的方式通过“商店窗口”来查看。而在编程中,浏览的意思是你知道要找什么而希望马上就找到。浏览程序能使你连续作出修改——比如要修改全部碰到的子程序名或变量名，一个好的浏览程序能在一组文件中搜索与替换。

一个浏览程序能在所有文件中找出要搜查的变量和调用子程序。它能作用于一个工程中的所有文件及所有有特定名字的文件。它能寻找变量和子程序名或寻找简单字符串。

多文件字符串搜寻

某些特别的浏览程序能在多文件中寻找你所指定的字符串，你可用它去搜寻一个全局变量所有出现的地方，或一个子程序会所有出现的地方。你可用它去查找某个程序员所编的所有文件，找出在文件中出现的程序员名字。这样，若你在一个文件中发现一个错误，你可用它去查找其它由同程序员编的文件中的类似错误。

你可用它去搜索特定的字符、相似的字符（不管大小写）或标准表达式。标准表达式特别有
用，因为你可用它去寻找复杂字符串．如果你要寻找所有含 0～9 数字下标的数组，你可这样查找：先找“ ┌”后跟几个或不跟空格，是数字 0～9，再后是几个或无空格，再后就是“┐ ”。一个用得很广的搜寻工具是 grep。一个 grep 查询数字的形式如下

grep　"\[*[0-9]\]" *.c**

你可使表达式更复杂以调整所要搜索的目标。

互相参照工具

一个互相参照工具在一张很大表中列出所有变量、子程序名及所出现的地方。这些面向批处理的工具已被大多数的交互式工具所取代，这些交互式工具根据需要产生必要的变量、子程序的信息。

调用结构生成程序

一个调用结构生成程序能产生所有有关子程序调用的信息。这在调试时有时是有用的。但更常用到的情况是在分析程序结构、或把程序包装成模块、或重复共用程序段时，有些生成程序生成所有调用情况表。其它的生成程序则生成由顶层调用开始的树状图。还有一些是向前或向后追踪程序调用情况的，即列出所有被一个程序调用的子程序（直接与间接）或所有调用一个子程序的程序（直接和间接）。

分析代码质量

这种工具检查静态源代码以测试其质量。

语法和语义检查程序

语法和语义检查程序提供比编译程序更多的检查代码的功能，一般的编译程序仅能提供检查基本的语法错误功能，而一个挑剔的语法检查程序则可利用语言间的细微差别检查出许多隐含的错误——那种编译程序不会指出你可能不愿那样编写的错误，例如，c 语言中：

while（i=0）...

是一句完全合法的语言，但它实际的意思是：

while（i= =0）....

第一句是有语法错误的，把 '= '和 '== '弄混是一常见的错误。Lint 是一个你在许多 C 环境中都能看到仔细的语法和语义检查员。Lint 提醒你，哪些变量未赋初值、哪些变量定义但完全

没用到、哪些变量赋值了但从来没用到、哪些通过子程序传递的参数没有赋值、可疑的指针运算、可疑的逻辑比较（像上面例子所示）、无法用到的代码及许多别的问题。

质量报告程序

有些工具分析你的代码并且报告你程序的质量。比如你可买那些能报告你每个程序复杂性的工具，这样你能集中精力对那最复杂子程序进行检查、测试、重新设计。有些工具计算代码的行数、数据的定义、注释、整个程序和单个子程序中的空行数。它们跟踪错误及与这些错误有关的语句，以使程序员能修改它们；提供可能的修改选择供程序员选择，它们还能计算软件的修改次数及指出经常修改的子程序。

重新组织源代码

有些工具能把源代码从一种格式转化成另一种格式。

重构程序

重构程序能把那些含 goto 的程序转换成无 goto 的结构化代码，这种情况下，重构程序就得做许多工作。如果原代码的逻辑结构令人感到可怕，那转换后的逻辑结构同样可怕，当然若你要靠手工来作这种转换，你可先用重构程序来做一般情况的转换，而用手工来做那些比较难的情况转换。当然你也可用重构程序来做全部的转换，用以启发你自己做手工转换。

代码翻译程序

有些工具能把代码从一种语言翻译成另一种语言。若你要把一个很大的代码从一种语言环境移植到另一种语言环境，翻译程序是很有作用的。跟重构程序带来的灾难性后果一样，若你的代码质量很差，翻译程序就照直把这种环的代码译成另一种语言。

版本控制

利用版本控制工具能帮助你应付软件版本的迅速升级，如；

- 源代码控制
- 风格控制

数据词典

数据词典包含程序变量名及描述它们的数据库。在一个很大的的工程中，数据词典对于跟踪成千上万个变量的定义是很有用的。在数据库工程中，数据词典对描述存在数据库中的数据是很有用的。在小组合作编程时，数据词典可避免起名字的不一致。这种不一致（或冲突）有时是直接的、语法上的冲突，即同一名字有两种意思；或是间接的、隐含的冲突，即不同名字是同一意思。

数据词典包含每个变量的名字、类型、属性，也包含如何使用的注释。在多人合作编程环境中，这个词典应当对每个程序员随时可查。就像程序员支持工具变得越来越强有力一样，数据词典也变得越来越重要；数据随时可能用到，但必须有个工具为它们服务。

20.3　执行代码工具

执行代码工具跟源代码工具一样丰富。

代码生成

本书叙述的工具能帮助生成代码。

链接程序

一个标准的链接程序，能链接一个或几个由源代码文件生成的目标文件，以生成一个可执行程序，许多功能强大的链接程序能链接用几种语言写成的模块。允许你选择最合适的语言而不管那些集成的细节问题。有些利用共用存储区的链接程序，能帮助你节省内存空间。这种链接程序生成的执行代码文件能一次只向内存装载部代码，而把其余部分保留在磁盘中。

代码库

在短时间内写成高质量代码的方法是，一次不全部都写出来，其中部分可以借用已有程序。至少下面这些部分你是可买得到的高质量的代码库：

- 键盘和鼠标输入
- 用户界面窗口的生成
- 屏幕和打印机输出
- 复杂的图形函数
- 多媒体应用生成
- 数据文件操作（包括常用数据库操作）
- 通讯
- 网络
- 文本编辑和字处理
- 数学运算
- 排序
- 数据压缩
- 构造编译程序
- 依赖平台的图形工具集

只要在 Microsoft Windows.OS／2 Presentation Manager,Apple Macintosh 和 X Window System
中把你写的代码重写编译一次就可运行。

代码生成程序

如果你买不到你所要的代码，让别人去写怎么样？你无需到处找人，你可买些工具回来，让它帮你写所需要的代码，代码生成工具着意于数据库应用，它包含了许多用途。普通的代码生成程序写些数据库、用户界面、编译程序方面的代码。这些代码当然不如人写的那样好，但许多应用场合用不着人工来编码。对许多用户来说，能得 10 个应用代码总比只能得到一个好代码强.。

代码生成程序也能生成代码原型，利用代码生成程序你可在短时间内描绘出一个用户界面的原型，你也可尝试用几种不同的设计方法。要做同样的工作靠手工可能要花上几个星期，你

又为何不用最便宜的方法呢？

宏预处理程序

如果你用 C 来编程而用到了宏预处理程序，你可能觉得没有预处理程序来编程是很困难的。宏允许你几乎不花什么时间就能产生一个简单的有名字的常量，比如用 MAX__EMPS 替代 5000，那么预处理程序就会在代码编译时用 5000 来替代 MAX__EMPS。

宏预处理程序也允许你生成一些复杂的函数，以便在程序中简单使用，它仅在编译时被替换回来而不花什么时间。这种方法使你的程序可读而易维护。因为你在宏中给出了一个好名字，所以你的程序更好读；又因为你把所有的名字放在一个地方，因而修改起来极其方便。

预处理程序功能对调试也很有好处。因为它很容易在改进程序时进行移植。在改进一个程序时，如果你想在每个子程序开头检查一下各内存段，那么你可以在每个子程序开头用一个宏。在修改以后，你可能不想把这些检查留在最后的代码中，这时你可重新定义这些宏使它不产生任何代码。同样的原因若你要面向不同的编译环境。如 MS－DOS 和 UNIX，宏预处理程序是很好的选择。

如果所用语言控制结构不好，比如 Fortran 和汇编，你可以写一个控制流预处理程序用来模仿 if－then－else 和 while 循环的结构化结构。

如果所用语言不支持预处理程序，你自己可写一个。这可参考(《Softwa Tools 》(Kernighan 和 Plauger 1976）中的第八章或《softwareTools in Pascal》(Kernlghan 和 Plauger 1981）《SoftwareTools》 中也有如何在 Fortran 中编写控制流的方法，也可用到汇编中去。

调试

这种工具在调试中有如下作用：

- 编译程序警告信息
- 给出程序框架
- 文件比较程序(比较源代码文件的不同版本)
- 执行显示程序
- 交互调试程序，软件和硬件的

下面讨论的测试工具与调试工具有关。

测试

用下面这些性能和工具能有效地帮助你测试：

- 给出程序框架
- 结果比较（比较数据文件、监视输出及屏幕图像）
- 自动测试生成程序
- 记录测试条件及重复功能
- 区域监视器（逻辑分析及执行显示器）
- 符号调试程序
- 系统扰乱程序（内存填充、存储器扰乱、有选择地让存储器出错，存储器存取性检查）
- 缺陷数据库

代码调整

这种工具帮助调整代码。

执行显示程序

执行显示程序在程序执行时显示代码运行情况，并且告诉你每条程序语句执行了多少次或花了多少时间。在程序执行时，显示代码犹如一个医生把听筒放在你胸前而让你咳嗽一样。它让你清楚地知道程序执行时的内部情况，告诉你哪是关键，哪个地方是你要重点调整的目标。

汇编列表和反汇编

有时你想看看由高级语言产生的汇编语言。有些高级语言编译程序能生成汇编列表；另一些则不能，所以你得用反汇编从机器码生成汇编语言。看着编译程序生成的汇编语言，它表明你的编译程序把高级语言转化成机器码的效率。它也告诉你为何高级语言看起来应当很快而实际上却运行很慢。在第二十九章的代码调整技巧中，几个标准检查程序的结果是不直观的。当用标准检查程序检查代码时，用汇编列表能更好地理解结构而高级语言却不能这样。

如果你觉得汇编语言很不舒服需要介绍，那最好的方法是把你用高级语言写的语句与编译程序产生的相应的汇编指令作个比较。可能你第一眼看到汇编时会感到不知所措，编译程序生成的代码你可能再也不喜欢看这类东西了。

20.4　面向工具的环境

有些环境非常适合于面向工具的编程。以下介绍三种:

UNIX

UNIX 和小而强有力的编程工具是不可分的。UNIX 的有名之处在于它收集了许多小工具，有着明确意义的名字，像什么以 grep,diff,sort,make,crypt,tar,lint,ctags,sed,awk,vi 及其它。和 UNIX 有密切联系的 C 语言也有着同样的基本原则，标准的 C 函数库就是一大堆小函数组成的，可由这些小函数汇合成大程序。

有些程序员用 UNIX 编程效率非常高，以至于他们随时把它带在身边，它们甚至把 UNIX 中的许多习惯用到 MS－DOS 和其它环境中去。UNIX 成功的原因之一是它把 UNIX 上的许多工具引放到 MS－DOS 机上。

CASE

CASE 是 Computer-aided software engineering 词头缩写，这种工具的功能是对软件的改进工作提供端到端的支持，如需求分析、结构组织、具体设计、编码、调试、单元测试、系统测试及维护等。若一个 CASE 工具真的把以上这些工作合成到一起的话，是相当强有力的，但不幸的是大多数 CASE 工具总有这方面或那方面不足。

- 它们仅支持部分改进功能。
- 它们支待改进所有功能，但有些方面功能太差而实际不能用。
- 它们改进各个单独部分但却没有合成到一起。
- 它们需要太多的额外开销。它们想支持任何功能，但却显得十分庞大而效率不高。
- 它们把着重点放在方法论上而不考虑其它的方面，因而把 CASE 转变成了 CASD (computer-aided software dogma) 。

在 1992 年的软件质量讨论中，Don Reifer 报告 CASE 编程的效率概述。

APSE

APSE 是 Ada progamming support Environment 的词头缩写。APSE 的目的是提供一套支持 Ada 编程的集合环境，它把重点放在软件内部应用上。美国国防部规定了 APSE 应有能力的要求如下：

- 代码生成工具（一个 Ada 程序编辑程序，一个好的打印机，一个编译程序）。
- 代码分析工具（静态和动态代码分析程序、性能测试方法）。
- 代码维护工具（文件管理程序、配置管理程序）。
- 项目支持工具（一个文件编制系统、一个方案控制系统、一个配置控制系统、一个出错报告
  系统、需求工具、设计工具）。

虽然有些代理商能提供一个 APSE，但却没有一个环境是出色的，不过 APSE 是开发工具的一个方向。

20.5　建立自己的编程工具

假如给你 5 个小时去做一项工作，而让你按以下两种方法去做：

1．很舒服地在 5 个小时内做完这项工作。

2．花 4 小时 45 分钟去建造一个工具，然后用 15 分钟利用这个工具去完成这项工作。

大多数编程员肯定会选择第一种做法。建造工具在编程中必不可少。几乎所有的大编程项目都有内部工具。许多工程有专门的分析和设计工具，这些工具甚至比市场上的工具要高级。

除了少数几个外：你完全可以编写出本章所提及的大多数工具。做这些事情可能不太值得，但实现这些工作在技术上是没有什么困难的。

各项目独有的工具

许多中型或大型项目需要一些自己独有的工具来支持编程。例如你需要一种工具去生成特殊的测试数据、检查数据文件的质量、模仿硬件等。以下是几个例子：

- 一个航空小组负责编制一套飞行软件，来控制和分析它所得到的数据。为了检查这个软件的
  性能，需用一些飞行数据来测试这个软件。工程师们写了一个传统的数据分析工具，用以分析飞行系统软件的性能，每次飞行以后，他们都用这个传统的工具去分析原始系统。
- Microsoft 计划发行一个窗口图形环境，其中包括一种新的字体技术。既然这套软件和字体
  数据文件都是新东西，错误很有可能从这两者中产生。Microsoft 的工程师编写几个传统的工具去检查数据文件中的错误，用以提高鉴别字体数据出错和软件出错的能力。
- 某保险公司开发了一套很大的系统用以计算保险费用的增加。因为这个系统相当复杂且精确
  性要求相当高，成百个算得的费用需仔细核查，但用手算一个费费需几分钟。公司编制了一套独立的软件一次就计算出这些费用。利用这套工具，公司计算每一个费用只需几秒钟时间，因而检查所用时间由原来占主要部分成为很小的一部分。

一个项目的计划中必须包含一些必要的工具，且要安排一些时间去做这项工作。

批处理

批处理是一种能自动做重复操作的工具。有些系统中批处理就叫批处理文件或宏。批处理可简单可复杂，它的好处是易写好用。比如你要写日记但又想保密，那你就加上密，除非你输入正确的口令，才可用。为了保证每次都能加密和解密，你得写一个批处理显示怎样给你的日记解密，怎样处理输入的词，这个批处理如下：

```
crypto C:\word\journal. * %1   /d   /Es   /s
word C:\word journal.doc
crypto C:\word\journal. * %1   /Es   /s
```

这里%1 是输入密码的地方。因而是不能在批处理中明显写出来的。批处理可以使你不用每次都输入所有的参数，而能保证所有的操作每次都按正确顺序执行。

如果你在一个时间内要经常输入超过 5 个字符的字符串，那最好把它写成批处理文件。用到批处理文件的例子有编译、连接指令序列、备份命令及带许多参数的命令。

批处理文件增强程序

批处理文件是一个 ASCll 文本文件，它包含一系列操作系统命令，其中每个批处理语言支持的命令，并不都是一样的，有些功能比另一些强。少数批处理语言如 JCL，竟和高级语言一样复杂；有些则实用功能不强。如果所用批处理语言功能太弱，那就找一个批处理文件增强程序，它会提供比标准批处理语言更强的能力。

20.6　理想编程环境

本书所讨论的理想编程环境仅仅是一种设想。这种软件还没有进入实用，本书仅就结构方面的问题预测一下这种结构工具的趋向，本书所讨论的那些性能是现有技术水平很容易达到的。而这些讲义对那些有志于在编程环境方面更上一层楼的人会是一种启发。在讨论中我们把想象中的环境叫“Cobbler”，它起源于传说中补鞋匠的典故，而对于今天的程序员和编程工具来说正是这种状态。

集成环境

Cobbler 环境包括编程中涉及到的具体设计，编码及调试活动。当涉及到源代码控制问题时，Cobbler 也包含源代码版本控制。

语言主持

在代码构造过程中，CObbler 提供了有关编程语言交互参考的帮助信息。帮助信息详细列举所使用语言功能的许多信息，帮助信息也列举了一般问题可能产生的错误信息。

环境也提供了语言结构的标准框架，这样你就无需知道 case 语句、for 循环或一个常用结构的精确语法了。帮助信息也给出了环境所提供的子程序的列表，你可参考这个列表选择子程序加入到你的代码中去。

具体的交叉参考

Cobbler 能让你得到一张变量出现地方及何时被赋值的表。你能从交互参考中得到变量定义的信息。同时还能得到关于变量说明等注释性的信息。

你可同样方便地得到检索有关于程序的信息。你可以从一个指定子程序名的地方，上溯或下查调用子程序的表链，你也可方便地检查一个子程序调用时各变量的类型。

用查询方式观察程序组织结构

Cobbler 从根本上改变了人们观察程序源文件的方法。编译程序读源文件是从上到下的，传统的编程环境也迫使你由上往下观察程序。这种顺序式的读程方法已不能满足人们从各个角度观察程序的需要

在 Cobbler 中，你可观察一大堆的子程序名、图形显示及具体观察一个子程序，也可选择组织许多程序的模式如分层、网络、模块、目标或按字母顺序列表等。从一个较高层次观察一个子程序可能不太清楚，要仔细看的话，可具体放大这部分（指具体调出这部分观察）。

当你深入观察一个子程序时，你能从几个层次具体地观察它，你可以仅观察子程序定义或注释序言部分；你可以只看具体的代码或只看注释；你也可只观察程序的控制结构而不显示控制块内的内容。在支持宏命令的语言中，你可观察含宏扩展和宏缩小的程序。若是宏缩小，你可方便地扩大每一个宏。

本书中用一个词：外形观察（outline View）。没有外形观察和没有研究外形的能力及没有放大具体的内容的能力，那么就不能很好地把握各章的结构。读每一章时平平淡淡地看，仅有的一点结构概念仅来自于从上而下地翻一下书。若在高层次与低层次间放大与缩小不断观察角度，那我们就能获得各章拓扑结构的概念，这对组织写作是至关重要的。

组织源代码跟组织写作一样难。当你有了多年组织写作的经验后，很难说你缺乏观察程序拓外结构的能力。

程序各组成成份也要改进。源文件的语义有时也很模糊，用模块或目标等观点来替代现有的观念。源文件的观念是独立的，你应当仔细考虑程序的语义，而不要考虑它存储时的物理结构。

交互式格式化

Cobbler 提供比现有环境更灵活的格式辅助模式。比如它的用户界面使得程序的外括号比内括号大。

环境根据用产规定的参数，而不依靠单独巧妙的打印机程序来格式化代码。一旦环境读入了程序，那就知道了程序的逻辑结构及在哪儿定义变量，在 Cobbler 中，你不需依靠一个单独的程序去格式化你的代码。现有的有些环境在你进入控制结构时提供自动的退格形式，但还不十分成熟。当你在修改程序时，若有 45 行需退后六列时，你只能靠你自己来做这项工作了，而在理想编程环境中，环境格式化代码是根据逻辑结构来做的，如果你修改了逻辑结构，环境相应地对代码的格式作调整。

说明

Cobbler 支持传统的注释方法并对其做格式化工作，并且支持声音注释方法。这种方法不是通过输人注释文字而是通过对鼠标讲几个词来存储你的思路。

性能调整

Cobbler 自动收集调整程序性能所需的信息。程序的哪一块花了全部执行时间的 90％？哪一块程序根本末被执行。它收集具体的信息为改进设计作准备：哪一块程序逻辑结构最复杂或有许多具体的数据；或哪些部分联系最紧？它收集修改的信息，哪些程序部分相对稳定？哪些部分是通用的？环境能做以上各项工作而无需你做什么。

环境特性

环境状态总是在几个窗口间切换，你可在这些窗口间上溯或下拉。同样你也可这样处理编辑程序：编辑程序有多层编辑、搜寻字符串、在单个子程模块或整个程序中搜寻和替换常规表达式等功能，而这些功能是有具体环境的。

和现有最好的调试程序相比，Cobbler 提供最完整的调试环境。现在的许多调试程序仅在某些方面令人满意，它们有的提供稳定逻辑结构、数据结构、数据的更改及目标的检查能力；有些提供跟踪执行和测试某个子程序的能力，甚至测试在一个大环境中有上下文的子程序。但在 Cobbler 中所有这些编辑、编译、调试诸功能都集成在一个环境中，在调试时你根本无需变换“模式”。

在改进程序的过程中，你总要最大限度地利用环境的编译能力去编译你的程序，你可以执行程序、修改部分程序，然后在不改变环境状态情形下继续执行程序。Cobbler 指出修改部分与哪些部分有依赖关系而需重新编译。

当然所有这些操作都是瞬时的，因为 Cobbler 利用了效率很高的后台处理。

最后 Cobbler 支持开发模式。它与已有模式间有着根本性的区别，现有模式强调小而快的功能；但开发模式着眼点不在如何减小机器周期，而着眼于最大限度地利用时间。强调尽快无痛苦地发现和改正错误。在开发过程中，Cobbler 环境发现并警告你出现一切常见的问题。如可疑指针、未赋初值、错误数组下标及数值的上溢等。编译程序很容易在程序中插入特殊代码去检查这些问题。

Cobbler 的主要优点

实现理想开发环境需要程序员的努力，下面是现有环境和理想环境之间主要区别：

- Cobbler 完全是交互式的。变量和子程序交互参考、子程序列表、语言参考信息等都可立即显示在屏幕。这些功能在现有的环境中都可能有，只是没有集成罢了。
- Cobble。是一个预知的环境。它预先就知道你的要求，而不需等你提出问题。环境在某些情况就知道你会对程序进行编译和链接。它为何要等到你去要求呢？假如有空闲时间，它为何不在后台进行编译呢？
- Cobbler 应用一切计算机时间，而节约人的时间。计算机时间变得越来越便宜，因而可以用它来节省人的时间，在随时编译策略指导下，计算机可能编译那些你还未完成的程序，这时有些编译是被浪费了，但在另外一些情形下呢？在这些情况下你就无需等待编译结果了。
- Cobbler 抛弃了纯源程序的观点。虽然你想清楚知道你编了什么代码，但你却不必像编译程序那样去看代码、数据定义、注释等。你也可以不去管那些具体的管理细节，如子程序在哪个源文件中等;你可以只管定义目标和模块及限制它们之间的相互作用。

以上所有这些功能都是现在技术水平可以达到的，许多已在 smalltalk 或 APL 环境中实现，其它的是很有名的超文本思想（hypertextiden)。所有这些功能并没有要求改变基本的编程语言如 C、Pascal、Basic、Fotran 及 Ada，也没有要求放松代码间的直接联系。

20.7　小　结

- 好的工具使编程更容易。
- 你可以编写大多数你需要的工具。
- 用今天的技术水平能编出高水平的编程工具。

第二十一章　项目大小如何影响创建

目录

相关章节

创建前提：见第 3 章
创建管理：见第 22 章

在软件开发中常出现小的项目随着开发的进展，项目规模不断变大的现象，假定你计划用 6 个月的工作量开发——5000 行的软件包，并在进行测试时发现 25 个错误。软件开发成功了，软件也可运行了，但可能导致程序的长度大大增加，甚至可达 50，000 行之多。

虽然最终软件的大小是所预期的 10 倍，并不意味着也需花费 10 倍的工作量，而可能是 20 倍的工作量。而且，20 倍的工作量并不意味着 20 倍的创建，也可能是 12 倍的创建和 40 倍的系统综合和测试。你也将可能获得不仅是 10 倍的错误，而是 15 倍的错误。

如果你习惯于开发小项目，你的第一个中等项目将是非常困难的，而且不是你预期中的令人愉快的成功。本章将讨论此问题并给出了具体应付方法。如果你已习惯开发中等项目，你可以利用你的经验进行一次小项目开发。本章也将告诉你如何对项目进行控制。

21.1　项目大小

你所在项目的大小是典型的吗？项目大小的范围很宽，意味着项目不是典型的。估计项目大小的一个方法是看项目开发组人员的多少。以下是二者间的粗略关系：

项目组人数	项目所占百分比
1～3	50%
4～8	33%
8～12	6%
12～20	5%
20～50	3%
50＋	2%

并不那么直观的数据，却反映了不同项目的大小和程序员人数之间的差别，大的项目往往使用更多的程序员，以下是对程序员和项目大小的粗略估计：

程序长度	程序员所占百分比
（以代码长度计算）	
2k	5～10％
2k～16k	5～10％
16k～64k	10～20％
64k～512k	30～40％
512k＋	30～40％

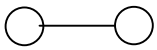
21.2　项目大小对开发活动的影响

如果只有你一人开发项目，对项目成功或失败影响最大的正是你自己。如果你所在项目开发组有 25 人，你也可能仍发挥最大作用，但更可能是大家各有一份功劳，整个集体对项目的成功或失败有更大的影响。

交流和大小

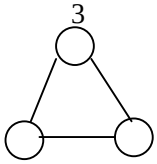
如果你是某项目的唯一开发者，你唯一交流的途径是你和用户所进行的交流，其作用就好似将你的左、右大脑联系起来。在项目组人数增多时，交流途径也相应增加了，但是二者的关系并不是线性的。交流途径是和人的平方数成线性关系的。以下是图示:

1



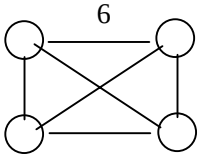
两个程序员之间的交流路径

3



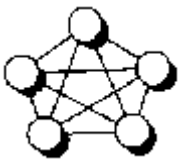
三个程序员之间的交流路径

6



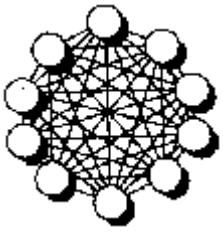
四个程序员之间的交流路径

10



五个程序员之间的交流路径

45



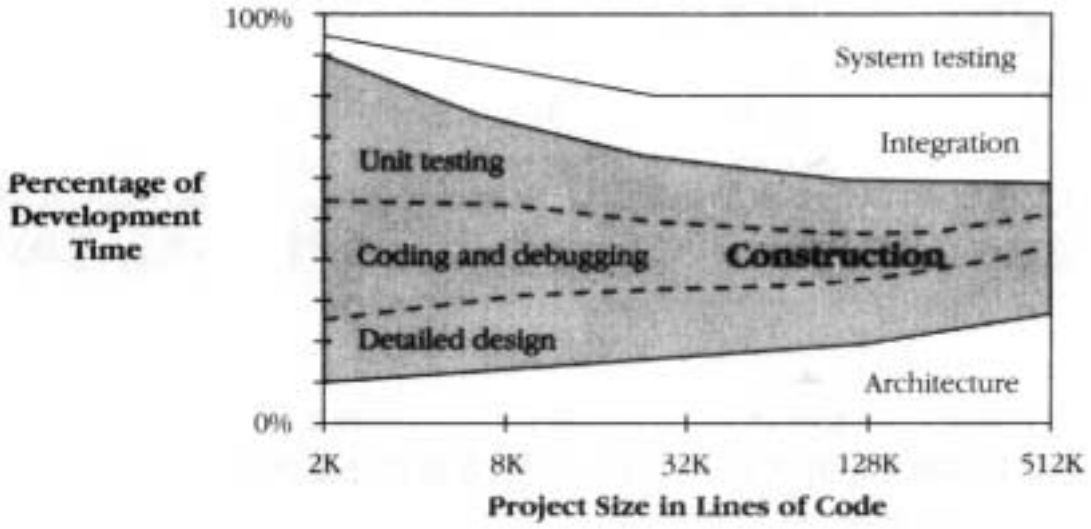
十个程序员之间的交流路径

你可看到，二位程序员之间仅有一条交流途径，五位程序员间有 10 条交流途径，而 10 位程序员间有 45 条交流途径——假定每一位程序员都可和其他程序员交谈。其中 2%的有 50 或更多程序员的项目，其交流途径至少有 1200 条可挖掘途径。交流途径越多，你放在交流上的时间越多，也就越容易出现错误。大的项目要求采用一定的方法以简化交流方法，以对其作一定程度的限制。

简化交流的典型方法是规范化交流方法。不是让 50 个人按各种可能方式相互间进行交流，而是让 50 个人都阅读和编写文档，有些是文本的，而有些则是图形的，有些是打印在纸上的，或是表格形式的。

活动比例和大小

正如项目越大，所要求的信息交流也相应增加，项目所需各种活动的种类也急剧地变化。以下是关于不同项目的大小和各种开发活动的比例的关系：

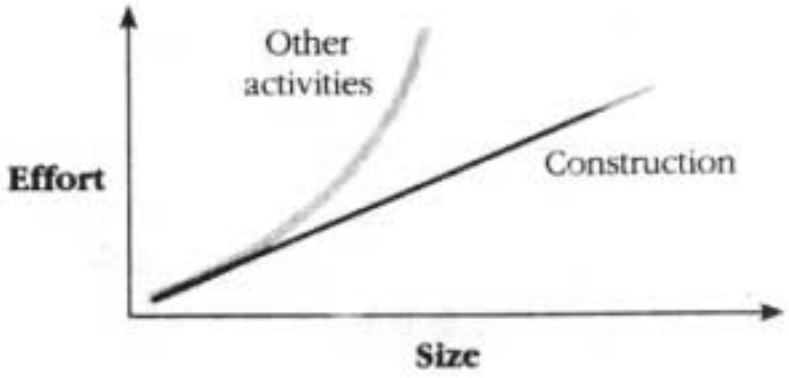


项目大小和代码行数

对于小项目，创建是最为突出的活动，占了整个开发活动时间的 80％。对于中项目，创建仍然是主导性的活动，但是所占比例下降了 50％，对很大的项目，结构、综合、系统测试和创建所占时间比例大致相等。总之，随着项目的增大，创建所占整个工作时间的比例将减小。从上图可看出随着项目的增大，创建最终将消失。

创建之所以随着项目的增大而变弱，是因为创建的各种活动——详细设计、调试、编码单元测试——是线性增长，而其它各种活动则是按乘幂的关系增长的。

以下是图示：



例如，某一项目是另一项目大小的二倍，则二者的创建工作量之比可能为 2 ： l ，而综合和系统测试工作量之比为 4 : l 。当最终开发所得的第一个项目的大小是另一个的 10 倍时，它可能需要 12 倍的创建工作量，100 倍的计划工作量和 40 倍的综合测试量。以下是随项目的增大而工作量增大的其它活动：

- 计划
- 管理
- 交流
- 需求开发
- 系统功能设计
- 接口设计和描述
- 总体结构
- 综合
- 错误消除
- 系统测试
- 文档生成

不论项目的大小如何，以下方法总是有价值的：结构化编码。其它程序员对逻辑和代码的检查、联机开发系统和高级语言的使用，对于本项目，这些方法可能并不充分，但是对小项目却是适用的。

方法和规模

方法被广泛用于各种大小的项目中。对小系统，方法往往是偶然的和本能的。对大项目，方法往往是精确和仔细计划的。

其中一些方法是可自由使用的，以致于程序员甚至不知道自己在使用它们。而部分程序员甚至说这些方法太教条化了，而不采用它们。程序员有时可能会不自觉地采用某一种方法，程序编制的任何一种途径都包含着某种方法的使用，不管这种方法是多么不自觉或原始的。就连早上起床后去上班，虽然不是一种创造性的方法，但也是一种基本的活动。坚持不使用方法的程序员，实际上只是不明确选用了方法——没有人不使用某一种方法。

正规的方法并不令人高兴。如果误用了，往往会引起麻烦。大的项目较为复杂，需加强对方法的自觉使用。正如建造摩天大楼需应用各种方法一样，相同大小的项目也需各种方法。不管你是否认为“军事才能”是一个矛盾的词，你应认识到军方是计算机的最大用户，并且也是最大编程研究的赞助者之一。它提倡的对问题的正规方法包括从 12 个方面对一个程序项目进行评分。在表 21 一 1 中给出了项目正规性明细表。

表 21－1　项目正规性明细表

	评 分					
因素	1	2	3	4	5	总分
1.初始要求	没有，对各种不同的设备重新编程	最小，有较多的苛刻要求	有限制；对环境应用新的界面	相当多的熟练应用现存状态	非常广泛地提高现存状态	__
2.通用性	高限制；单目的	有限制；规定一定范围的能力	限制灵活性；允许格式修改	多目的；格性灵活对象范围广	非常灵活；在不同的设备上不同，可适应对象范围很广	__

因素	评分 1	2	3	4	5	总分
3. 操作跨度	局部的或工具的	部分命令	单一命令	多命令	广泛地适用于各军事部门	—
4. 范围和对象的修改	无	不经常	偶然	经常	连续	—
5. 设备复杂性	单机进程处理	单机子进程处理，扩展外围系统	多计算机标准外围系统	多计算机高级编程，复杂外围系统	主控制系统，多计算机，自动输入--输出和显示设备	—
6. 人事安排	1－2	3－5	5－10	10－18	18 以上	—
7. 开发代价	3－15k	15－70k	70－200k	200－500k	大于 500k	—
8. 关键	数据处理	子程序操作	人事安全	单元残存	国家防务	—
9. 对程序修改的平均反应时间	>=2 周	1－2 周	3－7 天	1－3 天	1－24 小时	—
10. 对数据输入的平均反应时间	>=2 周	1-2 周	1－7 天	1－24 小时	9－60 分钟（交互式）	—
11. 编程语言	高级语言	高级语言和限量制汇编语言	高级语言和扩充汇编语言	汇编语言	机器语言	—
12. 软件开发中的合作性	无	有限	中等	广泛的	深入的	—
总计						—

使用项目正规明细表，一程序的得分可在 12－60 之间。得 12 分意味着对项目的要求是轻微的，且对其正规性要求很少。得 60 分意味着项目要求很高，且对结构的规定也很多。在社交场合，越是正式，你的服饰就越让人不安（高跟鞋，领带等）。在软件开发中，项目越正规，你就越需付出更多的工作量。根据项目正规明细表，表 21-2 给出了文件建议。

表 21－2　文件建议

正规评分	建议文件
12－15	用户指南和小程序文件
15－26	低级文件以及操作使用手册，维修手册，测试计划、管理计划、结构配置划
24－38	低级文件加功能描述、质量确保计划
36－50	低级文件以及系统和子系统描述．测试分析报表
48－60	低级文件以及程序描述

对于特定的项目、数据要求文件、数据库描述和执行程序也可能在建议之列。

你不必因某种原因而创建这种文件。你编写配置管理计划并不是为了运行有关程序，而只是强迫自己能向其它人解释。当你计划或创建一软件系统时，文件对你的实际工作的副作用是可见的。如果你自己感到要编写一般的文件，这就有点不对头了。

程序、产品、系统、系统产品

并不只是代码行数和参加人数对项目的大小有影响。一个更微妙的影响是软件的质量和复杂性。最初的 Gigatron 和 Gigatron Jr. 可能只需一个月的时间就能完成编写和调试工作。它只是单一的程序，有一个测试。如果 2500 行的 Gigatron Jr. 软件花一月时间，为什么 5000 行的 Gigatron 需 6 个月呢？

一类最简单的软件是由用户自行开发的单一程序。更为复杂的一种程序类型称为——软件“产品”——它是为了让其它人而不仅是用户自己使用。软件使用环境和生成环境是不同的。在其交付之前，软件接受过深入的调试，并被文档化，有可能被其它人所维护，软件产生的代价是程序开发代价的 3 倍。

另一种复杂类型是要求一群程序员相互协作进行开发。这样的系统称之为软件“系统”。开发一个系统要比开发一个程序更为复杂，因为系统需开发各不同部分之间的接口以便能将其有机地综合起来。一般开发一个系统所费代价是单一程序的 3 倍。

在开发“系统产品”时，它将加装产品外壳（用户界面），并将系统各部分综合起来。系统产品所费代价是单一程序的 9 倍。

程序、产品、系统和系统产品的复杂性，是导致估计错误最为常见的原因，程序员在利用过去的编程经验对创建一系统产品进行评估时，可能会低估近 10 倍。如果你用自己编写 2K 行代码的经验对开发一个 2K 行的程序进行评估，你所估时间仅为实际所需时间的 80％。编写一个 2K 行代码的程序并不相同。如果你没有考虑各种非创建活动所费时间，你实际所花费时间将比你所估计的多花 25％。

如果你所在项目变大，创建所占工作量将会减少。如果你的估计仅基于创建经验，发生的估计错误的机会增大了。如果你用自己 2K 的创建经验评估 32K 程序所花时间，你所估计的时间仅为实际需要的 50％。

估计错误主要是由于你对项目大小和开发较大程序的关系理解不全。如果你不理解产品其它工作的重要性，估计错误数可增加 3 倍或更多。

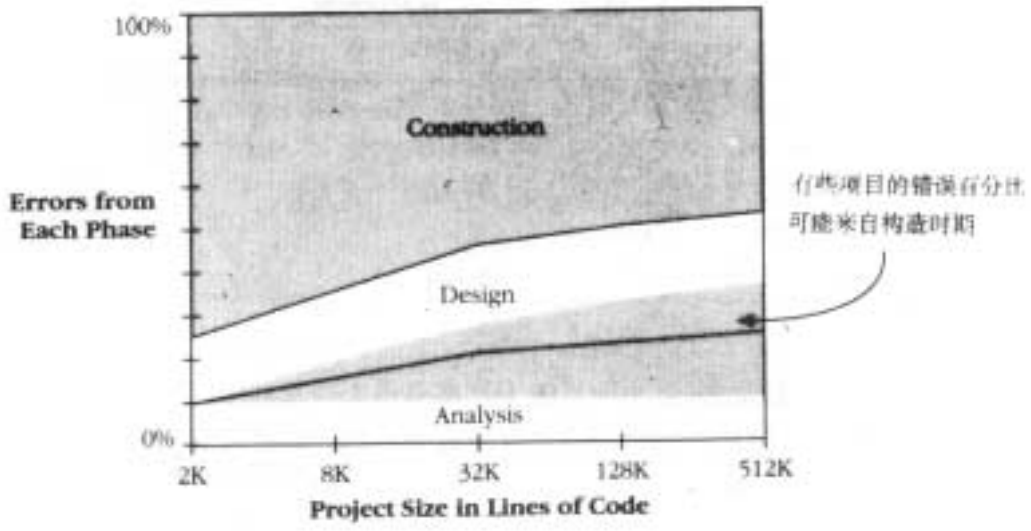
21.3　项目大小对错误的影响

质量和错误都要受到项目大小的影响。你可能认为这种类型的错误将不受影响，但是随着项目的变大，相当大的一部分错误通常要归于分析和设计错误。以下是一个图示：

对一些小项目，创建错误可占整个错误的 75％，方法对代码质量影响不大。对程序质量影响最大的是个人编写程序的技巧。

对大项目程序，创建错误可占整个错误的 50％。分析和结构错误所占比例是不同的。事实上较大项目需要更多的分析和设计，所以此类活动中发生错误的机会相应也大一些。对一些很大的项目，创建错误比是较高的。对一些 500,000 行代码的程序，75％以上的错误是创建错误。

错误的类型随项目大小而变化，所发生的错误数也会发生变化。你可能会自然想到，某项目大小为另一项目的二倍时，其所发生的错误总数应为小项目的二倍。但是，每行代码出现的错误概率和错误数也会增大。当某一产品大小是另一产品的二倍时，它所出现的错误是另一产品的二倍多。表 21－3 给出了项目大小和错误数关系：



项目代码行数

表 21－3　项目大小和错误数

项目大小（代码行数）	每 1000 行代码所发生错误数
小于 2k	0－25
2k－16k	0－40
16k－64k	0.5－50
64k－512k	2-70
52k 或更多	4－100

本表原载《Program Quality and Programmer Productivity》 (Jones l977)。

以上数据只是来自一些特定项目，你实际所出现的错误数可能和以上数据略有出入。然而本数据仍是有一定启发意义的。它表明随着项目的变大，其所出现错误数会急剧增加。大项目出现错误数是小项目的 4 倍多。此数据也表明对超过一定大小的项目，不能编写出没有错误的代码，错误总可以潜入，而不管采用什么方法预防错误。

21.4　项目大小对生产效率的影响

当和项目大小联系在一起时，生产效率和软件质量有许多共同之处。对于小项目（200 行或更小），对生产效率影响最大的是单个程序员的技巧。随着项目的增大，开发组人数和组织对生产效率影响迅速增大。

项目应为多大时，开发组人数才对生产效率有影响：Boehm 和 Gray 曾说过，较少人数的开发组的效率比人数较多的开发数高 39％。

表 21--4 给出了项目大小和生产效率的一般关系：

生产效率实际上是由以下因素所决定：人员的素质、编程语言、方法、产品复杂性、编程环境、工具支持和其它许多因素所决定的，所以可能表 21－4 的数据并不能直接用于你的编程环境，你可视具体情况而定。

表 21－4　项目大小与生产效率的关系

项目大小（代码行数）	每月代码行数
小于 2k	333－2000
2k –16k	200-1250
16k-64k	125－1000
64k-512k	67－500
512k 或更多	36－250

本表原载《Program Quality and Programmer Productivity》（Jones1977）。

但是，本数据也是有所启发的，最小项目的生产率是最大项目的 10 倍。即使从某一中等项目转移到另一中等项目——如从千行代码的项目转到 9 千行代码的项目——生产效率也可降低一半。

21.5　小　结

- 在一些小项目中的活动并不能想当然地用于大项目中，你应仔细计划它们。随着项目增大，创建作用减弱。
- 随着项目的增大，交流方法应简化。各种方法的使用应因时而异。
- 在相同条件下，大项目的生产率比小项目低一些。
- 在相同条件下，大项目的每行错误数比小项目的每行错误数多。

第二十二章　创建管理

目录

相关章节

创建前提：见第 3 章
程序长度：见第 21 章
软件质量：见第 23 章
系统集成：见第 27 章
设计较好软件的要旨：见第 32 章

软件开发管理已成为二十世纪晚期人们所不可轻视的一个挑战。对软件计划管理进行一般的讨论，已超出本书的范围，但是本文将讨论和创建有关的一些管理主题。由于本文讨论了许多主题内容，在另外几个部分你也可以找到更多有关的内容。

本章主要讨论和创建有关的软件管理主题。

22.1　使用好的代码

由于代码是创建的主要输出，所以创建管理的关键问题是养成使用好代码的习惯。一般说来，从开始就用标准并不是一个好的方法。程序员倾向于将管理者视为技术层次中的水平较低者，甚至认为其差距犹如单细胞生物和冰川世纪灭绝的猛犸像一样大。如果有编程标准的话，程序员可以买一个。

标准设置的考虑

如果你避免使用标准，标准对你也就不那么重要了，可考虑使用标准的替代：灵活的原则，或者采用一些建议而不用原则，或者一些包含最好实际情况的例子。

方法

以下是取得较好代码效果的几个方法。这些方法不应被视为教条的代码标准。

将二人分在计划的每一部分。如果二人不得不各自编码，你应确保至少这二人认为代码能

正常运行同时可读。协调二人的开发方法将涉及到指导者和受训者的配合。

如果要了解注释的详细内容，参看第二十四章“注释”。

注释每一行代码。一条代码注释常包括程序员和至少两个注释者，这意味着至少有三人将阅读代码的每一行。同级注释的另一个名字是“同级压力”。除了防止程序员离开计划体系外，让程序员知道另外一些人还将阅读本代码，将会提高代码质量。即使你所在部门没有明确的代码标准，注释也将会提供一个较好的途径，使代码向一个代码标准的方向发展，这些标准是群体在注释过程中所得到的一些结论，而且，随着时间的推移，群体将派生出自己的标准。

采用代码许可。在一些领域中，技术草图将由管理工程师批准和鉴署。鉴名就意味着对管理工程师来说，本草图在技术上是可行的，而且是没有错误的，一些公司也采用这种方法。在代码将完成之前，高级技术主管将签署代码表。

用好的注释作示例。管理的一个重要方面，就是向你的人员清楚地表达你的意图。其中的一条途径就是让你的程序员见识一些好的代码，或将其在公共场合张贴出来。这样作将会提供清楚的代码示例，而这正是你所期望的。同样，一本代码标准手册将包括一些“最好的代码表”，指定一些表作为最好的代码以便其它人能效仿。这样一本手册将比一本英语标准手册要容易改进，而且，无须费多大劲就能展示代码风格的精细，如果靠平铺直叙一点一点描述是困难的。

强调代码表是公用财产。程序员有时会觉得代码是“他们的代码”，好像这是其私人财产一样。虽然这是程序员自己心血的结晶，但是代码是整个计划的一部分，计划中需要它的人应有权得到它。代码在注释和维护阶段应当被其它人所看到。

曾经报道过的一个最为成功的计划是，一个用 11 人年的工作量开发的 83,000 行的代码。在其先期 13 个月的运行中仅发现一个系统错误。当你知道这计划是在 60 年代晚期没有联机编译和交互式调试条件下完成的时候，你会感到更为惊讶的。要知道，在 60 年代晚期——7500 行代码所花费的人年工作量，二倍于今天 3750 行代码所耗费的人年工作量。本计划的一个主程序员说，计划成功的关键之一是明确确定所有的计算机运算（错误或其它）是公共的而不是个人的财产。

奖赏好的代码。用奖励办法促进好的代码开发工作。当你开发你的系统时，请牢记以下几点：

- 此奖励应是程序员所需要的（许多程序员发现“哄小孩”似的奖励是令人生厌的，尤其是这些来自那些非技术管理人员的时候）。
- 所奖励的代码应是非常好的。如果你奖励一个大家都知道其工作干得不好的程序员，你将收到相反的效果。至于程序员是否态度好或是否按时上班则不是重要的。如果你的奖励和相应技术价值不一致，你将失去信誉。如果你的技术不够高，不能判断哪一个是好的代码，那么，你就根本无须奖励，或者让你的合作者们选择。

一个简单的标准。如果你管理一个编程计划，同时你也有编程经验的话，做好工作的一个有效的方法是说：“我必须能够阅读和理解本计划所编的所有代码”。管理者不是技术尖子，但防止“聪明”或恶作剧式编码也是有好处的。

本书的角色

本书的大部分是讨论良好的编程习惯，并不打算纠正教条的标准，甚至也不打算被用作为教条的标准。将本书作这样的用场将和本书的一些重要主题相矛盾．将本书视为讨论的基础，

作为一本有良好编程风格的参考资料，同时也可检查你所在环境中的编程习惯是否有益。

22.2　配置管理

软件计划是变化：代码在变化、设计在变化、需求在变化，而且需求的变化引起设计上更多的变化，设计上的变化又会引起代码和测试情况的更多变化。

何调配置管理

配置管理是全面地处理各种变化，以便系统随着时间的推移能保证其完整性。其另一个名字是“修改控制”。它包括如下各种方法，评估各种建议处理各种修改、在不同的时间保留系统的各种备份。

如果你不控制设计上的修改，你最终将会发现，设计中某些部分的代码和最初的大相径庭，你所写的代码也和新设计的部分不兼容。直到系统集成时你才知道它带来了许多不兼容性的问题，而这时恰恰是最关键的时刻，因为谁也不知道这之间是怎样一回事。

如果你不控制代码的修改，你在改变子程序而同时也有其它人正在改变它。成功地将你的修改和他人的修改协调一致将会避免这些问题。无节制的代码改变将会付出更多的测试量。已经测试过的版本也可能是老的、未曾变更的版本；变更的版本也可能没有接受过测试。没有一个良好的修改控制，在改变一个程序的同时，如果发现了新的错误，也不能回到原来的可能正常工作的程序状态。

问题也会长期存在。如果不能有条理地处理各种修改，你将好比在雾中胡乱前进而不是朝着一个明确的目标径直向前。没有好的修改控制，与其说是开发代码，倒不如说是在浪费你的时间翻来复去。配置管理可帮助你有效地使用时间。

尽管结构配置有其明显的必要性，许多程序员过去并不使用它。一项调查发现超过三分之一的程序员对结构配置并不熟悉。

结构配置并不是由程序员所最先提出。但是由于程序设计是多变的，所以结构配置对程序员异常有用。对于软件计划来说，结构配置通常称为软件配置管理，或 SCM（通常也叫“scum”）。SCM 着重于程序的源代码文档和测试数据。

SCM 的问题在于过程。防止汽车事故最可信的方法是禁止行车，而防止软件开发中问题的途径是停止所有的软件开发。虽然这是一条控制修改的方法，但这是软件开发中的一条可怕的途径。你将不得不仔细地计划，这样 SCM 才是一笔财富而不是你的包袱。

有关计划尺度对创建的影响，参看第二十一章。

在一个人的小计划中，不采用 SCM 而仅需考虑一下一般的定期备份，你可能照样干得较好。在一个 50 人的大计划体制中，你可能需要一个完整的 SCM 计划，它包括备份档案正规的过程，要求对设计进行修改控制及文档、源代码和测试的控制。

如果你的计划不大也不小，你不得不在以上两种极端方法中采用一种折衷的方法。以下几个部分描述了完成一个 SCM 计划的一些可选择项。

软件设计修改

在开发过程中，你一定会被怎样改进系统的性能所困扰。如果你只是在每个修改发生时才

去考虑它，你会发现自己进入了软件开发的歧路——整个系统在变化，离目标的完成也是遥遥无期。以下是控制设计修改的几条准则：

遵循正规修改控制流程。正如第三章所指出的那样，正规修改控制流程是你处理许多修改要求的有力手段。建立正规流程，会对如何在整个计划的范围内考虑修改有一个清晰的了解。

建立修改控制委员会。修改控制委员会的工作是在有修改请求时，从“米糠”中将“小麦”区分开来。任何想修改的人，将变化请求提交给变化控制委员会。“修改请求”这个词指的是任何将改变软件的请示：一个有新特点的主题、一个有特点的修改，一个报告真实错误的“错误报告”。委员会定期对所提修改请示进行检查。它可能同意或不同意或不予理睬这些修改请求。

集成考虑修改请求。本规则倾向于完成容易的修改。这种方式处理修改所带来的问题是，好的修改机会可能会失去。当你按照预定计划已经完成了计划的 25%时，你想作一下简单的修改，当然可以这样做。而如果你滞后于原定计划进度且已完成了计划的 90%时，你不可能再这样作简单的修改。当你在计划的收尾阶段已经超出了预定时间期限时，至于比第一次修改要好 10 倍是无关紧要的——你没有必要去作一些无谓的修改。你也可能失去一些最好的机会，因为你想到要作修改已经太迟了。

一个非正式的解决这个问题的方法是写下所有的主意和建议，而不管完成它的难易程度如何，同时也应及时保留这些建议和主意直到你有时间去处理它们。最终，将其作为一个整体看待，选择其中最有实效的一个。

估计改变所花的代价。不管何时你的客户、你的老板、或者你自己打算改变系统时，先估计修改花费的时间，其中包括修改所引起的代码注释和再测试整个系统，同时也应包括处理由于用户程序的变化所引起的设计、编码、测试需求等的修改所需时间。让有关各方知道软件是异常复杂交错的，即使乍看起来修改很小，但是时间预估仍然是必要的。不管改变刚提出时你是多么乐观，你应避免作出草率的估计，匆忙的估计错误率可能会达到 10 或 100 倍。

从另外一个角度看待处理各种修改，参看 3.3 节“创建过程中对各种改变请求的处理”这一节。

谨慎对待主要修改。如果顾客建议进行重大修改，这是对你的工作没有满足要求的警告。如果在高级设计时提出这些建议，你应停止设计工作并回到初始设计要求状态。在进行设计之前你应耐心等待，直到修改要求有所缓和为止。

如果在代码开发过程中，你的顾客坚持要作重大修改，你应坐下来和你的顾客谈一谈，并指出它对已经进行的工作所产生的影响。其中一个明显的影响就是早期的一些工作将废弃。

软件代码改变

另外一个配置管理主题是控制源代码。如果当你改变了代码并发现其中产生的一个新的错误，表面上看来和你的修改无关，你可能会想到将其和老版本的代码相比较，以便找出各种错误源。如果这仍然一无所收获，你可能会去查阅更老一级的版本。如果你有版本控制工具且保留源代码的多个版本的话，这种类型的版本回溯过程是很容易的。

版本控制软件。一些版本控制软件工作起来是如此地得心应手以致于你很少注意到你正在使用它。在使用开发计划中版本控制软件更是非常有帮助。典型地，当你需要对一特定文件的源代码进行操作时，在版本控制软件之下你查询本文件，如果别人已经注册登记了它，你将被告知你无法调出此文件。当你能够调出文件时，你可以在无版本控制下随意对其进行操作直到

你将其登录。当你登录文件时，版本控制软件询问你为何修改它，你对此回答你的理由。

由于你对此付出了最大努力，你将会得到如下好处：

- 别人正在操作某一文件时你不致于和他发生冲突。
- 你能轻易地将所有文件副本升级为当前的版本，通常你仅需使用一条命令即可。
- 你能够回溯任何由版本控制登录的文件版本。
- 你能得到任何文件的任何版本修改表。
- 你无需担心文件备份，因为版本控制拷贝备份是自动的。

版本控制对合作开发计划是不缺少的，它是如此的有效，以致于微软公司的应用部，发现源代码版本控制是具有重要竞争力的方便措施。

制作。一种特定类型的版本控制工具是和 UNIX 以及 C 语言相联系的实用程序。制作的目的是将生成目标文件所需要的时间减少为最少。对工作文件中每一个目标文件及目标文件所依赖的文件都可以应用本控制工具及怎样去制作它。

假定你有一个名叫 user.obj 的目标文件，在制作文件中，你说明你想制作 userface.obj 文件，你必须编译 userface.c。你同时指明 userface.c 将依靠 userface.h，stdilb.h 和 project.h。“依靠”这个概念仅意味着如果 userface.h，stdib.h 或者 project.h 改变了 userface.c 也需要再编译。

当你创建你的程序时，制作检查所有的依赖，同时确定需再编译的文件。如果你的 25 个源文件中的 5 个依赖于 userface.h 中所定义的数据的，并且 userface.h 已经改变了，制作自动再编译 5 个依赖于它的文件。制作不会再编译另外 20 个不依赖于 userface. h 的文件。使用制作可避免再编译所有 25 个文件或单独编译每一个文件。漏掉其中某一个文件，将会得到一些奇怪的错误。总的来讲，制作实际上比一般的编译、链接、运行循环将更节省时间和提高可靠性。

备份计划

备份计划并不是令人兴奋的概念。备份计划即定期备份你的工作。如果你正在编写一本书，你不会将其放到门廊上。如果你这样作，他们可能将被雨淋湿或被风吹走。你最好将其放在某个安全的地方。软件不像纸这样真实可触摸，所以你更易忘记你已在机器上留下了一些错误的东西。

计算机里的数据也会产生许多问题，磁盘可能失效，你或别人也许在无意中删掉一些重要文件。一位愤怒的雇员能破坏你的机器，你也可能由于盗窃、洪水、火灾等不幸事件而失去你的机器。

你应采用一些措施以便确保你的工作成果。你的备份计划应当包括定期进行备份，定期将备份转移至安全处，除了源代码以外，还应包括项目所有的重要资料、文档、图形、注释。

人们在设计备份计划时，常忽视的方面就是对备份程序的测试，不时测试一下你保存下来的东西以确保备份含有你所需要的一切，恢复文件时也能正常工作。

当你成功地完成项目后，你应保存下再开发所需要的东西——源代码、编译程序、工具、需求、设计、文档等，并且要将它们保存在安全的地方。

检查表

配置管理

一般

- 你的软件配置管理计划是否用于帮助程序员，并能将额外开销减至最少？
- 你使用 SCM 手段能否避免对项目失控？
- 你所在组是否有修改请求？请示控制可以是非正式方式或正式的方式。
- 你是否能比较正确地估计每次修改的影响？
- 你是否将重要修改视为需求分析不充分的警告？

工具

- 你是否使用版本控制软件以便配置管理？
- 你是否使用版本控制软件以减少开发时的协调问题？
- 你是否使用制作或其它控制依赖软件使编程更为有效和可靠？

备份

- 你是否将所有项目材料定期备份？
- 你是否定期将所有项目备份移到安全地点存放？
- 包括源代码、文档、图形和重要注释在内的所有材料都备份了吗？
- 你是否对备份程序进行了测试？

因为本书主要是关于创建的，本节主要是从创建的观点讨论修改控制。但是修改从各级水平上影响项目，需要有一集成修改控制策略才行。

22.3　评估创建计划

软件工程管理是二十世纪晚期对人类来说不可轻视的挑战之一。评估工程的大小和完成时间是软件工程管理最为棘手的问题之一。一般的大软件工程完成需一年多时间，同时所需费用也可能会超出预算的 100%（Jones 1986）。这主要是由于对工程的大小和所需时间估计有误和开发过程不努力有关。本节讨论如何评估软件工程，并阐明如何获得更多的信息。

评估方法

你能从以下几个方面评估软件的大小和所需时间：

如果想获得评估方法的更多知识，参看《软件工程质度和模块》或《软件工程经济》等书。

- 使用进度软件
- 使用诸如 COCOMO 的算法，Barry Boehm 的估计模型
- 使用外面其它的专家评估
- 举行预评估会议
- 评估项目的每一部分，然后再集成这些评估
- 让人们评估自己的那部分，再进行集成评估
- 评估完成整个项目所需时间，然后将时间细分到项目的每一部分
- 参考以前项目的评估

- 保存以前的项目估计看看它们的准确程度。用它们调整你的评估

这种方法摘自《软件工程经济》（伯亨利，1981）。

评估对象。你在评估什么？你为何要评估？你的对象需多大的评估精度？评估需多大确定性？乐观和悲观的评估到底能有何差别？

使评估有充足时间并计划评估。匆忙评估是不精确的。如果你要评估一个大的软件工程，你应将评估视为一个小的项目，并花费时间去评估时间。

要获得软件工程的要求的更多信息，参看 3.3 节“需求前提”，以便你能作好评估。

不要草率估计软件工程的各种开销。在某事物还没有被定义前，别人让你去估计产生它需要多少工作量是不明智的。在作出估计之前应定义要求，或对初始探索阶段进行计划。

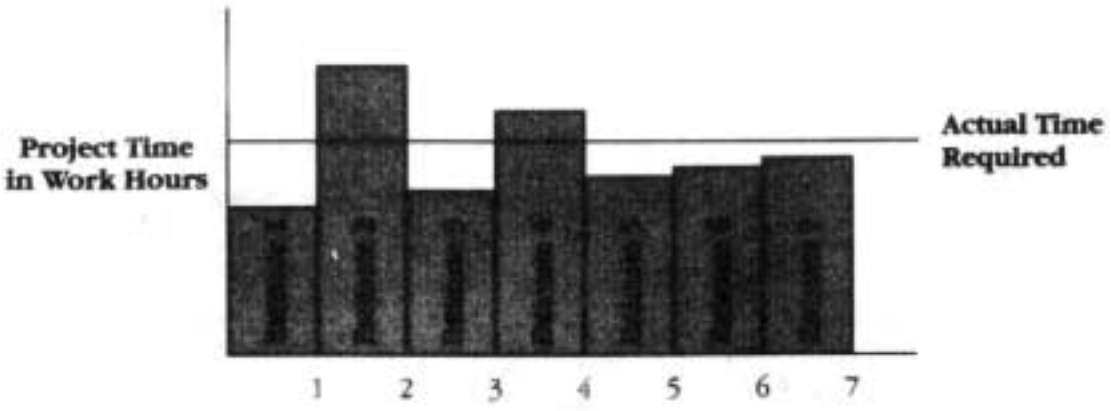
对你所需判别的对象从层次细节上进行评估。从对项目活动的各细节上进行评估，一般说来，你的检查越详细，你的评估就越难确，大多数定理指出总和的错误比错误的总和要大，换句话说，某较大部分 10%的错误，在 50 个小部分中，10%的错误往往会抵消掉。

在软件开发中随处可见重复利用技术，这是重复利用的一个场合，如需了解重复技术的摘要，参见“重复、重复、重复”32.7 这一节。

使用不同的评估方法并比较其结果。在本节的开头给出的评估方法表，说明有几种方法。他们将不会产生同样的结果，所以将这几种方法都试验一下，比较一下结果的不同。

小孩较早就知道如果他们向每一位家长要 1 / 3 碗的冰琪淋，比仅向一位家长要有多一些获准同意的机会。有时家长们聪明地给予相同的回答，而有时他们则不。你应从不同的评估方法中，看一看究竟能得出何种不同的答案。并不是每一种方法都是万能的，而且它们之间的差别应是明显的。

定期再评估。软件工程的各种因素在评估后都会有所变化，所以你应对定期再评估有所计划。在项目接近尾声时，你能评估得更好，同时你也可掂量一下你的初始评估。使用你的评估结果精化以后的评估。在项目收尾之时，你的评估精度也会有所提高。

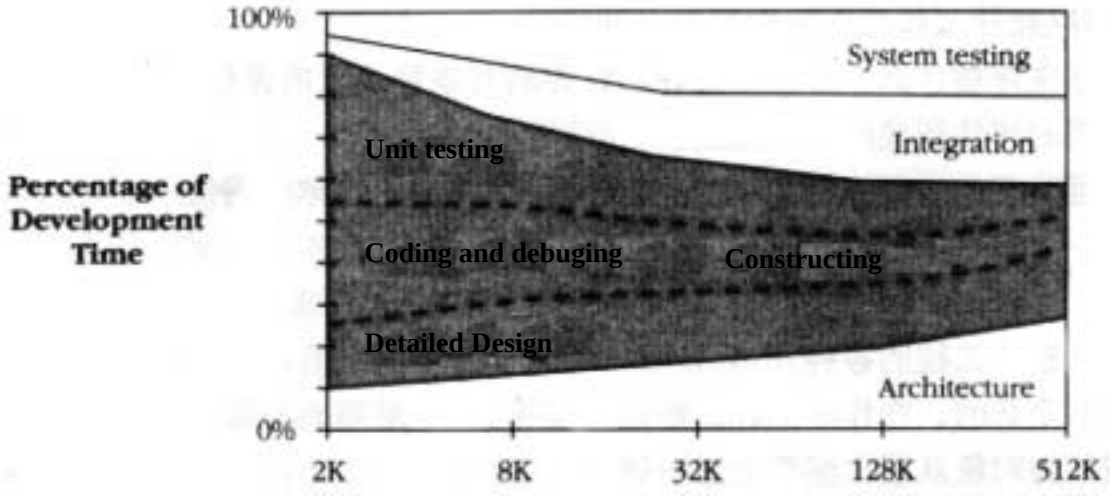


Milestones 软件开发进度表

评估创建工作量

有关不同大小项目的编码工作是细节，参看“工作量分配和大小”这一节（21.2）。

创建对项目进度的影响取决于项目的各项分配——一般为详细设计、编码、调试和单元测试。正如下图所示，各项分配随不同大小的项目而有所不同。



项目代码长度

除非你的公司有自己的项目设计数据，上图所给出项目的各项时间分配对开始评估你的工程是颇有益处的。

对完成一个项目需要多少创建的最佳回答随项目和组织的不同各项分配有所不同。你在开发项目过程中，应当利用你过去的工作经验来估计项目所需花费的时间。

项目进度的影响因素

程序长度对软件效率和质量的影响并不总是凭直觉就能轻易体会得到的，要知道程序大小对创建的影响，参着第二十一章“程序长度怎样影响创建”。

对软件开发进度影响最大的是程序的长度，但是也有许多其它因素影响逻辑开发进度。在表 22－1 中给出了对商品程序产生影响的一些因素。为了利用这张表，你首先要对你项目的大小作一个基本估计，然后用所给表中的每一行的因子去乘基本估计，调整所作估计。

以下为影响逻辑开发进度的一些不能轻易作结论的因素。这们都出自于《Software Engineering Economics》（1981，Barr Boehm）和《A Method of Programming Measurement and Estimation》（C．E．Wdston 和 C．P．Felis）。

- 协作动机
- 管理质量
- 代码再利用量
- 人事变动
- 语言水平
- 需求变更
- 和客户的关系如何
- 用户对要求的参与程度
- 用户的应用经验
- 什么范围内的程序员参与需求分析
- 对计算机、程序、数据的安全确保体系
- 文档量大小
- 项目对象（进度表、质量、可用性和其它可能对象）

虽然以上因素不能轻易下结论，但它们是重要的，所以应将其同表 22—1 各种因素一并考虑。

表 22—1

比 率

项目	很低	低	一般	高	很高	超高	可能影响范围
可靠性要求（不可靠性影响）	.75 轻度不便	.88 低损失易恢复	1.00 中等，损失可恢复	1.15 较高经济损失	1.40 对人有生命威胁		1.87
数据库容量		.94 数据库长度/代码行 <10	1.00 10≤数据库长度/代码行 <100	1.08 100≤数据库长度/代码行 <1000	1.16 数据库长度/代码行 >1000	1.23	
生产	.70	.85	1.00	1.15	1.30	1.65	2.36
复杂性	控制计算，设备依赖，数据管理很简单	控制计算，设备依赖，数据管理简单	控制计算，设备依赖，数据管理一般	控制计算，设备依赖，数据管理中等	控制计算，设备依赖，数据管理复杂	控制计算，设备依赖，数据管理异常复杂	
执行时间要求			1.00 小于执行时间的 50%	1.11 小于执行时间的 70%	1.30 小于执行时间的 85%	1.66 约为执行时间的 95%	1.66
主存限制			1.00≤80% 可用容易	1.07≤70% 可用容易	1.21≤85% 可用容易	1.56 约为 95% 可用容易	1.56
硬、软件的不确定性（操作系统）		.87 每 12 个月大修改，每 1 个月小修改	1.00 每 6 个月大修改，每 1 个月小修改	1.15 每 2 个月大修改，每 1 周小修改	1.30 每 2 个月大修改，每 2 天小修改		1.40
计算开发周转期		.87 交互式	1.00 一般周转期≤4 小时	1.07 一般周转期约为 4-12 小时	1.15 一般周转期≥12 小时		1.32
分析员能力、才能，效率，交流协作	1.46 15%	1.19 35%	1.00 55%	0.86 75%	.71 90%		2.06

项目	极低	低	一般	高	极高	超高	可能影响范围
小组成员在应用领域的经验	1.29 ≤4个月	1.19 1年	1.00 3年	.91 6年	.82 12年		1.57
程序员能力、才能、效率、交流、合作	1.42 15%	1.17 35%	1.00 55%	.86 75%	.70 90%		2.03
程序员基本软硬件经验(操作系统)	1.21 ≤1个月	1.10 ≤4个月	1.00 1年	.90 3年			1.34
编程语言经验	1.14 ≤1个月	1.07 4个月	1.00 1年	.95 3年			1.20
在现代编程中的实际应用	1.24 没有用	1.00 刚开始用	1.00 应用了一些	.91 一般应用	.82 常规应用		1.51
软件工具的使用	1.24 基本处理器工具	1.10 基本工具最小	1.00 基本最小/最大工具	.91 强大编测试工具	.83 增加请求、设计、管理文档工具		1.49
开发进度情况	1.23 额定值的75%	1.08 额定值的85%	1.00 额定值的100%	1.04 额定值的130%	1.10 额定值的160%		1.23

评估和控制

重要问题是：你是想要预测，还是想要控制？

评估是使软件工程按时完成计划的重要组成部分。一旦你确定了交货日期和产品性能，剩下的主要问题是怎样控制人力和技术资源的开销，以便能按时交付产品。从此种意义上来讲，成功地利用各种资源以满足计划要求，比初始估计的准确性更为重要。

如果你落后了怎么办

很多软件开发进度落后于预定要求，对一些实际软件开发的调查结果表明，评估所需时间要比实际所用时间少 20～30%。

当你落后时，增加时间并不是一个明智选择，如果你别无办法，可那样做。否则你可试一试一个或几个方法：

努力赶上。当项目进度落后时，通常的反应是对此抱有乐观态度。较为理智的想法是这样：“所用时间比我们所预期的要长一点，但如今这是不可变更的，我们以后能挽回这点时间”，但

是实际上人们很少这样作，对超过 300 个软件工程的调查指出越接近收尾阶段，延误和超进度情况越来越严重。所以人们在以后不仅不能挽回已失去的时间，而是越来越落后。

扩大队伍人数。对一个已经落后于预定时间的软件工程，增加人数只会使其更加落后。这好比向火上添油。这种解释是颇有说服力的，在新手能富有成效地工作之前，他们需要有一段时间熟悉工作。对他们进行培训浪费了已受训人员的时间。而且增加人数也增加了软件开发的复杂性和相互间的通信量。布鲁克指出一个妇女能在 9 个月中生下一个孩子，并不意味着 9 个妇女能在一个月中生下 9 个孩子。无疑对布鲁克的告诫是应该注意的。它企图使人们全身心地投入开发并按时交付。管理者应该知道开发软件不像柳接薄金属板一样；更多的工人并不就意味着完成更多的工作量。

增加程序员到一个落后于预定时间的项目，会使其更加落后之说，也掩盖了如下一个事实：在某些情况下增加人数有可能加快开发进度。正如布鲁克所指出的那样，对于一个任务不能再分割和不能独自工作的软件开发来说，增加人手无济于事。但是如果项目任务还可细分，并可将其分派给不同的人，甚至是刚进入本项目的人员。另外一些研究人员也证明了你可以在软件开发落后时增加人员不会使其更落后。

减少项目范围，减少项目的有力方法常被人们所忽略。如果你去掉某一特点，也就去掉了相应的设计、编码、调试、测试和文档工作，同时你也就无须再设计这点和其它点的接口。

当你最初计划你的开发时，将开发产品的能力划分为“必要有”、“有更好”、“可选项”几部分。如果你落在这里，先考虑“有更好”和“可选项”并将某些剔除掉。

将某特征从总体上减少一些，能得到一个具有同样功能但是更便宜的一个版本。你也可能准时提交版本，但是它没有经过实际性能调试。也可能提供一个最不重要的功能勉强得到实现的版本。你可能对时间要求放松一些，因为提供一个时间不急的版本是很容易的，你还可能放松空间需求，因为提供扩展存储版本是很容易的。

对重要的功能开发时间，应再评估。在 2 小时、2 天或者 2 星期之内你能提供什么功能？制作一个二星期后的版本，比二天与二个小时的版本有什么更多功能？

22.4　度　量

“度量”这个词指的是和软件开发有关的衡量。代码行数、错误数、每千行代码错误、每个模块的错误、全局变量的数目、完成项目所需的时间等都是度量。

对开发过程进行度量比根本不度量要好，虽然度量并不很准确；也可能难以制定一个度量标准；它需要随着时间能推移而不断精化。但是它能使你能对软件开发过程进行控制，而没有度量则不行。

如果数据在科学实验中被用到，则它应先量化。为了评估软件开发方式，你应首先度量它们，仅仅作出“这种新方式看上去更有成效”的断定是不充分的。

有时度量不必知道项目究竟是怎么一回事。当度量项目的某一方面时，并不知道项目的究竟是变大了，是变小了，还是没有变。本度量仅是项目中这一方面的窗口。在你逐步精化你的度量之前，你的这窗口也可能是较小和模糊不清的，但是这比没有窗口要好。

你能够度量软件开发过程的各个方面，表 22－2 给出了一些被证明为行之有效的度量。

表 22－2　有用的度量

长度	总质量
代码总行数	总的错误数量
注释总行数	每个子程序的错误数目
数据说明语句总数	每千行代码平均错误总数
空行总数	失效的平均时间
	编译错误
生产率	
总的项目时间	维护性
每个程序所花时间	每个子程序所用参数数目
每个程序修改次数	每个子程序所用局部变量数目
项目费用	每个子程序被其它子程序调用次数
每行源代码费用	每个子程序的断点数目
每个缺陷所耗费用	每个子程序的控制流向复杂性
	每个子程序的代码行数
错误跟踪	每个子程序的注释行数
	每个子程序的数据描述数
每个错误的严重程度	每个子程序的空白行数
错误的位置	每个子程序的 goto 语句数
纠正每个错误的方法	每个子程序的输入 / 输出语句总数
对每个错误所负责的人	
纠正每个错误所影响的代码行数	
纠正每一个错误所用时间	
发现一个错误所用平均时间	
修正一个错误所用时间	
纠正每一个错误的尝试次数	
由于纠正错误所引起的新的错误数目	

使用当前可用的软件工具，你能得到大多数度量。本书的讨论指出每种度量都是有用的。当前，大多数度量并不能明确区分程序、模块和子程序（Shepperd 和 Ince l989）。度量主要用来鉴别“外来”子程序。一个子程序的反常度量是说明子程序质量低需再检查它。不要对所有的可能度量都想收集其数据——你将会置身于数据的汪洋大海之中而不知所然。你应当从诸如错误总数、工作耗费时间、总费用、总的代码行数等简单的度量开始。

在项目开始过程中，标准各种度量，逐步精化它们，以作为衡量进展情况的标准。你应确保你是抱着某种目的而收集数据，确立目标，从而确定实现目标还有何问题，最终度量这些问题。美国宇航局软件工程实验室的一份数据收集总结指出，在过去的 15 年中所得出的最重要的教训就是，度量之前你应定义度量目标。

22.5　将程序员视为普通人

编程活动的抽象性要求程序员有适应办公室环境的天性，并且和使用者有密切的交往。高技术公司常向其雇员提供公园式的合作园区，有系统的组织结构、舒适的办公环境以及其它“高

级”环境以平衡精深的有时也是异常智力型的工作。最成功的公司其成功的原因在于高技术和高接触的有机统一（Naisbitt 1982）。本节指出了程序员并不仅是其自我的反映。

程序员们怎样花费自己的时间

程序员不仅要花费其时间编程，也要花费时间开会、接受培训、阅读邮寄材料和思考。1964年对贝尔实验室的调查发现程序员从以下几个方面花费他们的时间：

活动	源代码	事务	私事	开会	培训	邮寄	技术手册	程序运行	程序测试	合计
听说	4%	17%	7%	3%				1%		32%
和管理人员谈话		1%								1%
打电话		2%	1%							3%
阅读	14%				2%	2%				18%
写/记录	13%				1%					14%
外出		4%	1%	4%	6%					15%
散步	2%	2%	1%			1%				6%
其它	2%	3%	3%			1%		1%	1%	11%
合计	35%	29%	13%	7%	6%	5%	2%	2%	1%	100%

以上数据是建立在对 70 位程序员的工作时间和动机时间研究的基础之上，它是过时了的，而且不同活动中的时间分配随不同的程序员而异，但是此结果还是有所启发的。一个程序员大约有 30％的时间花在和项目没有直接联系的活动之中，散步、私事等等。在以上调查中，程序员花费了 6％的行路时间。这意味着他们一年有 125 个小时，一周有 2.5 个小时在路上消耗掉了。你也许认为这并没有什么，但是当你看到程序员行路时间和他们花费在培训上的时间相当，并且是三倍于他们阅读技术手册的时间，六倍于他们和管理人员谈话的时间之后，你会有所触动的。

能力差别

不同程序员的才能和努力的区别是巨大的，就像其他领域一样。一项对写作、足球、发明创造、治安和飞行等不同职业的研究表明，其中 20％的精英人物却取得了 50％的成就。以上研究结果是建立在对各项得分、专利、已解决案例等数据之上的。由于不少没有实际可见的贡献（如足球运动员没有得分、发明者没有自己的专利，侦探没有破获案子等），本数据也可能低估了实际的效率差异。

对编程来说，许多研究表明对不同的程序员，其在程序书写质量、程序长度和效率方面存在较大差别。

个人差别

在 60 年代，Sackman，Erikson 和 Grant 的研究初步表明，程序员在编程效率方面存在重大差别。他们研究了平均工作经验为 7 年的专业程序员的编程过程，发现最好和最差的程序员的初始编码时间比为 20：1，调试时间比 25：1，程序长度比为 5：l，程序执行速度比是 10：l，他们也发现，程序员的工作经验和编码质量或效率并不存在必然联系。

虽然像 25：l 这样的比例并不特别有意义，但是一般的常识如“在程序员之间存在重大差别”是有其意义的，并且已经被对许多专业程序员的研究所证实(Curtis1981，Mills 1983，DeMarco 和 Lister 1985，Curtisal 1986，Card 1987，Boehm 和 Papaccio 1988，ValettMcGarry 1989)。

组间差别

不同的编程组之间也有软件质量和生产效率的差别。好的程序员往往趋于聚集在一起，正如差的程序员一样。这个结论已经被对 18 个组织的 166 个专业程序员的研究所证实（DeMacro 和 Lister 1985）。

对七个相同的项目研究表明，其所耗工作量的范围之比为 3：4：l，同时程序长度之比可达 3：1。尽管存在种种差别，所调查的程序员并不是完全不同的组别。他们都是有几年工作经验的专业程序员，并且都是计算机专业的毕业生。由此看来如果各级之间稍不同将会产生更大的差别。

一早期研究表明，不同的编程小组完成同一项的程序，长度比可达到 5：1，并且所需时间可能为 2.6：l。在研究了有关编程数据之后，柏雷·玻利亨认为，一个组间程序员能力差别为 15％的编程小组所需开发时间 4 倍于组间差别为 90％的编程小组。玻利亨和其它研究者还发现 80％的贡献来自于 20％的研究人员（1987）。

补充雇用人员是经常的。如果你为了得到高水平的程序员而比得到低水平的程序员要多付出 2 倍的报酬的话，请你抓住这个机会。你将因雇用的程序员而在质量和效率方面得到回报，而且这也会对你所组织的其它程序员的质量和生产率产生影响，因为好的程序员倾向于聚在一起。

个人风格问题

编程计划管理人员往往不太清楚个人心理等对编程有较大影响。如果你试图要求遵守一定的编程习惯，你可能会激怒你的程序员。以下为一些个人心理因素：

- goto 语句的使用
- 编程语言
- 缩进风格
- 大括弧和 begin－end 关键词的使用
- 文本编辑的选择
- 注释风格
- 效率和可靠性的互换
- 方法的选用——例如，面向对象设计或创建设计
- 实用程序
- 变量命名习惯
- 局部变量的使用
- 度量，尤其是生产效率量度（如每天代码行数）

以上讨论的共同特征是每个程序员的个人风格的反映。如果你想以上述各项来要求你的程序员，请考虑以下各点。

清楚地知道你是在处理一个敏感的领域。在你开始你的行动之前弄清醒每位程序员的思想。

使用“建议”要同时避免使用教条的“规则”或“标准”。

规避使用明确的控制。为了控制缩进风格或大括弧使用，在源代码宣告完成之前要先经过整齐打印格式化程序运行。让整齐打印机作格式化工作。为了控制注释风格，要求所有的代码要经过检查，不清楚的代码要作修改。

你的程序员是否开发出了自己的标准。正如在其它地方所指出的那样，特定标准经常不如一些已存在的标准重要。你不必为你的程序员设置标准。但是你应坚持要求你的程序员标准化你认为是重要的地方。

为什么个人心理因素最重要以至于能保证不入歧途？对任何次要问题的限制可能不足以弥补士气所产生的影响。如果你发现无区分地使用 goto 语句、全局变量、不可读风格或其它影响整个项目的行为，你得为了提高代码质量而容忍一些摩擦。如果你的程序员是一个认真负责的人，这倒并不是一个问题。最大的困难莫过于代码风格的细微差别，如果对整个项目无甚损失你可不必理睬这些。

物理环境

以下是一个实验，走到乡间去，找到一个农场，再见到农场主。问他对平均每个工人来讲其设备价值多少？农场主会看看他的仓库，看一看拖拉机、运输车、玉米、豌豆然后告诉你对每个雇员来讲价值 100，000 美元。

接着再到城里，寻找到一个软件开发部，见到经理，问你对每个雇员来说，其设备价值多少？经理看看办公室，扫视一下桌子、一把椅子、一些书籍、一台计算机并告诉你每雇员为 25，000 美元。

物理环境对生产率有重大影响。DeMacro 和 Lister 询问了 35 个组织的 166 位程序员关于他们物理环境的质量。许多雇员对自己的物理环境并不满意。在后来的开发竞争中，前 25％的高水平程序员有着自己的更大、更安静的办公室，并且被来访者和各种电话打断机会要少。以下为最好和最差程序员办公室环境差异摘要：

环境因素	最好的 25％	最差的 25％
未用房间空间大小	78 平方英尺	46 平方英尺
可接受工作环境	57％是	29％是
可接受个人环境	62％是	19％是
不受电话打扰情况	52％是	10％是
可不受来客打扰情况	76％是	19％是
频繁的不必要打扰	38％是	76％是
对工作空间的满意程度	57％是	29％是

以上数据表明了生产率和工作空间质量的紧密关系。最好的 25％的程序员的效率是最差的 25％程序员的 2.6 倍。DeMacro 和 Lister 首先认为好的程序员由于得到提升，他们的办公条件可能一开始就好些，但是进一步调查证明，事实并不是这样。来自相同组织的程序员虽有相同的条件，但是他们各自的表现并不一样。大的软件开发组织有相同的经验。Xerox，TRW，IBM 和 Bell 实验室表明平均每人 10,000 到 30,000 美元的投资对大大提高生产率是必要的（Boehm 1987），而这笔数目还能从提高的生产率中获得回报。有了较好的办公条件，估计可得到 39～

47%的生产率增长(Boch 等，1984)。概括地讲，将你的工作环境从最差的 25%的水平提高到最好的 25%的水平可能会导致最少为 100%的生产率增长。

22.6　如何对待上司

在软件开发过程中，非技术人员往往是管理者。最为例外的情况是管理者有工作经验，但是已是十年未再干过了。会技术的管理者是少见的。如果你为某一人工作，应尽力保住你的饭碗。这不是一件容易的事情。作为一个阶层，每一位雇员都想升到他并不能胜任的层次。

如果你的上司并不是一个特别的人，你将不得不学会如何面对你的上司。“控制你的上司”意味着你应告诉你的上司怎样去做，而不是用其他方法。其要决在于用这样一种方法使你的上司相信你仍是受他管理的一员。以下是一些对付你上司的方法：

- 拒绝按照你上司的吩咐去做，坚持按正确方法继续你的工作。
- 假装按照你上司的吩咐去做，暗地里按照正确的方法去做。
- 先对自己如何做有一个全盘计划，等着上司对你的见解做评论并让人如何去做。
- 告诉你上司正确的方法，这是一个中途应变方法，因为你的上司经常提升、调动或被解雇。
- 寻找另一份工作。

最好的解决方法是努力说服你的上司。这并不是一件容易的事情，但是你可阅读《How to Win Friends and Influence People》(《怎样赢得朋友和影响他人》)一书以学会如何准备这件事。

22.7　小　结

- 配置管理，适当应用时，可使程序的工作变得更容易进行。
- 你能找到某种方法度量项目的某一方面，这样比根本不度量好。准确的度量是完善的计划、质量控制和提高开发速度的关键。
- 程序员和管理者都是普通的人，当他们受到礼待时往往干得更好。
- 合适的软件工程评估是软件开发管理最富挑战性的方面，你不妨尝试几种方法，看看评估的差别，以加深你对项目的认识。

第二十三章　软件质量概述

目录

相关章节

评审：见第 24 章
单元测试：见第 25 章
调试：见第 26 章
创建前提：见第 3 章

本章研究软件质量技术。虽然本书是讨论软件质量的提高问题，但是本章主要是对每种活动的质量和质量保证进行讨论。它主要给出一个大概的轮廓而不是对具体细节的讨论。如果你想了解对评审、调试、测试的有关细节，请看下三章。

23.1　软件质量特点

软件既有外部也有内部质量特征。软件的外部特征是用户应了解的软件产品属性，它包括：

- 正确性。整个系统受说明、设计和实现的错误影响程度。
- 可用性。用户学会和使用系统的难易程度。
- 效率。对系统资源的最小利用，包括存储和执行时间。
- 可靠性。系统在一定条件下执行特定功能的能力——在每次失效之间有一个较长的平均时间。
- 完整性。防止非法或不适当地访问。完整性思想包括：限制非法用户访问，同时确保证数据恰当访问；并行数据表进行并行修改；数据段仅含有有效数据等等。
- 适应性。系统在应用或其它环境下不作修改就能使用的能力，而不必经过特定的设计。
- 精确性。系统不受错误影响的程度，尤其是数据输出方面。精确性和正确性是不同的。精确性是对系统完成其工作性能良好的衡量，而不是它设计得是否正确。
- 坚固性。系统对无效输入或压力环境中能继续执行其功能的能力。

以上有些方面是重复的，但是它们在某些场合有其特定的意义而在一些场合则没有。

外部特征是用户所关心的特征。用户常关心软件是否易使用，而不是是否容易修改。他们也关心软件是否能正确工作，而不是代码是否可读或结构较好。

但是，程序员既关心内部特征也关心外部特征。本书着重讨论代码，所以着重于内部特征，其包括：

- 可维护性。修改一个软件系统，提高其性能或修正其错误的能力。
- 灵活性。修改系统使其能适应于不同的用途或环境的能力，而不必对系统进行特定的设计。
- 可移植性。能修改所设计的某一系统使其能在其它环境下运行的能力。
- 可重用性。能将系统的一部分用于其它系统的难易程度。
- 可读性。能读懂或理解系统源代码的能力，尤其是在细节说明这一级上。
- 可测试性。对整个系统进行单元或系统测试以证实其满足所有需求性能的测试难易程度。
- 可理解性。能从整个系统水平或细节说明这一级上理解整个系统的难易程度。可理解性要比可读性从更一般的水平上讨论系统的紧密性。

从所列的外部性能特点可知，它们和一些内部特征重合了，但是它们各自还是有着不同的意义。

系统的内部性能是本书所讨论的主题，但是在本章中不对其进行深入讨论。

内部特征和外部特征不是非常分明的，因为从某些方面来说，内部特征影响外部特征。软件的内部特征，如可理解性或可维护性的不佳将损害你纠正错误的能力，这会影响系统的外部性能如正确性和可靠性。软件设计的非灵活性不能对用户的要求反应迅速，反过来这也会影响外部性能，如可用性。关键是一些性能是适用用户的，而另外一些是适于程序员的，你应知道如何选择。

为了获得某些最大性能，不可避免会跟其它性能发生矛盾。从互相矛盾的事物中挑选一个最优解答，使软件开发成为一工程活动。图 23−1 给出了一些外部性能影响其它特性的途径。软件质量的内部特征也同样有此关系。

被影响特性 / 特性	正确性	可用性	效 率	可靠性	完整性	适应性	精确性	坚固性
正确性	↑		↑	↑			↑	↓
可用性		↑				↑	↑	
效 率	↓		↑	↓	↓	↓	↓	↓
可靠性	↑	↑		↑	↑	↑	↑	↓
完整性			↓	↑	↑			
适应性					↓	↑		↑
精确性	↑		↓	↑		↓	↑	↓
坚固性	↓	↑	↓	↓	↓	↑	↓	↑

23-1 对软件质量其它特性产生积极消极或无影响的外部关系表（其中；↑帮助，↓影响）

最有趣的是本章着重于并不总是和其它特性有联系的一些特性。有时一种特性阻碍另一种特性，有时则有助于另一种特性，或者既不妨碍也不促进。例如，正确性是关于满足规范要求的特性。坚固性是关于在非预期环境继续进行工作的特性。着重于正确性将会影响坚固性，反

之亦然，而着重于适应性将会损害坚固性，反之亦然。

本图仅指出了质量特点的各典型联系。对任一给定的项目，某两特点间的联系可能和其典型联系有所不同。对特定质量目标和各目标间的相互影响有一个通盘的考虑是有益的。

23.2　提高软件质量的方法

软件质量保证，是保证系统满足性能要求的有计划、有组织的活动。开发高质量产品的最为有效的方法是提高产品本身的质量。软件质量保证最好的方法是控制软件的开发过程。以下是关于软件质量管理计划的组成部分：

质量管理目标。提高软件质量一个有效方法，是从上节所讨论过的外部和内部特征中挑选出明确的目标。没有明确的目标，程序员就可能着重于并不是你所要求的特性。本书以下几部分讨论了建立明确目标的益处。

确定质量保证活动。对质量保证的一般看法是将质量视为一个目标。的确，在一些组织中，急促和草率的编程往往是一件常见的事。程序代码充满错误但能很快完成编程的程序员往往能得到更多的奖励。而高质量的程序员。虽然编出的程序优秀而且确保其是可用的，却往往得不到这种礼遇。在这样的组织中，人们也就不会对程序员不再把编程质量放在首位而惊讶了。所以应该让程序员明白质量是第一的。进行独立的质量保证活动使这种重要性得以体现出来，而程序员也能作出相应的反应。

测试策略。如想了解测试的详情，请参看第二十五章“单元测试”。

执行时间可以作为对产品可靠性的估计。开发者往往将调试作为质量评估和质量提高的主要手段。本章的其余部分更加详细地说明了调试的重要性。设计在高质量的软件创建过程中起着重要的作用。而质量保证在对产品的质量、结构和设计有所约束的同时，也可得出相应的测试策略。

软件工程准则。如想了解和创建有关的软件工程准则，请看第 3.6“编程约定”这一节。

在软件开发过程中，有一些准则用以控制某技术特性。这些准则可应用于开发过程中的各个阶段，如问题定义、需求分析、结构、创建和系统测试。本书的准则，从某种程度上说，是软件工程创建的准则（详细设计、编码、单元调试和集成）。

非正式技术检查。许多软件开发者在将工作送交正式检查之前，往往先要对其进行非正式检查。非正规检查包括手工检查设计、代码或对整个代码普查一下。

正规技术检查。软件开发管理是在最低级阶段发现问题，即在问题花费最少的阶段发现问题。为了实现此目标，软件工程开发者常使用“质量门”——（quality gate）定期的检查以确定某阶段的产品质量是否达到了继续下一步的要求。质量门常在需求分析和结构、结构和详细设计、详细设计和编码、编码和测试之间使用。质量门可以是普查、客户检查等检查。

外部检查。外部检查是一种特定类型的技术检查，以决定项目的状态或正在开发之中的软件质量。检查组来自外部并将其调查结果交给委托者，通常是你的上司。

开发过程。以上所提的各部分和软件质量有明确的关系，而与软件开发过程有着暗含的关系。含质量保证活动的开发过程将比不含质量保证活动的过程会开发出更好的软件。其它不和软件质量有明显关系的活动也将会影响软件的质量。

开发过程中一个并不明确、但能影响软件质量的情况是危险管理的使用。如果危险是可预

测的、可靠的，则软件也是可预测的和可靠的。如果危险是特别的，则所编软件将是混乱的——反映出产生它的组织的结构。

修改控制过程。妨碍软件质量的一个障碍是失控的修改。失控的修改会导致编码和设计的毁坏。结构或设计上的失控修改会导致代码和设计不一致。而修改代码以适应新的设计将比按预定计划需要更多的时间。对代码的失控修改，也会导致代码本身内部的不一致性，因不知道哪些代码被调试过、检查过而产生的不一致性。而失控的文档修改——需求、结构、设计和代码的集成表述，能产生以上各种影响。所以，有效地处理各种修改是进行有效开发的关键。

结果的定量。质量确保计划应是可衡量的，否则你无法清楚计划是否产生了效果。度量告诉你计划的成功或失败，也允许你对进度作出一定的修改，以看看进度是否有所加快。度量能产生另外一些影响。人们常因度量的使用而注意到什么被度量了。仔细挑选你的度量对象，人们习惯于着重那些被度量的工作，而忽略没有度量的工作。

原型。原型是对系统关键功能的可实现模块的开发。开发者应将用户界面部分地原型化以确定可用性，同时进行计算以决定执行时间，以及数据安排所应满足的存储要求。对 16 个发表的和 8 个未发表的用例调查，将原型和传统的规范开发方式相比较。这比较提示了原型可导致更好的设计、更好地满足用户需求，以及提高可维护性。

设置目标

明确地设置质量目标是确保软件质量一个有效的步骤，但是它易被忽略掉。你可能怀疑如果当你设置明确的质量目标之后，程序员们能否实现它们？回答是他们实现了的，只要他们知道目标并且目标本身也是合理的。如果目标经常改变或根本不可能实现，程序员对目标是会无动于衷的。

Gerald Weinberg 和 Edward Schulman 进行了一项有趣的试验，以研究设立质量目标对程序员所产生的影响（1974）。其研究对象是分别工作于 5 个不同程序版本的 5 组程序员。这 5 个组的质量目标是大致同的，每个组被告知可尽力去实现其不同的某方面目标。一个组被告知满足最小存储要求，另一个组应产生最为清晰的输出，一个组被告知建立最好懂代码，另一个组被要求使用最少的语句，最后一组则被告知用最少的时间完成程序的编制工作。以下是其结果：

每组目标范围

目标	最少存储	输入可读性	程序可读性	最少语句	最小编程时间
最少存储	1	4	4	2	5
输出可读性	5	1	1	5	3
程序可读性	3	2	2	3	4
最少语句	2	5	3	1	3
最少编程时间	4	3	5	4	1

本表摘自《Goals and Performance in Computer Programming》(Weinberg and Schulman 1974)。

本研究的结果是显而易见的。5 个组中的 4 个最先完成了被告知应优化的目标。而另一个组是其次才完成了应优化的目标。以上各组对各自的目标都完成得相当好。

以上调查最令人吃惊的是人们往往按照所要求的去作。程序员有着较高的成功动机，他们

会按所要求的目标去作，但是他们必须知道目标。第二个含义是，正如所预计的那样。由于目标间常相互冲突，通常并不可能满足所有的目标。

23.3　各种方法的效果

不同的质量确保活动将会导致不同的效果。人们已经研究了多种方法，其发现和纠正错误的效果也是可知的。本节讨论质量确保活动的效果的几个方面。

有一些方法较其它方法能更好地发现错误，不同的方法将导致不同类型错误的发现。评估错误检查方式的一个方法是，确定在产品寿命周期中所发现的错误对全部错误的百分比。表23-1（摘自 Caper Jone 的《编程效率》）一书给出了 10 种常见方法发现错误的百分比。

表 23－1　错误发现百分比

步骤	最低比	中等比	最高比
对设计文件的人工检查	15%	35%	70%
非正式组内设计检查	30%	40%	60%
正式设计检查	35%	55%	75%
正式代码检查	30%	60%	70%
模型或原型	35%	65%	80%
人工代码检查	20%	40%	60%
单元测试（单个子程序）	10%	25%	50%
功能测试（相关于程序）	20%	35%	55%
系统测试（整个系统）	25%	45%	60%
段测试（动态数据）	35%	50%	65%
集成测试	25%	35%	40%

本表摘自《Programming Productivity》（Jones 1986）

最有趣的是本数据提示出对任何单一方法其中等比不会超过 65%。而且，对于大多数一般类型的错误如单元测试，其中等比仅为 25%。

本数据有力地说明，如果项目开发者争取得到一个较高的错误检查比，他们就需要集成应用几种方法。Glenfotd Myers 的研究就证实了这点。Myers 研究了一组工作经验最少为 7 年而平均工作经验有 11 年的程序员。在给出一个有 15 个已知错误的程序之后，他让每一位程序员使用以下几种方法寻找错误：

- 描述测试
- 对源代码测试
- 对描述和源代码进行普查

Myers 发现了很大的差别。每位程序员所发现的错误数从 1 到 9 个错误不等。而平均发现错误数为 5.1，或者是程序错误总数的三分之一。

当单独使用时，不能说出以上各种方法谁优谁劣。人们所发现的错误数差别是如此之大，但是，任何二种方法的组合使用（包括独立的两组使用相同的方法）就可能将所发现的错误总数提高近二倍。对美国宇航局软件工程实验室的调查指出不同的人所发现的错误是不同的，只有两位代码阅读者通过代码阅读发现了 29%的错误。

Glenford Myers 指出在发现某一类错误时，人工检查比计算机调试更为有效，反之亦然。以上事实后来被下面的研究结果所证实：通过阅读代码能发现较多的控制错误。

由此得出的结论是，成对地使用错误检查方法要比单独使用好。Jones 也通过观察到集成错误检查效率比任何单一方法要高许多。任何使用单一测试方法其效果并不明显。 Jones 指出多种测试方法如单元测试、功能测试、系统测试的综合应用所产生的集成错误检查率将不会低于60%，这对于商业软件来说是合适的。

发现错误的代价

有些错误检查方法将比其它方法代价高。其中最为经济的方法是发现每个错误的代码都是最低。这要求每件事物都是平等的。所有事情都等同这个条件是不可能的，因为每个错误所花代价是受总的错误数影响的。另外每个错误都应被发现。而且这种评估只考虑错误检查方法并不估计其它因素。

在 1978 年 Myers 的研究之前，二种测试运行方法平均发现每个错误的代价差别不大，但是人工检查方法所付出的代价是测试方法的二倍。不幸的是，以上结果并不是不容置疑的，因为此次的研究对象缺乏检查经验。

当人们有了经验后，他们的检查工作也会更为有效。因此，最近的研究结果已有力地说明检查比调试更为合算。有人对一个系统的三次交付进行了研究，第一次交付时，使用各种方法也仅发现了 15%的错误。第二次交付时发现了 41%的错误，第三次则发现了 61%的错误。如将以上事实用于 Myers 的研究，就有可能得出，对每个错误来说检查的代价仅为调试的一半而不是调试的二倍的结论。对软件工程实验室的研究发现，阅读代码要比调试平均每小时多发现80%多的错误。

修改错误的代价

参见第 3.1 节“求助于数据”和第 25.4 节“典型的错误”。

发现错误的代价是总代价的一部分。另一个问题就是修改错误的代价。乍看起来你可能以为修改一个错误应花费相同的代价。

其实并不是这样，错误留在系统中的时间越长，将其除去所花的代价也越大，能较早发现错误的方法将导致低的错误改正代价。而且，有些方法如人工检查，一步就可完成发现和改正错误的全过程。另外一些方法如测试，在发现错误之后还需另外的工作以分析和改正错误。从总体上看，一步方法要比二步方法更为合算。微软公司的应用部已经发现，用代码检查的一步方法可在 3 小时之内发现和改正一个错误，而用调试二步方法 12 小时之内才能发现和改正一个错误。Collofello 和 Woodfied 曾经报道过一个由超过 400 人开发的 700,000 行程序，他们发现代码检查和调试所获得的收效之比为 1.38:0.17。

以下是关于有效的软件质量程序必须包括的几种技术问题：

- 对系统关键部分的正式设计检查
- 使用快速原型化技术进行模块化或原型化
- 代码阅读或检查
- 运行测试

23.4　何时应作质量保证

正如第三章所指出的那样，错误进入软件的时间越早，它就越深藏于软件的其它部分中，也就越不易将其移去。一个分析上的错误能在设计上产生一个或多个相应的错误，而这也会导致代码产生许多错误。分析的错误能导致不必要的结构或坏的结构选择。多余的结构将导致多余的代码、测试和文档。正如在浇铸地基之前，在设计蓝图上找出各种缺陷一样，在各活动之前找出分析和结构错误不失为一种好方法。

另外，由于需求分析和结构比创建更为包罗万象，所以一个单一的结构错误能影响好几个模块和不少子程序，而单一的创建错误不太可能影响多于一个的子程序或模块。由于这种原因，尽力发现各种错误也是合算的。

在软件开发的各个阶段都有可能潜入错误。因此，你在开发过程中，应自始至终强调质量保证工作。在工作一开始就应将其作为整个项目计划的一个部分。而且，质量保证工作也应坚持到项目结束为止，这样才能最后确保产品的质量。

23.5　软件质量的一般原则

如同没有免费午饭，或者即使有也不能保证其质量较好一样。软件开发和烹饪是全然不同的事情，而且从某种意义上看，软件质量是不一般的。软件质量的一般原则是提高其质量并少各种花费。

要理解此原则应基于对以下事实的理解：提高效率和质量的最好方法是减少代码再加工的时间，不论再加工是由于要求的变更、设计的修改或调试。软件产品的工业平均生产率是每人每天约 8 到 20 行代码。要编制 8 到 20 行代码是只需要几分钟的事情，那么，其它的时间干什么去了呢？

开发一软件要比仅开发一个程序所花的时间多得多，而且软件开发所做的事要比仅编码多得多。但这不是剩下的时间所作之事。

剩余的时间通常用于调试。调试通常要占一个传统的初始软件开发周期的 50%。消除掉防止错误的软件调试可提高生产率。因此，缩短软件开发时间最为明显的方法是提高产品质量，减少调试和再开发软件所需时间。

对实际数据的分析征实了这点，在对含超过 400 人年工作量和近三百万行代码的 50 个软件项目进行调查后，美国字航局软件工程实验室发现，改进质量保证和降低错误相联系，但是并不意味着总的开发费用降低。

对 IBM 公司的研究也得出了以下调查结果：如果不顾质量而只是想用最短的时间将软件开发出来，往往很可能需要较长的时间和花费超出。从一开始就着眼于取得最高可能质量和可靠性的软件开发，易于取得最好的开发进度、最高的生产率甚至是最好的市场成功率。

在一个较低的水平上也同样存在这种情况。在 1985 年的某一研究中，让 166 位专业程序员对同一描述编写程序。结果是程序平均行数为 220 行，并且只有少数人仅用 5 小时就完成了编程工作。最为有趣的是花费中等时间的程序员所编的程序错误要少得多。图 23-2 给出了这种结果：

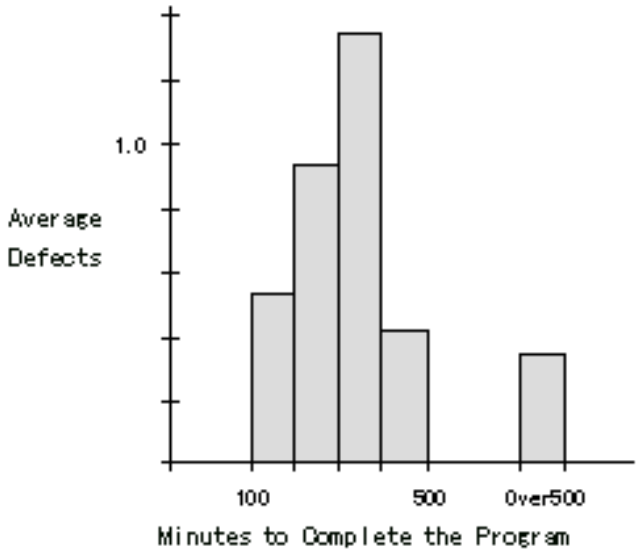


图 23-2　产生最多错误的软件开发并不是最快或最慢的软件开发

最快的组需花 5 倍的时间才能取得和最慢的组相同的错误率。往往并不一定是开发无错误的软件需更多的时间，而是开发没有错误所需时间少些。正如上图所示，没有错误的开发所需时间更少。

虽然，对某一类项目，质量保证需有所花费。如果你在为航天飞机或生命保障系统编写代码，可靠性需求会使本项目花费更多。

跟传统的编码——测试——调试相比，一个明确的软件质量程序有助于各种费用的节省。它避开重新分配资源以进行前期质量确保活动。前期活动较后期对产品质量有更大的影响，你在前期活动中所投入的时间将会节省更多的后期时间。其结果是较少的错误、较短的开发时间和较低的代价。在下二章中你将会看到软件质量的一般原则的更多例子。

检查表

质量保证步骤

- 你是否有项目较为重要的特定质量描述？
- 你是否让其它人明白项目的质量目标？
- 你是否要求不同的外部和内部特征？
- 你是否考虑过某些特性可能和其它相互矛盾或采用共同促进的方式？
- 你的项目是否需要几种不同的错误检查方法，以便能发现几种类型的错误？
- 你的项目是否包括这样一个计划，以便在软件开发过程中采取措施确保软件质量？
- 你的软件质量是否能按某种方式度量，以确定软件质量是提高或下降了？
- 你在管理中是否知道质量确保在刚开始可能增加费用，但今后的花费能节省？

23.6　小　结

- 并不是所有质量保证目标都能实现。确定你想实现的目标，并将其让你所在组的每一个人知道。

- 各种错误检查方法就其自身来说并不十分有效。仅用一种方法消除错误并非有效。成功的质量确保计划使用几种不同的方法，以确定不同类型的错误。
- 你可在创建之前或创建过程中使用几种有效的方法以发现错误。你发现错误越早，其所引起的损失也越小。
- 软件开发领域中的质量保证是面向过程的。软件开发并不包括影响最终产品的重复开发阶段，所以产品开发过程控制着产品的质量。
- 质量最终是主动的，它要求对系统的资源进行再分配，以便能预防错误而不是花费较大地去提高质量。

第二十四章　评　审

目录

相关章节

软件质量概要：见第 23 章
单元测试：见第 25 章
调试：见第 26 章
创建前提：见第 3 章

你可能同其它程序员一样有了一定的经验。你为你的用户作好了演示的准备并已从程序中排除了所有错误。在演示过程中，用户提议采用奇怪的输入，按随意秩序操作键盘，或者将程序运行 10 次。突然毛病不知从什么地方冒出来了。所有的评审都试图用一种或多种方式使你的工作在用户见到前显得更完美。如果你已读过检查的有关章节，你可能从本章得不到多少新知识。在第 24.l 节中所给出的检查效果的数据可能会使你吃惊，你也可能没有意识到第 24.3 节所述的代码阅读检错法的重要性。但是如果你目前了解的都来自自己的经验，请读下去！其它人既然有着不同的经验，你也可从中获得不少新见解。

24.1　评审在软件质量保证中的地位

所有的技术评审，虽有着不同之处，都是建立在本思想之上：开发者对其工作中的一些故障点是一无所知的，而另外一些人则不存在这个盲区。所以让别人来检查你的工作是有益的。

解释一下问题，足以使你找到问题症结之所在。常见到一些程序员走进另一个程序员的办公室并说道：“请你看看我的程序，好吗？我遇到麻烦了。”于是程序员开始解释问题，约三分钟后，在“帮助者”未发一言之前，被帮助者自己就已明白了错误之所在。

有些人从某种特定的技术角度使用“评审”这个词。在本章中，“评审”是检查、初查、代码阅读以及别人查看你的工作的其它方法的总称。

评审和其它质量保证方法互补

评审的主要技术目的是提高软件质量。正如第二十三章所指出的那样，单一的软件测试方法只能取得有限的效果——单元测试的检错比仅为 25%，功能测试为 35%，而集成测试为 45%，相比之下，设计和代码检查的检错比平均为 55%、60%。而评审的辅助作用在于减少了开发时间，从而降低了开发费用。对评审结果事例的研究情况是令人难忘的：

- 在一次软件维护中，在引入代码检查前，55%的联机维护修改是错误的，而在使用代码检查后，只有 2%的修改是错误的。当所有的修改一并考虑时，应用代码检查法一次就可改正 95%的错误，而未用代码检查法，一次只能纠正 20%以下的错误。
- 在同一开发组开发 11 个程序的过程中，开发的 5 个没有用评审的检查方法，其余的 6 个则用到评审的检查方法。在所有的程序都提交生产后，头 5 个程序平均每百行代码有 4.5 个错误，而经过评审的其余 6 个，则每百行代码只有 0.82 个错误。评审竟减少了 80%以上的错误。
- Aetna 保险公司发现，程序中 82%的错误可通过检查发现并且检查可将开发资源减少 25%。
- IBM 公司的 500,000 行的 Orbit 项目使用了 11 种检查方法。它不仅交付时间短，而且所产生的错误仅为正常情况时的 1%。
- 对 AT&T 公司的一个超过 20D 人的机构研究表明采用评审检查方法后生产率可提高 14%，而错误可减少 90%。
- 喷气推进实验室估计，在早期发现和定位错误，每次检查可节省开支 25,000 美元。

以上结果有力地说明了软件质量的一般原则，即减少软件中的错误数也能缩短开发时间。

不同的研究表明，在除了发现错误方面较调试更为有效之外，评审较调试能发现各种不同类型的错误。另外一个影响是，当人们意识到其工作将接受评审时，他们往往会仔细检查自己的工作。所以，即使调试相当有效，评审对一个程序的集成性质量保证是必需的。

评审传活使用技巧和编程经验

软件标准可被记录和传播，但是如果没有人谈论它或鼓励别人去使用它的话，它们将不会被遵循。评审是对程序员的代码有所反馈的一种重要方法。代码、标准、代码和标准相一致的原因等都是评审所涉及的话题。

除了能得到遵循标准程序的反馈外，程序员还需要以编程方面获得更多的反馈信息：格式化、注释、变量名、局部和全局变量的使用、设计方法、处理本地事情的方法等等。没有经验的程序员需要从熟练的程序员处获得指导。有经验的程序员往往显得很忙，应鼓励他们抽出时间进行经验交流。评审给有经验或缺乏经验的程序员们提供了交流技术的途径。因此，无论在现在或将来，评审给提高质量提供了机会。

评审质量和进度

评审一般包括两种类型：管理的和技术的，管理评审用于评估进度和建议纠正之中。它们着眼于花费和计划。管理评审是重要的，但是不是本书讨论的主题。

技术评审也可用于评估进度，但是它要求技术进度是可评估的。这通常意味着以下二个问题：（1）技术工作是否正在进行之中？（2）技术工作是否做得较好？以上问答是技术评审的副产品。

在创建中应自始至终多用评审方法

本书是讨论创建的，所以对细节的设计和代码的评审是本章的主题。但是，本章对评审的大部分讨论同样适于管理、计划、需求分析、结构和维护的评审，读过本章后，你就可将评审

应用于软件开发的任何阶段。

24.2　检　查

检查是一种特殊类型的评审，它在错误检查中被证明是行之有效的，并且和测试相比，它是一种相对较为经济的办法。检查方法由 Michael　Fagan 所发明，并在 IBM 应用了几年，最终由 Fagan 将其出版发行。虽然任何评审方法都包括阅读设计资料或代码，检查和走马观花式的参观是不同的：

- 检查表将检查者的注意力集中在过去所遇到的问题的领域中
- 检查侧重于错误检查而不是纠错
- 评审者在检查之前应先举行预备会议，而他们最后得出的问题应是经过共同讨论的
- 所有的参加者被分配以不同的任务
- 检查协调者并不是被检查产品一方的人
- 协调者在协调检查方面受过一定的训练
- 每次检查所得数据都被收集起来，以便反馈给今后的检查以提高检查质量
- 一般管理人员并不参加检查会议，而技术主管则有可能参加会议

你能从检查中获何收益

一般来说，设计和代码检查的联合使用可去除产品中 60%到 90%的错误。检查可较早地发现有错误苗头的子程序，Fagan 指出检查比预查每 1000 行代码要少 30%的错误。设计者和编码者通过参与检查可以提高工作能力，同时检查也可提高生产率达 20%左右。在使用设计和编码检查的项目中，检查可花费项目总时间的 15%。

检查中的各项分工

检查的一个重要特点是每个人都各司其职。以下是各种分工：

协调者。协调者负责控制检查进度，使其既有一定的检查速度又能最大限度地发现错误。

协调者必须有一定的技术胜任能力，当然并不苛求他是受检查的设计或代码方面的专家。但是他应能了解相关细节。这类人员负责检查的一切方面如分配受检的设计、代码或检查表、确立会议室地点、报告检查结果、落实检查会议所确定的有关内容。

项目主持者。设计者或代码编写人员在检查中扮演着相对次要的角色。检查的部分目标是保证设计或代码能站得住脚，如在评审、检查过程中发现设计或代码并不清晰，主持者将被告之修改，以使它更为清楚。否则，主持者应解释不清晰部分的设计或代码，甚至，让他解释为什么看起来有错误的部分例实际上可以接受。如果检查者对项目本身并不熟悉，主持者应对整个项目作简单介绍以便为检查会议作准备。

检查者。检查者是任何对设计或代码有直接兴趣者，但他不是本项目的主持者。设计的检查者可能是对设计有帮助的程序员。检查者中也可能有测试者或高水平的结构人员。检查者的任务是发现缺陷。他们常在准备过程中发现错误，当检查会议讨论设计或代码时，整个组将会发现更大的错误。

记录员。记录员记录所发现的错误和检查会议上的情况。有时记录员也可由协调者或由另

外一些人兼任。但主持者或检查者不作为记录员的兼任。

管理者。软件检查并不纯粹是技术检查。管理者的出现改变了技术间的相互作用。人们常感到他们不是检查材料，而是需要评估的。这也会将问题的重心从技术上转移到政治上去。管理者有权知道检查的结果，检查报告也应让管理者知道。

同时，检查结果不应该用来作为对性能的评估。不要杀死正在下金蛋的天鹅。检查中的代码仍需不断完善。性能评估应建立在最终产品的基本之上，而不是基本未完成的工作之下。

总的说来，一次检查至少应有三个参加者。协调者、主持者、检查者，并且以上三角色不能各自兼任。一般将检查组规模控制在 6 人左右，因为再增加人数的话，就难以管理了。对喷气推进实验室的研究表明，三人以上的检查组并不能提高所发现的错误总数。

检查的一般过程

检查包括以下几个明显的阶段：

计划。主管者将设计或代码提交给协调者，协调者决定谁将参与检查，并确定何时何地召开检查会议，接着将设计、代码或引人注目的检查表分发给每一位检查员。

总览。当检查员对要受检查的项目不太熟悉时，主管项目者应花费一小时左右的时间介绍一下生成设计或代码的环境。但是这并不是件容易的事情，因为它往往给需检查的设计或代码造成不清晰的假象。设计或代码应能自己站得往脚。总览不应提及它。

准备。每一位检查者在正式工作之前，先花 90 分钟的时间熟悉设计或代码。检查者使用检查表以激励或指导对检查材料的检查工作。

为了对用高级语言编写的应用程序代码进行评估，检查员可先每小时阅读 700 行代码。而为了对用高级语言编写的系统程序代码进行评估，检查员每小时只能阅读 125 行代码。最有效的检查速度差别很大，所以你应记录下你所在组中的各种速度以便在你的环境中最有效地应用它们。

检查会议。协调者选择一些人——通常是项目主持者解释设计或代码的阅读。所有的逻辑都要解释，包括每个逻辑结构的分枝。在检查时，记录所发现的错误，如果检查确认一个错误时，往往会停止对错误的讨论。记录员记下这种类型的错误并标明其重要性，检查工作继续进行。

对设计或代码的检查不能太快也不能太慢。如果太慢了，易使人的注意力迟缓同时，效率也太低了。如果检查过快，就可以漏掉很多应发现的错误。最优检查速度随环境而异，正如准备速度一样。你应作好记录工作，以便随着时间的推移你能找出你所在环境的最佳速度。有些组织已经发现对系统码来说，最佳检查速度是每小时 90 行代码。对应用程序代码来说，最佳检查速度可提高到每小时 500 行代码。

在会议过程中不必讨论问题的解答，应着重于确定错误。有些检查组甚至不允许讨论一个错误是否为一个真正的错误。他们认为如果被错误的是非弄混淆了，相应的设计、代码和文档也同样需得到澄清。

会议时间通常不应超过两小时。这并不意味着你制造一个假的火灾警报以便让人们在二小时之内都出来。但是 IBM 和其它公司的经验表明，检查者并不能一次在超过 2 小时的时间内全神贯注地工作。同样，在同一天中计划多次检查也是不明智的。

检查报告。在一天中的检查会议后，协调者编制一份列出每个错误的类型和严重性的检查

报告。检查报告有助于纠正所有错误并且可得出对程序组有特别意义的错误表。如果你能坚持收集每次所花时间和所发现错误数的数据，你就可用这些硬数据确定检查员的工作效率。否则，你无法清楚地知道检查员是否工作得更好。你也能明白检查员是否适于在你的环境中工作从而决定是变动还是不聘用他们。数据收集工作是重要的，因为任何一种新方法都需要用某种标准衡量其优劣。

再工作。协调者将错误提交给某些人，通常是主管者，以供其修改。他们将清单上所列错误一一改正。

执行。协调者负责检查再工作的执行。如果超过 5%的设计或代码需再加工，整个检查过程应重新进行。

如果低于 5%，协调者仍然可以要求进行再检查或亲自证实再加工。

虽然在检查过程中参与者不能讨论所产生问题的解答，有些人仍想探讨有关问题。但是，不能不管大家喜欢不喜欢就随便作修改。将评审过程“规范化”以便你能知道你的修改是否有益。

检查过程的所有部分都是必需的。已经发现，去掉或合并其中任意几个部分都将耗去更多的代码。如果你想改变检查过程而没有一种可以度量改变的方法，那么你就别这样做。如果你可对整个过程进行度量，而且知道改变后检查过程工作得更好，那么你就应坚持下去。

在检查过程中，你将发现某些类型的错误较其它错误更经常发生。建立一个引起人们对这类错误注意的检查表，以便检查者能将重点放在这类错误上。随着时间的流逝，你将会发现一些检查表上所没有的错误，所以你可将其加进检查表中。你也可能发现初始检查表上出现的一些错误停止出现了，这时你可将其从检查表中去掉。在进行一些检查以后，你所在组就能得到一个符合需要的错误检查表，同时对故障区域有一个清晰的了解，这样才可能让程序员有针对性地进行训练和得到帮助。将你的检查表限制在一页之内，较长的检查表在所要求的细节水平上是很难使用的。

检查本身

检查本身是为了发现设计或代码中的错误。它不是为了寻找选择对象或者争辨问题的是非。当然也不应批评设计或代码的主持者。检查对项目主持者来说应是有积极意义的，它应明显表明全体参与者都可提高程序质量，并且对所有参与者来说他都能从中获得不少体验。另外，检查组不能向项目主持者说有一些人是笨蛋应让其卷起铺盖滚蛋。像“任何知道 Pascal 语言的人都知道从 Num 循环到 0 是更为有效”的此类评论是根本不合适的。如果真是这样，协调者应将这种情况明白无误地表达出来。

由于检查要对设计或代码作出评论，项目主持者可能感到自己和它们有某种牵连，主持者自然觉得自己对代码有一种亲密之情。主持者应能预测到检查组所发现的一些错误其实并非错误，而且有一些可能是有争议的问题。尽管那样，主持者应接受每个已指出的错误并继续下去。接受某一评论并不意味着主持者就认为它是正确的。项目主持者尤不应庇护受检查的工作。在检查之后，主持者可独立地考虑每个问题并决定其是否有效。

检查者应懂得项目主持者是最终负责怎样处理一个错误的人。喜欢发现问题是好的（有时除了检查之外，还提出解答方法），但是每一位检查者都应尊重主持者最后决定怎样处理这些错误的权力。

检查表

有效检查法

- 你的检查表是否着重将检查员的注意力引向过去常发生错误的地方？
- 是否侧重于缺陷检查而不是纠错？
- 在检查会议之前检查员是否有足够的准备时间？每一位检查员都作好了准备吗？
- 每一位参与者是否都扮演不同的角色？
- 会议是否开得富有成果？
- 会议是否限制在 2 小时之内？
- 协调者在指导检查方面接受过特殊的训练吗？
- 在每次检查中，错误类型数据是否都作了收集，以便于你今后制作检查表？
- 是否收集了准备和检查率，以便可以优化将来的准备和检查？
- 每次检查所指定的条款是否都落实了？是由协调员本人还是重新作了检查？
- 管理员是否明白为什么他不参加检查会议？

检查概述

检查表可使检查员注意力集中在某一类错误上。由于检查过程有标准的检查表和标准的各司其职的人员，它是一个有组织的过程。它也是一个自我优化过程，因为它使用有正规的反馈循环以提高检查表质量、监控准备和检查速度。有了以上对过程的控制和优化，不管它怎样开始的，检查很快就成为一种强有力的方法。软件工程学会（SEI）已经定义了用于度量软件开发过程效率的技术标准。检查过程表明了何为最高水平。同时，检查过程应是有组织的、可重复的，并能使用可度量反馈以提高检查质量。你同样可将本方法实际应用于本书所讨论过的任何技术中。在开发过程中将这些思想归纳起来，那么简单地说，它们可使你所在机构向着最高质量和生产率的层次进军。

24.3　其它评审方法

其它评审方法不像检查方法那样得到实际经验的有力支持，所以目前它们所涉及的范围不广。本节所讨论的评审方法有：普查、代码阅读、软件演示。

普查

普查是一种流行的评审方法。普查这个词定义并不严密，因为人们实际上可称任何类型的评审方法为“普查”。

由于普查定义不严密，所以也就很难对其给出确切的定义。当然，普查包括二人或多人讨论设计或代码这种情况。普查可能就像即席的随意探讨会一样不正式。有时它也可能像一个预定的会议或送给管理者的总结报告一样正规。在某种意义上讲，“什么地方有两或三个人聚在一起”，什么地方就存在普查。普查方法的支持者赞成使用这样一个宽松的定义。所以本文只打算找出普查的一些共同之处，剩余的细节留给你自己去处理。

普查通常由接受评审的设计或代码的主持者所采用。

- 普查的目的是为了提高程序的技术质量，而不是评估程序。
- 所有参与者通过阅读设计或代码为普查做准备并寻找错误。
- 普查可为老资格程序员向新手提供传播经验和合作精神的机会，它也为年轻的程序员提出新方法，并向一些过时的结论挑战提供机会。
- 普查通常需花费 30 到 60 分钟的时间。
- 普查侧重于发现错误，而不是纠正错误。
- 管理人员并不参加普查。
- 普查这个概念是灵活的，它也适应于特定的场合。

你能通过普查获何收益

如果用得恰当，普查可得到和其它评审方法类似的结果，就是说，它一般能发现程序中 30%到 70%的错误（Myers 1979,Boehm 1987，Yourdon 1989）。普查的检错效率比评审稍低一些，但是在一些场合，普查还是相当有效的。

如使用不得当，普查可能会造成不少麻烦。普查的最低效率为 30%，这并无多大价值。至少一个组织已经发现对代码的扫视检查将是“异常不合算”的（Boeing Computer Services）。Boeing 发现，说服项目人员自始至终应用普查方法是困难的，而且当压力增大时，应用普查方法几乎是不可能的（Glass 1982）。

检查和普查的对比

检查和普查有何差别？以下为二者差别摘要。

性质	检查	普查
正规协调者培训	是	否
参与者分工明确	是	否
谁“主持”检查或普查	协调者	通常为作者
“怎样发现错误”——检查表	是	否
集中评审量——寻找最常见类型的错误	是	否
正规执行以减少不正确的定位	是	否
由于对程序员的详细错误反馈所导致的较少后期错误	是	影响不大
对结果的分析导致检查效率的提高	是	否
对过程中导致发现错误的数据的分析反过来也导致检查效率的提高	是	否

在排除错误方面，检查比普查显得更为有效。但是为什么有人爱用普查呢？

如果你拥有一个较大的评审组，普查不失为一种较好的评审方法，因为它给接受评审的程序带来多种不同的见解。如果所有参加普查的人都相信解答是正确的，就不会有大的缺陷。

如果有来自其它组织中的评审员，普查也可能是可行的。在检查中，每个人的职责分工是明确的，在人们有效地运用它们之前需要一个实践过程，让以前未曾参加过检查的人当评审员是不利的。如果你想让他们出一份力，普查可能是最好的选择。

检查比普查更有所侧重也能获得更好的收获。如果你想为所在组织选择一个评审标准,在作

出选择之前好好考虑一下。

代码阅读

代码阅读是对检查和普查的另一种选择。在代码阅读时，你能读源代码并寻找错误。你也同时对代码的质量如其设计、风格、可读性、可维护性和有效性提出你的见解。

对美国宇航局软件工程实验室的一项研究表明，代码阅读平均每小时可发现 3.3 个错误。调试平均每小时可发现 1.8 个错误（Card 1987）。代码阅读在软件生存期中要比其它调试方式多发现 20%到 60%的错误。

同普查一样,代码阅读的定义并不严格。代码阅读通常是二人或多人各自阅读代码然后和代码的开发者一起讨论。以下是阅读的方法：

- 在开会之前，代码开发者将源代码分发给代码阅读者。代码长度通常从 1000 到 10000 行不等。典型的是 4000 行。
- 一个或多个人共同阅读代码。最少应有 2 个人，以便激励在评审者间的竞争。如果你使用的人数多于 2 个，你应度量每个人的贡献，以便了解多余的几个人究竟有多大的贡献。
- 评审者独立地阅读代码。其速度大约是每天 1000 行。
- 当评审者完成代码阅读后，代码开发者主持召开代码阅读讨论会。会议持续时间为 1 个或 2 个小时，其侧重点是代码阅读者所发现的问题。通常人们并不是一行一行通读代码。本会议不是必需的。
- 代码开发者确定由评审者所发现的错误。

代码阅读和检查、普查的区别在于代码阅读侧重于个人对代码的评审,而不是着重于会议上的讨论。其结果是,评审者的时间大都花费在寻找代码的问题了。这样，花费在开会上的时间将减少因为每人只需花费少部分时间在这上面。因为会议所花时间对一个中等组来说是可观的。除非各组员能在二小时内碰头,开会耽搁的时间也少。代码阅读在当评审员不在一起时是相当有价值的。

软件演示

软件演示是一种将软件产品向用户展示的方法。在与政府所签合同的软件开发过程中,用户评审是常见的,因为这在需求、设计和代码方面对评审有所要求。软件演示的目的是向用户表明软件质量是好的，所以软件演示是管理评审而不是技术评审。

你不要指望依靠软件演示来提高产品的技术质量。准备软件演示可能对技术质量有间接影响，因为大部分时间都花费在使软件用户界面较好，而不是用在提高软件技术质量上面。

24.4　小　结

- 总的来说，评审在发现错误方面较测试要好。
- 评审侧重于错误发现而不是纠错。
- 评审往往比测试要能发现更多种不同的错误，意味着你应使用评审或调试方法以确保软件的质量。

- 使用检查表、准备良好的分工、连续的处理以便最大限制地提高检错能力，检查往往较普查能发现更多的错误。
- 普查和代码阅读是检查的候补方法。代码阅读可有效地利用每个人的时间。

第二十五章　单元测试

目录

相关章节

软件质量概述：见第 23 章
评审：见第 24 章
集成；见第 27 章
调试：见第 26 章
创建前提：见第 3 章

测试是最为流行的提高质量的方法——它受到工业研究和大学研究的有力支持，而且也得到商业研究的支持。在本章中，测试是指单元测试——对单个子程序和模块而不是对整个系统的测试。“测试”在本章中也可指“玻璃盒”测试,你可用它来测试你自己的代码——而功能测试是一独立测试者用基本的功能描述来检查一个程序。

如果一个系统足够小，你可应用本章的技术来测试整个系统。如果系统稍大一点，它可能由一个专门的测试组织来测试，而你所需作的是，在其合并成为一个总系统之前对每个子程序进行单元测试。

有些程序员将“测试”和“调试”混用，但是细心的程序员是区分这两种活动的。测试是发现错误的方法，而调试是对错误进行诊断并改正它们。本章仅讨论错误检查。错误纠正将在第 26 章中详细讨论。

25.1　单元测试在软件质量中的作用

测试对任何一个软件质量来说是重要的,在许多场合，它只是其中一部分。这是不幸的，因为评审的各种方式证明，它们比测试要能发现更多的错误，而且评审发现每个错误所花费仅为测试的一半左右(Card 1987)。单个测试方法(单元测试、功能测试、部分测试、系统测试)，通常只能发现少于 50％的错误数。几种测试方法的集成使用，只会发现少于 60％的错误（Jones 1986）。然而,由于测试已被广泛使用，而且它比评审要能发现更多种类型的错误，因此它应该

是软件开发周期的一部分。

如果你列出一些软件开发活动,并问“哪件事情和其它事情不同?”回答将是“测试”。由于以下原因，测试对许多开发者来说都是一项较难掌握的活动：

- 测试的目的是同其它开发活动的目的相矛盾的。测试是为了发现错误。一次成功的测试将中断软件的开发。而其它开发活动是为了防止错误和防止软件的终止。
- 测试结果不能证实错误的多少，只能证实其存在。如果你进行了广泛的测试并发现了成千上万的错误，并不意味着你已经发现了所有的错误。未发现错误，也并不意味着软件是完美无缺的。
- 仅用测试并不能提高软件质量。测试结果可以说明软件质量的好坏，但是并不能提高它。想靠增大测试量来提高软件质量，就如你想靠不断给自己称重而达到减肥的目的一样。在你称重前你所吃的东西将决定你到底有多重，你所用的软件开发技术决定了你调试所发现的错误数。你如想减肥，不必再买一台新磅秤。你只需改变你的饮食习惯。如果你想提高软件质量，不必进行大量测试；你应认真在开发过程中下功夫。
- 测试需要你假定你能在代码中找到错误，如果你认为不会找到错误，你倒可能真的找不到，你应对此有一个粗略计划。如果你执行一个程序，并希望找不到错误，你就可能忽视许多你应发现的错误。Glend Myers 让一群有经验的程序员对一个有 15 个错误的程序进行测试。程序员所发现的平均错误个数仅为 5 个。最好的是仅发现了 9 个错误。其它错误未被发现的原因是由于没有仔细检查错误的输出结果。其实这些错误是可见的，但是程序员并没有注意到它们。

你应期望在代码中能发现错误。存在这样的想法可能被视为是反常的，但是指望发现错误的正是你而不是他人。

一个重要问题是，对一个典型项目进行单元测试究竟要花费多少时间？普遍的数据是测试占项目时间的 50%，但是由于以下原因它并不是正确的。首先，以上数据包括了调试和测试，测试本身所花时间会少些；其次，以上数据是典型所花时间量而不是应花费的时间量。最后,以上数据既包括了系统测试也包括了单元测试。

正如图 25—1 所示那样，依项目的大小和复杂性不同，单元测试所占总的项目时间从 8%到 35%不等。这和已经报道的许多数据是相一致的。

其次，你怎样处理单元测试的结果呢？通常，你可借此评估所开发产品的可靠性。即使你无法改正测试所发现的错误，你也可以知道软件的可靠程度，测试结果的另一个用途就是可以利用它们引导对软件的修改。最后，经过一段时间的积累，你就可通过测试中所记录错误发现最常见的错误类型。你可以利用这些数据选择合适的训练课程，指导今后的技术评审活动以及今后的测试。

创建中的测试

正如本章前言所指出的那样，整个测试的范围要比创建过程的测试范围广泛得多。本书并不讨论广义的测试：系统测试、功能测试、黑盒子测试等等。

以上测试方法有时忽略了本章的“玻璃盒”或“白盒子”测试方法。你通常愿意将子程序设计成黑盒子形状——子程序的用户只需了解接口要求，而不必知道子程序的内部情况。在测试子程序时，应将其视为玻璃盒子，既看子程序的内部源代码,也检查其输入和输出，这样作是

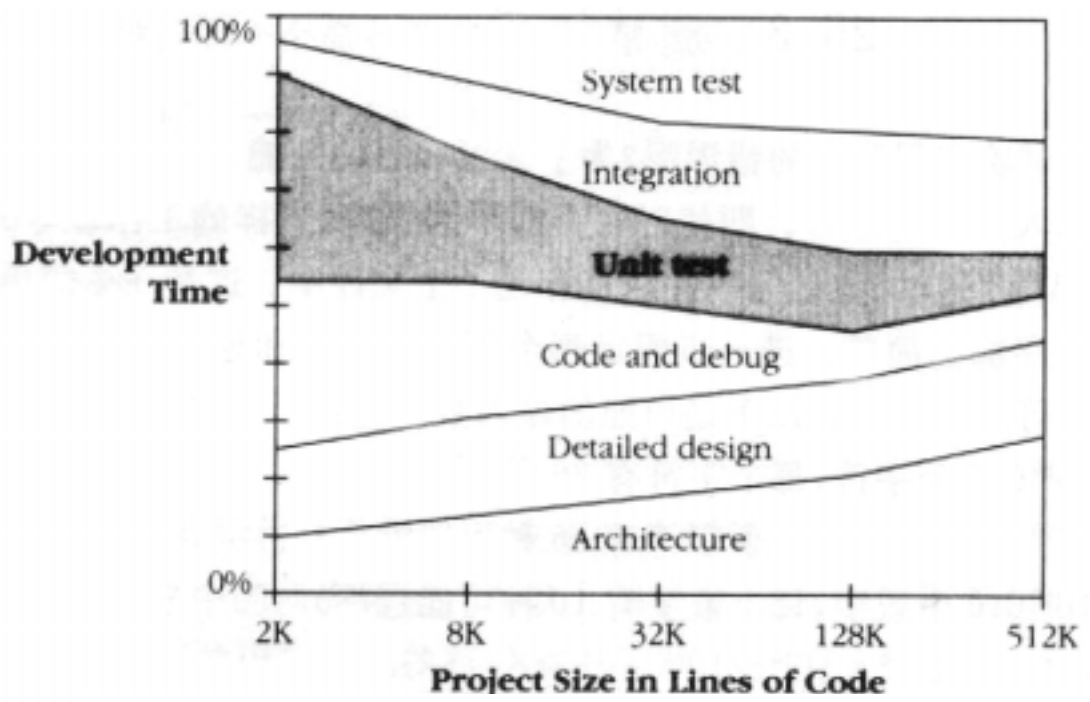


图 25－1　随着项目大小（代码行数）的增加，单元测试占整个开发时间的百分比减小

较好的。当然如果了解盒子的内部情况，你就可更仔细地测试你的子程序，至少你有更多的机会发现错误。和不熟悉代码者相比较，你处在发现错误的更有利位置上。

在创建过程中，你编写子程序，对其进行手工检查，然后评审或测试它。不论你的集成或系统测试策略如何，在你将其合并之前，你应仔细地测试每个单元。如果你在编写几个子程序，你应不时测试一下它们。单个测试子程序并不容易，但是要调试它们是容易的。如果你马上将未测试的子程序组合起来并且发现了错误，那么其中任何一个子程序都有可能有错误。如果你每次将一个子程序加进已经测试过的子程序中，你就能知道任何一个新错误可能是由新的子程序或者是由新、旧子程序的相互作用所引起的。这时，使用调试方法较为简单。

25.2　单元测试的一般方法

一个有条理的单元测试方法，可使你用最小的努力最大限度地发现各种类型的错误。请记住以下各点：

- 对每个需求进行测试，以便确保需求得到实现。在需求阶段上计划测试或尽量使测试早一些——在你编写单元测试前。你应考虑测试对需求的遗漏。安全性、存储、安装程序、系统可靠性都是测试的对象，并且在需求分析时它们都易被疏忽。
- 对和设计有关的程序进行测试以确保设计得到了实现。在设计阶段尽早计划测试——在你开始进行子程序的详细编码工作之前测试。
- 在详细测试的基础上对需求和设计测试增加基本测试。使用数据流测试和其它测试方法仔细检查你的代码。从最低限度来说，你应对每行代码进行测试。下节将讨论基本测试和数据流测试。

在产品生成过程中应编制测试用例。这样就能避免需求和设计中的错误，而改正这类错误所费代码要比代码错误大。尽早计划测试和发现错误以便能合理地改正错误。

25.3　测试技巧

为什么有可能利用测试确定程序中的错误呢？为了测试程序的性能，你不得不对你的程序测试每一种输入数据或输入数据的组合。即使对一个简单的程序，这样的工作也令人难以容忍。例如，你的程序接收人名、地址、电话号码并将其存在一个文件中。这是一个简单的程序，并且比任何让你厌烦的程序都要简单。进一步假定每个可能的名字和地址是 20 个字符的长度，并且它们要用到 26 个可能的字符。以下是可能的输入数据量：

名字　　26^{20}（20 个字符，每个字符有 26 种可能选择）
地址　　26^{20}（20 个字符，每个字符有 26 种可能选择）
电话号码　　10^{10}（10 个数字，每个数字有 10 种可能选择）
总的可能　　$=26^{20}*26^{20}*10^{10}=10^{66}$

即使是这样较小输入量，你也将有 10^{66} 种测试用例。如果诺亚走出其方舟并以每秒 1 兆次的速度对程序进行测试，他即使到现在也才完成了全部工作量的 l%。显然，如果你增大实际数据量，对全部可能性进行测试几乎是不可能的。

不完全测试

对各种可能全部进行测试是不可能的，实际说来，测试的艺术在于从所有测试用例中找出最能发现错误的示例来。在 10 种可能测试示例中，只有少部分可能发现错误。你应侧重于从所有测试示例中找出能提示不同点的用例，而不是那些不断地重复着的用例。

当你计划测试时，你应排除那些没有告诉你任何新东西的用例，这就是说，此时对新数据的测试将不会产生错误。人们已提出了不少有效的非实例测试法，下文将讨论其中的一些方法。

善于结构的测试

尽管其名字是粗糙的，基于结构的测试，其实是一个简单概念。其意思是你应对你程序中的每一条语句至少测试一次，如果本语句是一条逻辑语句，通常是用 if 或 while，你就应根据 if 或 while 表达式中的复杂性仔细测试，这样才能确保每条语句都经过了测试。确保所有语句都经过测试的最简单的方法是由程序计算路径数，然后设计最少数量的测试用例，以确保所有路径都得到了测试。你可能听说过“代码覆盖”测试或“逻辑覆盖”测试；它们都是使你的测试程序中所有路径的方法。由于这两种方法覆盖了所有路径，它们和基于结构的测试是相似的，但是这二种方法覆盖所有路径时并不使用最小的测试用例集。如果你使用代码覆盖或逻辑覆盖方法，你所需的测试用例可能比你用基于结构的测试所需的测试用例要多。

你可用表 25－1 所给出的方法计算所需的最少测试用例数。

表 25－1　确定基于结构的测试方法所需的测试用例

1. 对于程序的第一条直接路径开始，设定计数值为 1。
2. 每遇到下一个关键词或其等价词： if，while，repeat，for，and 和 or 计数值加 1。
3. 在 case 语句中每遇到一个 case，if 数值加 1。如果 case 语句中不含缺省语句，计数值再加 1。

以下是一个例子：

计算路径数目的一个简单的 Pascal 程序例子：

```
Statement1；　　－－开始计数 1
Statement2；
if X＜ 10 then　　－－遇到 if 计数 2
    begin
    Statement3；
    End；1
Statement4；
```

本例中，你一开始将计数置为 1，在遇到 if 语句后计数值变为 2，这意味着你至少需要 2 个用例以便覆盖程序中的所有路径。在以上例子中，你应有以下示例：

- 受 if 控制的语句得到执行（X＜10）
- 不受 if 控制的语句得到执行：（X＞＝10）

代码例子还应更实际一点，以便对测试工作有一个清晰的了解。例子中实际还包括有缺陷的代码。

下面的程序稍复杂一点。在本章以后都使用这个例子并且它可能有一些错误。

确定基于结构的测试所需测试用例的一个 Pascal 例子；

```
1   { Compute Net Pay}　　程序开始，计数 1
2
3   TtlWithholdings: ＝0；
4
5   for  ID: ＝1 to NumEInPloyees　　——for，计数 2
6   begin
7
8   { compute social security withholding, if below the maximum
9   if（Employee［ID］.SSWithheld＜MAX_SOCIAL_SECURITY ＝then　——if，计数 3
10      begin
11      SocialSecurity: ＝ComputeSocialSecurity（Employee［ID］）
12      end；
13
14   ｛set default to no retirement contribution｝
15   Retirement: ＝0；
16
17   {determine discretionary employee retirement contribution}
18   if（Employee［ID］.WantsRetirement）and　——if，计数 4 and，计数 5
19      (EligibleFotRetirement( Employee[ID]))  then
20      begin
21      Retirement: ＝GetRetirement（Employee[ID]）；
22      end；
23
24   GrossPay: ＝ComputeGrossPay（Employee[ID]）；
```

```
25
26   { determine IRA contribution }
27   IRA ：＝0;
28   if（ EligibleForIRA（ EmPloyee［ID］)） then  ——if，计数 6
29       begin
30       IRA：＝IRAContribution（Employee［ID］，Retirement，GrossPay）
31       end;
32
33    { make weekly paycheck}
34   Withholding：＝ComputeWithholding（Employee[ID]）;
35   NetPay     ：＝GrossPay－Withholding－Retirement－
36                    SocialSecurity－IRA;
37   PayEmployee（EmPloyee［ID］，NetPay）;
38
39   { add this employee's paycheck to total for accounting}
40   TtlWithholdings      ：＝ TtlWithholdingst ＋Withholding;
41   TtlSocialbourity     ：＝ TtlSocialSecurity ＋SocialSecurity;
42   TtlRetirement        ：＝TtlRetirement     ＋ Retirement;
43   end;   {for}
44
45   SavePayRecords（ Ttlwithholdings， Ttlsocialsecurity， TtlRetirement）;
```

本例中，你需要有一个初始计数值，每遇到 5 个关键词的一个，计数值加 1。但是并不意味着任意 6 个测试用例就覆盖所有示例。它只是表明最少需要 6 个用例。除非这些用例构造得相当好，否则将不会覆盖所有情况。关键是当你计算所需用例的数目时，你应注意你所用的相同的关键词。代码中每个关键词代表了或真或假的一类事物。你能确信对每个真或假应至少有一个测试用例与其一一对应。

以下是覆盖了上例中所有基数的测试用例：

用例	测试描述	测试数据
1.	无用例	所有布尔值为真
2.	初始 for 条件为假	NumEmployees<1
3.	第一个 if 为假	Employee［ID］SSWithheld>＝ MAX_SOCIAL_SECURITY
4.	第二个 if 为假（因为 and 的第一部分为假）	not Employee[ID]. WantsRetirement
5.	第二个 if 为假（因为 and 的第二部分为假）	not EligilbleForRetirement（Employee[ID]）
6.	第三个 if 为假	not ElgibleForIRA（Employee[ID]）

如果你的子程序比以上所讨论程序还要复杂，你所用测试用例以便覆盖所有路径的数目将会大大增加。短的子程序有较小的测试路径。没有较多 and 和 or 的布尔表达式需测试的变量数也较少。测试的容易也正说明了子程序较短并且你的布尔表达式较为简单。

既然已为你的子程序创建了 6 个测试用例，并且满足了基于结构的测试需求，你是否以为

你的子程序已经测试完了？不是，这种类型的测试只能保证代码是可执行的。它并不能解释数据间的差别。

数据流测试

本章最后一节和本节可使你明白在计算程序中控制流和数据流是何等重要的。

数据流测试是基于数据使用至少同控制流一样易受错误影响这个事实。Boris Beizer 断言现代程序中至少含有一半的数据说明和初始化（1990）。

数据的存在有以下三种状态：

已定义数据。数据已经初始化，但是还没有被使用。

已使用数据。数据已经作为子程序或其它程序的变量用于计算了。

已无效的数据。数据曾一度定义过，但是在某种程序上已失去定义。例如，如果数据是一个指针，可能它已经被释放了。如果数据是一个 for 循环指针，可能程序已经跳出了循环，而程序对超出 for 循环的指针没有定义。如果数据是一个文件指针而当文件关闭后，记录指针就无效了。

除了“定义”、“使用”、“失效”这些词外，使用一些描述立即或退出子程序的词是方便的：

进入。控制流在对变元处理前立即进入子程序。

退出。控制流在对变元处理后立即退出子程序。

数据状态的组合。数据组合的一般状态是某变量已被定义使用了一次或多次，可能失效，请看以下方式：

定义——定义。如果某变量在使用前已被定义过 2 次。那么你需要的不是一个好程序，而是一台好计算机。这样浪费较大且易出错，即使实际上没有错误。

定义——退出。如果变量是一个局部变量，没有必要知道是否定义它，也可能没有使用本变量就退出子程序。如果是子程序常量或局部变量的话，以上行为没有任何影响。

定义——失效。如果你定义了一个变量但是没有使用它的话，这是一种浪费。你所用变量多了，可能你需要使程序更小。

进入——失效。这是对局部变量而言的，如果你没有定义或使用一个变量，你用不着使其失效。另一方面，如果是一子程序常量或局部变量，只要在使其失效之前对其定义，本模式也同样有效。

进人——使用。本模式也同样是对局部变量而言的。变量在使用之前需定义。另一方面，如果是一子程序常量或局部变量，只要在某使用之前进行定义，本数据模式也同样有效。

失效——失效。变量不需失效二次，变量也无需恢复，除非你需要一台新的计算机。否则，恢复变量的使用意味着编程的马虎，双重失效对指针来说是重要的——挂起你机器的最好方法是释放一指针两次。

失效——使用。使用一个已失效的变量有时也有作用，但是这总是一个逻辑错误。如果代码看起来也能工作，但这只是偶然，在某个时候，它会产生故障引起更大的损害。

使用——定义。使用和随之定义一个变量是否带来问题取决于变量在使用前是否已定义。当你见到一个使用一定义型数据时，你应检查以前的定义。

在开始测试之前检查反常的数据状态序列。在检查所有反常的数据序列后，编写数据流测试的用例关键是检查所有可能的定义——使用路径。你检查的广度可随情况而异，它包括：

- 所有定义。对每个变量的所有定义进行测试（这就是说，任何一个变量每接收一个数据时都应对其测试）。这是不好的策略，因为如果你想测试代码的每一行的话，你可能会失败。
- 所有定义——使用混合数据状态。测试每一个某处定义然后在另一处使用的变量。这较测试所有定义是一种较好的策略，因为它仅执行每一行代码，而不确保每一定义——使用类型的数据将接受测试。以下是一个例子：

将接受数据流测试的一个程序例子：

```
if (Condition 1)
   x = a;
else
   x = b;

if (Condition 2)
   y = x + 1;
else
   y = x – 1;
```

为了覆盖程序中的所有路径，你需要使条件 1 是真或假的测试用例。你也需要使条件 2 为真或假的测试用例。以上可以用二个测试用例来处理：用例 1(条件 1 为真，条件 2 为真)和用例 2(条件 1 为假而条件 2 为假)。以上二个用例是你用基于结构的测试方法所必需的。它们也是你执行定义变量的每一行代码所必需的；它们可自动地进行简单的数据流测试。

为了覆盖每一种定义一使用类型，你需增加一些测试用例。你应还有使条件 1 和条件 2 同为真或假的测试用例：

```
x = a;
……
y = x + 1;
and
x = b;
y = x – 1;
```

但是你还是需要更多的用例以测试定义——使用混合类型的数据。你需要: (1)x=a 然后 y=x-1; (2)x=b 然后 y=x+1。在本例中，你可增加 2 个用例而得到以上组合：用例 3(条件 1 为真，条件 2 为假)和用例 4(条件 1 为假条件 2 为真)。

开发测试用例的较好方法是从基于结构化的测试开始，它可向你提供一些定义使用数据流。然后增加一些测试用例以便得到完整的定义使用数据流测试用例。

正如上一节所讨论的那样，基于结构的测试给第 25.3 节中的一个 45 行的 Pascal 语言程序提供了 6 个测试用例。对每个定义一使用类型的数据流进行测试需要有更多的用例。可能其中一些被一些已存在的测试用例所覆盖。以下是在基于结构的测试所产生的测试用例上所增加的

数据流混合类型用例。

用例	测试描述
7	在第 15 行中定义在第 30 行第一次使用，且没有被以前其它测试用例覆盖。
8	在第 15 行中定义在第 35 行第一次使用，且没有被以前其它测试用例所覆盖。
9	在第 21 行中定义在第 35 行第一次使用，且没有被以前其它测试用例所覆盖。

当你做过几次数据流测试用例后，你就能明白哪些测试用例是颇有成效的，哪些是已被覆盖了的。当你测试受阻时，你可列出所有的定义一使用混合类型数据。虽然看起来可能费事，但是，它可使你明白一些用基于结构测试的方法所不能得到的用例。

等效类划分

一个好的测试用例应覆盖相当大一部分可能的数据输入。如果二个测试用例提示出相同的错误，你可挑选其中的任一个。“等效类划分”的概念是以上思想的体系化并可减少所需的测试用例数。

本书第 25.3 节的 45 行 Pascal 程序中，第 9 行是使用等效类划分的好地方。测试条件是 Employee[ID].SSWithheld < MAX_SOCIAL_SECURITY。此时测试用例有二类：第一类是 Employee[ID].SSWithheld 比 MAX_SOCIAL_SECURITY 小，第二类是 Employee[ID].SSWithheld 大于或等于 MAX_SOCIAL_SECURITY。程序的其它部分可能有其它的等效类，这意味着你可能将测试比 Employee[ID].SSWithheld 二个值更多的用例，但是对于我们所讨论的这个程序来说，只需讨论 2 个即可。

当你已经对程序进行了基本和数据流测试时，再进行等效类划分将不会使你对程序有深入的认识。当你从子程序的外部看它（如从描述而不是从源代码时），或数据较为复杂而这种复杂并没有在程序的逻辑结构中有所反应时，等效类划分是异常有用的。

错误猜测

除了使用正规的测试技术外，好的程序员常使用一些非正规的、直接推断方法以提示代码中的错误。对应用编程来说，测试方法的不同往往导致测试结果的不同。例外情况是实时处理，但它不在本文的讨论范围内。

其中的一个直接推断方法是错误猜测。“错误猜测”这个词是对一个合理的概念所取的平庸的名字。错误猜测意味着测试用例建筑在对程序可能发生错误处的猜测上，错误猜测是需要一定经验的。

你对错误的猜测是建筑在直觉或过去的经验上。第 24 章指出检查的一个优点是它们能指出和列出常见错误表。错误表可用来检查新的代码。当你将过去所遇到的错误记录下来，你就有可能增大你的错误猜测所发现错误的可能性。

以下几部分讨论了几种可进行错误猜测的错误类型。

边界分析

测试的一个最有收效的领域是边界条件仅发生微妙的错误，如将 Num – 1 值认为是 Num 的值。或将 >= 误为 > 。

边界分析就是写出测试边界条件的用例来。如果你正在调试小于 MAX 范围内的数，你有以下三种可能条件，显示如下：

正如上图所示，有三种可能的边界用例：小于最大值，最大值自身，以及大于最大值。一般需要以上三个测试用例，以便确保没有发生常见的错误。

在 25.3 节中所示 45 行的 Pascal 程序包含着这样一个测试：Employee[ID].SSWithheld > MAX_SOCIAL_SECURITY，根据边界分析的原则，需检查三个用例：

用例	测试描述
1	定义用例 1 是使 Employee[ID].SSWithheld＜MAX__SOCIAL_SECURITY 这个条件为真时边界条件为真。于是，用例 1 使 Employee[ID].SSWithheld 为 MAX_SOCIAL_SECURITY – 1 值。本测试用例已经产生。
3	定义用例 3 是使布尔条件 Employee[ID].SSWithheld＜MAX_SOCIAL_SECURITY 为假时，边界条件为假。于是. 用例了使 Employee[ID].SSWithheld 之值为 MAX_SOCIAL_SECURITY＋l 本测试用例也已经产生。
10	另外一个测试事件是测试 Employee[ID].SSWithheld=MAX_SOCLAL_SECURITY 这是个死循环事件。

复合边界

边界条件分析也同样存在最少和最大容许值。在本例中，它可能是最小或最大总开支、总收入或总贡献，但是由于对这些值的计算已超出子程序的范围，对它们的测试用例在此不作深入探讨。

当边界含有几个变量是，一种更微妙的边界条件将从中产生。例如，二个变量相乘，当 2 个数都是大正数时将会出现什么情况呢？大的负数呢？0 呢？如果传递给一个子程序的字符串都是非同一般的呢？在第 25.3 节的 45 行 Pascal 程序，当每个雇员的工资数相当多时——如一帮年薪为 250，000 美元的程序员（我们总是这样希望的），此时你会想到变量 TtlWitholdings，Ttlsocialsecurity 和 Ttlretirement 到底有多大吗？以上是需要另外一个测试用例的：

用例	测试描述
11	现有一大帮雇员，每个人的薪水相当高——由于某种原因，1000 位雇员每人年薪为 250,000 美元，他们不需交任何社会保险机井区所有人都扣除退休保险金。

以下是一个测试用例，但是和以上用例相反雇员人数少而且每个人的薪水为 0.00 美元

用例	测试描述
12	10 位雇员，每人年薪为 0.00 美元

坏数据的排序

除了猜测边界条件时所发生的错误外，你也可猜测和测试几种其它类型的坏数据。典型的坏数据测试用例包括：

- 太小的数据（或无数据）
- 太大的数据
- 错误类型的数据（无效数据）
- 错误长度的数据
- 未赋初值的数据

如果你听从已经提出的建议，你就能想到一些测试用例。例如，用例 2 或用例 12 包括了“太小的数据”，你提出一些含有“错误长度”的数据的用例是困难的。下面提出了一些用例：

用例	测试描述
13	一共有 32,768 位雇员．要测试的数据是很大的．当然，数据到底有多大要因系统而异．但是，从本例中你可看到这个数据是相当大的．
14	薪水为负值，是错误类型的数据．
15	雇员人数为负数。是错误类型的数据。

好数据排序

当你试图找出程序中的错误时，往往忽视微小的用例中也往往包含错误。通常，基本测试段中的微小用例常是一种较好类型的数据。以下是值得你去检查的较好类型的数据。

- 微小事件
- 最小正常配置
- 最大正常配置
- 和其它旧数据的兼容性

对以上各种类型的数据进行检查能发现各种错误，这取决于测试对象。

最小正常配置不仅可用于一种测试，还可用于多种测试。它和边界条件的许多最小数值在本质上是相似的，但是最小正常配置是建立最小数值集合而不是正常期望值集。一个例子是在测试数据表时，需存储一张空数据表。为了测试字处理程序，你需要存储一个空文件。在对例子的运用过程中，测试最小正常配置需增加以下测试用例：

用例	测试描述
16	一位雇员．测试最小正常配置

而最大正常配置和最小配置相反。它和边界测试本质上相似，但是，它是建立最大数值集而不是正常期望数值。其中的例子是存储数据表。或者打印最大尺寸数据表格。对一个字处理器来说，它应能保存一份最大建议长度的文件。在对程序的测试过程，对最大正常配置的测试取决于最大正常雇员人数。如假定最大人数为 500，你应增加以下测试用例：

用例	测试描述
17	最大人数为 500，测试最大正常配置

最后一类正常数据测试，是测试和旧数据的兼容性。其使用场合是当用一个程序或子程序代换一个过时的程序或子程序时，新的子程序应能和旧程序一样利用这些旧数据产生相同的结果，当旧的子程序存在错误时，此种版本的连续性是回归测试的基础，其目的是为了确保修正后能保持原来的质量而不至于后退。在运行程序时，兼容性标准不应增加测试用例。

检查表

测试用例

- 每个子程序的要求是否有自己的测试用例？
- 子程序结构的每个部分是否都有自己的测试用例？
- 程序中每一行代码都是否至少被一个测试用例所测试过？这是否是由通过计算测试每一行代码所需的最少用例来确定的？
- 所有定义——使用数据流路径是否被至少一个测试用例所测试过？
- 代码是否被看起来不大正确的数据流模式所检查过？比如定义一定义，定义一退出，和定义一失效？
- 是否使用常见错误表以便编写测试用例来发现过去常出现的错误？
- 是否所有的简单边界都得到了测试：最大、最小或易混淆边界？
- 是否所有复合边界都得到了测试？
- 是否对各种错误类型的数据都进行了测试？
- 是否所有典型的中间数都得到了测试？
- 是否对最小正常配置进行了测试？
- 是否对最大正常配置进行了测试？
- 是否测试了和旧数据的兼容性？是否所有保留下来的硬件、操作系统旧的版本，以及其它软件旧版本的接口都得到了测试？
- 测试用例是否便于手工检查？

使用便于手工检查的用例

假定你为某一工资系统编写测试用例，你需输入某人的薪水，其方法是你可随意敲进几个数，试敲入：

1239078382346

这是一个相当高的薪水，比 1 万亿美元还要多，你可截短一部分得到一个更为实际的数字：39,078.38 美元。

现在，进一步假定本测试用例成功了，就是说，发现了错误。你怎样知道这是一个错误？现在，再假定你知道答案，因为你用手工计算出了正确的答案。当你用一个奇怪的数字如 34,078.38 美元进行计算时，你在得到程序结果的同时，自己却将结果计算错了。另一方面，一个较好的数字如 20,000 美元你一眼就能记住它。0 非常容易送入计算机，对 2 相乘是绝大多数程序员轻而易举就能完成的。

你可能认为一些令人厌烦的数据如 39,078.73 美元可能更易提示出错误，但是它确实是不如其它数有效。

25.4　典型错误

本书可使你明白当对错误有深刻了解时，你能测试得很好。

哪一个程序含最多的错误

你可能很自然地想到错误均分在你的整个源代码中。如果你平均每 1000 行代码发现 10 错误，你可假定平均第 100 行子程序你可发现一个错误。这确实是一个很自然的假设，但是它是错误的。实际上，绝大多数错误往往倾向于集中在少数有缺陷的子程序中。以下是错误和代码的一般联系：

- 80%的错误往往出现在 20%的子程序中（Endres 1975，Gremillion 1984，Boehm 1987）。
- 50%的错误往往出现在 10%的子程序中（Endres 1975，Gremillion 1984）。

在你认清必然之前，你也可能认为这种联系是并不重要的。

首先，20%的项目子程序占用了 80%的开发代价。但是，这并不意味着这花费最多的 20%的子程序就是含最多错误的 20%的子程序。

其次，尽管高缺陷率子程序所耗代价比是一定的，它所耗代码是异常昂贵的。在 60 年代，IBM 公司对 OS/360 操作系统进行研究时发现，所有错误并不是均分于整个子程序中而是集中在少数子程序中。那些常带有错误的子程序是“程序开发中昂贵的实体”（Jones 1986）。在 1000 行代码中。这些子程序所含错误数可高达 50 个，修改这些错误常常是十倍于开发整个系统所需的平均时间。

第三，开发代价高昂的子程序的含义也是显而易见的。正如一古老的名言所说一样，“时间就是金钱”，其推论是“金钱就是时间”。如果你避开那些令人讨厌的子程序，就可将各种开支削减 80%左右，同时将整个项目的工作量削减相当多。这是软件质量一般原则的鲜明表述，即提高开发质量就能提高开发进度。

第四，避开令人生厌的子程序，对维护的含意也是清晰的。维护侧重于判断、重设计、重写那些被认为是有错的子程序。Gerald Weinberg 曾报道过一个允许维护程序员花费少量时间选择和重写那些产生最多问题子程序的例子。在相当短的时间内，总错误比和维护量都减少了很多。

错误排序

已有研究者试图按其类型将错误排序，并确定每种错误类型的范围。每位程序员都曾遇到过令人生厌的错误：边界错误、忘记重新初始化循环变量等等。本书中的所有检查表提供了更多的细节。

Baris Beizer 集成了几种研究数据，得出了一种不容置疑的详细错误排序方法。以下是其研究结果的摘要：

25.18%	结构错误
22.44%	数据错误
16.19%	功能实现错误
9.88%	实现错误
8 98%	系统错误
8.12%	功能需求错误
2.76%	测试定义或执行
1.74%	系统、软件结构错误

4.71%　　其它

Beizer 的研究结果为两位有效数字，但是对错误类型的研究通常并不是不可置疑的，目前人们已经报道了许多类型的错误，研究错误的所得结果相差也很大。其结果差别可能为 50%而不是一个百分点。不同的报道其结果相差大，像 Beizer 这样将各种研究结果集成起来所得出的结论可能是没有什么意义的。但是即使所得结论不是无可置疑的，它还是有一定建设性意义的。以下是由 Beizer 的研究所得出的一些推论：

大多数的错误的范围是相当有限的。某一项研究表明：不用修改多于一个的子程序，你就可发现 85%的错误（Endres1975）。

许多错误并不是结构性错误。研究人员进行了 97 次采访发现三个最常见的错误源为：贫乏的应用控制知识、需求的冲突和缺乏信息交流和协调(Curits , Krasner 和 Iscoe1988)。

大多数实现错误来自程序员。在所有错误中，有 95%的错误是由程序员本人引起的，2%的错误由系统软件所引起，l%是由硬件引起(Brown 和 sampaon19733,Ostrand 和 Weyuker1984)。

书写错误是一个相当普遍的错误源。一项研究表明 36%的错误是书写错误（Weiss1975）。在 1987 年对接近了百万行的动态飞行软件研究表明 18%的错误是书写错误（Card1989）.另一研究表明：4%的错误是拼写错误（Endre1975）。在作者本人的一个程序中。一位同事发现几个拼写错误只是由于通过一个拼写检查程序运行一个可执行文件所引起的。你应对细节计算有所注意，如你对此有所怀疑，你应考虑三种最为昂贵的编程错误。Gerald Weinberg 报道这三种错误的花费分别为 16 亿美元、9 亿美元、2.45 亿美元。以上每一花费都包括了对前一个修正程序的任何修改。对设计的误解是许多程序常犯的错误。

Beizer 的研究发现 16.19%的错误是由于对设计的误解。而另一项研究则表明 19%的错误是由于对设计的误解（1990）。花费时间对设计有一个深刻的理解是值得的。虽然这并不能产生立竿见影的效果，但是在整个项目的生存周期中你将会得到回报的。

避免赋值语句的错误是质量的关键。一项研究表明 41%的错误来自赋值语句，它和绝大部分错误是边界错误或循环错误是相矛盾的（Youngs1974），这项研究还同时表明：发现赋值语句错误所花时间是发现其它错误所需时间的 3 倍（Gould1975），它指出了一个主要的盲区。在此以前，我们似乎没有给予赋值错误以和边界错误相同的地位。但是花费时间考虑如何避免它们是值得的。

大多数错误是容易改正的。大约 85%的错误在几个小时之内就可改好。而约 15%的错误修改时间为几小时或几天不等，其余 1%的错误所需时间稍长一点。本结果也得到 Beizer 的约 20%的错误需花费 80%的资源来改错的观点支持。通过回溯设计进行分析和设计评审可避免出现较多的错误。

用错误数度量你所在组的经验。本节中所讨论的结果不同，说明了不同的人其实际经验是相差悬殊的、这使你很难利用其它组的经验。一些结果跟通常的直觉是相矛盾的、你可能需要用其它工具弥补你的直觉的不足。一个较好的方法是度量你的进度，这样你就能够知道问题之所在。

错误创建所导致的出错比较

如果错误划分不明确，许多数据错误就可归于由不同的开发活动所引起的。其中，创建总

会引起不少错误。有时，人们认为改正创建错误要比改正分析或结构错误省事些。改正单个创建错误可能是省事一些，但这并不意味着定位全部创建错误的代价小。

- 在小规模项目中，实现错误占了整个错误相当大的一部分比例。在一次对一个小项目(1000 行代码)的代码研究中，75%的错误来自编码，10%错误来自分析，15%的错误来自设计（Jones1986a）。这种错误划分可视为许多小项目的错误分配。
- 实现错误至少要占整个错误的 40%。虽然对大一点的项目这个数字稍低些。有些研究人员甚至发现,对一些非常大的项目,实现错误甚至可占整个错误的 75%（Grady1987）。一般说来，对应用领域懂得越多，整个结构也就越好。这样，错误就倾向于集中在细节设计和编码上面（Basili 和 Perricone1984）。
- 创建错误的修改虽然比对分析和设计错误的修改要省事一些，但是代价仍是昂贵的。对 Hewlett-Packard 的两个项目研究发现：改正创建错误所花代码是设计错误的 25%到 50%。但是当从整体上考虑较大数量的创建错误时，改正创建错误的总代码是改正设计错误代价的 1 到 2 倍。

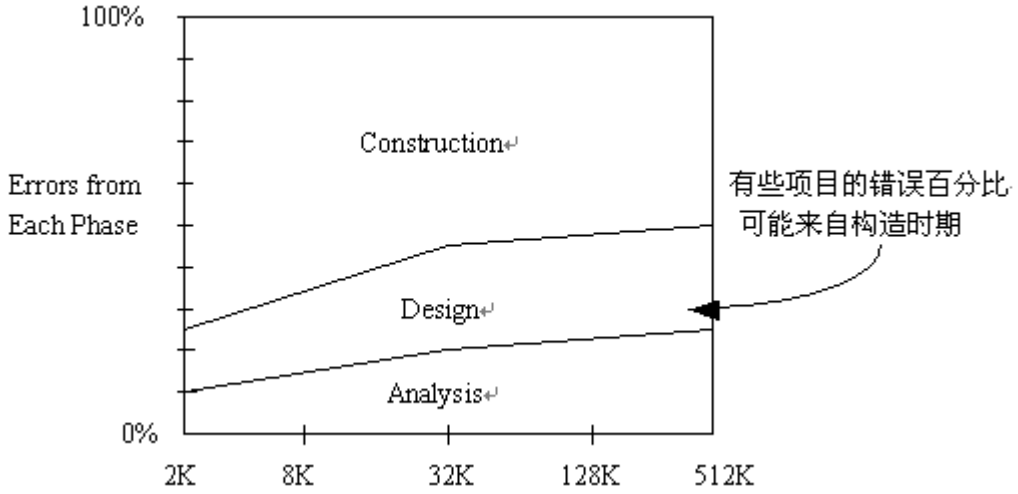


图 25-2　给出了项目大小和错误数的大致关系

你能发现多少错误

你所能发现错误的数量随开发过程的质量而异。以下是可能值：

- 对于所交付软件，一般是每千行代码约 15 到 20 个错误。软件开发通常采用联合开发技术，可能包括结构化程序开发。出现 1 / 10 错误的软件开发是少见的，而 10 倍以上错误的软件往往不会报道（它们可能永远也不会完工）。
- 微软公司开发部的经验是，在开发过程中每 1000 行代码为 10 到 20 个错误，而产品投放市场后平均每千行发现 0．5 个错误。之所以能取得这样的水平是因为采用了第 243 节所述的代码阅读技术和独立测试技术。
- Harlan Mils 开创了”无尘开发”的方法，这种方法可以得到每千行代码为 3 个错误，而在产品投放后每千行代码为 0.1 个错误的错误率。有一些项目，如航天飞机软件，通过采用各种开发方法,如代码阅读、评审、统计调试等，甚至可以做到 500,000 行代码

中不出现一个错误。

无尘开发的结果有力地证实了软件质量的基本原则：开发一高质量软件要比开发一低质量软件并改正其错误要省事。对一个 80,000 行使用无尘开发的项目，其生产率是 740 行代码每人月，而一般方法其生产率是约 150 行代码每人年。代价小而生产率高的原因是由于无尘开发实际上不需调试。Mills 宣称当一个开发组在完成 3 或 4 个无尘开发项目后，它有可能将所出现错误数减少 100 倍，而同时却能提高生产率近 10 倍。这真是值得啊！

测试自身的错误

你可能有如下体验：你发现软件有错误，你凭直觉觉得某部分代码可能出错了，但是所有的代码似乎又都没有错。你试着用几种新的测试用例发现错误，但是所得结果仍是正确的。你花费几小时反复阅读代码并且用手工计算结果，但是仍不能发现错误。过了几小时之后，你重新检查测试数据，你终于找到了错误！原来是测试数据本身出了错。你会感到你浪费数小时的时间来寻找测试数据而不是代码中的错误是一件多么愚蠢的事啊！

这也是一个常遇到的体验。测试用例可能比接收测试的代码含有更多的错误。原因很简单——主要是在开发者编写测试用例时疏忽所致。这些测试用例往往是即席制作，而不是在仔细设计和实现过程中得到的，它们通常被视为一次性测试，并且是可以抛弃的某种东西。

你可按以下几种方法减少你测试用例中的错误数：

检查你的工作。你应如同开发代码一样仔细地开发你的测试用例。你当然也要对自己的测试用例进行二次检查，可在调试器程序一行一行地调试你的测试代码，就如你调试你的程序代码一样。普查和检查你的测试数据是较为合理的方法。

在软件开发过程中计划测试用例。有效测试计划应从需求分析阶段或你接受指定的程序任务时开始，这有助于减少测试用例中的错误。

保留你的测试用例。花费一点时间提高你的测试用例质量。将它们保留下来以便进行循环测试和对第二版本的利用。如果你习惯于保留测试用例，你就可以很容易地发现问题之所在。

25.5　测试支持工具

本节讨论你可购买到或自己开发的测试工具。在此也无法说出某一种具体产品，因为在你读到此处时它们可能都过时了。你可查阅最近一些你最喜爱的程序员杂志。

建立“脚手架”以便测试你的子程序

“脚手架”这词来自建筑术语。“脚手架”用于方便工人能到达建筑物的一些地方。软件“脚手架”的唯一目的是为了能方便地测试代码。一种类型的软件“脚手架”是可被接受测试的高级子程序调用的低级子程序。这样的子程序可称为“残桩”。残桩的可实现程序由你确定，这取决于所需的正确程度。它应能：

- 不需作任何动作就能马上交还控制。
- 能输出诊断信息，可能是输人参数的反应。
- 能测试反馈给它的数据。

- 能从交互式输入中的返回值。
- 不论输入如何都能返回一个标准答案。
- 可增加分配给实时子程序的时钟周期。
- 其功能相当于一缓慢、厚实、简单或欠精确的实时子程序。

另一种类型的“脚手架”是调用正在测试中的伪子程序。这称为“驱动”或“测试工具”。它应能：

- 能调固定输入集的子程序。
- 对交互式输入进行提示，并调用有关子程序。
- 从命令行中接收变量（操作系统应能支持这种使用）并调用有关子程序。
- 从文件中读变元，并调用子程序。
- 由预定义输入数据集对有关子程序多次调用。

另一种“脚手架”是虚拟文件，它和实际文件有着相同的构成。一个较小的虚拟文件可提供许多方便。由于它较小，你可了解它的确切内容并且确信文件本身无错，由于你是特意将其用作测试的，你可设计其内容以使其间的任何错误是显而易见的。

显然，创建“脚手架”需要付出劳动，如果子程序中的错误已被发现，你还可重新利用“脚手架”。如果你使“脚手架”，你就可测试子程序而不必担心其受别的子程序的影响。当你使用一种精妙的算法时，“脚手架”是非常有用的。受测试的代码和其它代码嵌套，使执行每个测试用例往往易使人墨守成规。“脚手架”允许你直接执行代码。你花在创建“脚手架”的几分钟时间可节省数小时的调试时间。

你可在一个文件中写入不少子程序，其中含一个 main()“脚手架”的子程序。main()程序从命令行接受变量输入，并且将传递给正在受测试的子程序，以便于将其和其它子程序集成之前能运行你自己的子程序。在你集成代码时，可先保留子程序和脚手架代码，然后使用预处理命令或注释以使脚手架代码失效。由于通过预处理使其失效，脚手架代码并不影响可执行代码，并且其保留在文件底部，它并不是可见的。留下它并无妨碍。你也可再次使用它，将其移去和存档是不需花费多少时间的。

结果比较

如果你有自动测试工具检查实际输出和预期输出的不同，回归测试和重复测试就将容易得多。检查输出的一个简单易行的方法是将实际输出送入一文件中，然后用文件比较工具将实际输出和已预先存入一文件中的期望数据相比较。如果输出结果不同，你可能发现了一个错误。

测试数据生成程序

你也可编写代码，有选择地运行程序的某一部分。几年以前，作者发明了一个加密算法，并编写了使用此算法的程序。程序的目的是对文件加密以使只有用正确的口令才能将其解密。这种算法并不是简单地改变文件的内容，而是将其彻底改变。将一个程序正确解密是重要的，否则你可能会破坏整个程序。

作者为此设计了一数据生成程序以便能充分地检查程序中的加密和解密部分。它产生一个随机字符的长度从 0K 到 500K 不等的随机文件。也产生长度从 1 到 255 不等的随机字符串，以

将其作为口令。对每一随机用例，数据生成程序产生两个相同随机文件；将其中一个加密。然后重新初始化自身，再解密加密文件。然后将解密文件和未加密的文件作比较。如果发现有不同之处，数据生成程序将其打印出来。

作者将测试用例加权以便使文件平均长度为 30K，这比最大长度 500K 要小得多。否则，文件长度为 0K 到 500K 不等，平均测试文件长度为 250K。较短的平均长度意味着可以测试更多的文件、口令、文件结束条件和其它案件。所得结果是令人满意的。在运行了 100 个测试用例后。作者发现了 2 个错误，它们都是由

实际中可能不会出现的特殊用例所引起，但是它们终究是错误，发现它们也是令人兴奋的。然后，作者试运行程序数周，对超过 100，000 个文件进行了测试而没有发现一个错误。经过对文件内容、长度和口令的测试，作者确信程序是正确的。

以下是作者从中得出的教训：

- 设计良好的随机数据生成程序可产生你没有想到的异常数据。
- 随机数据生成程序要比人工更能深入地检查你的程序。
- 随着时间的流逝，你就能精细随机生成测试用例，以便能侧重于实际范围的数据输入。这意味着着重测试最可能为用户所用的领域，而使这些领域的可靠性最高。
- 模块化设计是有利于测试的。作者能将加密和解密代码抽取出来，并将其各自用于用户界面代码中，这样，编写测试驱动程序的工作是相当容易的。
- 如果所测试的代码有所变更，你可重新利用测试驱动程序。一旦在作者改正了两个早期错误后，马上就开始了重新测试的工作。

覆盖监控

Robert Grady 曾报道过没有检查代码覆盖的测试通常只运行了 55%的代码。覆盖监控是一种跟踪已运行和未运行代码的工具。覆盖监控对系统测试非常有用，因为它可告诉你测试用例是否运行了所有代码。如果你已运行了所有测试用例而覆盖监控指示还有一些代码未执行，

这时你便可知道你还需增加测试用例。

符号调试程序

符号调试程序是对代码普查和检查的技术补充。调试程序可一行一行调试代码，对变量值进行追踪，同计算机一样解释代码。在调试程序中对代码单步调试，并观察其工作情况是非常有用的。对你的代码用调试程序进行普查和让别的程序员对你的代码遂行评审，在许多方面是相似的。你的人工检查和调试程序有着不同的盲点。你可向前和向后看一看你的高级语言代码以及汇编代码，以了解高级语言代码是如何翻译成汇编代码的。你也可以查看寄存器和堆栈以了解变量是如何传递的。你也可查看被你的编译程序所优化的代码以了解各种优化的性能。以上各种优点和调试的使用目的并无太大的联系，是确诊已发现的错误，但是对调试的创造性使用会产生许多和调试初始特性大不相同的好处。

系统的测试

另外一些测试支持工具是用来测试系统的。许多人都听到过一个程序的 99%能正常工作但

最少 1%老是错误的故事。问题一般在于未初始化某一个变量,而且使用它是困难的,因为 99%的未初始化变量其值为 0。

这类测试工具应有如下能力：

- 存储器填充。你应确保不存在任何未初始化变量。有些测试工具在你运行程序之前将特定存储单元设置为随意的某个值，以使未初始化变量不致被设置为 0。在有些场合，存储器单元可能被设为某个特定值。例如，对 8086 微处理器，0XCC 是关于中断点的机器语言代码。如果你将有关存储器单元填以 0XCC，当某个错误执行了你没有料到的地方时，会触发调试程序中的断点，这样你就能发现错误。
- 存储器检查。在多任务系统中，当程序运行时有些工具可重新安排存储器，以便确保你没有将有关代码数据安排在绝对存储器单元，而是安排在相对存储器单元中。
- 可选择存储器失效。存储器驱动器可模拟使某一个程序运行时存储器容易不够的小容量存储环境：存储器请求无效，在失效之前给出准许任一个存储器请求，或在准许某一存储器请求前使另外任意一个请求失效。这对测试用到动态存储分配的复杂程序是非常有用的。
- 存储器访问检查（边界检查）。如果你工作在保护模式的操作系统之下，如 OS / 2 操作系统，你没有必要知道是否会发生指针错误，因为操作系统并不允许你使用超出程序之外的存储单元指针。如果你工作于一个实时操作系统如 MS－DOS 或 Apple Macintosh 之下，你应知道你所用指针是否正确，因为操作系统并不检查存储器越界。值得庆幸的是，你可买到存储器访问检查程序来检查指针运算以确保你的指针不出错。这样的测试工具可用于检查未初始化或悬挂指针。

错误数据库

一个强力测试工具是为所报道过错误而建立的数据库。这样的数据库既是管理也是技术工具。它使你能检查重复发生的错误、追踪新错误被发现和修正的速度、以及跟踪打开和关闭错误的状态及其严重性。如果你想了解有关错误数据库的更多的信息，请看 25.7 节。

25.6　提高测试质量

提高测试质量的方法和其它提高任何过程的质量的方法是相似的。你应对整个过程有相当的了解，这样可稍微变更过程并观察不同情况之间的差异。当观察到一种变更带来了积极的影响时，便可修改此过程，以便使其更好。以下讨论了如何提高测试质量的方法。

计划测试

进行有效测试的关键是从项目开始就计划测试。将测试看成和设过、编码同样重要意味着时间将分配给测试，测试被认为是重要的。并且是一个高质量过程。测试计划应能使测试过程可重复。如果你不能重复某一测试过程，你也就不能提高测试的质量。

再测试（回归测试）

假定你已经对你的产品进行了仔细的调试且没有发现错误，然后在某处对产品作了改动并

想确证它仍能通过所有测试——就是说修改并不引入新的错误。用于防止使软件质量倒退或“回归”的测试叫“回归测试”。

一项对数据处理人员的研究发现 52%被调查者对“回归测试”并不熟悉。这是一个不妙的信号，因为除非你在修改后再对系统地重新测试，否则几乎不可能得到一个高质量的软件产品。如果每做一次修改都进行不同的测试，你将无法确证你是否引进了新的错误。因此，每次都应用到回归测试。有时，当某一产品趋于成熟时增加新的测试，但是旧测试仍保留下来。

管理回归测试的唯一可行方法是使回归测试自动化。当人们将相同的测试进行许多次并看到许多次的结果都相同时；他们会对此厌烦，这会导致人们非常容易忽视错误，使回归测试目的破产。

25.7　测试记录

除了使测试过程可重复之外，你还需度量项目以便于你能明白修改是提高还是恶化了软件质量。以下是一些可供你度量所在项目的数据：

- 对错误的管理性描述（所提交的数据、编写报告的人所修改的数据）
- 对问题的充分描述
- 重复问题的方法
- 对问题的建议处理方法
- 相关错误
- 问题的严重性——例如，致命错误，令人讨厌的错误或无关紧要的错误
- 错误排序——分析、设计、代码或测试错误
- 代码错误的排序——边界错误、赋值错误、数组指针错误、子程序调用错误等等
- 所确定的错误位置。
- 由修改所引起的模块和子程序修改
- 个人对错误态度（这是有争议的，且可能对士气有一定影响）
- 每个错误所影响行数
- 发现错误所需时间
- 改正错误所需时间

一旦你得到以上数据，你可对其进行分析以确定项目质量是好还是不好。

- 每个子程序的错误数。从最坏到最好的顺序排序
- 发现每个错误所需平均测试时间
- 发现每个错误所需平均测试用例数
- 定位每个错误所需平均编程时间
- 测试用例所覆盖代码的百分比
- 每种严重错误中突出错误数

25.8　小 结

- 开发者使用测试是完成测试策略的一个重要组成部分。模块和系统测试也是重要的，

但它们不在本书的讨论之内。

- 各种测试方法的混合使用也是搞好软件质量程序的一个方面。高质量开发方法，包括有条理地对需求和设计进行测试。检查和普查在发现错误方面也至少同调试方法一样有效，并且可发现各种类型的错误。
- 通过采用基本测试、数据流分析和错误猜测你可生成许多测试用例。
- 错误倾向于集中在少数几个子程序中。找到这几个子程序，重新设计和重写它们。
- 测试数据往往比受测试的代码含有更多的错误。这种找错方法只会浪费时间而不会提高代码质量，测试数据错误往往比编程错误更令人讨厌，这可通过仔细地进行测试来避免。
- 最终提高测试质量的最好方法是，有条理的度量它，利用你的知识提高测试质量。

第二十六章　调　试

目录

相关章节

软件质量概括：见第 23 章
单元测试：见第 25 章
软件升级：见第 30 章

调试用于发现错误的根源并改正它，而测试恰好相反，它是用来发现错误的。对有些项目，调试可占到整个开发时间的 50%。对许多程序员来说，调试是编程最为困难的部分。

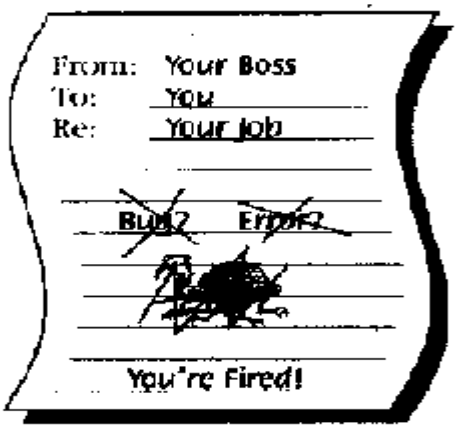
调试其实并不是很困难的。如能遵循以下建议，调试是很容易的，你也只有少数错误要用到调试。大多数错误是由于疏忽所致，通过检查源代码或通过调试程序进行单步调试是很容易发现这种错误的。

26.1　概　述

最近，COBOL 语言的设计者 Rear Admiral Grace Hopper 常提到“bug”。这个词的历史，要回溯到第一台大型数字计算机 Mark I 时代(IEEE 1992)。程序员发现一次电路故障是由一只进入计算机内部的飞蛾所致，从这以后，计算机故障都被称为“bug（臭虫)”。

“臭虫”是一个生动的词，它是由以下图示小虫派生出来的。

将其用于软件缺陷中，就是当你忘了喷洒杀虫剂的时候，臭虫作为不速之客潜入了你的代码中。它们通常可视为错误。软件中的一只臭虫意味着一个错误。错误后果却并不是上图所示那样生动。它更可能同以下便条一样：



在本文中，代码中错误的技术名称是“错误”或“缺陷”。

26.1.1　调试在软件质量中的作用

同测试一样，调试并不是提高软件质量的一种方法。它只用于改正错误。软件质量从项目的开始便应确保。提高软件质量的最佳方法是遵循详细需求分析、有一个出色的设计、高质量编码方法。调试为最终的一个不得已之举。

26.1.2　调试差别

为何要提到调试？是否人人都懂调试？事实并不是这样。对有经验的程序员的研究发现，相同错误所需时间比从 20 到 1 不等。而且，有些程序员不仅能迅速发现错误还能准确地改正错误。以下是对至少四年工作经验的专业程序员调试一个含有 12 个错误的程序的效率的研究结果：

	最快的 **3** 个程序员	最慢的 **3** 个程序员
平均调试时间（分钟）	**5.0**	**14.1**
平均未发现错误数	**0.7**	**1.7**
平均产生新错误数	**3.0**	**7.7**

三个最好的程序员在调试中发现错误所用时间为三个最差程序员的三分之一，而所产生错误数仅为最差程序员的五分之二。最好的程序员可发现所有的错误，并改正它们而不再产生新的错误。最差的程序员漏掉了 12 个错误中的 4 个, 而且在改正了 8 个错误后却引入了另 11 个错误。许多其它研究也已经证实了这种大的差别。

以上例子除了使我们对调试有一个深入的了解，也可证明软件质量的基本原则：提高开发质量可降低开发消耗。最好的程序能发现最多的错误、最快地发现错误，也能最经常地进行错误改正。你不必在质量、代价和时间上作出选择——它们是相互影响的。

26.1.3　使你有所收获的错误

错误意味着什么？如果你并不完全明白你的程序将作何用时在程序中会出现错误。如果你并不十分清楚你将告诉计算机去做些什么，你就不过是在尝试不同的事物。如果你是靠尝试来编程，产生错误是必然的。你不必清楚怎样去改错，你应学会怎样一开始就避免它们。

大部分人是易犯错误的，然而，你可能是一个有着深刻洞察力的优秀程序员。如果是这样，你程序中的错误对你是一个很好的机会：

从中对程序加深了解。如果你已经改正了错误，你能从程序中学到一些东西。

了解错误类型。你编写程序并在其中引入了错误。但不是每天每个错误都是清楚可见的，终有一天你有机会发现某个错误。一旦你发现了某个错误，你应问问自己为什么会产生此错误？你怎样更迅速地发现它？怎样预防错误的发生？是否还有类似错误？你是否能在其引起麻烦之前改正它？

从别人的角度了解代码质量。为了发现错误你不得不阅读代码。这就为评估你的代码性能提供机会。是否代码易于阅读？能否提高代码质量？使用你的发现提高以后所编代码的质量。

了解解决问题的方法。解决问题的方法是否能使你有信心？是否寻找工作的方法？是否能迅速发现错误？是否能迅速调试并发现错误？是否有痛苦和受挫折感？是否常胡乱猜测？是否需提高解决问题的能力。考虑到许多项目花费在调试上的时间量，如你能仔细观察调试的进行，你将不会浪费时间。花时间分析和改变调试方法，可能是减少开发一个程序所需时间量的最佳途径。

了解如何改正。除了会发现错误之外，你应懂得如何改正错误。使用 **goto** 手段只能补偿症状而不是修改问题本身。你是否能容易地改正错误？或通过对问题的精确诊断和对症下药，你是否对问题有了系统的修正？

考虑了各种可能后，调试是培植你自身进步的异常肥沃的土壤。它是所有创建道路的交叉路口（可读性、代码质量）。这也是对创建好的代码的报答——尤其是代码质量好到使你无需经常调试。

26.1.4　一个有效的方法

不幸的是，院校里的编程人员很少提供调试程序的结构框架。如果你在学习编程，你可能已遇到过对调试的讨论了。虽然作者受到了很好的计算机教育，作者所接受的调试建议是“将程序中插入输出语句以检查错误”。这并不是完美的。如果别的程序员所受教育和作者本人类似的话，许多程序员将不得不发明自己的调试方法。这是一种浪费！

26.1.5　调试的误区

在 **Dante** 看来，地狱底层是为魔鬼准备的。在现代社会，过时的方法有可能让不懂如何进行有效调试的程序员如同进入了地狱最底层一般。使用以下几种调试方法可能使程序员备受痛苦：

靠猜测发现错误。为了发现错误，在程序中胡乱插入输出语句，以便检查错误之所在。如果用输出语句仍不能发现错误，再尝试程序中的其它地方直到有了一点眉目。甚至不回到程序的最初版本上, 也不用各种修改记录。如果你并不清楚程序在干些什么，编程是令人痛苦的。你

应存一些可乐饮料之类的东西，因为你在到达终点之间将是漫漫长夜。

不花费时间理解问题。由于问题是试验性的，认为无需完全理解就能改正它，甚至简单地认为发现它就足够了。

用最为明显的方式改正错误。你应改正你所发现的特定问题，而不是去作一些大的模糊不清的修改，否则可能会影响整个程序，以下是一个较好的例子：

```
X ＝ Compute（ Y ）
if（ Y = 17 ）
   X = $25.15 { Compute（）doesn't work for y = 17,so fix it}
```

你只需在明显的地方加入一个特殊用例，不要为了 17 这个值的问题而进入 Compute（）函数的内部。

对调试的迷信。魔鬼已为迷信调试者在地狱留出了部分地方。每组中都有一个程序员可能为无尽头的问题所困扰：可恶的机器、神秘的编译错误、当月圆时才出现的语言错误、坏数据、丢失重要的修改、编辑程序不正确地保存程序。这就是“编程迷信”。

如果你所编程序出现了问题，这是你自己的过错。这不是计算机也不是编译程序的过失。程序本身不会作某些事情。它不会自己编写自己，而是你编写了它，所以你应对它负责。

即使一个错误刚开始似乎不是你的过失，但是你应仍有兴趣弄清楚是否真是这样。这有助于调试，你想找到代码中的错误是困难的，而当你认为你的代码无错时则更是困难。当你宣称某人的代码中存在错误，其它程序员会相信你已对问题进行了仔细检查，这样可能增大你言行不一致的缺点。假设错误是自己的，可使你免受宣称某个错误是别人，而最后发现是你的而不得不改口的窘迫处境。

26.2　找　错

调试包括发现和改正错误。发现错误（并理解错误）将化费 90％的调试时间。

幸运的是，为了找到一种比胡乱猜测更好的调试方法，你无需跟魔鬼有任何关系。和恶魔期望的恰恰相反，边思考边调试要比随意调试更有效和更令人感兴趣。

假定你被要求调查一件凶杀案。哪一种方法更使人感兴趣呢？：挨家挨户检查并询问每个人在 10 月 17 日晚上都干了些什么？或从所发现线索推断凶手的特征？大多数大都倾向于推断凶手的特征，而大多数程序员则发现合理的调试方法更令人满意。甚至，低效程序员中二十分之一的程序员对如何改错也不是任意猜测的。他们使用的是科学调试方法。

26.2.1　科学调试方法

以下是你使用科学调试方法的步骤：

1．通过重复实验收集数据
2．建立假设以解释尽可能多的相关数据
3．设计实验以便证实或否定假设
4．证实或否定假设
5．按要求重复以上步骤

以上过程和调试有着对应的关系。以下是发现错误的有效方法：

1．固定错误
2．确定错误源
3．改正错误
4．测试修改
5．寻找类似错误

以上第一步和科学方法的第一步相似之处在于它依赖于重复性。如果你能使某一错误可靠地发生的话，你也就能方便地确诊它。以上第二步则利用了科学方法的所有各步。你收集导致错误的数据，分析所产生的结果，然后形成对错误源的假设。你设计测试用例并检查以便能评估假说，然后你就能恰如其分地宣告你的成功或重复进行你的工作。

让我们通过一个具体例子看一看以上各步。

假定你有一个不时出错的雇员数据库程序。这个程序按升序打印出雇员表及其收入：

Formating,Fred Freeform	$5,877
Goto,Gray	$1,666
Modula,Mildred	$10,788
Many-Loop,Mavis	$8,889
Statement,sue switch	$4,000
Whileloop,Wendy	$7,860

所出现错误是 Many-Loop,Mavis 和 Modula,Mildred 的结果不对。

固定错误

如果某一错误不是可靠地发生，对其进行诊断是不可能的。使一个间歇性错误能定期发生是调试程序最富有挑战性的任务之一。

不定期发生错误通常是由于未对变量进行初始化或使用了悬挂指针。如果某一数有时错有时对，这可能是计算中所涉及到的某个变量没有被正确地初始化——大多数情况下此变量初始值被置为 0。如果你使用了指针，并且所发生的现象是奇怪的和不可预测的，你可能使用了一个未初始化指针，或对已分配的存储器单元使用了指针。

固定一个错误通常需要一个以上的测试用例来产生错误。它包括将测试用例减至最少而仍能产生错误的情况。如果你所在组织有专门的测试组，有时使测试用例更为简单是测试组的工作。但在大多数情况下，这是你自己的工作。

为了简化测试用例，你引入了科学的方法。假定你有 10 种可用于组合和产生错误的因素。对产生错误的非相关因素建立假说，改变假想非相关因素，然后返回测试用例。如果你仍然发现错误，便可排除这些因素简化测试。这样你可试着进一步简化测试。如果你未曾发现错误，你可否定这些特定假说，这样你便能明白更多的东西。也可能是一些微妙的变化仍将产生错误，但是你最少能明白一个不产生错误的特定修改。

在雇员收入程序中，当最初运行程序时，Many-Loop,Mavis 是列在 Modula,Mildred 之后。当程序第二次运行时，列表是正确的：

Formating,Fred Freeform	$5,877
Goto,Gary	$1,666
Many-Loop,Mavis	$8,889
Modula,Mildred	$10,788

Statement,Sue switch	$4,000
Whileloop,Wendy	$7,860

直到在 Fruit-Loop,Frita 进入收入表并出现在错误的位置时你才记起 Modula,Mildred 已经进入了收入表中。如果这二人分别输入情况又会怎样呢？通常，雇员是成组输入到程序中去的。

你设想：问题可能和输入单个新雇员有关。如果以上设想属实，再次运算此程序并输入 Fruit-Loop,Frita，以下是第二次运行的结果；

Formating,Fred Freeform	$5,877
Fruit-Loop,Frita	$5,771
Goto,Gary	$1,666
Many-Loop,Mavis	$8,889
Modula,Mildred	$10,788
Statement,Sue Switch	$4,000
Whileloop,Wendy	$7,860

以上成功的运行结果证实了设想。为了确证这点，你再次试着输入一个新雇员，一次输入一个，看看它们是否按照错误的顺序出现和第二次运行时顺序是否已改变了。

确定错误位置

简化测试用例的目的是，改变测试用例的任何一方面是否都能引起错误的出现。然后，细心地改变测试用例，并观察在一定条件下错误的表现形式，这样你才能诊断错误。

确定错误的位置也同样需要使用科学的方法，你可能怀疑错误是由某一特定问题，比如边界条件错误所引起。你可将你怀疑的常量取不同值——一个低于边界值，一个恰为边界值，另一个高于边界值——以此来确定你的假想是否正确。

在以上测试用例中，错误的根源在于你增加一位新雇员可能导致边界条件出错，但是当你增进两位或更多时不会使边界条件出错。通过检查代码，你没有发现明显的边界条件错误。求助于计划 B, 使用增一位雇员的测试用例看问题是否真是这样。你增加 Hardcase,Henry 作为一位新雇员输入并设想有关他的记录将会出错。以下是有关结果：

Formating,Fred Freeform	$5,877
Fruit-Loop,Frita	$5,771
Goto,Gary	$1,666
Hardcase,Henry	$493
Many-Loop,Mavis	$8,889
Modula,Mildred	$10,788
Statement,Sue Switch	$4,000
Whileloop,Wendy	$7,860

有关 Hardcase,Henry 这行正是它应该的位置，这意味着你的第一个假设是错误的。问题并不是由一次增加一位新雇员所引起。它可能是一个较为复杂的问题或者是某个完全不同的问题。

再次检查测试输出, 你注意到 Fruit-Loop,Frita 和 Many-Loop,Mavis 是含有下横线的名字。Fruit-Loop 首先被输入时发生了错误，但是 Many-Loop 却并未出错。虽然你并没有最初输入的输出表, 第一次出错时, Modula,Midred 看起来所排位置不对, 它是紧接着 Many-Loop 的。可能

是 Many-Loop 出错而 Modula 没有发生错误。

你设想：问题可能在于带有下横线的名字，而不是单个输入的名字。

但是又怎样解释错误仅在第一次输入雇员时才发生呢？你通过查阅你的代码发现用到了两个不同的排序子程序。一个在输入雇员时用到，另一个在保存数据时用到。当第一个雇员名字输入时对所用子程序的检查发现，它并没有对数据进行排序。它只是将数据按大概次序排一下以加速保存子程序的排序过程。于是，问题在于数据被排序之前就被打印。问题在于带有划线的雇员名因为排序子程序并不处理标点字符。现在，你就能对你的假想进一步求精。

你假想：带有标点字符的名字没有被正确排序就保存下来了。你后来用另外的测试用例证实了你的这个假想。

26.2.2　发现错误的诀窍

一旦你已固定了某个错误且求精了产生错误的测试用例，找到其错误源的工作可能是繁琐的或富有挑战的，这取决于你所编代码的好坏。你可能由于所编代码质量不高费了不少时间才找到错误。你也可能不愿听到这些，但这是真实的。如果你遇到了麻烦，可采用以下方法：

使用所有可能数据进行假设。当你对错误源进行假设时, 你应考虑尽可能多的数据。在上例中，你可能注意到 **Fruit-Loop,Frita** 所在位置并不正确于是你作出“**F**”开头的名字的排序是不正确的。这是一个不正确的假设，因为它并不能解释 **Medula,Midred** 也是处于不正确位置或第二次排序是正确的这个事实。如果数据并不符合假设，你也不应抛弃这些数据——问问它们为何不符合并重新作出假设。

对上例的第二个假设，是认为错误来自带有下横线的名字，而不是单个输入的名字。它似乎也不能解释名字第二次输入时排序是正确的这个事实。然而，第二个假设是更能证实错误的较为求精的假设。刚开始假设不能解释所有数据是很自然的，但是只要你保持对假设求精，最终是能和有关数据是吻合的。

求精产生错误的测试用例。如果你不能发现错误源，可试着进一步求精测试用例。你可使一个常量取代你所设想的还要多的值，你侧重于某一个常量可使你取得重要的突破。

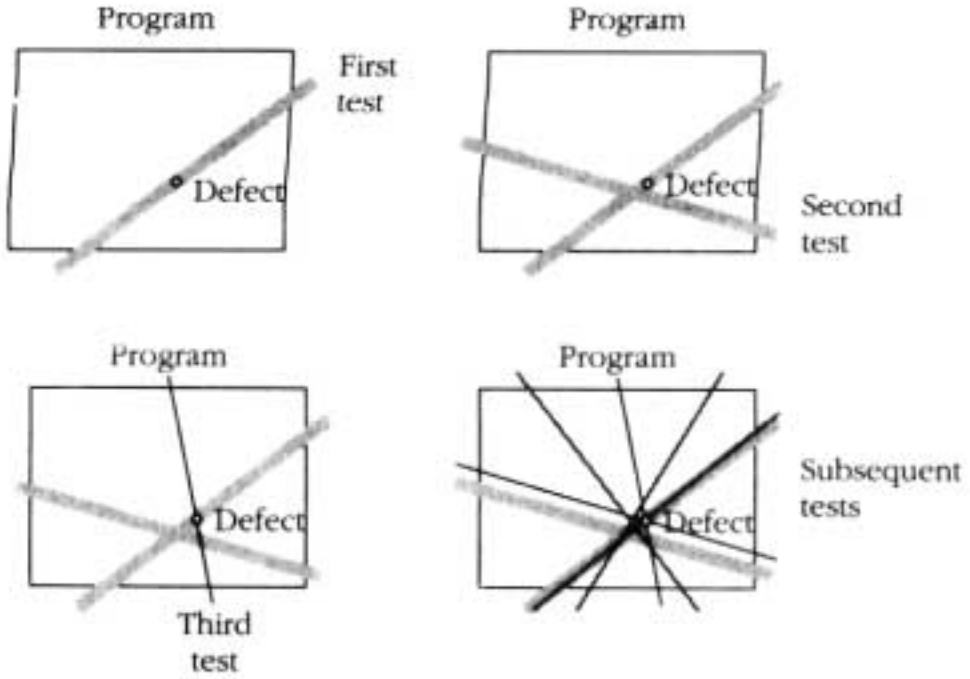
通过不同的方式再生错误。有时试用相似但并不相同的测试用例是有益的。如你用某个测试用例得到一个错误修改，而用另一测试用例得到另一个错误的修改，你就能确定错误之所在。

通过不同的方法再生错误有助于确诊产生错误的原因。一旦你认为判明了某个错误，试运行一个产生类似错误的测试用例。如果此测试用例产生了错误，你也就懂得问题之所在了。错误常常源于各种因素，仅用某一测试来论断错误有时并不能发现问题根本之所在。

生成更多的数据以产生更多的假设。选择测试用例然后运算它们，以产生更多的数据，将其加到你的可能假设中去。

使用否定测试的结果。设想你建立了一个假设并运行了一个测试用例来证实它。再假定有一测试用例否定了你所作假设，如果你还是不知道错误之所在。你仍需要有新的测试用例——即错误并不是发生在你所认为的领域。这缩小了你的研究领域和可能的假设范围。

提出尽可能多的假设。不是将自己限制在一个假设之中，你应尽可能提出更多的假设。你首先不必分析它——你只需提出尽可能多的假设，然后检查每个假说并考虑用测试用例证实或否定它。这种智力练习是有益的。因为它有助于打破由于侧重于某一个原因所引起的调试僵局。



通过不同的方法再生错误已确定产生错误的原因

缩小可疑代码区域。如果你正在测试整个程序，或整个模块、子程序，你应测试较小的部分。有条理地移去部分程序，再看看错误是否还发生。否则，你就知道了你所移去的那部分程序中含有错误。而移去部分程序后错误还发生。你就知道错误发生在留下的那部分程序中。

你应细分和诊断整个程序，而不是任意地移去程序的某个部分。对你的寻找采用二分搜索算法。首先试着移去一半的代码，并确定这一半是否存在错误，然后将二分这部分代码。并检查其中的一半是否含有错误，继续二分下去，直到你发现错误为止。

如果你使用了许多小的子程序，你可通过注释对子程序的调用来达到细分整个程序的目的。或者，你可通过使用预处理命令移去代码。

如果你使用了调试程序，你不必移去代码，你可在程序中某处设置断点，然后检查运行结果。如果你的调试程序允许你跳过对子程序的调用，你就可跳过对一些子程序的执行，然后再看看错误是否发生。调试过程和将程序的某些部分作物理移去是相似的。

怀疑已发生过错误的子程序。已发生过错误的子程序有可能发生错误。在过去已发生过麻烦的子程序比以前未发生过错误的子程序更有可能含有一个新的错误。再检查发生过错误的子程序。

检查最近修改过的子程序。如果你不得不诊断一个新错误，它通常和最近作过修改的代码有关。它可能是全新的代码或已修改过的旧的代码。如果没有发现错误，你可运行一下程序的旧版本，看看是否有错误发生。如果也没有发生错误，你就能明白发生在新版本中的错误是由子程序中的相互作用所引起。

扩展可疑代码区域。将注意力集中在代码的某一小部分区域是容易的，因为你确信“错误必定发生在这段代码中”。如果你没有发现错误，考虑错误不在本段代码的可能性。扩展你所怀疑的代码区域，然后用二分法寻找其中的错误。

逐步集成。如果你每次在系统中加进一部分代码，调试的进行是容易的。如果在增加代码后你发现了一个新的错误，你可再移去这部分代码并单独调试它。借助于测试工具运行子程序

你就可确定它是否错了。

耐心检查。如果你使用集成方法并发现了一个错误，你就可测试某一小段代码以检查错误。有时你可运行集成代码而不是分解代码并检查新的子程序来发现错误。对集成系统运算测试用例，可能需花费较多的时间，而运行某一段特定代码花费的时间会少得多。如果前一、二次你没有发现错误，你就得运行整个系统，忍辱负重，分解代码并单独调试新的代码。

为迅速调试设立最大时间。人们往往习惯于进行迅速的猜测而不是有系统地测试代码以发现全部错误。我们中间的赌徒往往愿意采用五分钟的时间就可能发现错误的方法，而不用花半小时发现错误的稳重方法。问题在于如果五分钟方法不顶用的话，你会变得固执。想通过捷径发现错误，往往是数小时的时间白白流逝而毫无收获。

当你决定采用速胜方法时，你应为其确定一个最大时间限制。如果你超过了时间限制，你应认为错误较难发现而不像你最初所想那样简单，这里你应改走较为费事的查错方法。这种方法能轻易地发现简单的错误而发现较隐蔽的错误也只需花较长一点的时间。

检查一般错误。使用代码质量检查表以激发你能考虑各种可能错误。如果你能遵循第 **24.2** 节所示的检查习惯，你就将拥有你所在环境中，常见错误的良好检查表。你也可使用贯穿于本书的检查表。请参看目录后的“检查表”。

跟其它人谈论有关问题。有些人称这为“交谈调试”。在你向别人解释问题的过程中你往往就发现了错误。例如，如果你在向别人解释工资程序中出现的问题，你可能会这样：“喂，Jennifer，你有时间吧？我遇到了一个问题。我希望程序能对我的雇员工资单进行排序，但是一些名字的排序是不正确的。当我将其打印出来除第一次外其它各次都是正确的。我检查问题是否输入新名字所致，于是我试了一些，发现并无问题。我知道当我第一次打印时程序是能对名字正确排序的，因为程序在雇员名字输入时对其排序，在保存时再一次排序，当输入雇员名字时程序并不对其排序。原来问题在这里。真是谢谢你，Jennifer，你对我帮助很大。”

Jennifer 没有说一句话，你同时也解决了你的问题。这种结果是典型的并且这种方法对你解决不同的错误是非常有效的。

暂时终止对问题的考虑。有时你非常专注，以致于你无法清醒地思考问题。有多少次你暂停下来去喝一杯咖啡当你向咖啡器走去时，你突然明白了问题的解答？或者在午饭时？或者在回家的路上。如果你的调试并无进展而且你已经尝试了各种选择，你应休息一下。或去散步，或作一下别的什么事情。让你的下意识在不自觉中得出对问题的解答。

暂时放弃调试的作用在于减少因调试而带来的焦虑。焦虑的出现是你应休息一下的信号。

26.2.3　语法错误

在给定诊断信息方面，编译程序的性能是有所提高了。你花费 2 小时的时间发现 Pascal 程序中不匹配的分号的日子已经是一去不复返了。以下是你可用来加速这些面临危险的生物的灭绝过程（语言错误问题就如猛玛象和利齿虎一样）的方法；

不要相信编译程序信息所给行数。当编译程序给出一个神秘的语法错误时，你怎么也找不到这个错误——是否是编译程序误解了问题或作出了错误的诊断。一旦你发现了真正的错误，你就弄清楚编译程序给出不正确错误信息的原因。对编译程序有一个较好的了解有助于你发现今后的错误。

不要相信编译信息。编译程序力图告诉你确切的错误，但是编译程序有时又像个捣蛋鬼一

样，你可能不得不一行一行阅读以弄清楚编译程序在提示些什么。如，在 UNIX C 中，当出现整数被零除错误时，你就能得到异常的信息。而在 Fortran 中，你能得到“在子程序中没有 END 结束符”的编译信息，你也能提出许多自己的例子。

不必相信编译程序的二次信息。有些编译程序能较好地发现多重错误。有些编译程序发现第一个错误变得晕晕然不能自制。它们给出并无任何意义的错误信息。而另外一些编译程序较有头脑，虽然它们在发现错误后有一种成功感，但是不会因此给出许多无用的信息。如果你不能很快地发现第二或第三次错误信息，你也不必焦虑。改正第一个错误并重新编译它。

分解。将程序分成几部分尤其有助于发现语法错误。如果你有某个令人讨厌的错误，你可移去部分代码并重新编译程序。编译结果可能是没有错误，给出相同的错误或给出不同的错误。

寻找另外的注释和引号。如果你的代码因为存在其它的引号或从某处开始注释将而编译程序给出错误信息的话，你可将以下符号有条理地插入到你的代码中以便能发现错误：

C　　/*"/* */

或

Pascal {' { }

对 Fortran 来说，就不存在此问题，对 Basic 同样也不存在此问题。

26.3　修改错误

困难在于发现错误。改正错误是一件容易的事情。但是正如其它容易的任务一样，越是容易就越容易出错。正如在第十章所指出的那样，第一次纠错仍有 50%的出错机会。以下是一些减少出错机会的方法：

在改正问题前真正了解其实质。正如“调试的误区”所指出的那样，使你进展困难并损害程序质量的最佳途径是没有真正了解问题的实质就对其作出修改。在你改正某个问题前，你应对存在问题有一深刻的了解。你应通过使用再生错误的用例或不再生错误的用例剖析错误，这样继续下去直到你对问题有了足够的了解以致于每次你都能预测其发生。

理解整个程序，而不只是了解某个问题。如果你对某个错误的上下文有一个较深的理解，你就有可能完全解决某个问题而不是问题的某一方面。一项对短程序的研究表明，对程序有着全局了解的程序员比那些只知某个问题的程序员有着更多的成功地改正错误的机会。由于上述所研究的程序较小（280 行），这并不是说你在改正一个 50,000 行的程序之前应对其完全理解。而是说你至少应理解和改错有关的“邻近”代码——“邻近”不只是几行而是几百行。

确诊错误。在你开始修改一个错误之前，你应确信你正确地诊断了错误。花时间运行测试用例以便证实或否定你的假设。如果你已确证你的错误只是由于某一种原因所致，你也不必有足够的证据；先排除其它原因。

放松自己。如果某程序员想去滑雪，你的产品交付期即将临近，而它已经落后了，并且它还有一个或多个错误需修改。此程序员改变源文件，并将其送交版本控制检查，它也没有重新编译程序且没有确证程序是否正确。

实际上，他所作修改是并不正确的，他的上司生气了。他怎么能对即将交付的代码作修改而不对其进行检查呢？还有什么比这更为糟糕的呢？这是否由欠考虑所致?

如果这不是由欠考虑所致，也可能是一个秘密的或一般的错误。急于解决某一个问题是最浪费时间的事情，它可导致草率的判断、不正确的错误诊断、不彻底的修改。一厢情愿可能会导致找不到问题的解答。压力通常是自己强加的——引起错误的解决方法，有时会不经过证实就自己认为解决问题的方法对的。

保存初始源代码。在改正一个错误之前，你应保存原来的版本以使你今后能利用上。你易忘记你所作修改哪一个是重要的。如果你保存有最初的的源代码，你至少可将新、旧文件作比较以确定修改了的地方。

修改错误问题，而不是症状。你当然应能修改症状，但是你应着重修改问题而不是将程序牢牢地包裹起来。如果你没有透彻地理解问题，你不应该修改代码。你仅修改症状只会使代码质量变坏。假想你有如下代码：

需作修改的 **Pascal** 代码：

```
for ClaimNumber:=1 to NumClaims[Client] do
    begin
    Sum[Client]:＝Sum[Client]＋ClaimAmount[ClaimNumber］
    end;
```

再进一步假定 client 之值为 45，而 sum 之值为 3.45 美元是错误的。以上是修改错误的方法：

对代码作了不正确修改的 Pascal 例子：

```
for ClaimNumber:=1 to NumClaims[Client] do
    begin
    Sum[Client]:=Sum[Client]＋ClaimAmount[ClaimNumber]
    end;
  if(Client＝45) then
    Sum[45]:=Sum[45]＋3.45;
```

现在再假定当 Client 为 37 时，NumClaims[Client]之值应为 0，但是你没有得到 0 值，这时你不正确地对程序作如下修改。

不正确地修改代码的 Pascal 例子(续)：

```
for ClaimNumber:=1 to NumClaims[Client] do
    begin
    Sum[Client]:=Sum[Client]＋ClaimAmount[ClaimNumber]
    end;
if (Client=45) then
    Sum[45]:=Sum[45]＋3.45
else if [Client=37) and (NumClaims[Client]=0.0) then
    Sum[37]:=0.0;
```

如果以上用例对你毫无作用，你也就没有从本书中学到什么东西。虽然本书不可能列出所有问题的可能解答方法，但是以下三个是最重要的：

- 修改并不是在任何情况下都奏效。问题也可能是初始化错误。初始化错误是由定义所致，是不可预测的。所以当今天 client 为 45 时 sum 为 3.45 美元，并不可从中得出明天的情况。明天 Sum 值可能为 10,000.02 美元，或是正确的。这就是初始化错误的本质。
- 它是不可维护的。代码由于存在特殊用例而可在出错时工作，那么特殊用例是代码最突出的特征。3.45 美元不总是 3.45 美元，以后也可能出现另外一个错误。代码也需作相应更改以便能处理新的特殊用例，但是 3.45 美元这个特殊用例不应排除掉。代码因特殊用例而逐渐作相应的修改。最终代码可能难以再继续支持特殊用例，于是它逐渐沉入大洋的底部——这是一个适合它的归宿。
- 使用计算机来计算比用人工计算更为有效。计算机擅长于可预测、有系统的计算，但是人可以创造性地处理各种数据。使用打印机而不是程序处理有关输出是明智的。

仅为某种原因修改代码。修改的病症是任意修改代码，直到表面看来代码工作正常。这种方法的典型原因如下：“本循环似乎含有一个错误。它可能是一个边界条件错误，所以我用-1 试一试。但是并不奏效。所以我再换用+1 试一试。”但是不应任意修改代码。你不理解地对代码作出越多的修改，你就越对其是否能正常工作失去信心。

在作出修改之前，确信此修改能奏效，进行错误的修改可能使你感到惊讶。它将会引起自我怀疑，个人重新估价和深层灵魂自疚。这种情况应很少发生。

每次作一个修改。当同时作几个修改过，修改可能会带来不少麻烦。当同时作二个修改时，可能会引起类似初始错误的微妙错误。这样你处在一种较为窘迫的位置，因为你不知道你是否改正了错误。你已改正错误但又引入了类似的新错误，或者你既未改正错误又引入了类似的新错误。记住，一次只作一个修改。

检查你的修改。你亲自检查程序，或让别人检查程序。运行相同的不同侧面的测试用例来确诊问题，并确证问题的所有方面都已经解决了。如果你只是解决了部分问题，你将发现你仍有事情要做。

重新运行整个程序以检查你所作修改的副作用。检查副作用最容易和最为有效的方法是通过回归测试运行整个程序。

寻找相似错误。当你发现一个错误之后，应寻找和它类似的错误。错误往往是成组地出现。仔细地观察错误类型的某个值时，你就能够改正此种类型的所有错误。寻找相似的错误需要你对问题能有深入的理解。如果你并不知道应怎样发现类似的错误，这就意味着你还没有把握问题的实质。

26.4　调试心理因素

调试同其它软件开发活动一样是智力活动。你的自我意识告诉你代码是好的，而即使你已看到了一个错误，你仍认为没有任何错误。你不得不仔细地思考——建立假说、收集数据、分析假说、有条理地剔除无用的数据——但许多人对这种正规过程不习惯。当你既要设计代码又要调试它的时间，你不得不在以下二者之间进行快速切换：进行创造性的测试和进行僵硬苛刻的调试。在你阅读你的代码时，你应试图与习惯心理做斗争并避免轻率地认为某些代码正是所期望的。

26.4.1　心理因素怎样影响调试

当你看到程序中的符号 Num 时，你知道了什么？你是将其误为“Numb”？或者你认为是“Number”？最可能的情况是你将其误为“Number”。这就是“心理设置”现象——你将你所见误认为是你所期望看到的东西。以下符号是何意?

在以上典型的迷惑测试中，人们常常只看到一个“the”。人们总是看到他们期望看到的东西。再往下看：

- 接受过三种结构控制创建教育的学生常常希望所有程序都是这样：当看到代码中有 goto 语句时，他们就希望此语句能遵循他们已经知道的三种模式的某一种。他们并没有认识到此 goto 语句可能还有其它模式。
- 学过 while 循环的学生常常希望一个循环被连续地估计，就是说，他们希望一旦 while 条件为假时循环就终止了。他们将 while 循环看成了自然语言中的“while”了。
- 发现赋值语句出现错误的难度是发现数组错误或其它“交互式错误”难度的三倍。程序员常能发现边界错误和副作用但是往往忽视简单语句中所常出现的问题。
- 程序员无意中使用 SYSTSTS 和 SYSSTSTS 作为同一变量。直到程序运行了上百次后他才发现这个问题，并且所编书上包含的结果也是错误的。

一个程序员查看如下代码：

```
if (X<Y) then
    Swap:=X
    X:=Y
    Y:=Swap
```

有时可能看到如下代码：

```
If (X<Y) then
    begin
    Swap:=X
    X:=Y
    Y:=Swap
    end
```

人们希望一种新现象类似于他们所见到过的其它现象。他们希望新的结构和老的结构一样。如将编程中所用“while”语句的意义看成和一般的“while”语句。将变量命名为他们以前所用的变量名。于是你将自己所见的东西误为自己所期望看到的东西，并忽视了其中的差别。

心理因素怎样影响调试?首先，它强调良好编程习惯的重要性。好的格式、注释、变量名。子程序名以及其它编程风格，有助于建立编程背景，这样所出现错误才能有所差别。

心理因素的第二个影响在于当错误出现时对部分程序的挑选上。研究发现最有效地进行调试的程序员能轻松地排除掉没有关系的程序段。一般说来，实践知识可使优秀程序员缩小其研究领域并能迅速地发现错误。当然，有时包含错误的部分程序也会被不正确地去除的。这样你可能会花费时间对某些代码段寻找错误而忽视了真正含有错误的代码段。当你在一十字路口走错了方向时，你必须转过身来再向前走。在第 26.2 节对发现错误诀窍的讨论中，有一些建议是可以用来克服“调试盲点”的。

以下是心理因素影响变量名的例子：

第一个变量	第二个变量	心理差距
STOPPT	ST0PPT	几乎不可见
SHIFTRE	SHIFTRT	几乎没有
CLAIMS1	CLAIMS2	小
GCOUNT	CCOUNT	小
PRODUCT	SUM	大

在你调试过程中，你应注意因心理因素所引起变量名和子程序名的差别。在你生成代码时，选择有较大差别的变量名以避免这个问题。

26.5　调试工具

使用很容易得到的调试工具，你就可以进行具体的调试工作。虽然还没有一个完善的调试工具，但是每年都有不少性能有所提高的调试工具推出来。

26.5.1　源代码比较程序

当你修改程序中的错误时，源代码比较程序是一个有用的工具。如果你对代码作了几处修改并且需要去掉一些你已忘记了的修改，比较程序能指出错误并提醒你的记忆。如果你在新版本中发现了在旧版本未曾有过的错误，你就可以比较这二个文件以确定所作修改。

26.5.2　编译警告错误

最简单和最有效的调试工具是你本人的编译程序：

使你的编译程序警告级为最高，修改你的代码以便不产生任何警告错误。忽视编译错误是草率的。而关掉警告信息是更为马虎的一种行为。孩子们有时认为如果自己闭上眼睛便不会看到你，这样你就好比走远了一样。将编译器的警告信息关掉意味着你看不到这些警告信息。这好比你闭上眼睛让一个成年人走远一样，实际上这些警告性错误是存在的。

假定编写编译程序的人比你更为了解有关语言，如果他们对你的程序提出警告，这意味着你可以从中学到关于某种语言的一些新东西。你应尽力弄懂这些警告信息。

将警告信息视为严重错误信息。有些编译程序让你将警告信息视为错误信息。使用本特征的一个原因是它强调了警告信息的重要性。正如你将你的手表拨快 5 分钟使你早 5 分钟一样，将编译程序给出的警告信息视为严重错误可使你更严肃地看待它们。另外一个将警告信息视为错误信息的原因在于它们经常影响你程序的编译。当你编译并连接一个程序时，警告错误通常

并不阻碍程序的连接，但是错误将会阻碍它。如果你想在连接之前检查警告错误，你可将编译程序开关设置为将警告错误和其它错误等同看待。

为整个项目的编译设立标准。为你的开发组的每个人使用相同的编译程序编译代码设立标准。否则，当你试图将不同的人用不同的标准编译所得的代码进行集成时，你将会得到一大堆莫名奇妙的错误信息。

26.5.3　扩展语法和逻辑检查

你可买到另外一些工具，这样你就能比你的编译程序更为深入地检查代码。如对 C 语言。lint 实用工具能仔细地检查初始化变量的使用，将==误写为=和其它类似微妙的错误。

26.5.4　执行剖析程序

你可能不会将执行剖析程序看成一个调试工具，但是对程序剖面进行几分钟的研究能揭示一些令人惊讶(和隐含)的错误。例如，我怀疑一存储管理子程序性能有问题。使用线性排列的数组指针后，存储管理并不是很难的事情。我用杂凑表代替线性排列数组期望能将程序执行时间至少减少一半。但当对代码进行剖析后，发现程序性能并无多大改观。我于是仔细地进行了检查并发现，一个内存分配算法的错误导致了很长一段时间的浪费。问题并不在于线性搜索方法、我最终也没有必要优化这种搜索方法。仔细检查执行程序剖析程序的输出以确信你的程序在每个领域所花时间都是合适的。

26.5.5　使用“脚手架”方法

正如在第 **26.2** 节所指出的那样，抽取一段令人讨厌的代码并执行它，是从有错误的程序中除去恶魔的一种最为有效的方法。

26.5.6　调试程序

高级符号调试程序要比输出语句有效得多，即使输出语句并不是任意地添加的。商业化调试程序在过去 10 年中发展迅速，在今天其能力可以改变你程序的许多方面。

一个好的调试程序允许你设置断点，当程序执行到某一行时，或第几次到达某一行，或当某全局变量改变时，或当某变量被赋给某一特定值时，你都可对其设置断点。它们可以一行一行调试程序，跳过或跟踪进子程序内部。它们允许程序向后执行，并且可向后运行到某个错误产生的地方。它们也可以允许你记录特定语句的执行情况——这和将“I'm here”这个输出语句分布于整个程序中是类似的。

一个好的调试程序允许对数据进行充分的检查，包括结构化数据和动态分配数据。这使得对链表指针或动态数组的检查变得相当容易。调试程序能很好地处理用户定义类型数据。调试程序允许你查询特殊的数据，分派新的数值和继续程序的执行。

你可以查看由你的编译程序生成的高级语言或汇编语言，如果你使用几种语言的话，调试程序自动地为每段程序显示正确的语言。你能看到对全部子程序的调用，并迅速查阅任何子程序的代码。你可在子程序环境中改变程序中常量的设置。一个好的调试程序应允许你选出某一个子程序并用调试程序直接运行它，但是作者本人还没有听说过有类似能力的调试器。

现在最好的调试程序能记住每个程序中要调试的量（断点、要观察的变量等等）。

硬件调试程序在硬件上运行，这样它就不会阻碍正在接受调试的程序的运行。当有时间或存储容量要求时，用硬件调试程序进行程序调试是有必要的。

由于现代调试程序所提供的强大能力，当你看到有人批评调试程序时你可能会感到惊讶。但是一些计算机权威建议不必使用调试器工具。他们的观点是调试只是一种工具，并且你通过对时间问题的思考要比依赖工具的更快发现错误。他们也建议不采用调试程序，而应用人工执行程序的方法发现错误。

但是人们对交互式调试程序效果的研究很少，作者本人对这种最近 **5** 年才开发出来的强大的调试工具也知之甚少。有些早期的研究指出人工调试并不是进行有效调试所必须的。

尽管由于存在实际的证据，对调试程序的反对是没有什么收效的。对工具的误用并不意味着应排除它。这正如你不因服药可能过量就避免服用阿斯匹林，也正如你不因可能划伤，就不用割草机收拾你的草坪一样。任何有力的工具被正常使用，也可能会被误用，调试程序也同样如此。

调试程序并不能代替较好的思考。但是，在有些领域，思考也不能代替调试器。最有效的方法是思考和好的调试程序的联合使用。

26.5.7　检查表

·调试

发现错误的方法：

- 使用所有数据建立假设
- 求精产生错误的测试用例
- 通过不同的方法再生错误
- 产生更多的数据以生成更多的假设
- 使用否定测试结果
- 提出尽可能多的假设
- 缩小可疑代码区
- 检查最近作过修改的代码
- 扩展可疑代码区
- 逐步集成
- 怀疑以前出过错的子程序
- 耐心检查
- 为迅速的草率的调试设定最大时间
- 检查一般错误
- 使用交谈调试法
- 中断对问题的思考

改正错误的方法：

- 理解问题的实质
- 理解整个程序
- 确诊错误
- 放松情绪

- 保存初始源代码
- 修改错误而不是修改症状
- 仅为某种原因修改代码
- 一次作一个修改
- 检查你的工作，验证修改
- 寻找相似错误

调试的一般方法

- 你是否将调试作为学习有关程序、错误、代码质量及解决问题方法的一次机会？
- 你是否避免使用试错法，或避免采用迷信的方法？
- 你是否认为错误是由于你的过错？
- 你是否使用科学方法以固定间歇性错误？
- 你是否使用科学方法发现错误？
- 你是否每次使用不同的方法，而不是只用一种方法发现错误？
- 你是否验证了修改信息是正确的？
- 你是否利用了警告信息、执行剖析程序、建立脚手架方法、交互式调试等？

26.6　小　结

- 调试是软件开发中的问题。最好的方法是使用本书的其他方法避免一开始就发生错误。你仍需花费时间提高你的调试技能——因为最好和最差的调试效率差别为 **10:1**，甚至更大。
- 注意你的编译警告信息，并及时改正编译所提示的错误。如果你忽略了明显错误的话，你就难以改正微妙的错误。
- 发现和改正某错误的关键是，有一个有条理的方法，注重你的测试，这样每次你都能向前迈进一步。
- 在你改正一个问题之前应理解问题的本质，对错误源的任意猜测和修改只会使你的程序更为糟糕。
- 调试是软件开发的有力工具。你应能利用调试工具。记住你还应好好动脑筋。

第二十七章　系统集成

目录

相关章节

单元测试：见第 25 章
调试：见第 26 章
控制结构：见第 22 章
软件改进：见第 30 章

集成是指一个软件开发过程，在这个过程中你要把各个分离的软件部分合并成一个统一系统。对于一个小的工程，集成可能只需要花费一上午的时间就将现有分支程序组合在一起。对于一个大的工程，它可能需要花费几个星期甚至几个月。但无论任务规模大小，它们应用的原理是相同的。

27.1　集成方法重要性

集成方法的重要性在工程领域比在软件方面可能更加显而易见。我住在太平洋彼岸，亲眼看到了由于拙劣的集成造成的危险事故，华盛顿大学的足球体育馆在建造过程部分倒塌。

华盛顿大学足球馆的坍塌原因是，因为在建造期间它没有足够的强度来承受自己的重量。在建设完毕时足球馆可能有足够的强度，但它被按照错误的过程次序来建造，这就属于集成错误。

足球馆在建造倒塌是因为它在结构上没有足够的强度来支承自己。这并不说明它在建成后没有足够的强度，而是它需要在建造过程中每一步都要有足够的强度。如果你用错误的顺序集成软件，系统编码、测试、调试就非常困难了。如果只有等到整个系统能工作时，各个分支软件才能工作。这个系统可能永远也不能完成。这也像足球场在建造中被自身重量压塌一样，即使在整个工作完成后系统可以工作。

因为只有当一个软件开发者完成了单元检测和系统连接测试后才能集合。所以集成过程有时也被认为是一个测试过程。虽然它很复杂，但被看成一个独立的过程。一个好的集成能给你带来以下益处：

- 易于诊断错误
- 更少的错误

- 少量连接框架
- 在短期内形成首次可工作系统
- 短期的全面开发计划
- 良好的用户关系
- 增强信心
- 增加工程完成的机会
- 更可靠的预测计划
- 更准确地了解工程情况
- 提高代码质量
- 减少文件

这些要求对于一个记性不好的系统测试者来说可能显得过高。但正因为尽管集成非常重要却常常被人忽视，我们才在本书中占用一章来讲述这个问题。

27.2　分段与递增集成

程序既可以通过分段的方法，也可以通过递增的方法来集成。

分段集成

直到前几年，分段集成还是一个规范方法，它按如下设计好的步骤进行。

1. 设计、编程、检查和调试。这个步骤叫“单元开发”。
2. 将各程序合并成一个非常大的系统，这叫“系统集成”。
3. 检查和设计整个系统。这叫做“系统再集成”。

分段集成的一个问题是：当各程序在系统中首次被放在一起时，新的问题不可避免地显露出来，并且问题的原因可能发生在的系统的任何部位。既然你有大量的程序还没有在一起工作过，事故原因可能是不易检查的两个程序间的接口错误，或是两个程序相互作用引起的错误。所有的程序都被怀疑。

任何特定问题发生地点的不确定性，再加上这些问题又同时突然出现。这就迫使你不仅要解决程序间相互作用引起的问题，还要解决各种难以诊断的问题，因为这些问题相互作用。正是这个原因分段集成才叫做“爆炸扩张集成”。

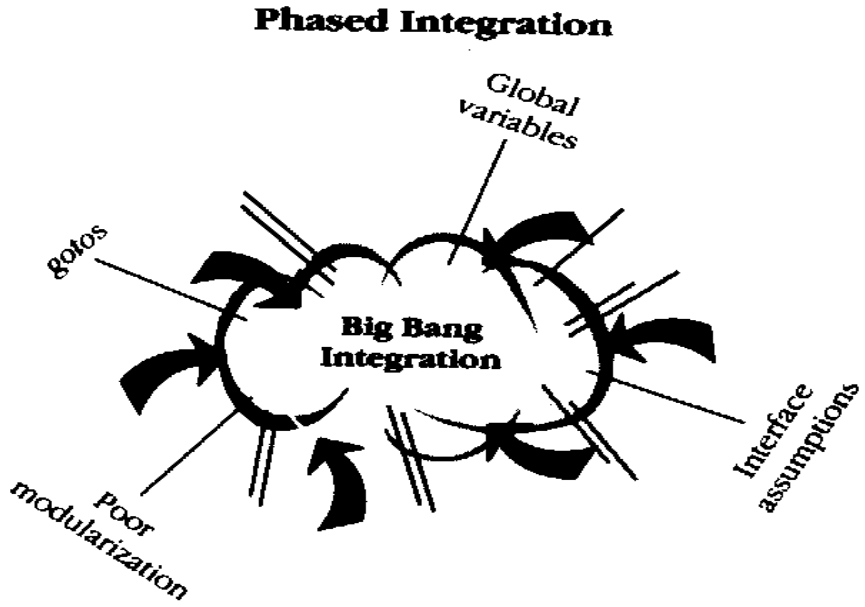
对于一个小的程序或者说是一个很小的程序——分段集成可能是一种最好的方法。如果程序仅由两段或三段分支程序组成，如果你幸运的话。分段集成可能节省你的时间。但在大多数情况下，另一种集成方法更好。

- **递增集成**

在隐喻章中提到：可惜现在还没有一个人写一本关于递增方法方面的书，因为它是一个技术上有效收集的方法。递增集成正是这样一个技术。

通用方法

在递增集成中，你在小的分块中写程序，然后将这些分块一次合成一块，这种一次一块



的集成方法是按如下步骤进行的：

1. 开发系统中一个小的功能块。它可以是最小的功能块，最硬的部分，或是一个关键部分。彻底地检查，调试这部分。它将当做一个骨架，在它上缚着肌肉、神经、皮肤，组成系统的其它部分。
2. 设计、编码、检查和调试程序。
3. 将这些新程序集成在脚手架上。检查、调试脚手架和这些新程序的组合，在加入新程序之前，一定要确保组合工作正确，如果其余工作已被完成，重复过程从第二步开始。

有时，你可能想要集成比子程序大的单元。例如，如果一个模块已经被彻底测试，并且模块的每一个部分程序通过了小的集成。你可以仍然递增集成法将它们集成为统一模块，在你用相同方法加进分块程序时，系统不断得到要求而增大，好像雪球从山上滚下时不断增大一样。

递增集成的优越性

无论你来用什么样的递增策略，递增方法比起传统的分段方法来，有许多优点。

容易确定错误位置。在递增集成中，当出现新问题时，错误一定出在新程序上，无论它是新程序与其余程序接口包含的错误，还是新程序和以前被集成的程序相互作用产生的错误，你都能准确地知道应该检查哪里。更进一步，因为你一次只出现少量问题，你将减少由于多个问题相互作用；或一个问题掩盖另一个问题的危险。在接口产生的问题越多，递增集成法帮助你完成工程的优越性越显著。一项工程的记录表明：39%是模块的接口错误。既然在许多工程中，开发者将 50%的时间花费在调试上，容易确定错误位置带来的益处，在性能和生产中都将最大限度地提高调试效率。

在整个工程中，系统可以尽早成功。当代码被集成并运行时，即使整个系统现在还不能应用，但它却显得很快就可以应用。用递增集成的方法，程序员可以在他们的工作中尽早看到结果，这样他们的信心比当他们怀疑自己的工程总不能首次时强得多。对于成功的感受，看到系统运行 50%比听到系统将被完成 99%，要实在得多。

各单元得到更充分的测试。在工程中很早地开始集成，当开发系统时，你宁愿每一个程序分别集成，而不愿意等到最后将它们一起集成，在这两种情况下，程序都被单元检查，但是作为整个系统的一部分，程序在递增集成中比在分块集成中更多地被运行。

你可以用少量的开发时间建立一个系统。如果集成过程被仔细安排，你可以在系统其它部

分正在编码的同时设计系统的这一部分。这样虽然没有减少开发完整设计和编码所需的工作时间，但一些工作可以平行进行，当上机时间很珍贵时，这也是一个重要优点。

递增集成支持和鼓励其它递增策略，递增原则应用于集成，它的优点是显而易见的。一个更加完整的递增方法——改进的分布法，将在 27.4 节讲述。

27.3　递增集成法

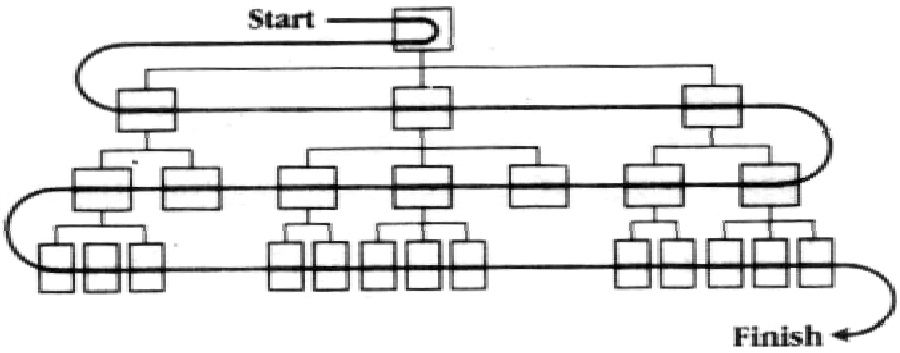
分段集成，你不必考虑工程中各部分建立的先后顺序。所有的部分都是在同一时间里被集成的．所以你可以按照任何一个次序建立他们，只要你能最终都将它们建立好。

关于递增集成，你必须做认真安排，大多数系统要求先集成一些部分再集成其它部分。集成过程的计划安排直接影响工程建立的计划安排；各个部分建立顺序必须决定于它们在集成中的次序。

集成次序的策略有不同形状规模，但没有一种在任何情况下都是最好策略。最佳集成方法随着工程不同而不同。并且最好的解决办法总是针对特定的工程中特定问题而产生的。明白这一点，在次序编号时，你会更好地洞察每一种可能的方法。

- **自顶向下**

在自顶向下集成法中，处在分层结构中顶层的程序最先被写入和集成。代码存根必须被写入以便执行顶层程序。然后，当程序以顶层向底层集成时，存根程序被实际程序所代码。下图表明了这种集成方法是如何工作的。



自顶向下集成法，你要将顶层的程序最先集成，底层的程序最后集成

对于顶层－底层集成法，重要一点是程序和模块间的接口必须认真规定。在调试中，绝大多数的麻烦，不是那些只影响一个程序的错误，而是那些由程序间微妙的相互作用产生的错误。规定接口并不是一个特殊的集成步骤，它只是为了确保接口。

另外，和其它任一种递增集成法相比较，自顶向下法的优点是系统的逻辑控制能够被尽快地测试，所有在分层结构中处于顶层的程序被执行，这样，大的概念性设计问题会被很快地暴露出来。

自顶向下集成的另一个优点是，如果你仔细安排集成次序，可以使系统的一部分先尽快工作。如果用户接口部分处在顶层，你可以先有一个基本的接口以便迅速工作。然后再增补具体内容，由于可以迅速看到工作结果，用户和程序员的信心都增强了。

在低层设计内容被完成之前，自顶向下递增集成法允许你设置一个起始代码，一旦设计程

序运行到所有层次中的某一底层，你就可以开始执行和集成上面高层部分的程序，这就好比在写字母时不用等待点“i”的点，或划“t”上的横线。

尽管有以上优点，但单纯的自顶向下集成法也存在许多缺点，它给你带来的麻烦常常比你想象的要多。

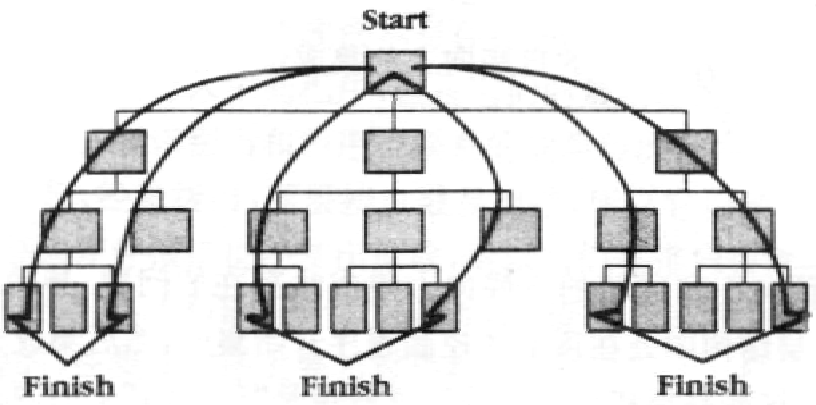
纯粹的自顶向下集成法，运行直到最后才能脱离复杂的硬件接口。如果硬件接口发生故障或执行中出了问题，人们通常希望运行能继续下去。但是低层的问题经常像冒烟一样，按照一定路径传到系统顶层，引起高层部分的变化，这样就削弱了尽快集成工作的优越性，减少冒泡问题，必须对早先集成的单元仔细测试，并对硬件接口程序的运行进行仔细分析。

纯粹的自顶向下集成法的另一个问题是它占据了所有的从顶到底的联系。这意味着在中间步骤中需要大量的接口，否则许多底层程序就不能被集成，但接口是易出问题的。当测试代码时，接口比精心设计的程序包含更多的错误，在支持新程序的新接口中出现的错误破坏了递增集成中抑制错误向新程序蔓延的原则。

单纯地实施自顶向下集成法几乎是不可能的。顶层－底层集成法按预定的顺序运行，你从顶层（叫它第一层）开始集成，然后集成下一层（第二层）的所有文件。当你在没有集成完第二层的所有文件之前，你不能集成从三层往后的各层。在纯粹的顶层向下集成法中，这种死板的规定完全是专断的。很难想象在运用纯粹的顶层向下集成法中不遇到困难。大多数人运用像部分的顶层向下集成法这样的混合方法来代替纯粹的顶层向下法。

最后，如果程序收集站不在顶层，就不能使用自顶向下集成法。如果你采用目标定向设计，收集站将不在顶层，你需要采用不同的集成策略，在许多相互作用的系统中，顶层的位置是主观的，尽管自顶向下集成与自顶向下设计不是一回事，但它们有一点相同的，就是顶层。如果你采用顶层向下设计法设计一个系统时，你至少应该知道需要一个顶层，你可以从它开始从上到下集成。

尽管纯粹的自顶向下集成法不实用，但对它的了解可以帮助你理解通用方法。一些在应用纯粹的自顶向下集成法时出现的好处和问题，对于放松的自项向下集成法并不那样明显。所以一定要记住它们。

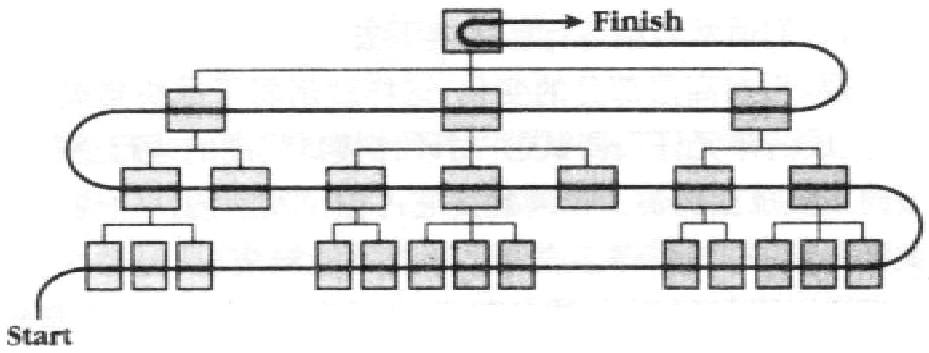


将工作过程变换成直接从顶层到底，你可以从顶层向底层中的一部分集成

- **自底向上的集成方法**

在自底向上的集成方法中，你在程序结构分层中首先从最底层的程序开始集成。然后每

次加入一个低层的程序，而不是将他们同时一起加入，这便是递增集成方法的自底向上的策略。你先从低层开始写入测试驱动程序以便运行低层程序。然后随着系统开发，不断在测试驱动程序的框架上加入新的程序。当你加到高层程序时，驱动程序被实际程序所代替，这便是在程序集成中以自底向上策略的运行次序。



在自底向上集成法中，你最先集成底层程序，最后集成顶层程序

自底向上的集成法具有递增集成法优点，它抑制了单个程序中的错误源被集成，所以错误出现位置很容易被确定，在整个工程中集成能尽快实现。

自底向上集成法中，早期的硬件接口也存在着潜在问题。由于硬件的限制决定了是否能完成系统的要求。所以必须确保所有硬件不出现问题。

那么自底向上集成法存在的问题是什么呢？主要问题是它必须直到最终才能脱离主要的，高层系统的集成接口，如果系统在高层有概念性的设计问题，那么在所有具体工作被执行完毕前，在执行过程中不会发现它们，如果设计要进行本质性的改动，那么一部分底层工作就白做了。

自底向上集成要求你在开始集成前必须完成系统的所有设计工作。如果你不这样做，那些本不应该影响设计的假设可能深深地插入底层代码中，你设计高层程序时总被底层程序的问题所困扰，产生十分不方便的局面，用低层的具体程序去驱动高层的程序设计。这和信息隐蔽、结构化设计的原则相矛盾，高层程序集成本身的问题比起你在完成高层程序设计前便开始执行低层代码所引起的问题来说只是沧海一粟。

和自顶向下集成法一样，纯粹的自底向上的集成也是很少见，你可以用混合集成法取而代之。它们包括部分集成法。

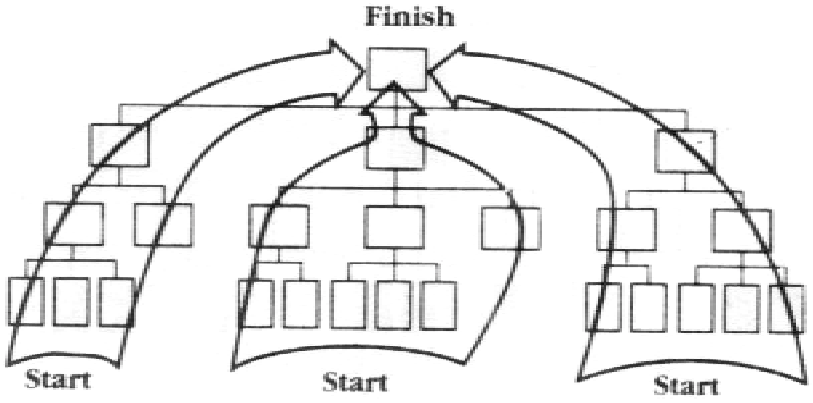
- **“三明治”集成法**

由于纯粹的自顶向下和纯粹的自底向上集成法存在的问题，导致程序员们推出了“三明治”集成法。你从分层结构中处在顶层的控制程序开始集成，然后集成处在底层的设备接口程序和大量实用程序。这些高层和低层的程序就好像是“三明治”上的两片面包。

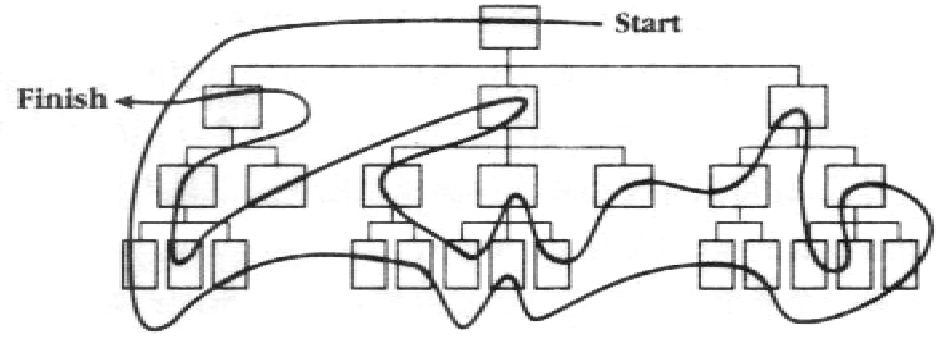
你最后从中层程序脱离集成。这些中层程序就好像是组成“三明治”中间的肉，奶酪和西红柿。如果你是素食者，你还可以用土豆，和豆芽做“三明治”馅，但是“三明治”集成法的使用在这一点上必须保持清醒——你的嘴可能已经塞满了。

下面是三明治方法的图示。

这种方法避免了纯粹的从底层向上或顶层向下的集成方法的死板模式，你从经常出错误



改变纯粹的从底层向上的集成过程，你可以在局部从底层向上集成。这样方法介于底层向上集成法和本章后面要讲述的定向集成法之间



在三明治集成法中，你从顶层和底层程序开始，最后集成中层程序

的程序开始集成，并将需要的框架数量减少到最小，这是一种可行的实用方法。

下一种方法和它相似，只是更加复杂。

- **定向冒险集成法**

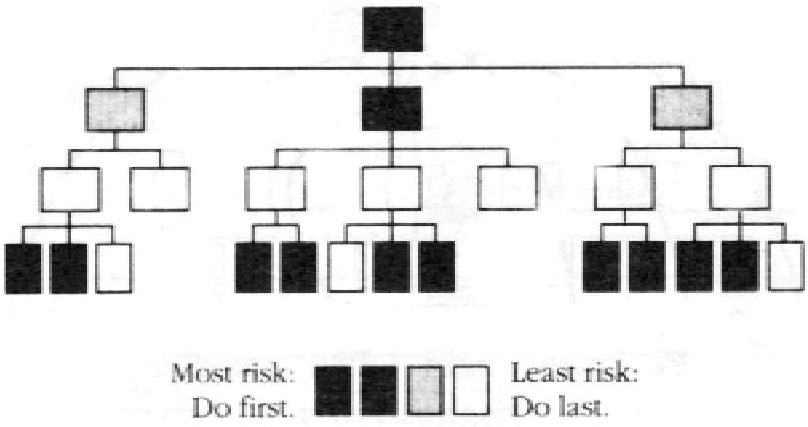
定向冒险集成法又叫“硬件部分优先”集成法。它和三明治集成法一样，是想寻求一种方法，避免纯粹的自顶向下和从底层向上集成法本身存在的缺点。恰巧，这种方法也是从顶层和底层程序中首先开始集成，最后存入中间程序，但是它们的最初目的却是不同的。

在定向冒险集成法，你必须识别出和程序相联系的危险层次。你必须决定运行哪一部分最危险，并从这部分开始运行，经验指出：顶层接口最易出错，所以它们经常排在危险表的开头。在分层结构中，处在底层的硬件接口也经常出危险，所以它们也排列在危险次序表的前头。你可能知道中间程序的危险部位，如可能是一个难以被理解的算法或者执行目标过高。这种程序也可以被认为是高风险的，在集成次序上要相对靠前。

剩余的代码都是不易出危险的，可以等到以后再集成，这些程序中的一部分可能比你想象的要复杂，但这是不可避免的。下面是走向冒险集成法的示意图。

- **功能定向集成法**

递增集成法的最后一种方法是一次集成具有某一功能的程序。这里的“功能”不是指某一模糊的概念。是你正在集成的系统的一种可识别的功能。如果你正在写一个文字处理系统，

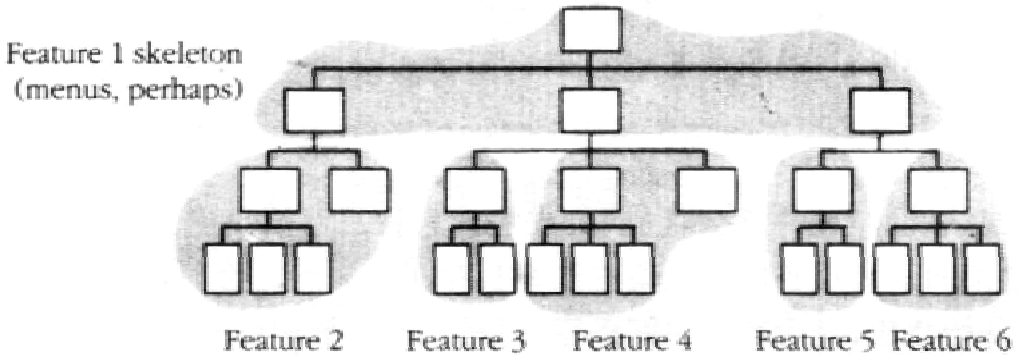


用定向冒险法集成时，被认最易出错的程序最先被集成，越容易执行的程序集成时次序越靠后

这个功能可能是屏幕显示，文件自动重新定格等诸如此类的东西。

许多功能在多个程序中被执行。这样在集成中，对一个程序只在一次被集成的这方面有所改变。当被集成的功能大于一个程序时，此方法降低了你对错误源的确定能力，也就削弱了递增法的优越性，但是只要你在集成具有某功能程序前对这些实施这个功能的程序进行彻底的测试，这个缺点就显得不重要了，你可以先将小块程序集成功能块。然后再将这样功能块集成系统。如此这股反复运用递增集成的策略（参见 27.4 节）。

人们通常希望选择一个能够支持其它功能的内架，从这个框架开始集成，在相互作用的系统中，第一个被集成的功能块可能是一个相互作用的菜单系统，你可以将其它功能块挂在你首先要集成的这个功能块上，看下图所示。



在功能定向集成法，你要将具有某一可识别功能的一组程序一起集成，通常一次不只集成一个程序，但也不一定总是这样

各部分被加在“功能树”——层次结构中，组成某一功能块的所有程序的收集站。如果功能块相对独立。集成就容易实现。允许调动低层库代码以调用其它功能块，但是不能在同一功能块中调用中层代码（可惜，低层库程序在上图中没有画出）。

功能定向集成法有三个优点。第一，除了低层的库程序外，它可以有效地消除其它程序的搭架内容。框架可能需要一些搭架的材料，或者在某一特定功能块被加入之前，部分框架不能

操作。但是当每一个功能块都被挂上框架后，不再需要另加的搭架程序。因为每一个功能块都

是自我包含的，每一个功能块包含了支持它所需的所有代码。

第二个主要优点是每一个最新被集成的功能块，能会产生一个功能递增，这就提供了一种工程稳定向前运动的证据。这个递增的部分是后面章节讨论的改进公布法的要素。

第三个优点是功能定向集成法非常适合目标定向设计。这个目标就像地图一样指出功能块，使功能走向集成成为目标定向系统的自然选择。

纯粹的功能定向集成法和纯粹的自顶向下或自底向上集成法一样，使用起来很困难。通常在特定的重要功能块被集成前，一些低层的指令首先被集成。

自底向上，从自顶向下，三明治法，冒险定向，功能定向。你是否从人们起的这些名字中体会到了这些方法的运行过程？它们是名符其实的。这些方法中没有一种是固定不变的教条，运用这些教条你可以按步就班地从第一步做到第 47 步，然后宣称任务完成。像软件设计方法一样，它们是启发性的而不是计算方法，也不是任何一种现成规范过程。你可以根据你的工程要求，创造出独一无二的集成方法来。

- **检查表**

递增集成策略

- 这种方法是否能辨别出程序或模块的最佳集成顺序？
- 集成次序是否和结构次序相同，以致于结果程序只能在适当的时候被集成？
- 这种方法易于故障诊断吗？
- 这种方法是否需要最少的搭架程序？
- 这种策略确实比其它方法好吗？
- 各部分间的接口是否已被指定好了？

（指定接口不是集成的任务，但证明是否已这样做是集成的任务）

合理使用速增集成法，对软件工程是十分有价值的。下节改进的公布法正是递增策略这种价值的体现。

27.4　改进的公布法

改进公布法是一种公布软件的递增方法。在某些方面它的概念比递增集成法更加广泛，但它的主要技术功能是递增集成和结构的有序化。

- **和探险的相似处**

如同探险安排，探险方法也可以用来安排软件开发过程。

安排的第一种方法是在地图上标出具体步骤中所需的每一件东西。用这种方法你要计划好旅程中的每一个方面——探险者人数和狗的数量、设备和食品的数量、你将要走的路线、什么时候吃饭、每天走的里程、什么时候到达、在哪儿呆多久并且做些什么。你留给自己很小的自由选择余地。

如果你计划得足够准确，你将可以得到一个可以预先确定的工程项目。如果你正工作在熟悉的环境中，你的用户确切地知道需要什么，并且你有一个严格的工作时刻表。探险式的计划是有效的。有一些工程中，你的用户可能要求有一个详细工作时刻表，这就迫使你必须采用探

险式设计计划。

这种方法存在几个缺点。所有工作必须按刻板的计划执行。工作环境的一点变化就能引起计划产生重大问题。探险可能遇上意想不到的暴风雪、疾病或雪橇的机械故障，只要比计划延迟一两天时间，就可能导致整个任务失败。你可能有死机时间，遇上了没有料想到的问题，或是个人私事，这些都能使你的计划延迟。

探险计划也抑制了你对用户环境变化的应变能力。假如探险发现了雪人，将不可能有时间停下来拍照。软件开发与此极其相似。死板的计划是建立在假设你的用户已经完全接受了文件的要求，不需要再做任何更改的基础上的，然而在代码写入之前，用户可能不能准确可靠地描述他们需要什么。对不知道他们到底想要什么的顾客的抱怨是可以理解的，但这方面的软件开发工作却是不可逃避的。随着用户和问题打交道的时间越长，他们对其所需要的东西了解得越清楚。正像你和问题打交道时间越长，所学到的解决问题的办法越多一样。你能提供给用户的任何灵活性，在竞争中都是优势。

第二种方法是制定使你能对环境变化有很好的应变性的计划。Meriwether Lewis 和 William Clark 探险和这种方法相似。

这种方法需要对物资精心选择：四轮马车、威士忌酒、马匹、人员、食物、货车、杂物等，所有这些东西的质量都要认真挑选，这种计划假定探险者能灵活应付在通往所有目的地途中遇到的各种意想不到的情况。一些目的地的情况已经了解的十分清楚，如找一条通往太平洋的线路。但在探险中另一些重要的路程，探险者却对它们的情况、所需物资一无所知。例如走哪条路线、需要多长时间、需要多少张浣熊皮帐蓬等。

在软件中，与 Lewis 和 Clark 式计划相类似的，是建立一个在市场上最实用的文字处理系统，其余的工作是研究“最实用”的含义。它可能意味着极好的打印能力、运行速度快或用户可支持等。它也可能意味着程序异常简单或用户接口功能非常强。具体的目标是无法事先确定的。你的计划必须做好尽量多的准备以应付各种可能性。计划的目的是保障途中没有人热死或冻死，计划要包括各种不可预测、难以想到的情况。这种方法对于人员、工具、标准迅速变化的市场十分有益。

- **通用方法**

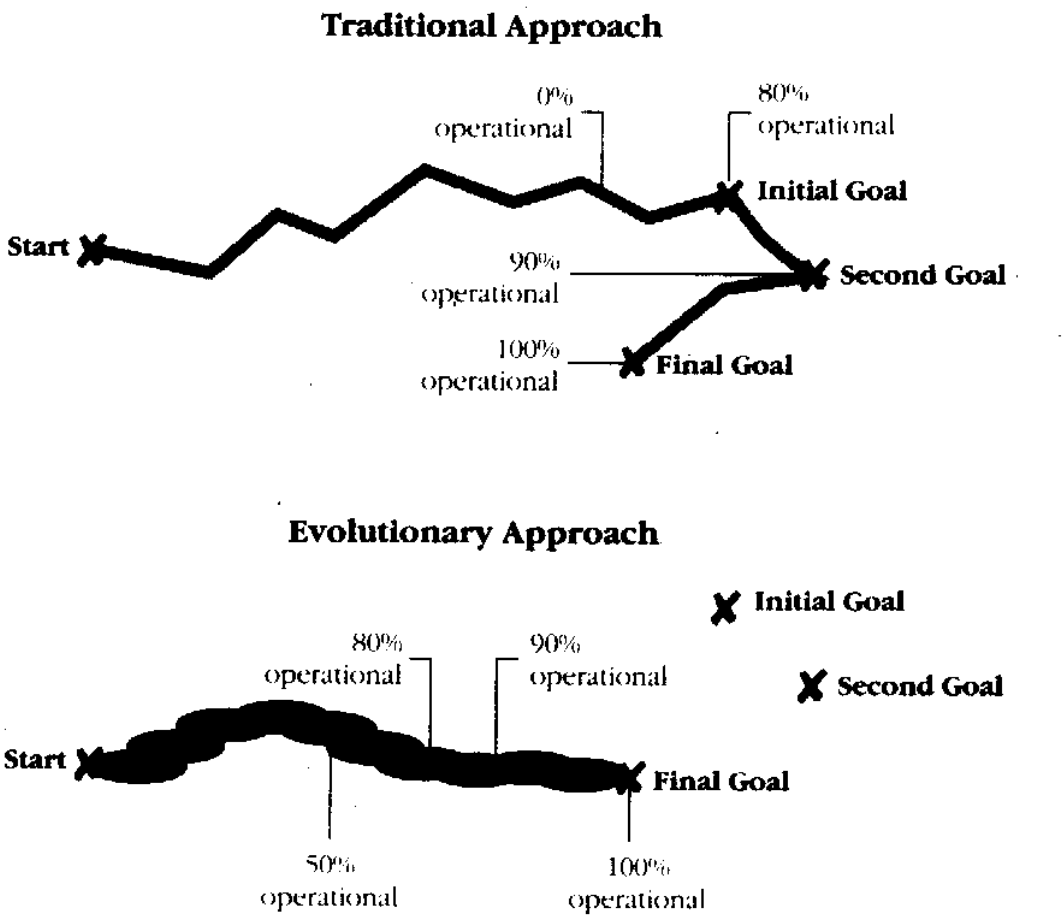
改进的公布法的本质是在完整连续层次中首先并公布一个程序，并且每一层就是这个在一定程序上可用的程序版本。例如，你要开发一个网状帐目分析表程序，你可以安排如下几个层次：

公布 1　实现基本接口、数学计算、支持简单的数据输入，但是许多复杂功能不能实现。
公布 2　实现公式和复杂的数据输入函数。
公布 3　实现存储文件功能。
公布 4　实现数据库操作。
公布 5　实现画图功能。
公布 6　实现与其它产品（数据库、ASCII 码文件其它帐目分析表）的接口。这个产品功能很强。
公布 7　实现调整运行产品，前面版本中的缺点已经识别和改进了。
公布 8　实现整个系统的产品。

改进的公布法的重点是：第一次公布是你最终传送产品的核心。顺序释放在一个仔细安排好的路径中加入更多的功能。

对于改进公布法。部分计划决定你将如何应付用户要求的变化。用户可能要求改变用户接口、功能函数、或执行过程。如果你要满足他们的要求，计划必须有灵活性来响应这些变化。

这个问题正是改进法与传统方法的不同所在。



改进公布法在开发过程中具有灵活性，鼓励中小过程正确，而不是大的结束过程

在传统方法中，工程要先完成初始目标，然后再完成第二个目标，最后达到最终目标。改进方法不必走完实现初始目标的所有路程。它应能稍微改变航线朝不同目标走，同样改进方法不必到达第二个目标，并再次改变航线到达最终目标。因为这种改进方法是递加，所以它能够比非递加法对航线进行更多的修正。并且使到达最终目标的航线更加直接。

在传统方法中，你必须同时开发所有产品。在绝大多数或全部产品都可操作前，没有一个产品是可操作的。多个释放和最终释放主要用于提高产品的可靠性。这个产品是测试性的而不是功能性的。

在改进方法中，完整详细的设计、编码和检查都要在每一次释放中执行。因为每一个释放规模小，你可以把它看成是一个小的程序。一个分析设计和释放的范围依赖于在执行过程中你想要的灵活性。如果你不需要十分灵活，在开始编码前你可以在更好的环境下设计。

在改进方法中，每一次释放计划是建立在前一次释放基础上的。无论你怎样分析设计，它都给你执行过程提供了一定量的灵活性。

第一次释放计划是独特的。它总是需要包括一些属于目标结构考虑的事情，像“软件结构是否充分打开以便支持修改”，包括多少还没有完全意料到的东西？在工程开始时，为软件制定一个通用目录。也是一个好想法。但是，建立的运行程序依赖于有多少灵活性，这只是一个猜测，你今后可以不去考虑它。

改进公布法没有限制新系统的完整性。每一次新的释放都能替换部分老系统并开发新的功能。在一些点上，你可以大量地替换老系统使它实际成为了一个新系统。新系统的结构需要仔细设计，使老系统的限制不影响它的传送。这可能意味着你不能将新系统的特定部分加到老系统上。这样做很不方便。但愿等到新系统能很好地支持这部分工作。

- **优点**

如果不说优点，改进方法是很难被描述清楚的。因为它的优点和运行过程相互缠绕。

集成照顾自己。软件在集成过程中存在的严重问题。它的发生率直接和两次连续集成尝试中的间隔时间成正比。如果两次尝试间隔时间越长，出现问题的机会越大。在快速频繁地公布软件时，当你采用改进公布法时，你必须快速、频繁地集成，这样做能确保你执行正确和减少问题。

Microsoft 公司应用部门的开发者已经成功地应用这项技术。当他们开发一个新产品时，他们每天晚上编译全部产品，每天早晨迅速测试，保证没有一个集成了坏的代码和不好的系统。由于产品总是在工作，应用部门的座右铭是“我们每天都能发送产品”。这个产品可能不具最终应具有的全部功能，但它有的是可以工作。

缩短产品发货周期。对于大众市场软件典型的开发方法是，你先确定第一版，并设计、编码、测试直到完成它，如果开发工作花费的时间比你计划的要长，那么产品发货就要推迟，从产品上获得收益的时间也晚，极端但并不少见的情况是，如果你释放第一版时晚了，你可能永远都得不到第二版。

用改进的公布方法，你可以在一版和二版之间定义一些内插版，你可以确定 2a 版到 2f 版并且在一定时间基数上，每版完成时间间隔一个月，你可以计划在一年中用 7 个月完成 2a 版，八月份完成 2b 版，九月份完成 2C 版…依此类推，如果你能完成你的一年计划，你将完成 2f 版并能够发货“完整”的产品，如果你不能完成 2f 版，你仍有可以工作的产品在 2a 版，2b 版或 2c 版，这些版本的任一个都是前一版的改进，你可以用它们代替 2f 自动发送，一旦你将它发送了，你就可以确定新的增加部分而继续这个过程。

这种方法能够加快产品释放的频率，提高产品释放的可测性，从用户观点看你释放了二个版本 1 版和 2 版，从你的观点看，你开发了 10 个版本但只释放了其中 2 个，这 2 个是在你公司需要它们时释放的，这种方法需要在用户文件与市场间保持一种复杂的协调，但是如果你工作在激烈的软件竞争世界，这是一个有效的办法保持产品发展循环稳定。

在建立过程中，使用户满意。对于任何一个工程，成功与否的一个重要标准是，它是否被用户接受。用改进公布法，一旦系统睁开它的眼睛，眨几下，用户便能为拥有如此工作方式的系统感到振奋。因为系统有很强的可变性，用户们可以尽早地看到它，尽早地提出改进建议，他们有更多的时间调整工作系统，并改进它使其更好读。另外，因为软件被设计成通过几次释放来公布的，这种结构已经假定是可以变化的，这再次增强了你应付变化要求的能力。

易于获得工程状态。技术工程难以管理的一个原因是很难知道工程的真实状况。如果编

码是“90%完成”，它意味着剩下 10%部分将使工程完全被实现呢？还是意味着剩下 10%将产生几百个错误，将工程拖延几个月，我们不希望那种错误无处不在的“90%完成”。改进的公布法提供了频率不容置疑的里程表，不论释放能否做，工作质量可以明显地从释放质量中看出，如果开发组遇到困难，你在第一次释放中寻找，你不能等到工程全部完成却不能工作时再找错误。

减少估计错误。软件工程的错误估计问题是当前软件工业面临的最大问题之一。Capers，Tones 估计平均每个大的工程都要延迟一年，100%地越出预算，不清楚主要问题是原始估计错误，还是工程管理差，或两者兼而有之。改进方法通过快速。频繁的发送会避免这个问题，代替对整个系统一个大的估计，你可以为几个小的释放做几个小的估计，每一次释放，你都可以从错误中学到估计它的方法，反复校正你的方法，使以后的估计提高精度。

更加平衡地分配开发、测试资源。一个典型的改进的公布工程由许多小的计划周期组成，需要分析、设计、编码和测试，分析不是集中在工程开始进行，编程不是集中在中间进行，测试也不是集中在最后进行，你可以在整个开发周期内，或多或少地统一分配分析，编程测试资源。

增强信心。用改进公布法，在第一次释放后，起码这部分产品可以工作，这部分工作的产品使每一个人有理由相信他们的工程会成功，这大大地提高了士气。

增加工程完成的可能性。由于如下几点，你会感到工程完成的可能性更大。

- 工程将不会被放弃，如果工程出现问题只要有 50%被完成，这 50%就可以工作，这比有 90%被完成却一点也不能工作更不易被人们放弃。
- 即使用户花了大钱，工程却只是部分被完成，依靠这些能够很好运行的部分，这个工程可能也是十分有用的，在许多情况下，工程最后 10%至 20%的部分是自由选择内容，并不是工程的核心部分，这样即使失去一点零头，用户可得到大部分必要的内容，想一想，比起你采用一次性全部完成的方法，完成 90%的内容却一点也不能执行来，你的用户得到了他所需的大部分功能并可执行，用户会是多么高兴！
- 你和用户都能同意早点结束，在一些情况下，你达到了 80%的指标，你的用户说，这正是我需要的，我不知道最后 20%部分有何用，这些对我已足够了。如果这种情况发生，你和你的用户都会成为你们公司的英雄，用户因为在达到指标 80%时停止了工程，因此节省了资金，而你却提前完成了工程。

改进的方法全面提高代码质量。在传统的方法中，你知道某些人一定要读你的代码并维护它，这对你去写一个好的编码只是一个间接动因；在改进的方法中，你知道你自己需要多次读和修改编码。自然地这会更强烈地刺激你去写一个可维护、易修改的编码。

你能在早期发现程序是否支持修改。在一次将全部软件发布完毕的工程中，设计和编码的初衷可能支持将来的修改，也可能不行。用改进的方法，你在整个工程的修改问题上会有一个额外收获，如果软件是通过多次释放来被发送的，这个设计一定能支持修改。如果软件不是这样发布的，你在几个释放（版本）内找出并改变它。

编码需要很少的文件编制。在第二十一章程序规模的讨论中，希望文件编制的工作量不随着工程的增大而增加，因为每一个改进的释放表示了少量工作量，你可将每一次释放当成一个小工程，附加适度的文件编制。这一系列小工程对文件编制要求的总量，比一个大的工程要求的文件编制总量少，这种节省是很有意义的，因为在大的工程中，文件编制的工作量占总工作量的 2/3。

- **与原型开发的关系**

改进的公布法是原型开发的一种形式吗？不，它们在几个方面不同，原型开发总是探测性的，它的目的可能是决定用户实际需要的功能，要点将是检查用户接口或计算设备或其它软件的功能。

改进的公布法的重点不是检查性的，产品的每一个版本都应该是可接受的，每一个版本都是实现最终产品的一个步骤。你可以计划安排应付用户要求的变化，也可以不这样做，重点可能只是将软件的初版尽快送到用户手中。

多版的原型开发和 Fred Brooks 的建议相一致，这就是建立一个版本再废弃它。这思想是指建立一个廉价版本概念，然后用一个传统的开发周期建立一个快速、小规模、稳定的版本。你不用为废弃而建立一个版本，你只要将结构任务排成序列以便系统一部分可用上。

一些原型开发的形式中包括保持原型的计划，最终将它改进成已完成的软件，这个方法和改进的公布法是一致的。没必要记住它，通常，原型开发发生在没有改进的情况下，改进的也只发生在没有原型开发的情况下。

- **改进的公布法的限制**

改进的公布法并不是万灵药，要记住这些限制：

改进的公布要求制订更多的计划。在以往的工程中，制订计划集中在工程的开始阶段，一个产品的多个发布需要比传统工程制订更多的计划，这并不是一个缺点，因为在工程的开始阶段集中制定计划通常是没有效率的。工程的管理范围应该涉及修改计划和通过你对工程获得的观察的估计，改进的公布法并不比一个传统方法需要更多的工作量。

改进的公布法需要更多的技术投资。产品的多个释放要求产品在开发周期中释放的几个点都保持足够的清洁。传统方法只需要产品在对公众释放那一点上保持清洁就可以。

如果开始时不十分了解应用目的，这也不是一个大缺点，不断集成不断释放确保代码群都朝同一方向移动。如果你等到工程结束时，发现代码群已经游离你想让它们到达的目标，这时很难再调转它们，使它们到达目标。

如果应用目的很明确、多个释放的经费可能是一种浪费，如果能确保每一个人都能同意工程在哪结束，你可以仅制定一个公布的计划使开发更有效率。

改进的公布法有时被用作苛刻计划、分析或设计的借口。改进的公布法最后一个缺点是它有时为完整的要求或不确定的设计寻找合理借口。如果你用改进法，要保证你不马上改变以前结构上的要求。

- **检查表**

改进的公布法

- 在工程完成前你已经有几个版本了吗？最后的功能将能否实现？
- 第一次释放是否包括了程序的核心，其它程序是否可以从这上面发展？
- 第一次释放是否能尽快地扩大？
- 第一次释放是否可用，至少在最低限度上可用？
- 在工程目标并不十分明确时，你是否尽量努力确定每个改进阶段的内容？

- 每一个释放都要加进重要内容吗？
- 执行过程是否有灵活性以响应用户反馈，如果没有灵活性，这种不灵活性是有意安排的吗？
- 每一个释放是否被看成一个小工程，有自己的编码和测试，在某些情况下，还有自己的分析和设计。
- 结构是否足够开放以支持几次释放后的许多变化。
- 你是否考虑以对现有程序进行修改为基础的改进的公布过程。
- 你是否要用到每一阶段的结果去改进、估计和计划下一阶段？

27.5 小　结

- 集成的计划安排影响程序模块设计、编码和调试的次序，它也影响你是否能顺利地测试、调试它们。
- 递增集成法有许多形式，除非工程非常琐碎，这些形式中的任一种都比分段集成好。
- 一个好的递增方法更接近 Lewis 和 Clark 探险的方法，而不是 Amundsen 的方法，此种（前者）方法假定路程中的一部分没有图标，灵活性是十分重要的，它的计划可以应付意想不到的情况。
- 改进公布法在工程中可将能工作的软件尽快送到用户手中，而传统方法必须使用户等到所有东西都送到后才能工作。
- 改进公布法是对双方都有益的一种方法，对于用户，它可以使他们早日确定工程是否成功，在工程管理上对工程进行有了清楚的了解；对于开发，它可以使人们知道在提高代码质量方面应该做些什么。

第二十八章　代码调整策略

目录

相关章节

代码调整技术：见第 29 章

这一章讨论功能改进的问题——这是一个在历史上有争议的问题。在 1960 年，计算机资源还十分有限，效率是被关心的最重要问题。1970 年计算机迅速发展，程序员认识到他们只注意功能，严重地破坏了系统的可读性和维护性。代码调整很少被注意，随着 1980 年微型计算机的革命，功能限制又把效率问题提出来

你可能在两个方面关心功能问题：策略和战术。这一章只讨论功能上提出的策略问题：什么是功能、它的重要性、获得它的通用方法。如果已经对功能策略有所了解，并正在查阅有关提高功能的特定的代码层技术，你可以看下一章，然而在你看手工作前，至少浏览一下本章信息，是不会浪费你时间的。

28.1　功能综述

代码调整是提高程序功能的一种方法，你可能还发现其它改进功能的方法，但没有一种方法能比代码调整更节省时间，并不损坏代码，这一节就要讲述这种方法。

性能质量与功能

一些人通过有色眼镜看世界，程序员就像你和我，也喜欢通过代码这个有色眼镜看世界，我们总是猜想只要编码做的越好，委托人和顾客就越喜欢我们的软件。

这一点和现实生活中的邮寄件地址在某些方面相似，但是它没街号，也没有自己的实际所在地区，用户对于可实际接触的程序特性比代码质量更感兴趣，只有当系统影响他们工作时，用户才对原始功能感兴趣，否则用户更关心程序的执行时间，而不管它的原始功能，按时发送软件，提供干净的用户接口，避免死机，所有这些可能更有意义。

这有一个说明：我一个星期写 10 封信，但我不想用手写地址，所以我就将信封装入打印机，一年前我要用我的文件处理器打印一个信封，我必须先打开一个信封文件，改变地址，调出对话框，打印信封，再调整出对话框，将打印机联机关上对话框，一星期要做十次这样乏味的工作。

在今年早些时候，我改进了我的字处理程序，现在我有一个信封按钮，不像过去要经过信封文件、联机、对话框等不必要的复杂手续，现在我只要按一个按钮，作为一个方案处理器的用户，我并不关心打印速度是否比过去慢了两倍，因为我的整个处理过程快了。我不知道文字处理器改进代码运行是快是慢，我只知道它的功能更好了。

功能与代码运行速度联系并不紧密，你的代码运行速度并不代表其它性能质量，如果你提高代码运行速度而牺牲其它性能，这不但不能改进功能，还会损害功能。

功能与代码调整

一旦你将效率作为优先考虑标准，不论是强调速度还是大小，你应该在选择改进代码的速度或大小前考虑几个选择，从以下几点考虑提高效率：

- 程序设计
- 模块和子程序设计
- 操作系统相互作用
- 代码编译
- 硬件
- 代码调整程序设计

程序设计

这个标准包括一个单独程序设计的主要过程，将一个程序分成若干模块的主要方法，一些程序设计成高性能系统，另一些程序又很难写成高性能系统。

考虑“real-world data－acquisition”程序作为例子，这个程序被认为是高层设计的，测量速度作为程度主要特性，每个测量量包括通电时间、测量值精度划分、测量值大小范围、变传感器的数据单位（如百万伏）成工程数据单位（如度）。

在这个例子中，如果设计中不寻找危险部分的地位，程序员会发现他们必须设法在软件中用数据方法估计一个 13 阶多项式。这可是一个 13 阶多项式，有 14 项，各变量最高次幂达到 13 次，如果不用这种方法估计多项式，他们就必须寻找问题地址，用不同的硬件和一系列多项式设计，这种变化是不能通过软件调整来改变的，任何软件调整方法也不可能解决这个问题，这是一个在程序设计阶段必须用寻址解决问题的例子。

如果你知道一个程序的规模和速度很重要，你就应该设计一个程序结构，以便你能合理地分配规模和速度指标，设计一个功能定向结构，然后为各系统和规模指定空间和速度的目标，这将在以下几个方面对你有帮助：

- 独立建立空间和速度目标及测试系统的最终功能，如果每个子系统都制订了自己的空间和速度目标，整个系统也就制订了自己的目标，由于子程序已经制定了目标，你可以直接识别它们，并为它们制定再设计和代码调试的目标。
- 只有使目标清楚明白，才能提高实现它们的可能性，程序员必须知道目标是什么后才能实现目标，目标越明白，实现越容易。
- 你可以制订一个现在不直接提高效率，但从长远看能提高效率的目标，通常效率通过前后相关的其它问题一起衡量更好。举例来说，要想实现高度可修改性的目标，你可以制订一个更好的基本效率目标，而不一定把问题本身的直接效率作为目标，用高效的模块，达到可修改设计。你能很容易地用低效率的元件读取一个更有效的目标

模块和程序设计

一旦你在程序设计层次上识别出了模块，就该设计模块和子程序的内部内容，这是另一种功能设计方法，在这一阶段你设计的功能，一个关键问题是选择数据和算法结构。这个问题经常影响存储器的使用和程序的运行速度。

如果你的程序是靠外部文件、动态存储器或输出设备工作的，它可能和操作系统相互作用，如果性能不好，它可能是因为操作系统程序太慢或太繁，你可能没有意识到程序和操作系统相互作用，有时你的编译器产生了你意想不到的系统调用。

代码编译

代码编译器是将易读的高级语言代码转变成可执行的机器代码。如果你选择一正确的语言和正确的编译器，你可以不需要再考虑优化速度，如果你的编译器和连接器组合起来支持覆盖，你可能也不用考虑空间问题。

硬件

有时候最便宜和最好的改进系统性能的方法是买一个新软件，如果你的程序是为全世界成千上万的顾客使用，买新硬件显然是不现实的选择，但是如果你正在开发的顾客软件只是为了几个用户，改进硬件可能是最便宜的选择，它能节省你改进性能的初始工作的投资，它还能提高每一个程序的性能，只要这个程序在这个硬件中运行。

代码调整

代码调整是修改正确代码的初衷，这种修改初衷是为了使程序更有效地运行，它是本章剩下部分的讨论主题，调整不是改变设计，它改变的只是实现。

既然从程序设计到代码调整，你在每一阶段都能使程序戏剧性地得到改进，Jone Bentleg 引证了一个争论，在一些系统中，每一阶段的性能都能被成倍地改进，既然你在六个阶段中的每一个阶段都能得到 10 倍数，这意味着有百万倍的潜在性能改进，尽管这个改进放大倍数要求程序在每一阶段的放大倍数是相互独立的，这种情况很少见，但系统的改进潜力还是很鼓舞人的。

28.2　代码调整介绍

为什么要用代码调整？它不是最有效的提高程序性能的办法。程序设计、数据结构选择和算法选择通常能产生更好的改进效果，它也不是最简单的改进性能的方法，买一个新的硬件，或更好的编译器更加简单，它还不是最便宜的改进功能的方法，它最初需要花费很多时间手编调整代码，并且以后手编调整代码也很不容易读。

用代码调整有几个原因。第一个吸引人的地方是这种方法似乎公然违抗自然规律。一个需要执行 $5X10^8$ 秒的程序，只调换几行，就减小到执行速度为 10^8 秒，这是难以置信的满意结果。

用代码调整的另一个原因是：掌握写出高效的代码的艺术是一个优秀程序员的必经之路，在网球中，你不会通过检球得任何分，但是你仍需要学习正确的检球方法，你不能只会弯下腰用手捡球。如果是你好的网球手，你应该拍头重击网球，等它反弹至腰部高度时接住它。重击超过三次，或第一次球没弹起都算捡球失败，捡网球并不是一个重要问题，但在网球文化中，捡球的方法能给你带来一定的声望,同样,除了你和其它程序员,没有人关心代码的紧凑程度,虽

然这样，在编程文化中，写一个短小有效的代码，能证明你是一个极好的程序员。

代码调试的问题是：有效的代码不一定是好的代码，这是下面几个小结要讨论的主题。

你所听到的许多关于代码调整的概念是错误的，下面是一些常见的误解。

减少高级语言中语句（代码）的行数可提高机器码的执行速度或减小机器码所占空间——错误！许多程序员都紧紧抓住这一信息，如果他们能写一个只有一两行的代码（语句），这才是最有效的结果。请看下面写的一行给数组五个元素赋初值的语句：

```
for i=1    to 5    do              a[i]=i
```

与下面五行相同作用的语句比较，你猜那个执行更快？

```
a[1]=1
a[2]=2
a[3]=3
a[4]=4
a[5]=5
```

如果你按行越少执行速度越快的老教条来考虑，你一定会猜是第一个快，因为它比第二个少了四行，哈！在 Pascal 和 Basic 语言中的测试结果表明：第二个结构比第一个快了 80%。下面是一些比较数据。

语言	for 循环时间	直接编码时间	节省时间	执行时间比值
Pascal	0.660	0.110	83%	6:1
Basic	0.379	0.051	87%	7:1

注（1）这个表和本章以后各表的时间单位用秒，它只具有比较的意义，实际执行时根据编译器，和设备的不同而不同。

注（2）实际各次测试结果有波动，这里只是平均结果。

注（3）这里没有指明编译器的牌子和版本，性能不随牌子和版本不同而改变。

注（4）这个结果不对各种语言都是有意义的。

一种操作也许比另一种操作更快或更省空间——错误！当你谈论性能时不存在“也许”这个概念。你的修改是对程序有益还是有害，必须通过对性能测量来判断。如对规则的每一次改变，你都要改变你的语言、编译器、甚至编译器的版本，在使用一种编译器的一台机器上所能运行的，在使用另一种编译器的另一台机器上可能就是错误的。

下面这个现象说明在一情况不能通过代码调整来改进性能，如果你想使程序简洁，在一种环境改进性能的技术，在另一种环境下使用可能反而降低了性能。如果你更换或改进了编译器，新的编译器可能会自动按照手工调整代码的方法优化代码，这时你的工作可能就白做了。

更严重的是你的代码调整破坏编译器的优化，它本是设计更加直接的代码。

你应该处处优化程序——错误！有一种理论：如果你写每一个程序时，都尽可能使其短小，运行速度快，那么你的整个程度将简洁并执行速度快，这是一种只见树木不见森林的看法，在这种情况下，程序员往往因为过于注重局部优化而忽视了更重要的整体优化。当你继续工作时，会出现以下几个重要问题：

- 在程序完整工作前，你几乎不知道哪个性能最弱，程序员不能把时间花在需要改进的地方。这导致时间花在了不需要的优化上。程序性能很差。因为需要优化的时间都被浪费了。

- 在极少的情况下开发者能正确判断性能弱点，他们过分注重已经认识的弱点而忽视其它弱点。这样做最终的影响是降低了系统的性能，开发者们将自己埋在算法分析和神秘数据中，而这些最后对用户意义并不大，对正确性、模块化、信息隐蔽和可读性的关心倒成为第二位目标，即使性能改进容易，一个性能只能影响 5%的程序代码，你想要运行中 5%的性能改进，还是想要 10%的可读性！

如果通过简化程序语句节省开发时间和用于优化运行程序比，其结果总是优化整体运行程序比只注重局部优化执行更快。

简而言之，过早的优化主要缺点是缺乏远见，用户感受的是最终的性能和程序质量。

在某些工程中，有些优化法不能满足性能要求，你将不得不更大地改动已编成的代码，在这些情况下，小的局部范围内的优化已经不能满足要求，这种情况中问题不是出在代码质量不好，而是软件结构不能满足要求。

如果你需要在程序完成前优化，就要通过在过程中建立透视来减小危险，这个方法是为程序或子系统优化的，建立这些目标的方法就好似当你计算某棵树有多大时一直保持一只眼睛看着森林。

一个快速的程序和一个正确的程序一样重要——错误! 将程序的快速性和简洁性放在正确之前考虑。这种说法一点道理也没有。Gerald Weinberg 讲了这样一个故事：一个程序员被邀请帮助调试一个出了麻烦的程序，程序员和程序开发小组的人们一起工作几天后得出结论程序不可救药了。

在回家的飞机上，程序员把情况仔细考虑了一下，认识到了问题所在，在下飞机前他已经写出了新的程序提纲，当他把新程序调试几天后打算又返回时，这时他收到一封电报说，因为那个程序没有希望调试成功，这项工程已经被放弃了。他赶紧回去劝说上级这项工程能够完成。

接着他又劝说负责这项工程原先的程序员，他们听从了他的劝说，当他干完后，原先系统的设计者问“执行你的程序用多长时间？”

“不可一概而论，但大约平均每个输入用 10 秒钟。”

“哎！但我的程序每个输入就用一秒钟。”这位老手斜靠在椅背上，他因为难倒了这位新手而感到满足。其它程序员似乎也同意这种看法，但这位新程序员也不服气。“你们说的对，像你们的程序不能工作，如果要求我的程序不工作，我也能使它快速运行还不需要花一分钱。”

对于某些类的工程，规模和速度是主要关心的问题，这类工程数量极少，对于这类工程，性能中薄弱环节必须在设计前提出，对另一些工程,过早优化可以严重威胁软件质量，包括性能。

Pareto 原理

Pareto 原理又叫 80/20 定律，内容是你可以用 20％的工作量得到 80％的结果，这个原理除了用于程序设计中，在其它领域也有应用，但它已明确地用于程序优化。

Barry Boehm 在报告中说 20%的程序段消耗了这个程序 80%的执行时间。在早期的论文《对 FORTRAN 程序的经验研究》中，Donald Knuth 发现不到 4%的程序经常占用超过 50%的运行时间（1971）。

Knuth 用一个线性计数表分析程序，发现了这样一个惊人的关系，它的含义对程序优化是清楚的。你应该测试你的程序，找出频繁使用的地方，然后对经常使用的那百分之几的代码进

行优化。Knuth 列表分析了他的线性计数程序并发现：两个循环花费了一半的程序执行时间，他改变了几行代码使原程序速度提高两倍。

Jon Bentley 讲述了一个例子：一个一千行的程序花费了它 80%的时间，在五行中求平方根，通过提高求平方根程序速度 3 倍的方法，他提高了原程序速度的 2 倍。

Bentley 也在报告中讲了一个例子。一个小组发现：一半的系统操作时间花费在一个小的循环语句上，他们重新用微代码写了循环的语句，使循环速度提高十倍，但是这样做并不能改变系统的性能——他们不得不又重写了系统原来松散的循环语句。

ALGOL 语言的设计小组——也是 Pascal、C 和 Ada（曾经是最有影响的语言之一）语言的先驱，接受了如下忠告：“最好”是“好”的敌人，追求完善可能阻碍了完成，首先要完成程序，然后再完善它，需要完善的部分通常是很少的。

测量

因为程序长短与所占时间并不成比例，本程序的一小部分经常占用很多运行时间，所以要测量你的代码，找出频繁执行的代码段。

一旦你发现频繁执行的代码段并优化它们后，就再次测量代码确定究竟将它改进了多少，性能的许多方面是不合常理的，在本意前面例子中，有一个 5 行代码却明显比一行代码（语句）速度更快、内存更小。这只是编码中使你-惊奇的一例。

经验也不能对优化有很大帮助，一个人的经验可能是从一种老的机型语言或编译器中得来。当这些东西中的一种改变时，所有预测就都作废，你再也不能确定一个优化的影响了，除非你去测量这个影响。

几年前，我写一个矩阵各元素相加的程序，源代码可看下面例子：

```
Sum=0;
for(Row=0; Row<RowCount; Row++)
  {
  for(Column=0; Column<ColumnCount; Column++)
      {
      Sum=Sum + Matrix[Row][Column];
      }
  }
```

这个代码是简单的，但是作为矩阵元素相加程序，它并不是最简的，我们知道所有数组运算和循环测试必定是花费大的（占机时多），我在计算机科学班时学过，每一次运行一个二维数组时，要执行花费很大的乘法和加法，对一个 10X10 矩阵，总共 100 次乘法和加法，再加上循环花费。通过变换指针标志，我推断可以用 100 个相对花费小的递增指针代替 100 个花费大的乘法运算，我仔细将代码变换成指针标志。得到下面程序：

```
Sum             =0;
Elementptr      =Matrix;
LastElementptr  =Matrix[RowCount-1][ColumnCount-1] + l;
While ( Elementptr < LastElementptr )
    {
```

```
Sum += *Elementptr++;
}
```

尽管这个代码不像第一个代码那样好读，尤其对于不熟悉 C 语言的程序员，但我自己却十分高兴，对于 10X10 矩阵，我估算一下可以节省 100 个乘法和许多循环花费，我很高兴，决定测量一下提高的速度。

你知道我发现什么了？

没有一点改进。对于 10X10 矩阵、3X3 矩阵、25X25 矩阵都没改进，编译器的优化器已经将第一个代码优化得足够好了，以致于我的优化工作没有一点帮助，我非常失望，我得到的教训是：没有测量性能保证的优化，其结果常常是使你的代码难读。如果认为用测量去证明高效性是不值的，我也就没有必要只为一个程序性能的冒险在这里现身说法了。

测量需要精确

性能测量要精确，用数一只象、二只象、三只象的方法测定程序时间是不够精确的，表分析工具非常有用，或者用你自己系统的时钟和记录去计算操作所用时间。

无论用别人的工具或自己写测量代码，一定保证你测量的只是你要调整的代码的执行时间。如果你正工作在多用户或多任务系统，用分配给你程序的 CPU 时钟周期的次数测量，而不用时间为单位测量。否则当系统将你的程序换成另外的程序，你的程序将被处罚，因为其它的程序执行的时间将算到你的程序上。同样地，尽量找出各种测量的方法，为了避免无论是源代码还是调整代码被不公平的处罚。

编译器优化

现代编译器的优化功能可能比你想象的要强大得多。在我前面举的例子中，编译器的优化和我用自认为更有效的方式重写代码进行的优化，两者相差无几。

购买编译器时，比较每一种编译器在你程序中的性能，每种编译器都有自己的长处和短处，有些编译器比其它的更适合于你的程序。

用编译器，简单程序比复杂程序效果好，如果你自作聪明地干一些像变换循环指针之类的蠢事，编译器不但能工作还会损害你的程序，在 18.5 节中“每行只用一次声明”的例子，在这个例子中，用编译器优化代码，简单直接方法比用复杂编码所能做到的快 15%，节省空间 45%。

用一个好的优化编译器，你的代码通过速度可提高 30％或更多，在下一章讲的技术，速度只能提高 20％，为什么不编写一个清楚的代码，让编译器去优化它呢？下面是几次测量的结果，检查一个优化器能将一个插入分类程序速度提高多少。

语言	没有编译器优化的时间	有编译器优化的时间	节省时间	执行时间比
Ada	2.80	1.60	43%	2:1
C compiler 1	2.25	1.65	27%	4:3
C compiler 2	2.58	2.58	0%	l:l
C compiler 3	2.31	0.99	57%	2:1

程序两个版本间唯一不同是对第一次编译时，编译器优化被关闭，第二次编译时被打开。显然，一些编译器的优化效果好于另一些，你可以检查一下你自己的编译器，测量一下它的编译效果，从三种 c 语言编译器的时间上可能看出，很难说哪种特定语言编译效果比另一种好。

用 Ada 语言编译器优化的代码比三种 C 语言编译器中的两种好，但比第三种差。

什么时候调整

使用一个高质量的设计，使程序正确，使程序模块化并易于修改，这样你今后的设计就容易了，当程序是完整正确时，检查它的功能，如果程序设计者能使它快速、简单，就先不要优化，等到你知道你需要对它优化时。

重复

一旦你找到了性能的弱点，用代码调整改进性能的效果会使你吃惊，用一种技术方法你将很难获得 10 倍的改进，但你可以有效地将几种技术组合起来，即使已发现一种改进方法仍要继续探索。

我曾经编过一个软件，进行数据加密，实际上我不是一次把它编完的，我大约编了 30 次。通过加密可以对数字数据编码，使数据没有口令就不会被发现，加密算法也是比较难的，似乎它自己不会被自动执行，实施的目标是用 IBM-PC 原装机在 37 秒内加密一个 18K 的文件，我的第一个实施程序执行了 21 分 40 秒，所以我还有许多工作要改进。

尽管大多数情况下只用一种优化方法效果很小，但是将它们积累起来就很有意义了。从改进效果百分比上看，没有任何一种 3%甚至 4%的优化能满足我程序性能目标要求，但最后将它们合并起来，效果就非常明显了，这个例子的含义是如果你挖掘得足够深，你就可以获得惊人的改进。

在这个例子中，我做的代码调整是我所做过的代码调整中最富挑战性的，同时最后的代码也是我所编代码中最难读，最不可维护的。最初的算法是复杂的，高级语言翻译出来的代码几乎不可读，翻译成汇编产生了一个连我都怕见的 500 行程序。通常，代码调整和代码质量间的关系都是这样，下表显示了我使用的优化方法。

优化方法	执行时间	改进
初始执行——直接法	21:40	——
变字段为数组	7:30	65%
最内层循环不滚动	6:00	20%
移动最后排列	5:24	10%
合并两个变量	5:06	5%
用逻辑判断合并		
两步加密算法	4:30	12%
使两个变量共享一个存储器		
减少内循环的数据穿梭	3:36	20%
使两个变量共享一个存储器		
减少外循环的数据穿梭	3:09	13%
不打开所有循环		
使用文字数组下标	1:36	49%
移动程序调动		
将所有代码放在一行	0:45	53%
用汇编重新整个程序	0:22	51%
最终	0:22	98%

注：本表的优化过程并不表示所有优化都按此进行，我也没有将全部优化过程写在上面

28.3　低效率情况

在代码调整中，你会发现程序中的某些部分慢得像冬天里的“糖浆”在流动，并且大得像“肥胖的 Alber”。你要改变它们，使它运行起来像加了润滑油，快得像闪电，并且还要使它们小得能藏在 RAM 中的字节间的缝隙里，你总要对程序进行透视，以便确切知道哪一部分运行慢，哪一部分过于复杂，但是一些操作有一个很长的行动缓慢和过于繁杂的历史，要先从调查它们开始工作。

低效率的一般来源

通常有下面几种原因造成系统效率低：

输入输出操作。造成效率低的一个重要原因之一是不必要的输入输出，如果你有一个小文件可以将它存入内存也可以将它存入磁盘（在使用前要将文件读入内存），这时要用内存数据结构。除非内存空间满了。下面是两个代码的功能比较，一个代码是随机存取到一个含 100 个元素的内存数组中，另一个代码是随机存取到一个含 100 个空间的磁盘文件中：

语言	外部文件时间	内存数据时间	时间节省	性能比
C	54.29	0.066	99.9%	800:1
Fortran	56.78	0.159	99.7%	350:l

通过这个数据，内存数据大约比在外部文件中的数据存取快 1000 倍，如果对于外部存取，测试用一个更慢的媒介，如没有磁盘调整缓冲的硬盘或软盘，两者差别还会更大，如果数组有 1000 个元素而不是 100 个，差别也会更大。

对于顺序存取性能比较与前面相似只是差别小一些。

语言	外部文件时间	内存数据时间	时间节省	性能比
C	5.65	0.066	98.8%	85:1
Fortran	38.32	0.159	99.6%	250:1

在顺序存取中内存存取的效果不像随机存取中那样，但仍然能比外部存取大约快 100 倍，足够使你在程序的一个速度临界部分考虑两次输入输出。

格式打印程序。格式打印程序会引起空间扩大和速度减慢，像 C 中的 print()，Basic 中的 PRINT，USING 和 Fortran 中的 FORMAT()语句，它们的代码都是很复杂的，使用一次格式打印就会加入一大堆复杂的代码。下面是格式语句占据的附加空间。

语言	未格式化的 打印空间（字节）	格式化的打印 空间字书	附加字节
Basic	12，310	15，614	3，304
C	7，308	8，921	1，613

这里的每个程序只是用标准输出设备打印一个简单的“Hi”字，它们之间的区别只是打印语言是否被格式化了。如果空间在程序中非常重要。那么打印语句是你寻找节省空间的一个好地方。

浮点操作。对于硬件不支持浮点操作的机器。如许多 PC 机，浮点操作在空间和速度上都

花费很大。速度在下一节里详细讨论，当考虑空间时，要认识到一个单独的浮点操作就能充满整个浮点库。如果你不用任何浮点操作，你就不会得到浮点库，如果你用了哪怕只一次浮点操作语句，你就会得到浮点库的所有影响。下面是一个格式化打印语句打印整数 10 和浮点 10.0 引起的变化结果：

语言	整型空间（字节）	浮点型空间（字节）	附加字节
C	8921	23821	14900
Pascal	8216	20664	12448

用两种编译器，差别是惊人的，在两个例子中，如果你用整型操作代替浮点操作语句，都能节省 15K 空间，第一次浮点操作就充满了整个库，接下来的操作不再带来任何影响。

你能靠这种方法节省空间吗？Basic 语言编译的结果就大不相同了：

语言	整型空间字节	浮点型空间字节	附加空间
Basic	15614	15614	0

在这个例子中，从整型到浮点型的变化没有引起任何差别，这种特殊的 Basic 编译器自动地包含浮点库，无论你的程序是否包含浮点操作。这不是 C、Pascal 语言与 Basic 语言的本质区别，只是编译器特性上的不同，这再次增强了用测量法验证你的优化能否达到你想要达到的目标的重要性。

分页法。引起操作系统更换存储器页码的操作比只在同一页存储器内工作的操作要慢得多。有时一个简单的变换可以产生巨大的不同。下一个的例子中，一个程序员写了一个初始化的循环，这个循环在系统中产生了许多缺页中断，系统用了 2K 页。

```
for Column:= 1 to 1000 do
      begin
      for Row:= 1 to 5 do
          begin
          Table[Row, Column]:= 0
          end
end;
```

这是一个采用合适变量名写成的非常标准的循环，它能有什么问题呢？问题在 Table 语句中的每一元素是 2 个字节长。因为 Table 中每一行（row）有 1000 个元素长。这意味着 Table 中的每一行是 2000 个字节长，也就是大约一页长。这意味着每一个单独的数组存取都要引起缺页中断。如果一个缺页中断用千分之一秒，则初始化代码要运行 5 秒钟。

程序员按下面方法重新构造循环：

```
for Row:= 1 to 5 do
  begin
  for Column:= 1 to 1000 do
    begin
    Table[Row, Column]:= 0
    end
end;
```

这个代码仍然在每次关闭行变量时引起缺页中断，但是它仅关闭 5 次行变量而不是 5000

次了。所以它运行的总时间千分之几秒而不是千分之几千秒。

系统调用。调用系统子程序常常是花费很大的，系统子程序包括磁盘、键盘、屏幕、打印机或其它设备的输入输出操作；内存管理程序和特定的应用程序。如果你的工作环境处在操作系统的顶层，如 Microsoft Windows，测出你的系统调用花费多大，如果它们花费很大，可考虑下面这些选择：

- 你自己写一个服务程序。有时你仅需要系统子程序提供功能中的一小部分，这时你可以从低层系统子程序中建立一个自己的服务程序，用自己的程序作替代程序。这能使程序更小更快，也更适合你自己的要求。
- 避免进入系统。
- 和系统的买主合作使调用更快。大多数买主想改进他们的产品，乐意听到自己每个薄弱环节（他们起初可能似乎有点抱怨，当实际上是很感兴趣的）。

常用操作的功能耗费（指占机时间）

尽管你在不测量的条件下，不能说一些操作比另一些操作花费更大，但是某些操作总是花费要大些。在你的程序中寻找“糖浆”（指运行很慢的程序部分）时，用表 28-1 可帮助你做一些初步判断，猜想你系统中的粘性部分。

表 28-1　常用操作的耗费

		相对消耗时间	
操作	例子	Pascal	C
整数赋值	i = j	1	1
整数加 / 减法	i=j-k	2	3
整数乘法	i=j*k	3	2
整数除法	i=j/k	5	4
用常量下际访问整数数组	i=a[5]	3	2
用变量下标访问整数数组	i=a[j]	3	4
用常量下标访问二维整数数组	i=[3.5]	3	2
用变量下标访问二维整形数组	i=a[j,k]	6	4
访问结构变量区段	i=rec.num	1	1
访问单个指针	i=rec^ ,num	2.5	2
被访双指针	i=rec^next^num^...	3.8	2.6
浮点赋值	x=y	5	85
浮点加 / 减法	x=y＋z	300	150
浮点乘 / 除法	x=y.z	300	150
浮点平方根	x=Sqrt（y）	500	300
浮点正弦	x=sin（y）	1000	700
浮点对数	x=log（y）	1800	1200
浮点指数	x=exp（y）	2000	1300
用常量下际访问浮点数组	x=z[5]	5	100
用整形变量			
下际访问浮点数组	x=z[j]	8	100

操作	例子	相对消耗时间 Pascal	C
用常量下标访问二维浮点数组	x=z[3，5]	5	100
用整形变量下标访问二维浮点数组	x=z[j,k]	10	100
无参数程序调用	foo()	6	6
带一个参数的程序调用	foo(i)	6	6
带两个参数的程序调用	foo(i, j)	8	8

注：本表的测量对于代码环境是敏感的，对编译器优化方法也非常敏感。

你可能从表 28-1 中发现了一些规律：浮点操作通常比整型操作占机时多（在这个例子中浮点操作是在软件中进行，而不是硬件中进行）； 乘除法比加减法占机时多；浮点函数占机时最多。

同这些规律一样有趣的是令人吃惊的现象，在浮点型数学运算操作中，C 语言的编译器大约比 Pascal 语言的编译器快两倍；在执行浮点变量赋值时，C 编译器却比 Pascal 编译器慢 10 倍多，这个差异是非常明显的：对于 Pascal 是 5，对于 C 是 85，如果你正在用这个 C 编译器工作，你要特别注意尽量避免浮点操作。按道理讲一个浮点操作不应该比整型操作多占这么多时间，这个事实说明一定要认真测试性能，而不能想当然。

在复杂操作中考虑两个编译器之间重要等级的不同时，参考表 28-1 中的数据，在你的环境测试一下你感兴趣的操作。

这张表（或是你自己制作的和其相似的表）是解决第二十九章中讲述的代码调整技术中的速度改进的关键，在每一种情况下，改进速度都是用快速的操作代替慢速的操作，下一章将提供如何改进的几个例子。

28.4　代码调整方法

当你考虑代码调整是否能帮助提高系统性能时，应该按如下步骤进行：

1　用高度模块化设计开发软件，这样易于理解，修改。
2　如果性能很差，测量系统，找出频繁执行位置。
3　判断性能的弱点是否是由不合格的设计数据结构算法引起的，判断代码调整是否合适；如果代码调整不合适，返回步骤 1。
4　调整步骤 3 中识别出来的系统中的薄弱环节，测量每一个改进，如果它不能提高系统性能就放弃重来。
5　重复步骤 2。

28.5　小　结

- 性能是软件整体质量的一方面，可能并非最具影响力的一方面。调整代码只是程序性能的一部分，恐怕亦非最重要的。程序结构、具体设计数据结构和算法选择，对于程序规模和执行速度的影响要比产生代码本身的工作大。

- 优化设计关键要用到定量测量。重要的是找到确实能改善程序性能的段落，在此就要强调优化的作用。
- 大多程序主要的时间花在小部分代码上。在执行程序及测试前不容易知道是哪部分代码。
- 为写出优良程序最好是写出清晰易读和易修改的源程序来。

第二十九章　代码调试技术

目录

相关章节

代码调试步骤：见第 28 章

尽管代码调试在这些年来被忽视，在计算机编程的历史上，它却一直是个时髦的课题，因此，一旦你觉得有必要提高程序质量，并希望达到代码水平，你就可以参阅这下面丰富的调试技术汇总。

本章重点涉及在如何提高速度，并给出减少代码量的一些提示。速度与规模是程序性能的标准，但减小规模更多地牵扯到重新设计数据结构而非代码调试，而调试指的主要是设计的执行方面，而非设计本身。

本章并没有给出几个普遍运用的调试技术手段，以便让你照搬不误。这里讨论的主要目的是作一下示范，以使你能灵活地应用于具体情形上。

29.1 循　环

由于循环段要被执行多次，程序的弱点多半在循环段内。此节力图使循环执行更快。

避免开关

开关指在循环体内设判断句，每次循环一次执行一下判断。如果循环执行时，判断指向不变，你可以在循环外设判断，对循环避免包含开关，将循环嵌入条件中而非相反。这里有个反切换的例子。

C 循环体内有开关的例子：

```
for（i＝0; i＜count; i＋＋）
{
    if（SumType = = Net）
    {
```

```
    NetSum=NetSum＋Amount［i］;
    }
  else / *SumType＝Gross* /
    GrossSum=GrossSum＋Amount［i］;
    }
  }
```

这段中，通过每次重复，if（sumType==Net）都被测试一遍，甚至当其状态相同也要一遍遍地检验。为提高速度，写成如下方式更好:

C 不含开关的例子：

```
if（SumType=Net）
  {
  for（i=0，i＜Count，i++）
    {
     NetSum=NetSum＋Amount[i];
     }
  }
else  / *SumType = = Gross* /
  }
  for（i=0;i<Count;i++）
   {
   GrossSum=GrossSum+Amount[i]
   }
  }
```

for 语句被多次重复，故代码占空间大些，但好处是省时。

语言	直接时间	调试时间	省时间
C	1.81	1.43	21%
Basic	4.07	3.30	19%

注： （1） 时间单位为秒。实际时间因编译器、编译选择方式及具体测试环境而异。

（2） 基准结果是由几次到几千次代码片段的结果加以采样，平滑得到的。

（3） 某些种类的编译器效果不同，因而性能因种类而异。

（4） 各种语言作的结果比较不都有意义，因为不同语言编译器不总是可比的。

本例的危险点在于两个循环须是并行的，若 Count 变为 Client Count，相应两处都得改变，这对于维护程序是件苦差事。

若你的代码真是如此简单，你就可以放过维护问题，在这循环中，引入新变量，将它分配到循环后的 Netsum 或 GrossSum 中。

冲突

或称“熔断”（fusion），是同一元素集被两个循环同时操作造成的。解决方法在于斩断循环，将其变成单个循环，这有个循环冲突的例子。

Basic 把会冲突的循环分离：

```
for i=1 to Num
    EmployeeName（i）=" "
next i
…
```

```
for i=1 to Num
    EmployeeEarnings(i)=0
Next i
```

如果你的程序有冲突竞争，你得将之合二为一。这通常意味着，循环计数器只能有一个。本例中两个循环用到 1 to Num，可能有冲突争先的危险，可以改成：

Basic 有冲突的循环：

```
for i＝1 to Num
  EmploreeName（i）＝""
  EmployeeEarnings（i）=0
next I
```

语言	直接时间	调试时间	节省时间
Basic	6.54	6.31	4%
C	4.83	4.72	2%
Fortran	6.15	5.88	4%

你可以看到，省时量不太显著，在有些情形，省时可能更为重要，但是你必须测量才知道。

循环的冲突会有两种危险：首先，两部分的标志可能因冲突而改变，因此不再相匹配；第二，每一循环的位置可能是重要的,在你合并之前，得明确应如何将其组成正确的顺序。

展开

循环展开的目的是减少循环内的操作，在第二十八章中，一个循环被展开，但是 5 行代码比一行要快。这时，循环被展开成一到五行。每行按顺序执行。

尽管在处理少量元素时，循环的展开是个快速方法且有效，元素多时就不适用了，或者是在你不清楚会用到多少元素时不适用。例如：

Ada 可展开的循环：

```
i:=1;
while(i<=Num) loop
    a(i):=i;
    i:=i-1;
 end loop
```

要部分展开循环，你要处理好每一种情况。这种展开损害循环的可读性,但不损害通用性。下面是展开的循环：

Ada 一次展开的循环：

```
i：=1
while（i＜Num＝ loop
```

```
        a(i)   := 1;
        a(i+1) := i+1;
        i := i+2;
    end loop
    if (i=Num) then
        a(Num):=Num
    end if;
```

当五行顺序式代码展成九行复杂代码时，维护就变得难多了，可读性也差了，若非能提高速度，它没什么可取之处。任何一种设计法则，都需经权衡取舍。所以既便某种技术显得劣拙，到了特定环境下，它又是上上人选。

本书方法将原来 a（i）：=i 换成两行，i 增加两个步长而不是 1。在 while 循环后的特殊代码是必要的，如果 Num 是奇数，且循环结束后还应再重写一次计数。下面是展开前后的比较：

语言	直接时间	调试时间	节省时间
Ada	1.59	0.82	21%
C	1.59	1.15	28%

从 21％到 28％的改善是可观的，循环展开的危险在于，可能发生边界错误。如果我们把循环更进一步展开，会有更多的优越性吗？下面是个二次展开的代码段：

Ada 二次展开的循环：

```
    i:=1；
    while（i＜Num－1＝ loop
      a（i） := 1;
      a（i＋l） := i+1;
      a（i＋2）:= i+2;
      i:=i+3;
    end loop
    if（i＜Num＝ then
      a（Num）:= Num;
    end if;
    if（i＝Num-1） then
      a（Num-l）:= Num-1;
     end if;
```

下面是循环展开后运行耗时结果的比较：

语言	直接时间	一次展开时间	二次展开时间	节省时间
Ada	1.04	0.82	0.72	10%
C	1.59	1.15%	0.99	10%

结果表明，进一步的展开能再度省时，我们最关心的是程序会变得多么繁琐，你可能认为它不那么复杂，可要是你还记得几页前,其不过是一个只有五行的循环，你就会注意到性能与可读性之间的平衡问题了。

循环内工作量的最小化

写出有效率程序段的关键在于，循环内的工作量最小。如果你能估计出全部或部分结果语句，并仅将结果用在循环内，这么作是有理由的，通常这是一种编程的好方法，有时还会增可加可读性。

假如你有如下程序，循环内有一复杂的指针表达式：

C 循环内带有复杂的指针表达式：

```
for（i＝0，i＜Num;i++)
    {
        NetRate[i]=BaseRate[i]*Rates->Discounts->Factors->Quantity
    }
```

此处将合适的名字赋给变量，并另配给复杂的指针表达式可以提高性能和可读性。

C 简化复杂指针表达式：

```
QuantityDiscounts=Rates->Discounts->Factors->Net;
for（i＝0；i＜Num;i++)
{
    NetRate[i]=BaseRate[i]*QuantityDiscount
}
```

特殊变量 QuantityDiscount，很清楚地使得数组 BaseRate 乘上一个数量因数，从而计算出网络比率，从循环中表达式看并不是那么清楚。若把复杂的指针表达式赋给循环外的变量，可防止每次运算循环时，指针都被三次引用。

语言	直接时间	调试时间	节省时间
C	9.56	8.29	13%
C++	8.51	8.40	1%
Pascal	10.44	10.33	1%

除了 C 编译器外，这种方法提高的效率是微不足道的，这提醒你们，开始设计代码时不必过多考虑执行速度，而应从可读性人手。

标志值

如果循环判断比较复杂的话，你可以简化判断句提高效率。如果循环是为了找数，一种方法就是使用标记值，把它安插在找数程序的末尾，并且保证终止找数检索。

关于使用标记值从而改善复杂测试，这里有一个典型例子，循环的检索部分检查是否找到标志值，判断是否偏离标志值。

C 检索循环内的复杂判断：

```
Found＝FALSE
i=0
while（(!found）＆＆（i＜ElementCount)）
    {
    if（Item[i]==TestValue）
```

```
            Found=True
        else
            i++;
        }
    if（Found）
……
```

这段中，循环对每一重！found 和 i<ElementCount 进行判断。！Found 判断是用来表示已找到所要的元素。i<ElementCount 是用来避免数组溢出。循环内，Item 的每一值都被分别测试，所以每执行一次循环，执行三次判断。

这种检索循环中，若你在检索段尾设标志来终止循环，并将三个判断合为一个，那么每循环一次就可作出判断。这时你再检查每一元素。如果头一个也正是末一个，你就知道了你要找的标志值并不存在。

C 用标志值加速循环：

```
/ * set sentinel valueprserving the original value* /
InitialValue=Item［ElementCount］;
Item[ElementCount]=TestValue;
i=0;
while（Item[i]！=TestValue）
      {
       i＋＋;
       }
  / * restore the value displaced by the sentinel * /
  Item［ElementCount］=Initialvalue;
  / * check if value was found * /
if（i＜ElementCount）
……
```

当 Item 是一整型数组时，省时效果是颇带戏剧性的：

语言	直接时间	调试时间	节省时间	性能比
C++	6.21	4.45	28%	1：1
Pascal	0.21	0.10	52%	1：2
Basic	6.65	0.60	91%	11：1

Basic 编译器的结果更是有趣，但结果都是好的。而当数组类型变了，结果也变了。下面是当 Item 是单精度浮点数时的结果：

语言	直接时间	调试时间	节省时间	性能比
C++	20.93	21.48	-3%	1：1
Pascal	21.42	21.97	-3%	1：1
Basic	59.84	30.07	49%	2：1

此时，结果变化十分显著，这再次表明，程序是否最优取决于具体的执行环境。

事实上在线性检索时，高标志技术通用于任意情形。要注意的是必须仔细选择标志值，及如何将其嵌入数组和程序中，记住如何保留原始值和替换它。

将最忙的循环放在里面

如果有多重循环，考虑好哪一个在内哪一个在外，下面是个如何改进多重循环的例子。

Pascal 改进多重循环：

```
For Column:=1 to 100 do
begin
for Now:=1 do 5 do
    begin
        sum：=sumTable[1000，column]
    end
end
```

改善程序的关键在于，外层循环要比内层执行项数多，循环每执行一次，先要初始化指针，每循环一次增加一，并随之检查计数。循环执行总数是外循环 100 次，内层循环 100*5＝500 次，总共是 600 次，单单切换内外循环，就能作到内层执行 500 次，外层只执行 5 次，这样总重复数量 505 次，从分析上，可望节省（600－505）/600=16％的计算量。

请看下面比较表：

语言	直接时间	调试时间	节省时间	性能比
Pasic	2.53	2.42	4%	1:1
Ada	2.14	2.03	5%	1:1
Fortran	1.97	3.57	-81%	1:2

Fortran 编译器重新设计循环使程序出现负增长效率 81％，既然 Fortran 直接执行时间在所有编译器中是短的，显然这种优化对 Fortran 语言不合适，这种矛盾的结果再次说明，需针对具体情形讨论程序的有效性。

降低运算强度

降低运算强度就是，将乘除等费时运算替换为加减。有时，循环内可能有一表达式，依赖于循环指针与某因子的乘积。因为加减比乘除通常要快，有时这样替换能更有效率，运行更快，例如：

Basic 循环指针乘法：

```
for i=1 to Num
    Commission(i)=i*Revenue*BaseCommission*Discount
next i
```

此段是顺序式的，但费时，你可以改写程序，使之将结果相加而非每次都用乘法。这样就减轻了运算强度。

Basic 加代替乘：

```
Increment=Revenue * BaseCommission * Discount
```

```
TotalCommission=Increment
for i＝1 to Num
    Commission（i）＝TotalCommission
    TotalCommission＝TotalCommission＋Increment
Next i
```

乘法是耗时的，用上述变形就带来某种意义上的优点。原代码程序中每增加一个 i，都要乘上 Revenue * BaseCommission * Discount，先是乘 1，后是加 2，再是 3，如此类推，优化后的程序则是令 Increment 等于 Revenue * BaseCommission * Discount，然后，每循环一次将 Increment 加到 TotolCommission 上。第一次循环只加一次，第二次加二次，第三次加三次，如此类推，其效果与第一例子是一样的，但省时多了。

关键在干，原乘法取决于循环指数，因为循环指数是唯一变化的表达式，所以表达式可以很经济地重写。请看下表：

语言	直接时间	调试时间	节省时间
Basic	3.85	2.85	26%
C	3.35	2.64	21%

29.2 逻 辑

许多程序运行是靠逻辑判断操纵的，这一节讲述怎么更好地操纵逻辑表达式。

当你知道答案时就停止判断

假定你有一个语句如下：

```
if（5＜x) and（x＜10)then…
```

一旦你已经确定 x 小于 5，就不需要再执行第二个判断了。

一些语言提供了一种改进的表达形式叫“短路改进”，它的意思是编译器产生代码，一旦知道答案就自动停止判断，短路判断是标准 C 的一部分，Ada 也支持带有关键词 and then 和 or else 的明确短路判断，前面的例子用短路版本写成如下形式：

```
if（5＜x)and then（x<10）then..
```

在另一些语言中，短路改进或是依靠编译器或是依靠你自己的判断，如果想要当答案已知时停止判断，你就必须避免用 and 和 or，取而代之的是增加一个逻辑判断，用短路判断上面的代码可换成如下代码：

```
if（5＜x＝  then
      if（x<10）  then ...
```

一旦答案已知就停止判断的原则在其它许多情况下也适用，一个搜索循环就是常见的例子，如果你正在查找输入数组中的负数，你只需要知道当前的数是否是负数，一种方法是检查每一个值，当你发现一个负数时，就设置一个 NegativeFound 的值，下面就是该搜索循环：

```
NegativeFound=False;
for（i＝0;i<Num;i＋＋）
{
```

```
    if（Input[i]＜0)
     {NegativeFound＝True}
  }
```

一个更好的方法是：你一发现负数就停止搜索，下面就是这种方法的几种实现：

- 在 NegativeFound=True 行后加一个 Break 语句，在 Ada 语言中用 exit 语句。
- 如果你用的语言没有 break，用 goto 代替 break 使程序执行循环体后的第一个语句。
- 将 for 循环语句改成 while 循环语句，同时检查 NegotiveFound 和循环计数是否超过 Num。
- 将 for 循环语言改成 while 循环语句，在原数组的最后一个元素后再加一个负数观察值，这样你只需在 While 简单地判断是否为负就行了，当循环体结束后，看一下负数的值是原数组中的元素，还是观察值，关于观察值在书中前面章节已经详细讨论了。

下面是 C 中用 break 语句和 Pascal 中用一个标准 while 语句的结果：

语言	直接时间	代码调整后时间	节省时间
C	2.03	1.48	27%
Pascal	2.97	2.69	9%

根据频率确定判断攻序

安排判断先后次序，使执行速度最快，逻辑值最可能为真的判断放在最前面执行，这种安排次序应该和正常情况相吻合，如果运行效率低，说明有例外现象。这个原则可以应用于 Case 语句和 if-then-else。

下面是用 case 语句编的键盘输入一个字的程序：

```
case （Inputchar）of
    '+',
    '=':        processMathSymbol(Inputchar)
    ',',
    '.',
    ':',
    '!',
    '?':        processPunctuation(Inputchar)
    '0'.. '9':  processDigit(Inputchar)
    ' ';        processSpace(Inputchar)
    'A'.. 'Z',
    'a'.. 'z';  processAlpha(Inputchar)
    else        processError(Inputchar)
    end
```

这个例子中 case 语句是按照最接近 ASCll 的分类次序来排列次序的，case 语句的执行结果和你写一大堆 if-then-else 语句相同，所以如果你用’a’作为输入字符，程序要先判断它是否是教学运算符号、标点符号、数字、空格符，最后才判断它是否是字母字符，如果你知道输入频率最高的字符，可以将最常见的情况放在最先判断，下面重新排列 case 语句的次序。

```
case（Inputchar）of
    ‘A’.. ‘Z’,
    ‘a’.. ‘z’:   processAlpha(Inputchar);
    ‘ ‘:         processSpace(Inputchar);
    ‘,’,
    ‘.’,
    ‘:’,
    ‘!’,
    ‘?’:         processPunctuation(Inputchar);
    ‘0’.. ‘9’:   processDigit(Inputchar);
    ‘+’,
    ‘=‘:         processMathsymbol(Inputchar);
    else         processError(Inputchar)
end
```

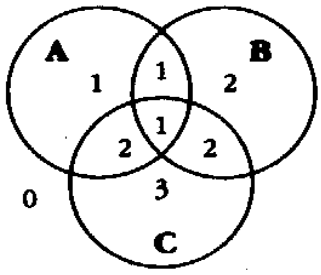
因为在优化代码中最常见的情况通常最快被判断，所以程序将执行少量的判断，下面是字符典型混合优化的结果：

语言	直接时间	代码调整后时间	节省时间
Pascal	3.13	1.97	37%
C	5.17	3.47	33%

注:混合输入的基准是 78％字母，17％空格符和 5％的标点符号。

查表法代替复杂判断

在一些情况下，查表法可能比沿着复杂的逻辑判断链执行更快。在复杂的判断链中，通常可将一些事物归成一类，然而按这种分类执行。作为一个抽象的例子，假设你想根据某一事情和 ABC 集合的不同关系，分配给它一个分类号。



下面是例子中分配分类号的复杂逻辑键：

```
if（（A&&!C）!!（A&&B&&C））
    Class=1;
else if((B&&!A))!!(A&&C&&!B))；
    Class=2;
```

```
else if((C&&!A&&!B))
        Class=3;
    else
        Class=0;
```

你可以用更易修改的、执行速度更快的查表法代替这些判断：

```
/*  define classTable */
Static int class Table [2][2][2]=;
    /*! B|C     B|C     !BC     BC              */
    {   0,       2,      3       2         /*!A */
        1,       1,      2,      1};       /*A   */
...
class=classTablel[A][B][C];
```

尽管表的定义非常难懂。但是如果它被更多地文件化后，它就一点也不比复杂的逻辑键代码难懂。如果规定变化了，这个表要比前面的逻辑判断容易维护，下面是执行结果：

语言	直接时间	代码调整后时间	节省时间	性能比
C	2.31	1.59	31%	1:1
Basic	58.88	23.00	60%	3:1

在这个例子中，查表法的另一个优点是所占空间小，在 C 语言中逻辑键法占 88 个字节，而查表法只占 43 个字节，包括表本身的空间在内。

偷懒改进法

我过去有一个同窗好友，他是一个大懒鬼。他为自己偷懒辩护说:人们急着做的许多事情是不需要做的，如果他把要办的事情拖得过久，他又说：这事不重要，应该拖到最后办，他不会浪费时间去做它的。

这偷懒改进法就是建立在我朋友所用的原则上，如果一程序用偷懒改进法，那么它就要避免作任何工作至到这个工作需要做时，偷懒改进法与时间判断策略相似，只有当接近需要时工作才开始做。

假设在一例子中，你的程序包含一个拥有 5000 个值的表，在开始阶段先建立整个表，然后当程序执行时调用它，如果程序只用表中很少的一点项目，更明智做法是表中哪部分项目需要时再计算它们，而不是一下子将表中所有项目都算出来，一个项目一旦被计算出来，将它存人表中以便将来使用。

29.3 数据转换

改变数据结构对于减小程序空间和提高执行速度可能是一个十分有力的帮助，数据结构设计已经超出了本书的讨论范围，但是执行特定数据结构的模式转换，可以提系统性能，下面是几种调整数据结构的方法。

尽量用整型数不用浮点数

在表 28-1 给出了常用操作占用机时的数据。整型数加法乘法运算总比浮点数快得多，至少对于浮点操作在软件中实现而不是在硬件中实现是这个样子的。将循环指针由浮点型变为整型的例子，这种调整可以节省时间，对许多语言浮点操作不会带来问题，但是在某些语言中，像 Basic，所有变量都自动定义成单精度整型数。这时对它们进行明确地整型说明会对程序有帮助。下面是一例子：

```
for i=1 to 100
   x(i)=0
next i
```

除非“i”被说明是一个整形变量，在 Basic 中，它是一个单精度浮点数,比较下面相似的 Basic 循环程序。它明确地用了整型标志。

```
for i%=1 to 100
     x(i%)＝0
next i%
```

这样做有多大不同，下面是这个 Basic 代码和功能相似的 Fortran 代码的执行结果：

语言	直接时间	代码调整后时间	节省时间	性能比
Basic	5.54	0.94	83%	6:1
Fortran	5.51	0.93	83%	6:l

尽量减少数组维数

通常多维数组占机时多，如果你能建立你的数据结构，使其成为一维数组而不是二维数组或三维数组，你就可能节省时间。

假设有一个如下的初始化代码：

```
for Row:=1 to NumRows do
     begin
     for Col:=lto NumRows do
          begin
          Matrix[Row，col]:＝0
          end;
end;
```

这个代码是运行 50 行 20 列，它花费 3.32 毫秒，但是，如果将这数组重新建立结构，用一维数组代替原来的二维数组，它只运行 2.28 毫秒，下面是修改后的代码：

```
for Entry:=1 to NumRows * NumCols do
     begin
     Matrix[Entry]:=0
     end;
```

下面是运行结果，附加 C 和 C++语言的结果比较：

语言	直接时间	代码调整后时间	节省时间	性能
Pascal	6.64	4.55	32%	4:3
C	7.57	5.71	25%	5:4
C++	4.29	7.69	-79%	1:2

改进结果 Pascal 和 C 是好的，C++却更糟。你可能想知道负的节省是否是由 C++编译器的一些缺点所致的，但是原先未优化的 C++代码比其它两程序编译器优化的代码还快，所以你不可能再对它有大的改进了。可能是你的代码调整妨碍了编译器的自动优化，在这个例子中编译器优化似乎比手工更有效。

这个结果说明，盲目听从任何一种代码调整建议都是危险的，在你没有在自己特定的环境下尝试过建议前，你不能做出任何肯定。

将二维数组转换为一维数组这个特殊技术的危险是：计算 NumRows * NumCols 可能无意识地引起 Entry 指针溢出，你可以用定义 Entry 为长整型量的方法解决问题，它将避免了溢出问题。但是这样改变后对性能可能起相反的影响。下面就是你将 Entry 由整型变换成长整型引起的性能变化：

语言	原先时间	Entry 为整型的时间	Entry 为长整型的时间	总共节省时间
Pascal	6.64	4.55	7.25	-9%
C	6.57	5.71	15.43	-104%
C++	4.29	7.69	9.62	-124%

转换成长整型后，性能下降范围从 9%到 124%，并且在这些例子中，没有一个优化比原先二维数组快。

减少数组访问

另一方面减少访问二维或三数组有利于减少数组访问次数，同样，循环反复用数组中的元素也适合用这种方法改进，下面是不必要的数组访问的例子：

```
for(DiscountLevel=0;Discountlevel<NumLevels;Discountlevel++)
{
    for(RateIdx=0;RateIdx<NumRates;RateIdx++)
    {
    Rate[Rateldx]=Rate[Rateldx]*Discount[DiscountLevel];
    }
}
```

当 RateIdx 在内循环中变化时，Discount[Discountlevel]不会改变，你可将 Discount[Discountlevel]移出内循环，这样你每执行一次外循环就只有一次数组访问，而不是每执行一次内循环就有一次访问。下个例子是修改过的代码：

```
for (discountlevel=0;DiscountLevel<NumLevels;DiscountLevel++)
    {
    ThisDiscount=Discount[DiscountLevel];
    for (RateIdx=0;RateIdx<NumRates;RateIdx++)
        {
```

```
            Rate[RateIdx]=Rate[RatIdx]*ThisDiscount;
        }
    }
```

结果如下：

语言	直接时间	代码调整后时间	节省时间
C	16.09	13.51	16%
C++	15.16	15.16	0%
Basic	31.09	21.37	31%

再一次看到，不同的编译器产生的结果截然不同。

运用辅助索引

运用辅助索引就是加上一个能更加有效地访问数据结构的相关数据，你可以把相关数据加到主数据中，也可以存入并行结构中。

串长度索引

在C语言中，串是以置0的字节作为结尾标志的。在Pascal中，长度字节放在每一串的开头，指示出本串的长度；确定C中一个串的长度，程序必须从每一串的开头开始数起至到发现置0字节为止；在Pascal中，程序只要看长度字节就可以。Pascal中的长度字节就是我们本节讨论的带索引数据结构的一个例子，在特定操作中，如计算串长度，它能加快速度。

你可以把长度索引的概念应用到任何具有可变长度的数据结构中去，它通常能比每一次你需要时都计算长度更有效地跟踪数据长度。

对于单向顺序表的索引

如果你想在单向顺序表中找到一个特定入口，你必须从表的起点开始检查每一个字符，直到发现你所需要的那一个，如果表很长，有1000个字符，平均说来，你在找到你想要的字符之前必须检查一半的字符。

如果性能要求很高，你可以将这个已链接的表和另一个小的已链接的表连在一起。另一个表作为主表的索引表，假设索引表有10个入口，第一个入口指向主表的起始段，第二个入口指向主表前十分之一内容，第三个入口十分之二，第四个入口十分之三……依此类推。当你寻找已链接主表中的一个字符时，先进入索引表，在你发现你要找的值处在主表的哪一个段时，你先要检查索引表中的入口，然后在确定的十分之一段中检查。平均要检查这一段的一个入口即可找要你想要的值。

如果是1000字长的表，用这种10字符扩充法，你平均要检查的字符数量从500减少到55，这种扩充法虽然增加插入字符的平均所需时间，但是你必须在搜索时间、插入时间和存储时间三者间折衷。

这种思想对于已链接的表，甚于磁盘数据访问，都是非常有用的，相似的表被用在存储B树上。

独立的并行索引结构

有时，操纵数据结构的索引比直接操纵数据结构本身更有效。如果数据结构中的项目太大或难移动(也许在磁盘上)，分类和搜索索引参考比直接用数据工作更快。如果每一个数据项目都很大。你可以产生一个辅助结构，它由描述详细信息的关键值和字符组成。你也可以将关键

字项存入内存中，所有的寻找，分类在内存中完成，当你想知道你要找的数据项的确切地址时，只需访问磁盘一次。

高速缓存运用

高速缓(冲)存储就是用以下原则存入一些值。最常用的值比不常用的值更容易获得。如果以一个程序随机地从磁盘上读取记录为例，高速缓存中存储读取频率最高的记录，当程序收到要求读记录的命令时，它先检查高速缓存区看是否有这个记录，如果有，记录就被从内存中直接返回，而不是从磁盘中返回。

除了可以缓存磁盘上的记录外，你还可以用此方法缓存其它区域。在 Microsoft Windows font-proofing 程序中，当这个字被显示时，高速缓冲检查近期最常用的字，大约可提高显示速度两倍。JonBentley 报告说：一个拼写修正程序，缓存了 1000 个最常用的字在内存中，程序在磁盘中收集一个 500000 字的字典，但是内存中的高速缓存使磁盘存取很少用了。

高速缓冲存取还可以节省计算时间，尤其对于参数计算简单情况，例如假设已知直角三角形两个直角边，求斜边的计算。直接运行程序如下：

```
function Hypotenuse
    (
    sideA:real;
    sideB:real
    ):real;
begin
    Hypotenuse:=Sqrt(SideA*SideA+sideB*sideB);
end;
```

如果你知道相同的值将要反复被调用，你可以用下述方法缓存，

```
function Hypotenuse
    (
    sideA:real;
    sideB:real
    ):real;
const
    cachedHypotenuse: real=0;
    cachedsideA:      real=0;
    cachedsideB:      real=0;
begin
    {check to see if the triangle is already in The cache}
    if((sideA=eachedsideA)and
        (sideB=cachedsideB))then
    begin
        Hypotenuse:=cachedHypoteuse;
    exit;
    end;
```

```
    {compute new hypotenuse and cache it}
     cachedHypotenuse:=sqrt(sideA*sideA+sideB*sideB);
     cachesideA       :=sideA;
     cachesideB       :=sideB;
     Hypotenuse       :=CachedHypotenuse;
end;
```

第二种程序版本比第一种复杂，所占空间也多，它以提高速度作为代价补偿。许多高速缓存计划不止存入一个元素，所以它们有可能花费更多的时间。下面出示两种版本的速度差异。

语言	直接时间	调整后时间	节省时间
Pascal	2.63	2.25	14%
C	1.99	1.93	3%
Basic	1.75	2.52	-44%

这再一次显示了不同编译器带来的巨大差异。成功地应用缓存取决于访问用缓存元素所需相对时间，产生未缓存元素的相对时间和存入一个新元素到缓存区所需相对时间，还取决于缓存信息的使用频率。在某些情况下，成功还可能取决于执行高速缓存的硬件。一般说来，产生一个新元素的时间越长，相同信息使用次数越多，高速缓存的价值越大。访问高速缓存区的元素和存入新元素到高速缓区所用时间越少，高速缓存价值越大。

29.4 表达式

程序中的部分工作是花在算术和逻辑表达式中。复杂的表达式往往要占用很多机时，所以这一节讨论减少其运行时间的方法。

充分利用数学中的等价关系

你可以根据数学上的等价关系，用一个占机少的表达式来代替一个占机时多的表达式。例如下面表达式在逻辑上是等价的。

```
not A and not B
not(A or B)
```

如果你用第二个表达式代替第一个，你就可以节一个 not 操作，如果你能判断，例如，在 Pascal 中用 repeat-until 循环代替 while-do 循环，你就可以从第二个选择中消去 not 这样就省了两个 not。

尽管避免一个 not 操作节省的时间微不足道，但是这种通用的原则却是十分有效的，JonBentley 描述 sqrt(x)<sqrt(y)的例子，当且仅当 sqrt(x)<sqrt(y)时，你可以用 **x<y** 代替第一个判断。如果 sqrt()程序占机时很多，那么你节省的时间是惊人的。下面是两者结果。

语言	直接时间	代码调整后时间	节省时间	性能比
C	3.63	0.71	80%	5:1
Basic	2.37	0.94	60%	3:1

运用强度削减

正如前面提到的，强度削减就是用一个执行时间少的程序代替一个执行时间多的程序。下面是几种可能的替换：

- 用加法代替乘法
- 用乘法代替指数运算
- 用等价三角函数代替三角函数子程序
- 用短整型代替长整型
- 用定点数浮点数代替双精度浮点数
- 用单精度浮点数代替双精度浮点数
- 用再次移位操作代替整型乘除法

下面是一个详细的例子，假设你计算一个多项式，多项式形式如下：

$$ax^2 + bx + c$$

字母 a、b、c 是系数，x 是变量。一般的计算 N 阶多项式程序如下所示：

```
value=Coefficient(0)
for power=l to order
    Value=Value+cofficient(power)*x^power
next power
```

如果你正在考虑强度削减，你将会感到指数操作很刺眼，一种解决方法是用乘法代替指数运算，在每次循环中，这种方法和前几节强度削减例子中用加法代替乘法相似，下面是用强度削减后的程序：

```
Value=Coefficient(0)
PowerOfx=x
for Power=1 to Order
    Value=Value+Coefficient(Power)*PowerOfx
    PowerOfx=PowerOfx*x
next power
```

如果你要算一个二阶多项式，多项式最高幂次项是平方项或更高阶多项式，这种方法产生了巨大改进：

语言	直接时间	代码调整后时间	节省时间	性能比
Basic	25.43	3.24	87%	8:1
Ada	28.72	11.42	60%	3:1

如果你对强度削减要求很严格，你对两个浮点乘法仍会感到不满意，你可以通过变量累加次数代替每一次相乘来进一步强度削减。下面是程序：

```
Value=0
for Power=Order to l step -1
    Value=Value+Coefficient(power)*x
next power
Value=Value+Cofficient(0)
```

这种方程消去了外加的变量 PowerOfx。并且在每一次循环中用一个乘法代替两个乘法，这一个程序相当快，但是看起来不直接：

语言	直接时间	第一次优化	第二次优化	比第一次优化节省时间
Basic	25.43	3.24	1.98	39%
Ada	28.72	11.42	6.97	39%

从这个例子中得到的启发是：第一次优化后不要停止，即使第一次产生了显著效果，第二次优化会更好。

编译期间初始化

如果你正在使用一个已被命名的常数，或是子程序调用中产生的数。如果这个数可预先算出，就把它放入常量以避免调用子程序。同样的原则适用于乘法、除法、加法和其它操作。

我曾经要计算一个以 2 为底的整数的对数，结果用截尾法代替整数，系统中没有以 2 为底对数子程序，所以我只能自己编一个，最快最容易的方法是用下面公式：

log(x)base=log(x)/log(base)

利用这个等价性我可以写一个程序：

```
unsigned int log2(unsigned int x)
    {
    return(unsigned int)(1og(x)/log(2));
    }
```

这个程序执行相当慢，因为 log(2)的值一直不变，我用计算好的值 0. 69314718 代替 log(2)，修改的程序是：

```
unsigned int log2(unsigned int x)
{
return((unsigned int)(log(x)/LOG2)); —LOG2 已为 0.6931478 命名常量
}
```

因为 Log()程序总是要占很多机时，比类型转换和除法所占机时多得多，你可以猜想除去一半的 log()函数，可以减少一半程序所需时间，下面是测量结果：

语言	直接时间	代码调整后时间	节省时间
C	29.00	21.74	25%
Fortran	43.06	32.95	23%

节省时间不像想象的那么多，显然是因为一个浮点除法和两个类型转换冲淡了优化效果，根据经验关于除法和类型转换的重要性，以及 50%的估计显然错了，在优化时，根据经验的估计常常出错。

密切注意系统程序

系统程序执行时间长并且提供的精度常常过高。经系统运算的程序，它的设计精度可以把宇航员送到目标上离目标只差士 2 英尺的范围内，如果你不需要这么高的精度，你也不需要花费这么长时间计算它。

在先前例子中，log()程序返回一个整型值，但是却用浮点函数 log()程序计算它，这远远地超过了整型结果的要求，所以在我第一次改进后我又写了一系列整型判断，它们对于计算一个整型的 log2 是足够精确了，下面是程序：

```
unsigned int log2(unsigned int x)
{
    if(x<2)         return(0);
    if(x<4)         return(1);
    if(x<8)         return(2);
    if(x<16)        return(3);
    if(x<32)        return(4);
    if(x<64)        return(5);
    if(x<128)           return(6);
    if(x<256)           return(7);
    if(x<512)           return(8);
    if(x<1024)          return(9);
    if(x<2048)          return(10);
    if(x<4096)          return(11);
    if(x<8192)          return(12);
    if(x<16384)         return(13);
    if(x<32768)         return(14);
    retun(15);
}
```

这个程序用整型操作，省去了类型转换，结果如下：

语言	直接时间	代码调整后时间	节省时间	性能比
C	29.00	0.15	99.5%	200:1
Fortran	43.04	0.44	98.9%	100:1

大多数的转换是按最坏情况设计的，即它们将变量在程序内部变成双精度浮点数。即使你给它们的是一个整型变量，如果你发现一个和程序紧密相联，却又不需要很高的精度，这时你要立即注意它。

另一种选择是利用右移字节操作相当于被 2 除的性质，一个数被 2 除并且商非 0 的次数就这个数 log2 的值，下面是根据这个原理编的程序.

```
unsigned int log2(unsigned int x)
{
unsigned int i=0；
while((x=(x>>1))!=0)
    {
    i++;
    }
return(i);
}
```

对不懂 C 语言的程序员来说，这程序是不易读的。while 条件是一个复杂的表达式，这是一个编程实践的例子，你平时应尽量避免这样用，除非你有充分的理由这样用。

这个程序运行了 0.21 秒，比上一例 0.15 秒多用了 40%的时间。但是它只占用了六分之一的内存，所以在某些情况下你更愿用它。

这个例子想要强调的重点的是：在一次成功地优化后不要停止。第一次优化节省 25%时间的显著改进，但是比起第二次或第三次优化成绩来相差很远，另一个值得注意的是，要在速度和内存之间寻求折衷。

正确使用常数类型

使用被命名的常数和文字，要求和它们被分配的变量保持一致，当常数和与其相对应的变量不一致时，编译器就得做类型转化，将常数类型转变成变量类型。一个好的编译器在编译时就进行类型转变，所以它对运行时间没有影响。

然而一个低级的编译器或解释器，在运行时产生一个类型变换代码，所以你应该注意，下面例子是在初始化浮点变量 x 和整型变量 i 两种情况下的不同。第一种情况如下：

x＝5

i＝3.14

需要类型转换。第二种情况是：

x＝3.14

i＝5

不需要类型转换。最后结果是:

语言	直接时间	代码调整后时间	节省时间	性能比
编译 Basic	5.38	5.38	0%	1:1
解释 Basic	15.27	6.42	58%	2:1
C	20.65	5.45	74%	4:1
C++	4.94	4.94	0%	1:1
Fortran	3.07	3.07	0%	1:1

除了一个编译器外，其它编译器都没有节省时间，Basic 的解释器时间改进了。这些结果都不令人吃惊，令人吃惊的是 C 语言编译器，当前的第二代流行产品，却做了极坏的工作，这表明每一个编译器都有其特定的优点和缺点，有时缺点是令人惊讶的。

预先计算结果

一个普通低级的设计决策是，选择在计算结果分步计算，还是一次将所有结果计算出来，存好，需要用时再查表。如果这个结果要被使用许多次，常常是一次将它们计算出来，因为查表寻找它们更省时间。

这个选择可以用几种方法清楚地表示它自己。对于最简单的情况，你可以在循环计算表达式的一部分而不是在循环体内。前面章节已经出现这样的例子，对于最复杂的情况，你可以在程序执行开始前，先一次性计算一个查寻表，或者你也可以将结果存在一个数据文件中或深深嵌入程序中。

以空间战争录像游戏为例，程序员要计算距太阳的不同距离，所以程序员可以预先计算出

几个重力因数，然后将他们存入一个 10 元素数组，数组查寻比复杂的计算快得多。

假设你有一个计算支付汽车贷款的程序，其代码如下：

```
function ComputePayment
    (
    LoanAmount      :Longint;
    Months          :integer;
    InterestRate    :real;
    ):real;
begin
    payment：=LoanAmount/
            (
            (1.0-Power(1.0+(InterestRate/12.0),Months))/
            (InterestRate/12.0)
            )
end;
```

计算贷款支付的分工是复杂的，并且很耗费时间。将这些信息放在一个表中，而不必每次都去计算他们，这样做可能会省时间。

那么表究竟要做多大？LoanAmount 是变化范围最大的变量。变量 InterestRate 的范围是从 5％到 20％，间隔步长为 0.25%，这样只需要 61 个不同的利率，Months 的范围从 12 到 72，需要 61 个不同的期限，LoanAmount 可能范围是从$1000 到$10,000。它比你一般要查的表需要更多的入口。

大部分的计算过程不依靠 LoanAmount，所以你可以将计算中最复杂的部分（大表达式中的分母部分放在表中，表通过变量 InterestRate 和 Months 查寻）。每一次都计算 LoanAmount 部分。下面是修改程序：

```
function ComputePayment
    (
    LoanAmount      :longint;
    Months          :integer;
    InterestRate    :real;
    ):real;
var
    InterstIdex:Integer;
begin
    lnerestldx:=
        Round((InterestRate-LOWEST_RATE)*GRANULARITY * 100.00);
    payment:=LoanAmount/LoanDivsor[InterstIdx，Months]
end;
```

在这个代码中，复杂的计算被一个数组索引和一数组访问所代替，下面是变化的结果：

根据你的环境，你将需要计算 LoanDivisor 数组，在程序初始时或从磁盘读取它，一种选择是你可能先将它初始值设为 0，每个元素只有当需要时才第一次计算它，然后将它存好以便以

后需要时可以查询，这就是前面讲的高速缓存形式。

语言	直接时间	代码调整后时间	节省时间	性能比
Pasal	3.13	0.82	74%	4:l
Ada	9.39	2.09	78%	4:1

你不一定非用表的形式提高速度。通过预先计算一个表达式同样可以达到这个目的，这个程序与前面例子中的程序相似，但它增加不同种类预先计算的可能性。假设你有一个计算贷款支付的程序如下：

```
function ComputePayments
    (
    Months:          integer;
    InterestRate：   real;
    )：real；
var
    LoanAmount:longint;
begin
    for LoanAmount:＝MIN_LOAN_AMOUNT to MAX_LOAN_AMOUNT do
        begin
            Payment: ＝LoanAmount／
                    (
                    (1.0－Power(1.0＋(InterestRate／12.0)-Months))／
                    (InterestRate/12.0）
                    );
     …
     end;
end;
```

即使没有预先计算表，你也可以在循环体外予先计算运算中的复杂部分，然后在循环体中用它，下面是它的程序：

```
function   ComputePayments
    (
    Months：          integer;
    InterestrRate: real;
    ): real；
var
    LoanAmount：Iongint;
    Divisor:real;
begin
    Divisor:=（1.0+(InterestRate/12.0),-Months))/
            (InterestRate/12.0);
    for LoanAmount:=MIN_ LOAN_ AMOUNT to MAX_ LOAN_ AMOUNT do
        begin
        Payment：= LoanAmount/Divisor;

 ...
```

```
        end;
    end;
```

这和前面介绍过的放置数组变量到循环体外的技术相似。这种方法的计算与第一次优化中用预先计算表的方法比较结果如下：

语言	开始时间	代码调整后时间	节省时间	性能比
Pascal	3.51	0.77	78%	5:1
Ada	10.33	1.70	84%	6:1

通过预先计算法优化程序可采用如下几种形式：

- 在程序执行前计算出结果，并将它们写成常量形式。
- 在程序执行前计算出结果，并将它们放入变量中。
- 在程序执行前计算出结果，并将它们放入一个文件中。
- 在程序起始段一次性计算出结果，以后每次需要时可以参考。
- 尽可能地在循环体外多计算，减少循环体内的工作量。
- 只在你需要时才第一次计算它们，并存储起来以便以后用时可以重新获得。

消除常用的子表达式

如果你发现某个表达式重复好几次，可分配给它一个变量，然后用变量代替重复计算的表达式。在贷款计算例子中就有可以消除的常用的子表达式，原程序如下：

```
payment：=LoanAmount/
            (
            (l.0-Power(l.0+(InterestRate/12.0),-Months)) /
            (InterestRate / 12.0）
            );
```

在这个例子中, 你可以分配给 InterestRate/12.0 一个变量，这样，程序使用两次变量代替计算两次表达式。如果你给变量起一个好的名字. 这样优化不仅增强了可读性，同时也改进系统性能，下一个例子是修改后的代码：

```
MonthlyInterest:=InterestRate/12.0
Payment:=LoanAmount/
            (
            (1.0－Power（1.0+Monthlylnterest，－Months）） /
            MonthlyInterest
            );
```

这个例子中，节省时间效果似乎不显著。

语言	直接时间	代码调很后时间	节省时间
Pascal	3.13	3.08	2%
Ada	9.56	9.33	2%

这里 Power()程序占据了大部分时间，以致掩盖了从消除了表达式中节省的时间。在其它例

子中，如果子表达式占据了整个表达式的更大部分的计算时间，这种优化效果将更加显著。

并不是所有子表达式的消除都产生相同的结果，例如，假设你想要消除了面代码结构中Mtx[InsertPos-1]表达式：

```
for（Boundary＝1,Boundary<NUM;Boundary＋＋）
 {
      InsertPos=Boundary;
      TestVal=Mtx[InsetPos-1];
      While(Insertpos＞0&&Mtx[InsertPos]＜TestVal）
           {
           SwapVal              ＝Mtx［InsertPos]
           Mtx［InsertPos]      ＝Mtx［InsertPos-1]
           Mtx［InsertPos-l]    ＝SwapVal;
           InsertPos --;
 }
```

你可以用两种方法代替这个常用的表达式，常用的方法是分配一个值，用这个值代替它。在这个例子中你可以用 TestVal 代替 Mix［InserPos-1］这个代码如下所示：

```
for(Boundary-1;Boundary＜NUM;Boundary＋＋）
  {
  InsertPos= Boundary;
  TestVal=Mtx[InsertPos-1];
  While(InsertPos＞)&&Mtx[InsertPos]<TestVal)
       {
       WwapVal                    ＝Mtx［InsertPos］;
       Mtx[InsertPos]             ＝TestVal;
       Mtx[InsertPos-1]           ＝SwaPVal;
       InsertPos--;
       TestVal＝Mtx[InsertPos-1];
       }
  }
```

这种方法在每次循环时只消除了一个数组变量和一次 InsertPos-1 计算，结果并不理想。

语言	直接时间	代码调整后时间	节省时间
C	1.53	1.58	-3%
Ada	1.43	1.59	-11％

这种方法实际上损害了系统的性能。但是 C 语言提供了另一种类型的指针，下面程序就是如何用一个指针 TestValPtr，去代替常用于表达式：

```
for(Boundary＝1;Boundary＜Num;Boundary＋＋）
  {
  InsertPos＝Boundary;
  TestValPtr＝& Mtx[InsertPos-1]

  while（InsertPos＞0&&Mtx［InsetPos］＜*TestValPtr）
```

```
        {
        SwapVal                 ＝Mtx［InsertPosj;
        Mtx[Insertpos]          ＝*TestValPtr;
        *TestValPtr             ＝SwapVal;
        InsetPos--;
        TestValPtr--;
        }
    }
```

这种方法每次循环中消除了三个数组变量和三个 InsertPos－1 的计算，结果也相应得到改进。

语言	直接时间	代码调整后时间	节省时间
C	1.53	1.33	13%

29.5 程 序

在代码调整中,最强有力的手段之一是好的程序分解。小的、明确的程序可以节省空间，因为它们把将要做的工作独立地放到多个地方。它们使程序更容易优化，因为你可以调用每个小程序。小程序相对地在汇编中容易被重写,长的、拐弯抹角编成的程序很难被理解，汇编也是不可能的。对于一个明确的任务，你可以从卖主手中买一个程序，因为他们可能比你更有时间去加工这个程序。

写子程序

在早年的计算机编程中，有些机器对于使用子程序的操作有惩罚性限制。调用子程序着操作系统得脱离程序转入子程序，转入要调用的子程序执行后，再重返调用它的原来程序。所有这些转换者要消耗资源使程序运行减慢。现代计算机——指的就是你现在使用的那种——对于调用子程序没有任何惩罚。而且，由于在程序中增加代码使得程序规模扩大，以及机器内存有限而附加切换次数增加，已经有研究表明使用较多的直接代码要比用调用子程序的耗时大些。

有些情况下，你可能将某段程序直接插入大程序而节省几毫秒。如果你用的机器是带微处理器的，你可以使用宏来控制代码。下面是这种审拷贝的程序的比较：

语言	直接时间	代码调整后时间	节省时间
C	1.81	1.65	9%
Pascal	1.20	1.04	13%

表中显示的增加额要比你通常所见的高些，由拷贝是简单的，所以子程序调用的要比其它运算方式好。每段子程序所做工作量越小，此法的好处也就越多。如果子程序的工作量较大，别指望会有什么大的捞头。

29.6 汇编语言再编码

一个不应不提的传统的经验是，当你的工作进入瓶颈口而进退两难时，就应考虑用汇编语言重新编码。汇编可以改进速度及代码空间。下面是优化程序使用汇编语言的典型方法：

1. 用 100%的高级语言写出程序。
2. 测试着是否满足要求。
3. 如果需要提高性能，分析源程序找出弱点。因为有 5%左右的程序通常消耗整个程序运行所需时间 50%以上，通常你可找出一些小片段。
4. 把这些小片段用汇编语言重编码以提高程序性能。

是否采纳上述步骤取决于你对自己高级语言编的程序的失望程度和你使用汇编语言的熟练程度。

我第一次提示汇编程序，是在我前几章曾提到的 DES 加密程序。我尝试了能想到的所有优化方法，然而程序运行速度仍比预想的低二倍。将某部分用汇编语言重编码怕是唯一的选择了。作为汇编语言新手，我只会从高级语言往汇编语言直译，但我仅刚起步，就赢得了 50%的程序速度的提高。

假设你有一个子程序用调制解调器将 H 进制码转换成大写的 ASCll 码。下面是一个 Pascal 的程序：

Pascal 用汇编语言提高质量:

```
 proedure HexGrow
     (
     var Source:        ByteArray;
     var Target:        WordArray;
              num:       Word;
     );
var
     Index:       integer;
     LOWerbyte：byte;
     UPPerByte：byte;
     Tgtlndex：  integer;
begin
     Tgtlndex：＝l;
     for Indes：=1 toNum do
          begin
          Targst［Tgtlndex]：       =((source［Index]and ＄FO）shr4）＋＄41;
          Target[Tgtlndex＋1⌋：    =(source[Index]and ＄ Of）＋＄41;
          Tgtlndex：= Tglndex＋2;
          end;
  end;
```

尽管看不出哪些比较臃肿，由于包含较多位操作，运行起来却很慢，体现不出 Pascal 的先

进，而位操作是汇编语言的专长，所以这段程序就成了重编码的好例子。

Pascal 用汇编语言编码的例子：

```
HexExpand       PROC    NEAR
        MOV     BX,SP           ；取栈指针
        Mov     CX,SS:[nx＋z]   ；取出字节数待扩展
        Mov     SI,SS:[BX＋8]   ；装入源偏移量
        MOV     DI,SS[BX＋4；]  ；装入目标偏移量
        XOR     XX,AX           ；数据偏移量清零
EXPAND:
        MOV     BX,AX           ；数组偏移量
        MOV     DL,DS[SI＋BX]   ；取源字节
        MOV     DH,DL           ；拷贝源字节
        AND     DH,15           ；取 msb
        ADD     DH,65           ；加 65 为调制解调器准备
        SHR     DL,l            ；传送 1sb 到位置
        SHR     DL,l            ；”
        SHR     DL,1            ；”
        SHR     DL,l            ；”
        AND     PL,15           ；取 lsb
        ADD     DL,65           ；加 65 为调制解调器准备
        SHL     BX,l            ；为目标数组指针留双倍偏离
        MOV     DS:[DI＋BX],DX  ；放目标字
        INC     AX              ；增加数组偏移
        L00P    EXPAND          ；重复（循环）直到完成
        RET     10              ；栈弹出并返回
HexExpand       ENDP
```

在这个例子里用汇编语言重新编码是成功的，能节省时间 65%。但如果原来的程序是 C 编的，C 本来是适合位操作的，重编码后就不像用 Pascal 那么显著了。

语言	高级语言时间	汇编语言时间	节省时间	性能比
Pascal	1.59	0.55	65%	3:1
C	0.83	0.38	54%	2:l

用汇编语言重写，不一定就弄出庞大难看的子程序来，有时候这些子程序通常是很朴素的，有时，汇编码几乎与高级语言一样紧凑。

用汇编语言重编码的一个相对简单有效的策略是，使用能产生汇编目录清单的编译器。抽取你需要调试的汇编码，存在一个单独的源文件里。使用编译器的汇编码作基础，手工地优化代码，每一步都要检查以便测量改进。有的编译器有将高级语言语句理解成汇编码的注释。如果你用的也是，将它们在汇编程序中当作文本保留。

29.7 调试技术快速参考

根据具体环境，你可能在自己的代码中选下列表中的某一项：

表 28-1 代码调试技术概括

	技术
	提高速度与规模
	冲突的循环
	用表查寻替代复杂逻辑
	用整型变量替代浮点变量
	在编译时初始化数据
	使用正确的常量
	预先计算结果
	消除常用子表达式
	将关键子程序翻成汇编语言
	提高速度
	在循环中不要包含 if 判断时使用反切换
	循环展开
	最小化内循环工作量
	在检索循环中使用标志
	将最忙的循环放入嵌套内层
	降低内层循环工作的运界强度
	如果可知结果停止判断
	将 case 与 if then else 判断顺序合理安排
	按频率划分
	使用索引
	变多维数组为一纸数组
	使数组参考量最少
	用索引指示增加数据结构
	将常用值高速缓存
	找等价关系式
	减少逻辑与数学运界强度
	注意系统子程序
	写子程序

29.8 小 结

- 优化的结果因不同语言、编译器和环境而大不相同。如果不对特定的优化进行测试，没人知道结果会怎样。
- 头一次优化未必是最好的。即使你找到了一个好的，试着寻找更完美的。
- 代码调试有点像原子能，是一个有争议的题目，有些人认为它对可靠性和可维护性是有害的，根本不该使用。有些人认为只要注意，它是有益的。如果你决定要使用本章描述的技术，千万小心从事。

第三十章　软件优化

目录

相关章节

高质量程序：见第 5 章
高水平设计：见第 7 章
全局数据：见 10.6 节
评审：见第 24 章
单元测试：见第 25 章

设想：一个精心策划的软件需要细心的编程分析才能确保运算的稳定性。设计跟着需求走，它必须仔细，使程序从头到尾顺利执行，这也意味着许多程序代码不得不忍痛割爱。按这种设想，程序作大改动的时候主要是在程序维护阶段，而有些修改则是在初始版本系统交付后进行的。

现实：程序在设计过程会有较大改动，从初始设计时的改动到程序维护时的改动几乎是差不多大的。编程，调试及单元测试要花费 25%～70%的工作量，这也取决于程序的规模。如果编程和单元测试是顺次进行的，它们只占总工程量的 20%左右。不过即使是规模良好的软件工程，软件从设计之初到结束也会有约 25%的设计要求上的改变，指标决定编程，有时两者联系相当紧。

现实：现代程序设计提高了改变程序结构的潜在可能性。传统的设计过程中，中心问题是要避免修改代码，这是保证成功的标志。但是现代方法中，设计及开发都成为难以预料的了。新方法以代码程序为中心。在程序设计完了，也可能会有大篇幅的修改。

30.1　软件进化种类

软件的进化和生物进化有相似之处，有些改变是有益的，可有些就不是，好的软件进化过程就像从猴子到古人猿，再到我们引以为荣的现代程序员。有时这种进化的力量设备也会帮了倒忙。

软件进化最清楚显著的标志是修改后的程序是更有效了呢还是质量下降？如果你认为修错是陷跳，质量就会下降。如果你将修改视作增进原始设计的机会，质量才能提高，当你看到软件质量下降了，说明软件进化的方向搞错了。

另一显著特征是:软件进化所作的调整是结构性的还是维护性。这两种类型侧重点是不同的。

结构调整通常由开发者完成，从而避免程序惨遭遗弃。系统还不是在线的，所以面临的压力只是进度上的——这时还不会出现 500 个愤怒的用户指责系统不好用的景象，同理，结构调整就相对自由些，系统还在动态调整中犯错误而受惩罚的可能不大。这些暗示着一种与你进行的软件维护类型不同的进化经历。

30.2 软件优化指南

程序员在软件优化中有个通病，就像身在山中，当局者迷一样。如果你能意识到优化是必不可少且很重要的，你就应当试着用它。软件优化的中心规则是提高程序的内在质量，下面是几条原则:

多设计子程序

提高程序质量的一条标准就是模块化；多设计些好的子程序。程序中有变化时，模块化就方便得多。如果程序中某一段编成子模块会使之更清晰，就再划出子模块。这一点在 30.3 节“编制新子程序”中详细讨论。

减少全局变量

当你用到程序中的全局变量时，注意检查一下它们。在编过某段程序后，你可能想出避免再用到某全局变量的方法。在刚开始设计程序时，可能你容易弄混全局变量；这样，你希望用更简洁的方案。你也可能因为忘记分离全局变量而备受其苦，应该将它们局限在分段的程序中，吃一堑长一智吧。只要你记得那些应该修订的部分就行了。应指出修改程序应在最初几版进行，这时最合适。

改进你的编程风格

当你回过头来修改时，正是整理变量、旧布局注释的好机会，此时记住把一切好的编程风格都用上。

改变管理

编程中你可能遇到新的要求及改错方案，使得你得修改代码。如果你改完程序而跟你的改动无关的新错出现了，你就会想到比较新旧两版进而找出错误之源。要是这不奏效，大概你就倾向于用老版本了。你可以使用版本控制工具，跟踪多版源程序来达到比较版本的目的。在第二十二章，结构管理一节中，有详尽的讨论。

重审修订后的程序

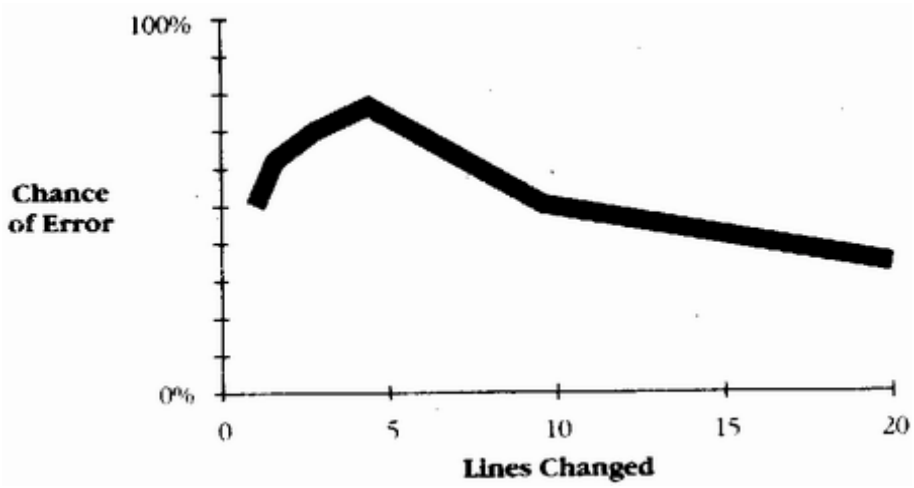
如果一次重审显得很重要，就更有第二次的必要，Ed Yourdon 说，程序员在初次编程时，要作 50％以上的修改。有趣的是，如果程序员针对一大段程序而不是仅仅几行着手工作，要作修改的机会就大得多。特别地，当修改的行数从一行变为 5 行，是越改越糟的可能性也大了。在这以后，改坏的可能就小了。

程序员小心地进行着修改。他们不应仅在办公室上修改程序。而应当试着运行一下来检证工作是否有效。

规则很简单，像对付复杂改动那样对付简单改动，有一个组织作过研究，审核前后错误率可能由 55％降到 2％。

重测试

应当用测试来验证修改的效果。在二十五章中，这种回归测试有详尽描述。Gerald Werberg



说，十个最昂贵的编程错误都涉及修改现存的程序。最贵的几个每个都值上千万美元，尽管只需改动一行。

软件优化的哲学

优化既是危险的又是趋向完善的机会，当你作了改动时，力图使以后修改变得更容易。在刚开始设计时，你可能不会想到遗留的问题有多少。有机会修改程序时，就尽你所学去做。在头脑中记住你的源程序及所做的修改。

30.3　编写新程序

修改时到底是提高了质量，还是损害了程序，一项原则是把一个老的程序分成 2 到 3 个新程序。

许多其它活动也能促使软件优化成功通常并不需要特别技巧，在本书其他它节已经阐述了这一值得讨论的主题。

一个拥有很多小的、定义良好的子程序的程序就成为了自给自足的程序。小的子程序要比加注释的大程序好，因为这方便了读者，况且子程序多半不像注释那么易于过时，Glass 和 Nosenx 在他们的《软件维护指南》中说，使维护简单的最重要的因素就是程序模块化。

创立新程序，减低复杂度

可能你最常创立的程序是用来增加程序可读性的那种。程序也可能很复杂，因为它涉及的问题很困难，也可能只是因为太长。本处不是要节省代码空间或是提高性能。你只不过在试图使复杂的程序易于理解而已。

创立新程序有时就像从源程序中抽出一段作子程序，再起个标题作名字。

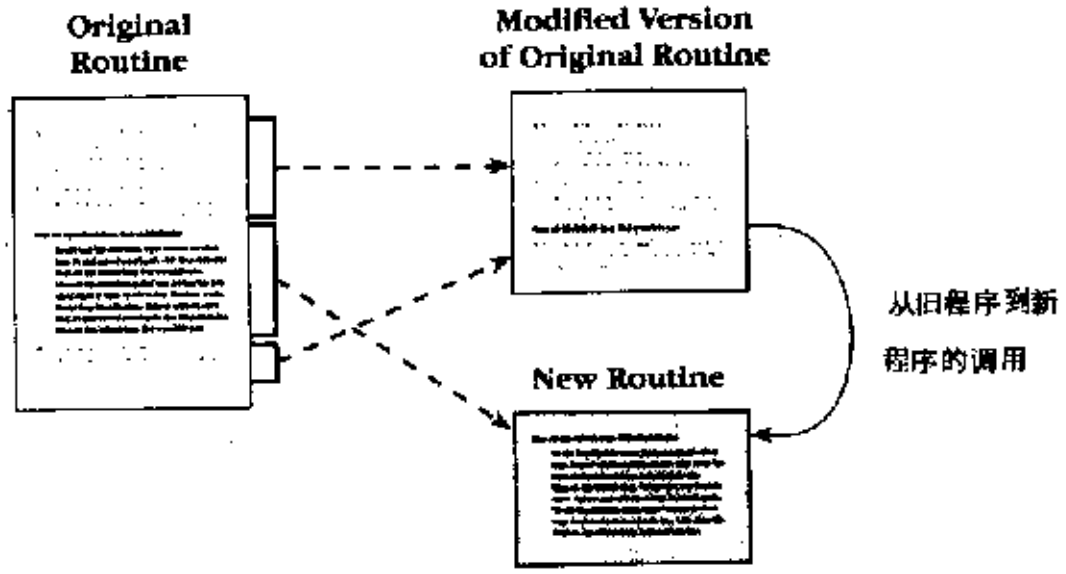
创立新程序有时简单得你无需费神思考它，如果你认为值得费劲把程序从这儿打断，那么就打断它，这种努力总是有益的。

用缩短程序来简化程序

缩短程序也是降低复杂程度的方法。试想你有一个解释命令的程序段，请看下例：

Pascal 一个将要修改的例子：

```
procedure DisplayCommand
```




```
        (
            CommandSentence: string;
            Var Command: COMMAND_TYPE;
            Var Error:    ERROR_TYPE
        );
{Display a readable version of a coded command string
    Command strings are of the form "# CA @AT arg1 @AT arg2..."
    "#"是命令开始处
    "CA"是命令缩写一个字母
    "@"是变量的开头
    "AT"是自变量的类型字母 }

Var
    Idx:                integer;
    CommandChar:        char;
    CommandWord:        String;
    Argument:           array[1...MAX_ARGUMENTS]of string;
    ArgCount:           integer;

begin
    { check for valid command}
    if((CommandSentence[1]<>COMMAND_SIGN) or
        (Length(CommandSentence)<MIN_COMMAND_LENGTH)) then begin
        Error:=InvalidCommandString;
        exit
    end;

    {expand the command abbreviation to a meaningful command word}

    case CommandSentence[2] of
```

```
        'L': CommandWord:＝'Left';
        'R': CommandWord:＝'Right'
        'U': CommandWord:='Up';
        'D': CommandWord:='Down';
        else FatalError('Unexpected Command in Display Command.')
    end；{case}

    ｛extract arguments｝
    ArgCount:=0；
    for Idx=3 to Length (CommandSentence) do begin
        CommandChar=CommandSentence[Idx]

        {begin new argument}
        if(CommandSentence=ARGUMENT SIGN) then begin
            ArgCount:＝ArgCount＋l;
            Argument[Argcount]:＝’’
        end
            { add next character to existing argument}
            else begin
            Argument[Argcount]:=Argument[Argcount]+CommandChar
        end
    end;{for};

    {display the command and its arguments in a pleasing format}
    writeln{'Command:',CommandWord};
    for Idx:=1 to ArgCount do begin
        writeln('        ',Argument[Idx])
    end
end；{DisplayCommand}
```

现在假设修改了程序，现在不是面临 4 个命令的程序段，而是十二个了。case 语句扩展成下面句子：

Pascal case 语句扩展：

```
        { expand the command abbreviation to a meaningful command word}
        case CommandSentence[2] of
            'L': CommandWord:＝'Left';
            'R': CommandWord:＝'Right'
            'U': CommandWord:＝'Up';
            'D': CommandWord:＝'Down';
            'E': CommandWord:＝'Enter';
            'X': CommandWord:＝'Delete';
```

```
            '*':CommandWord:＝'Undo';
            'J': CommandWord:＝'Join';
            'B': CommandWord:＝'Break';
            'S': CommandWord:＝'Save';
            'O': CommandWord:＝'Open';
            'C': CommandWord:＝'Close';
            else FatalError('Unexpected command In DisplayCommand.')
        end; {case}
```

你可以将 case 语句留在程序中，也可将其另编程。新设的程序就使你的程序清晰易读。下面还有一个例子：

Pascal 一段子程序扩展为二段：

```
    function CommandFromword(CommandCode:char);string;
    begin

    {expand the command abbreviation to a meaningful command word}
    case CommadCode of
        'L': CommandFromWord:＝'Left';
        'R': CommandFromWord:＝'Right'
        'U': CommandFromWord:＝'Up';
        'D': CommandFromWord:＝'Down';
        'E': CommandFromWord:＝'Enter';
        'X': CommandFromWord:＝'Delete';
        '*': CommandFromWord:＝'Undo';
        'J': CommandFromWord:＝'Join';
        'B': CommandFromWord:＝'Break';
        'S': CommandFromWord:＝'Save';
        'O': CommandFromWord:＝'Open';
        'C': CommandFromWord:＝'Close';
        else FatalError('Unexpected command In DisplayCommand.')
    end; {case}
    end {CommandFromWord }
```

在老程序中，DisplayCommand()用 case 语句转成 CommandFromWord();

```
    CommandWord:＝CommandFromWord(CommandFromWord(CommandSentence[2])
```

这个例子可能是修改中创立新子程序的典型事件从已有的逻辑结构中取出部分内容形成新段落，这段程序可作为一段子程序，因为它既有一定的逻辑结构，又能执行单一的定义好的功能。

用减少相互嵌套降低复杂度

除了把程序短化，将某段从相嵌套的逻辑结构中分离出来还可以降低复杂度，也使可读性增强，这里有一个相互嵌套的例子。

C 相互嵌套的例程分成单独子程序：

```
void ReadEmployeeData
    (
.   EMPLOYEE        Employee[MAX_EMPLOYEES];
    int *           EmployeeCount;
    )

    { EMPLOYEE      NewEmployee；
    BOOLEAN         ValidRecord;
    Char            TextChar；
    int             FieldInx;
    int             CharIdx；
    FILE*           EmployeeFile；
    int             Length；

    EmployeeFile=OpenEmployeeFile(EmployeeFile);

    {Read and check each employee record}
    *EmployeeCount=0;
    ReadEmployeeRec(&NewEmployee);
    do
    {
    /*  check each record's validity */
    Valid Record=TRUE;
    For(FieldIdx＝0；FieldIdx＜NexEmployee.NumFields;FieldIdx++)
    {
    /*  check each record's validity */
    Length=strlen(NewEmployee.Field，[FieldIdx++])；
    for(CharIdx=0;CharIdx<Length;CharIdx++)
        {
        TestChar=NewEmployee.Filed[FieldIdx][CharIdx]
        if (!
          ('a'<=Testchar&& Testchar<＝'z'||
          ('A'<=Testchar&& Testchar<＝'Z'||
          '0'<＝Testchar&&Testchar<='9')
          ValidRecord=FALSE;
        }
    }
If(ValidRecord)
```

```
        {
            Employee[*EmployeeCount]=NewEmployee；
            (*EmployeeCount)++;
            }
        ReadEmployeeRec(NewEmployee);
        }
        /* do */
        while(*EmployeeCount<MAX_EMPLOYEES&&!feof(EmployeeFile));
        …
    } /* ReadEmployeeData*/
```

如果你想改变程序，而使得内循环变得不复杂或者有利于修改，你可将任意一个循环转化成自主的程序，从而会使程序模块化，可读性和可维护性都加强了。

你可以把大的循环变为名叫 ValidateEmployeeRec( )，可将小的叫 ValidateInputField()。

下面的例子是大循环变为自主程序后会是什么样子。

C 将相互嵌套的循环改成自主程序：

```
    BOLLEAN ValidateEmployeeRec(EMPLOYEE Employee)
    {
    int             FieldIdx;
    int             CharIdx;
    FILE*           EmployeeFile；
    Char            TextChar；
    BOOLEAN         ValidRecord;
    int             Length;

    ValidRecord=TRUE;
    for(FieldIdx=0;FieldIdxEmployee<Employee.NumFields;FieldIdx++)
    {
        length=strlen(Employee.Field[FieldIdx]);
        for(CharIdx=0;CharIdx<Length;CharIdx++);
        {
            TestChar=EmployeeField[FieldIdx][CharIdx];
                if(!
                    ('a'<=Testchar&& Testchar<＝'z'||
                    ('A'<=Testchar&& Testchar<＝'Z'||
                    '0'<＝Testchar&&Testchar<='9'))
                ValidRecord=FALSE;
                } /* for */
            } /* for */
        return(ValidRecord);
```

```
        }
```

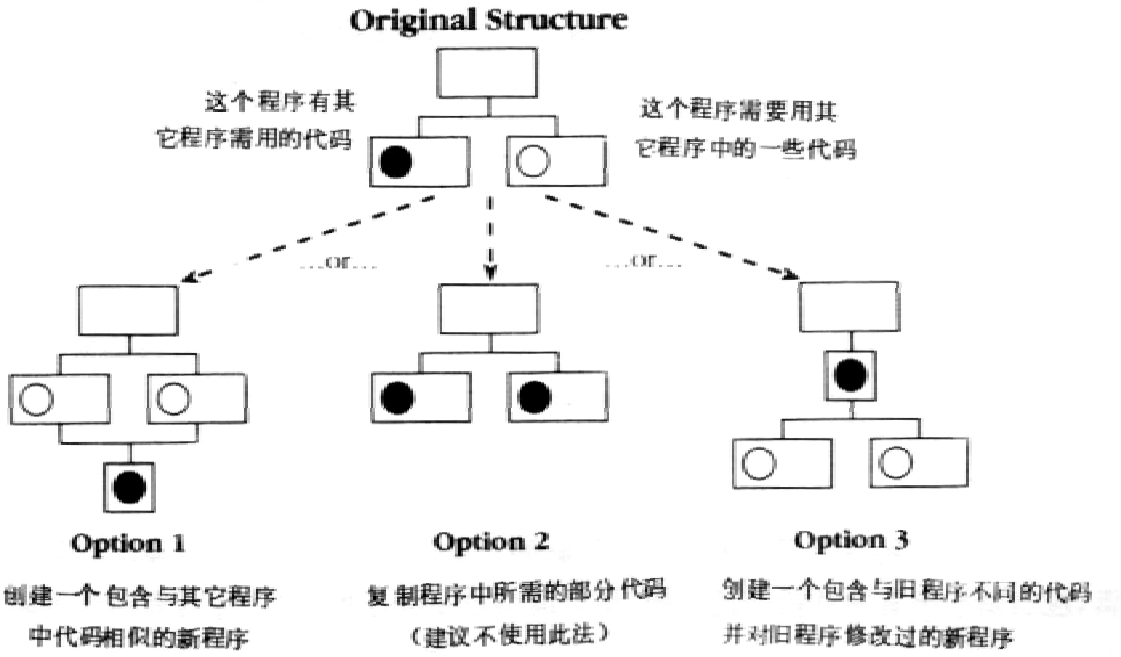
旧程序中的 for 循环将被替换成一个新程序。

C 从原来子程序到新程序：

```
    void ReadEmployeeData
        …
        { read and check each employee record}
        *EmployeeCount＝0;
        ReadEmployeeRec(&NewEmployee);
        do
            {
            /*    check each record's validity*/
            ValidRecord=ValidateEmployeeRec(NewEmployee);
            if(ValidRecord)
                …
```

程序在此处被调用最大的优点在于修改后程序既短，嵌套又浅。

程序优化使得复杂的循环变成简单的程序，这个例子并未增加某种功能，但却显示了良好的技术；在不改变功能性的前提下简化了程序。这样你对程序理解更深了，程序增加了可读性可修改性和模块化。你可试着用同样的技术简化 case 语句。



加一段代码，然后加上标志，告诉程序是做原来的工作还是新的工作。

很难说这是种最佳方案。还有更天才的设想，那就是拆散新旧程序都要作的功能，而不是只改动一点，将每部分功能定义为一段自主程序。这样两个新程序就体现了新程序的区别，新

程序通常有比你原先想象的更多的通用场合。给新程序起个清晰的名字说明目的，这样你的程序将更接近设计。

由于你是在软件优化时作的这些修改，原先的程序并不要求两段程序有相同的部分。因此，遗留的问题是如何共享这些代码。

图中显明了三种可能的修改策略。

考虑这些方案：

- 在较低水平上强调相似性。为共享代码段创立新程序。从原程序中抽取代码，并从源程序及其它程序中调用代码。这是最通常的解决方案，可以提高模块化程度。
- 重点放在不同情形下的相似性上。在所有的程序中复制一遍，有相应功能的代码（类似宏）。你可以从原程序中拷贝一段，并手工地安插到合适的位置。因为代码段重复多次，此方案有些占空间。因此带来的维护问题也令人头痛，因为算法的改变可能使多处地方需重新书写。这不能称之为好方案。
- 把对相似点的侧重放在高层上。就是说，创立新程序包含共享代码。写了新的，用低层次的程序来管理程序的差异处。原来的高层次子程序处理普遍情形——相同的共性——并调用新的低层的子程序以处理细微的差别。

Pascal 一段将要共享的程序段：

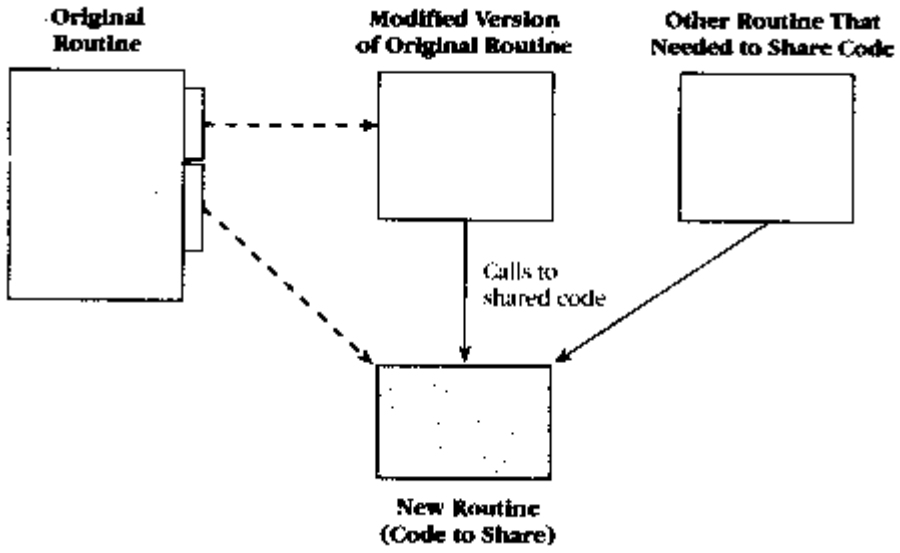
```
…
AllowableTemperature:=MaxAllowableTemperature(Environment);

/* get most recent temperature from top of stack */

Temperature                    :=StackTemperature[Stack.Top]
StackTemperature[Stack.Top]    :=INITAL_TEMPERATURE;
if(Stack.Top>0)    then
    begin
    Stack.Top:=Stack.Top-1
    end;

if(Temperature>AllowableTemperature) then
    begin
    ShutdownReactor(Enviromentm,Temperature)
    end,
…
```

再进一步假定．你还有个程序需要知道最新的温度。你可以复制堆栈弹出部分的代码，但是按逻辑，应当是抽出有关栈操作的代码，使之成为自主程序段，然后在需要的地方调用它。用图解表示即：



下面是相共享的新程序（新程序具有新的标题）：

```
 procedure    MostRecentTemperature
      (
var stack:              STACK_TYPE;
var Temperature:    1..100000
)

begin

/* get most recent temperature from top of stack */

Temperature                         :=StackTemperature[Stack.Top]
StackTemperature[Stack.Top]     :=INITAL_TEMPERATURE;
if(Stack.Top>0) then
     begin
     Stack.Top:=Stack.Top-1
     end
end;
```

既然你已将获得最新温度的代码段并生成了自主程序，下例是说明如何从原位置处进行调用。

```
AllowableTemperature:=MaxAllowableTemperature(Environment)
MostRecentTemperature(Stack,Temperature);
if(Temperature>AllowableTemperature) then
     begin
     ShutdownReactor(Enviromentm,Temperature)
     end;
…
```

现在你可以在用要的地方调用此程序了，除了分享代码处，你已经作了程序归档，所有的栈弹出都用来得到最新温度。由此可见新程序使之清楚易懂。

你还隐藏了执行跟踪温度的程序的数据结构。整个程序无须知晓在栈内的数据结构。只有某些程序知道，而你已经编出来了。在命名栈的高层程序中使用数据时，你还有些小问题，栈的命名有些计算机术语化而非面向问题。在高层程序中，并不是直接操纵数据结构的一个更好的变量名是温度记录，在低层访问程序中你仍可以称之为栈。

总之，新程序极大地改善了源代码。它增进了代码共享、提高抽象程度和信息隐蔽。

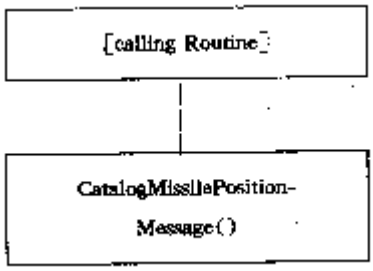
调动程序共享代码的例子

假如你是火箭专家，为一家航天公司工作，你正在编制有关导弹信息目录的程序。起初，程序只需记录导弹位置的信息。下面是你编制目录信息要采取的步骤：

- 读信息
- 检查信息标志着是否是导弹位置信息
- 检查信息的各个方面
- 存储信息

如果所有操作都简单，你可以将程序命名作 CatalogMissilePositionMessage(),这个例子是较简单的。调用等级如下图：

现假设除了导弹位置的信息外，你还得对新导弹信息排序，导弹更新信息以及导弹辨识信息。你已写下的程序已能完成大多的任务。先读取新信息，再辨识、检验，再存储。如何转换代码以使你能最大地共享代码呢？



在导弹位置信息程序的例子中，你可能将程序段从 CatalogMissilePositionMessage()抽出，并创立新的程序 Readmessage(),Checkmessage(),CheckmessageFields()及 SaveMessage()来处理信息。然后你还可再编制程序，新程序叫做较低层程序。在顶部，你可能还要编写程序辨别信息种类并在正确的程序间传送这些信息。

新程序被放在图的灰框中，因为，它们之间复杂地联系着，你还得自己编出新程序来满足要求。

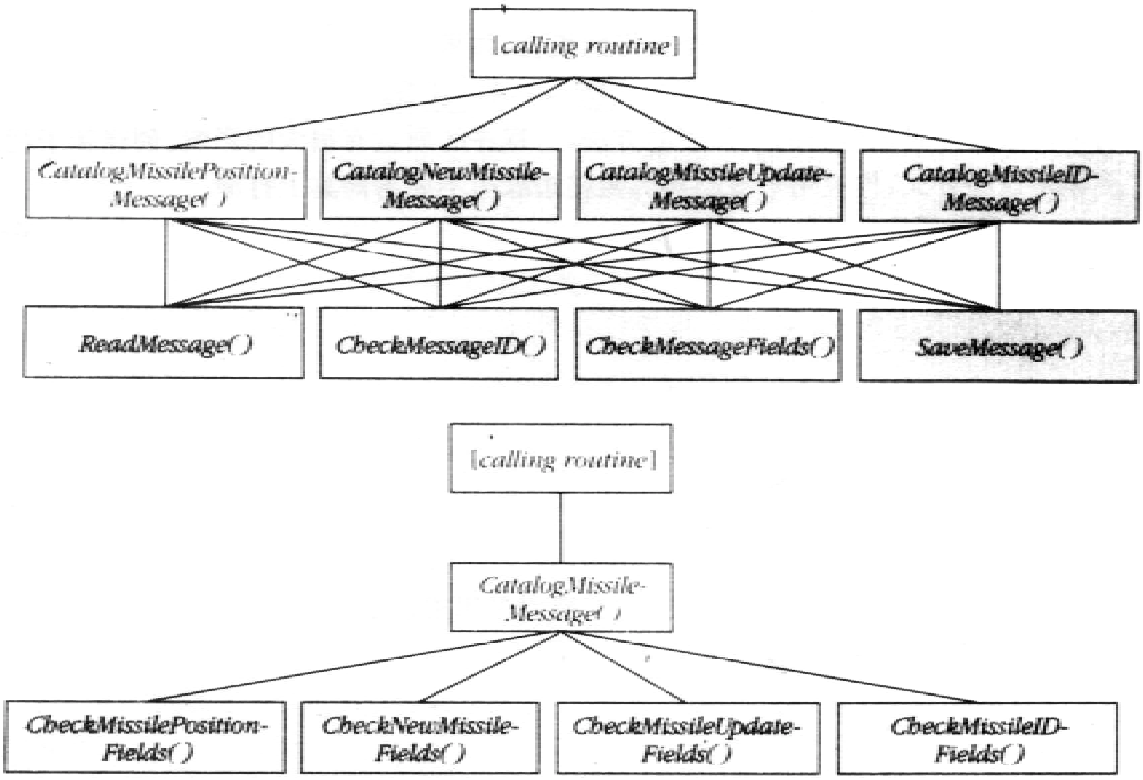
在高层次共享代码

另一种出路是编写共享代码，作比较低层的程序主要处理新程序间的差异部分。这样你有许多地方可调用新代码段，而不是从许多地方分别调用新代码。

读信息、识别信息及存储信息工作可能会是非常通用的，以致于不需要修改就能支持新的信息。你可以不作改动，如在 CatalogMissilePositionMessage()，如果唯一工作就是检查信息领域的不同处，你就可以用低层程序来说明这些差异，从上面程序中反映出更普遍的功能，但在此之前，唯一的变化是调用一个特殊程序来测试信息栈。

新程序已不那么多联系也不那么复杂了。

显然，低层共享代码在高层收益是很大的。多数情形下不像这样，你可更多地从低层共享



代码中受益，但不要有在高层分离代码的念头。在一些情况下，你会很高兴自己所做的事。

检查表

修改程序

- 每次改变都是按修改策略的吗？
- 变动是否彻底地检查过了？
- 软件是否作了重测试，确保修改未使性能变坏？
- 修改是否提高了程序的内在质量？
- 是否尽可能将程序拆成子程序从而模块化？
- 是否尽可能地减少了全局变量？
- 是否改进了程序风格？
- 如果为了共享代码，必须作些改动，你是否考虑过共享代码在高层程序上而非在低层程序上？
- 如此改动后，是否为以后的修改创造便利？

30.4　小 结

- 初始开发中，升级是传统开发方法的一个典型事例，并且随着新方法的使用，它可能变得更为突出。
- 使用升级，软件质量可能是提高或恶化了。软件升级的基本规则其内部质量应随时间的流逝而提高。虽然在扩充维护阶段，软件退化是能避免的，在创建中软件退化就不

妙了。

- 开发对提高你的程序质量是一个最好的机会，如果你想有所收益，你应很好的利用每一次机会。
- 当你修改程序时创建新的子程序的方法，对程序的质量有很大的影响。创建新的子程序是你用这种方式：你应用品化结构的粗糙边缘而不要再向其上打洞。

第三十一章　个人性格

目录

相关章节

软件技术主题：见第 32 章
控制结构和复杂性：见第 17.7 节

在软件开发中人们很少注意个人性格问题。自从 1965 年 Edsger Dijkstra 的有里程碑意义的文章“程序开发是一种人类活动”发表以来，程序员性格被认为是合理的和有成效的研究领域，虽然有些题目如“大桥建筑者的心理”和“对律师行为的研究实验”看起来可能是荒唐的，而在计算机领域，“计算机编程中的心理”和“对程序员行为的研究实验”等题目则是常见的。

每个领域的工程人员都知道工具和他们所用材料的局限性。如果你是一位电机工程师，你就应明白各种材料的导电性和使用电压表的各种方法。如果你是一位建筑师，你就应明白木材、混凝土、钢材的性能。而如果你是一位软件工程师，你的基本建筑材料是人的聪明才智，并且你的主要工具是你自己。建筑师是将建筑物结构进行详细的设计，然后将设计蓝图交给其它人去建造，而你则是一旦当你从细节上对软件作出设计后，软件生成过程也就结束了。

编程的整个工作就如建造空中楼阁一样—它并不是纯粹的人工活动。于是，当软件工程师研究工具和材料的必需性时，他们发现自己正在研究人的智力、性格，不像木材、混凝土和钢材等可见的东西。

31.1　个人性格是否和本书的主题无关

编程工作极强的内部特点使得个人特点异常重要，你想想一天全神贯注地工作八小时有多么困难，你可能有过由于精力过份集中而今天无精打彩的体验。或由于上个月过份投入而本月没有一点精神，你也可能在某一天从上午 8 点工作到下午 2 点，以致于精神快要坍塌了。有时你从下午 2 点拼命于到 5 点，然后花费一周的时间修改你在其间所写的东西。

人们难以对编程工作进行检查，因为没有人知道你真正干些什么，我们经常有过这样的体验，我们花费 80%的时间进行我们所感兴趣的 20%的工作，同时花费 20%的时间生成其余 80%的程序。

你的老板并不能强迫你成为一个好的程序员，甚至过了很长一段时间你的老板也无法判断你是否是一个称职的程序员，如果你想成为一个高手，你得全靠你自己下功夫。它和你个人性格有关。

一旦你自己决定成为一个高级程序员，你发展的潜力是很大的，各种研究发现，创建一个程序所需的时间比可达到 10：1，同时也发现调试一个程序时间，程序实现长度、速度、错误率和所发现错误数对不同的程序员其差别可达 10：1。

你无法改变自己的聪明程度，但是你可在一定程度上改变自己的性格，已发现在程序员成为高级程序员的过程，性格是更有决定意义的因素。

31.2　聪明和谦虚

聪明看起来似乎不是个人性格的一个贡献。它也的确不是。恰巧的是，好的智力是和成为一个好的程序员有着并不严密关系的因素。

为什么？难道这并不要求你有一个好的智力吗？

不，你不需这样，没有人真正同计算机一样迅速敏捷。完全理解一个一般的程序需要你有吸收细节的很强的能力，并能同时理解所有细节，你很好地利用你的聪明要比你有多聪明更为重要。

在 1972 年，Edsger Dijkstra 发表一篇论文，名字叫作“谦虚的程序员”。他在此文中主张所有的程序员都应尽力弥补他们很有限制的智力。那些最精通编程序的人往往是那些认为自己的头脑是多么有限的人，他们是谦虚的。而那些最为糟糕的程序员往往是那些拒绝承认自己的能力不适应工作任务的程序员。他们的自我妨碍自己成为优秀程序员，你学到越多的东西来弥补你的大脑，你就越能成为一个好的程序员，你越谦虚，你取得的进步也就越快。

许多良好的编程风格的目的是减少你大脑的负担，以下是一些例子：

- “分解”一个系统的目的是为了使其更为简单易懂。人们往往易于理解几条简单的信息而不是一条复杂的信息。所有软件设计方法的目的是将复杂的问题分解为简单的几部分，不论你是否使用结构化、自顶向下或是面向对象的设计，以上目标都相同。
- 进行评审、检查和测试是弥补人的错误的一种方法，评审方法部份源于“无错编程”，如果你没有任何错误，你就用不看评审你的软件，但是当你知道自己的能力是有限时，你就应和别人讨论以提高你的软件质量。
- 将子程序编短一些有助于减少你的工作量。你根据问题而不是计算机科学术语编写程序并使用尽可能高级的抽象思维，有助于减少你的工作量。
- 使用各种交谈方式可将你从编程的死胡同中解放出来。

你也许认为靠聪明能更好地开发人的智力活动，所以你无需这些帮助。你也可能认为使用智力帮助的程序员是走弯路。实际上，研究表明那些使用各种方式弥补其错误的谦虚的程序员们所编写的程序既易为自己也易为别人所理解，并且其程序中所含错误也少。实际的弯路是出现错误和影响进度的路。

31.3　好奇心

一旦你认为自己理解程序的能力是有限的，而且你意识到，进行有效的编程是补偿自己能力的方法时，你就开始了你生涯中漫长的探索过程。

在变成高级程序员的过程中，对技术的好奇心是很重要的。有关的技术信息变化迅速。许多 PC 程序员没有在什么机器上编过程，而许多程序员还没有用过电脑的穿孔卡片。技术环境的特定特征每隔 5 到 10 年就发生变化。如果你跟不上这种变化，你将面临落伍的威胁。

程序员往往很忙碌，以致于他们没有时间对更好地工作或对工作发生兴趣。如果你真是这样，你也不必在意大多，因为许多人都同你一样，以下是一些培养你的好奇心的方法，你真应该好好学一学它。

在开发过程中建立自我意识。你对开发过程越了解，不管你是通过对开发过程的阅读或自己的观察得来的，你就越能了解各种修改，并使你所在开发组向一个更好的方向前进。如果分配你的工作任务很少而不能提高你的技能，你也应对此满足。如果正在开发有良好市场前景的软件，你所学的一半知识将会在今后三年内过时，如果你不再学习新知识，你将会落伍。

在 1988 到 2000 年中，美国平均工作人数可增加 11％到 19％，计算机系统分析员可增长 53％。程序员可增长 48％，计算机操作员可增长 29％──在现有的 1，237，000 份工作的基础上再增加 556，000 份新工作。如果你在工作中学不到什么，你可试着找一份新工作。

实验。了解编程的一个有效途径是对编程和开发过程进行实验，如果你对所用语言的工作过程不甚了解，你可编写一个短程序以检查此特征并看其是如何工作的，你可在调试器中观看程序的执行，你用一个短程序而不是一个不甚了解的大程序来测试一个概念是很好的。

如果短程序的运算表明程序运行结果并不是你所期望的，这时你应怎么办？这正是你所需要的，最好用一个短程序在有效编程的一个关键方法上迅速制造错误，每次你可从中有所收益，制造错误并不是罪过，没有从中学到什么才是罪过。

阅读解决问题的有关方法。解决问题是软件开发中的一个重要活动，Herbert Simon 曾报道过人工解决问题的一系列例子。他们发现人们自己通常不能发现解决问题的方法，即使这种方法很容易学到。这意味着即使你想自己创造车轮，你也不能指望成功，你可能设计出方车轮。

在你行动之前进行分析和计划。你将会发现在分析和行动之前存在着矛盾，有时你不得不放弃的数据和行动，对大多数程序员来说，问题并不在于过分分析和过分使用。

学习成功项目的开发经验。学习编程的一种非常好的方法是向一些优秀程序员学习。Jon Bentley 认为你应静心坐下来，准备一杯白兰地，一枝好雪茄烟，然后如同读小说一样阅读程序。实际上可能并不是这样，许多人往往不愿牺牲其休息时间来阅读──500 页的源程序，但是许多人往往乐意研究一个高级程序的设计，并有选择地研究一些具体细节。

软件工程领域很少利用过去成功或失败的例子。如果你对建筑学有所兴趣，你可能会研究 Louis Sullivan，Frank Lloyd Wright 和 I.M.Pei 的设计图，你也可能会参观他们的建筑物，如果你对结构工程有兴趣，你可以研究 Broolyn 大桥，Tacoma Narrows 大桥以及其它混凝土、钢铁和木材建筑，你应研究你所在领域中成功或失败的例子。

Thomas Kuhn 指出，任何成熟的科学，实际上是通过解决问题而发展起来的，而这些问题通常被看作本领域良好工作的例子，并且可用作将来进行工作的例子。软件工程是刚入成熟阶

段的一门科学，在 1990 年，计算机科学和技术委员会曾指出，在软件工程领域很少有对成功和失败的例子进行研究的文件，在 1992 年 3 月的“ACM 通信”中有一篇文章主张对别人编程中出现的问题进行研究，总之，学习别人的编程是有重要意义的。

其中一个最受欢迎的栏目是“编程拾萃”，它专门研究编程过程中出现的问题，这对于我们是有启发的。

你可能有或没有一本研究编程的书，但是你可阅读高级程序员所编写的代码，阅读你所尊敬的程序员的代码，或阅读你并不喜欢的程序员的代码，再将他们的代码和你自己的代码比较。它们之间有何异同？为什么会有差异？哪一个更好？为什么？

除了阅读他人的代码之外，你也应让其它高水平程序员评价你的代码质量，找一些较高水平的程序员评论你的代码，从他们评论中，你可剔除那些带个人色彩的东西而着重于那些重要的东西。这样可提高你的编程质量。

阅读手册。手册恐惧症在程序员中很流行。一般来说，手册的编写和组织都不好，但是程序员对手册的恐惧也和他们对书本的过分恐惧有很大关系。手册含有一些重要的东西。所以花费时间阅读手册是值得的，忽视手册中的信息正如忽视一些常见的首字母简略词。

现代语言产品一般都带有大量程序库，这时，你花费时间查阅参考手册是值得的，通常提供语言产品的公司，已经编写了许多你可以调用的子程序。如果是这样，你应弄懂有关手册，每隔一段时间阅读一下手册。

阅读有关书籍和期刊。你应为自己阅读本书感到幸运。你已经学到了软件工程的许多知识，因为一本书每年都要被许多程序员所阅读，读一些东西可能使你的专业知识向前迈进一步，如果你每二个月阅读一本好的计算机书籍，你的知识将会大大提高并能在同行中脱颖而出。

31.4　诚　实

编程生涯成熟的部分标志是不折不挠地坚持诚实，诚实通常表现在以下几个方面：

- 不假装你是一个编程能手
- 乐于承认自己的错误
- 力图理解编译器警告信息而不是对其置之不理
- 对你的程序有一个清晰的了解，而不是进行编译看其是否有错
- 提供实际状态报告
- 提供实际方案评估，在你的上司面前坚持自己的意见

前二个方面—承认你不知道一些事情或承认你犯了一个错误是你谦虚的反映。如果你不懂装懂你又怎么能指望学到新东西呢？你最好是假装自己知之甚少，听别人的解释，向他们学习新的东西，并评估他们是否真正了解其正在谈论的东西。

你应对自己的能力作某种程度的估计，如果你对自己的评价很完美，这可是一个不妙的信号。

拒绝承认错误是一个令人讨厌的习惯，如果 sally 拒绝承认错误，她看起来相信自己没有错，可能会使其它人相信她确实是无辜的，但是事实证明 sally 出错误了，这样，每个人都知道她犯了错误。错误正如潮流一样是一种复杂的活动，如果她在过去没有发生过错误，谁也不

会将错误归咎于她。

如果她拒绝承认错误，到头来她只能自食其果。其它人都知道他们在同一个不诚实的人工作。这比仅犯一个错误更令人反感。如果你犯了一个错误，你应迅速主动地承认错误。

对编译器错误信息不懂装懂是另外一个常见错误。如果你不理解某一编译警告信息或你认为时间太紧迫来不及检查，你想想这是不是真正浪费时间？编译器将问题明白无误地向你展示出来，而你却不试图解决问题，我碰到过不少人在调试过程中请求帮助的事，我问他们是否有一个完好的编译器，他们回答是。于是开始解释问题的症状，我说：“这看起来像是未对指针进行初始比。但是编译器应对此给出了警告信息。”他们就说：“哦，编译器确实给出了警告信息，我们以为它是指其它事情。”你自己所出的错误难以蒙蔽别人，也更难以愚弄计算机，所以你用不着浪费时间这样作。

另外一种疏忽是当你并不完全了解程序时，你“编译它看是否能运行”。在这种条件下，其实并不意味着程序能运行，因为连你自己都不清楚程序的有关情况。请记住，测试仅能发现错误的存在，而不能保证一定不存在某种错误。如果你不理解程序，你就不能进行深入的测试，你如果觉得应编译一下程序以便了解程序的运算情况的话，这可是一个不妙的信号，这可能意味着你不清楚在干些什么。在将你的程序编译之前你应对其有一个深刻的理解。

状态报告也同样是一个令人反感的领域。如果程序员在最后 50％的项目时说，程序中 90％是完整可靠的，他们将声名狼藉。问题在于你对自己的进度缺乏了解，你应对你的工作加强了解。但是，你为了迎奉上司而不愿说出真实情况的话，可就不同了。一般来说上司都愿意听到对项目状态的真实报告，即使不是他们所希望听到的，如果你的观察和见解是中肯的，你应客观地将其说出来，上司需要有准确的信息以便协调各种开发活动，而充分的合作是必需的。

和不准确的状态报告有关的一个问题是不正确的估计。典型的情况是这样：上司问 Bert 要花多少时间才能开发出一个新的数据库产品。Bert 和一些程序员交谈了一下，讨论了一些问题，最后认为需 8 个程序员和 6 个月的时间，但是他的上司说：“这并不是我们所需要的，你能不能使用较少的程序员在短时间内完成工作？”Bert 考虑了一段时间，并认为可以通过削减培训时间和假期以及让每个人的工作时间稍微延长一点来达到上司的要求。他于是作出了需 6 个程序员和 4 个月时间的估计，他的上司说：“这就行了。这是一个相对较为低优先级的项目。你应及时完成它，因为预算不允许你超时。”Bert 所犯错误在于，他没有认识到评估是不可商量的，他可以将估计作得更为准确，但是他和老板的商量结果并不能改变开发一个项目所需的时间。 IBM 公司的 Bill Weimer 说；“我们发现技术人员一般都能准确地估计项目。问题在于他们能否坚持自己的决定；他们需要学会坚持自己的意见。”Bert 许诺在 4 个月里交付产品而实际上 6 个月才交付产品，肯定会使他的老板不高兴的。时间一长，他可能会因妥协而失去信任的。否则，他会因坚持自己的估计而得到尊敬的。

如果上司施加压力要改变你的估计，你应认识到决定要怎样作是上司职权范围内的事。你可以说：“看，这是项目的开销，我无法说此开支对本公司是否值得—这是你的工作。我不能和你‘商量”项目所花的时间，这正如我们不能协商确定一里究竟有多少英尺一样——这是不可变更的。你不能商定自然界的规律，我们只能商定本项目中影响进度的各方面，然后重新评估。我们能减少一些特征，降低性能，分阶段开发项目。或者是使用更少的人但时间延长一点，或者是使用稍多的人，而相应地减少一些时间。”

我曾在一次软件开发管理讨论会上听到一个奇怪的说法。主讲者是一本销售很好的软件工程管理书籍的作者。一听众问：“你的上司让你评估某一项目，你知道当你得出准确的评估时你的上司可能认为代价太高而放弃项目开发。这时你认为应怎么办？”主讲者回答说，你说服你的上司作出开发项目的决定时，他们就对整个情况了如指掌了。

这是一个错误的回答。你的上司是负责整个公司的运转的。如果开发某一软件需 100000 美元而你的估计是 200000 美元，你的公司就不会开发软件。这是要由上司作出决定的。上面这位主讲者对项目的开支说假话，告诉上司将比实际的要少，他这是在损害上司的权威，如果你认为项目是有前途的，它能为公司带来新的重大突破，或能提供有价值的培训，你应将其说出来。你的上司也会考虑这些因素的。你哄骗上司作出错误的决定将会使公司蒙受损失。如果你失去了你的工作，你将会明白你最终得到了什么。

31.5　交流和合作

真正优秀的程序员应学会怎样和别人工作和娱乐，编写可读代码是对程序员作为组中一员的要求之一。

计算机也就同其它人一样能读懂你的代码，但是它要比其它人更能阅读质量差的代码。作为可读性原则，你应将修改你的代码的人时刻记在心上。开发程序首先应同程序员交流，其次则是和计算机交流。

绝大多数高水平程序员喜欢使自己程序的可读性强，并抽出充足的时间这样作。虽然只有一些人能坚持到底，而且其中一些人还是高水平的代码编写者，对开发中程序员级别的了解，有助于解释什么地方适合于此原则：

级别 1：初学者

初学者是能使用一种语言基本能力的程序员，这样的人能够使用子程序、循环、条件语句和其它许多语言特征。

级别 2：中间者

中间级程序员有使用多种语言的能力，并且至少非常熟悉某一种语言。

级别 3：专家

编程专家对其语言或环境或对这二者有着很深的造诣，这种级别的程序员对公司有价值的，而且有些程序员往往就停留在这个水平上。

级别 4：大师

大师有着专家那样的专业知识，并能意识到编程只是 15％和计算机交流，其余 85％是和人打交道。一般程序员只有 30％的时间甚至更少。大师所编写的代码与其说是给计算机看倒不如说是给人看的。真正的大师级程序员所编写的代码是十分清晰易懂的，而且他们注意建立有关文档。他们也不想浪费其精力去重建本来用一句注释就能说清楚的代码段的逻辑结构。

一位不强调可读性的高水平代码者可能停留在级别 3 的水平上，但是问题还不止此。依作者本人的经验，人们编写不可读代码的主要原因在于他们所编代码质量较差。他们并不是自言自语地说：“我所编代码不好，所以我要使其难以读懂”，而是他们并不能完整地理解自己的代码以致于不能使其是可读的，这就使他们只能停留在 1 或 2 级的水平上。我所见的最差的代码是由一个任何人看了她的程序后都会望而生畏的人所编写的。最终，她的上司威胁说如她再不

改正就要解雇她。她的代码是不作注释的，并且其程序中充满了如 x，xx，xxx，xx1 和 xx2 这样的全局变量。她的代码给了她大量的机会显示她的改错能力。

你不必为自己是初学者或中间者而内疚，你同样不必为自己是专家而不是大师自愧，在你知道怎样提高自己的水平后，你倒是应为自己停留在初学者或专家的水平上有多长时间而内疚。

31.6　创造力和纪律

当我走出校门时，我自认为是世界上最好的程序员。我会编辑令人容忍的井字游戏程序，也能用 5 种不同的计算机语言编写一个 1000 行的 WORKED 程序。然后我进了 Real World 公司。我在 Real World 公司的第一个任务是阅读和理解一个 200000 行的 Fortran 程序，然后我使其运行速度提高了 2 倍。任何真正的程序员将会告诉你所有结构化编码将无助于你解决问题。

“Real Programmers Don’t write Pascal”

向一位刚走出校门的计算机科学毕业生解释为何需要约定和工程纪律是困难的。当我还是一个大学生的时候，我所编写的最大的代码是 500 行的可执行代码，作为一个专业程序员，我也已编写了许多小于 500 行的实用工具，但是一般项目的长度为 5000 到 25000 行，并且我参加过超过 50 万行的项目的开发工作，这种类型的工作不是需要较高的技巧，也不需要使用新的技巧。虽然一些有创造性的程序员将各种标准和约定视为对其创造力的阻碍，但是，对大项目来说，如果没有标准和约定，项目的实现是不可能的，而此时要发挥创造性也是不可能的。不要在一些无关紧要的领域建立约定，这样你就可在你认为值得的地方集中发挥你的创造力。

McGarry 和 Pajerski 在对美国宇航局的软件工程实验室过去 15 年的工作回顾中说，强调纪律的方法和工具是非常有效的。许多有很高创造力的人都能很好地遵守纪律，高水平的建筑师在材料的物理性能、时间和代价的限定范围内进行工作，艺术家同样如此，许多看过 Lenoard 的设计的人，都为他在细节上对约定的遵守产生由衷的敬重。当 Michelangelo 设计天花板时，使用了各种均衡的几何形式如三角形、圆周和正方形，他按一定层次将以上三种图形安排在三个区域，如果没有自我约束和结构，这 300 个人物的排列将是混乱的而不是有机地结合在一起的艺术杰作。

一个杰出的程序员需要遵守许多规则。如果你在开始编码之前不分析需求就进行设计，你将在编码过程中学不到关于项目的许多东西，你工作的结果看起来更像一个三岁小孩的手指画，而不是一件艺术作品。

31.7　懒惰

- 懒惰表面形式有以下几种：拖延自己讨厌的工作
- 迅速地将自己讨厌的任务作完以摆脱任务
- 编写一工具来完成自己讨厌的工作以解脱自己

当然，有一些懒惰形式要比其它方式好一些。第一种方式是没有任何益处的。你可能有这样的体会：你常常花费几小时来作一些没必要作的工作，而不愿面对自己所无法避免的次要的

工作，我讨厌数据输入，但是许多程序需要少量的数据输入。别人都知道我已拖延了数天的工作仅因为为了拖延无法摆脱的用手工输入几个数据的任务，这种习惯是“真正的懒惰”，你编译某一子程序以检查有关情况，这样你可以避免人工检查程序同样也是一种懒惰行为。

这些小任务并不像看起来那样令人反感，如果你养成马上完成这些任务的习惯你就能克服拖延这种懒惰。这种习惯叫“明懒惰”——懒惰的第二种方式，你仍然是懒惰，但是你是通过在自己所讨厌问题上花费尽量少的时间来避开问题的。

第三种选择是编写工具来作这令人讨厌的工作。这是“长期懒惰”。它无疑是懒惰中最有积极性的一种形式，只要你通过编写工具最终节省了时间，通过讨论可知，一定程度的懒惰是有益的。

当你不是透过玻璃看问题的时候，你就看到了懒惰的另一方面。“赶着做”或“努力”并不能发出炫目的光芒。赶着做是一种多余和没有必要的努力。它只是说明你的焦急而不是你进行工作的努力程度，在有效编程中最为重要的思考是人们在思考中往往显得并不忙。如果我和一位看起来一直很忙的程序员一起工作，我将认为他并不是一位好的程序员，因为他并不是在使用对他来说是最有价值的工具和自己的头脑。

31.8　不是你想象中那样起作用的性格

“赶着做”并不是唯一的一种看起来可能受敬重而实际上并不起多大作用的性格。

坚持

依赖于环境，“坚持”可能是一笔财富也可能是一种不利条件，和其它许多多义概念一样，对它有不同的解释，这取决于你认为它是一种好的特性或坏的。如果你想将坚持定义为坏的性质，你可能说它是“顽固”，如果你认为是一种好的品格，你可称其为“坚强”或“坚持”。

在大多数情况下，软件开发中的坚持是顽固的意思，在你碰到某段新代码时，你再固执己见并不是什么好事。你应试着用另一个子程序，用另一种编码方法，或返回原来的地方，当某种方法并不起作用时，你应换用另一种方法。

在调试中，当你终于发现一个烦扰你达 4 小时之久的错误时，你一定感到非常满意。但是如果你在一段时间—通常为 15 分钟没有取得任何进展时，你应放弃找错。用你的潜意识去思考问题，尝试用别的方法解决问题，重写全部令人厌烦的代码段。当你的精神有所恢复时重新回到原来的问题上。和计算机错误作斗争是不明智的，你应尽量避免它们。

知道在什么时候放弃是困难的，但是这是你必须面对的一个问题。当你觉得自己受挫折时，你可向自己提出这个问题，你问问自己并不意味着放弃，但可能意味着是对自己的行动设置规范的时候了：“如果我不能用这种方法在 30 分钟时间内解决问题，我将用几分钟时间考虑不同的方法，并在下一小时内尝试不同的方法。

经验

和书本知识比起来，软件开发中经验的价值要比其它领域小，这有几种原因。在许多其它领域中，基本知识变化缓慢，以致于 10 年前毕业的某人所学到的知识在现在仍没有什么变化。而在软件开发中，即使基本的知识也发展迅速，在你以后 10 年毕业的某个人可能学到了二倍

于你的有效编程方法，一些老的程序员往往被另眼相看，不是由于他们对某些特定方法缺乏接触，而由于他们在走出校门后对一些闻名的基本编程概念缺乏了解。

在其它领域中，你今天在工作中学到的东西可能对你明天的工作有所帮助，在软件开发中，如果你不改变你在使用从前的编程语言中的思维方式或你在你的旧机器上得出的代码调试方式的习惯，你的经验将不值一文。许多进行软件开发的人往往花费时间准备上一次的战斗而不是下一次，如果你不因时间而作出应变，你的经验与其说是帮助倒不如说是一个阻碍。

除了软件开发中的迅速变化外，人们常从其经验中得出错误的结论，客观地对自己进行检查是困难的，你也可能忽视经验中使你能得出不同结论的重要之处，阅读其它程序员的研究材料是有益的，因为研究材料揭示了其它人的经验—它们都经过充分的精炼，你可客观地对其进行检查。

人们也往往荒唐地强调程序员的经验。“我们需要有五年以上 C 语言编程经验的程序员”就是其中一例，如果一程序员在头一、二年没有学 C 语言，第三年学也不会产生很大区别。这种类型的经验和其工作能力没有多大区别。

在程序开发中，知识更新迅速使此领域中“经验”处在一种奇怪的地位上，在其它许多领域，过去有着成功历史的专业人员，往往令人放心，并且因其一串成功的事情而得到尊敬。退步很快的人将很快和潮流格格不入。为了使自己有所价值，你必须紧跟潮流。对年青的、求知欲旺盛的程序员，他们往往在这点上有优势，而有些老的程序员认为自己有所资格了而讨厌一年接一年都要证实自己的能力。

最后一个问题是：如果你已工作了 10 年，你得到了 10 年的经验应当是真正的经验，你如能坚持不断地学习，你就能得到经验，如果你并不想学到什么，不管多少年你也学不到什么。

计算机迷

如果你还没有至少在一相同的项目上花费一个月的时间——一天工作 16 个小时；为了发现你的程序中最后一个错误睡眠中你也念念不忘它，你接连几天没日没夜地工作——即使你所编的程序并不复杂，那么你可能不会意识到编程中有某种令人兴奋的东西。

Edward Yourdon

这种对编程的痴迷纯粹是胡闹，并且几乎注定要失败。但是那些通宵程序员使你觉得他们是世界上最好的程序员，但是随后你不得不花费几周的时间来修正你在这短时间的辉煌中所带来的错误，你可能对编程非常热爱，但是你应能冷静地处理这个问题。

31.9　习惯

好的习惯起作用是由于你为一个程序员所作的大部分事情是你在无意识中所完成的，例如，有时你可能会感到以前爱采用缩进循环，但是现在每当你编写一个新的循环时你不会这样想了。这种情况确实在建立程序格式时存在。你最后一次向自己提出这个问题是在什么时候？如果你已经有五年实际编程经验，你就存在较多的机会，如果你最后一次向自己提出疑问的时间在 4 年半之前，剩下的便是受习惯的支配时间了。

你在许多地方都存在习惯。例如，程序员往往爱仔细地检查循环变量而少检查赋值语句，这就使得发现赋值语句中的错误要比发现循环变量的错误困难一些。你能对别人的批评作出

友好或不友好的反应。你一直在寻找使代码可读或编码速度更快的方法，也可能你无意寻找它们，如果你不得不在可读性和编码速度方面作出选择，你每次都会作出相同的选择，当然，你并不是在真正选择；你是在习惯性地作出反应。

成为某方面好的或差的程序员，主要是靠你自己的所作所为，建筑师要通过建筑而程序员要通过编程。你所作成为习惯，决定了你的编程品行，最终，你的习惯好坏决定了你是否能成为一位好的程序员。

微软公司的 Bill Gates，Chairman 和 CEO 曾说过，任何好程序员在开始的几年都做得很好。从那以后，程序员的好坏便基本定型了。在你进行编程很长一段时间后，很难见到你突然说“我怎样才能依循环进行得更快呢？”或“我怎样才能使代码更可读呢？”这些都是好的程序员一开始便养成的习惯。

当你开始学某一件事时，你应按正确的方式学好它，当你开始学时，你已对其进行了思考，并且你可在正确或错误的途径间作出轻易的选择，在你作过一段时间后，你对你所作的不太注意，此时“习惯的力量”会开始起作用。确保起作用的习惯是你所希望的。

如果你没有养成最有效的习惯你应怎么办？对这些问题没有一个明确的答案，以下是对此问题的部分回答，你无法用没有习惯取代坏的习惯，这就是为什么突然停止抽烟或节食的人如果不用一些别的什么替代的话会存在很大困难的原因。用一种新习惯代替旧习惯比完全戒除旧习惯要容易一些，在编程中，应尽力养成良好的习惯。你应养成在编写代码之前编写 PDL（流程图）和在编译之前阅读代码的习惯，你不必为失去坏习惯而多虑。在用新习惯取代后坏习惯会自然而然消失的。

31.10　小结

- 你的个人性格直接影响你编写计算机程序的能力。
- 最有明显作用的性格为：谦虚、好奇心、诚实、创造性和纪律，还有文明的“懒惰”。
- 高级程序员的发展和生成与天才并无多大联系，任何事情都和个人的发展有关。
- 令人吃惊的是，小聪明、经验、坚持和欲望既可帮助你也能妨碍你。
- 许多程序员不主动去吸收新信息和新技术，而是靠偶然地上获得一些新信息，如果你抽出少量时间学习别人的编程经验，过一段时间后，你将在你的同行中脱颖而出。
- 好的性格对养成良好习惯有很大影响，为了成为一位高水平的程序员，你应养成良好的习惯，其余的就会随之而来。

第三十二章　软件开发方法的有关问题

目录

本书着重于软件创建的细节：高质量子程序、变量名、循环、代码设计、系统综合等等，本书不着重于抽象的讨论而是强调有关具体的主题。

一旦你对本书以前各部分的基本概念有了一定的了解，你只需从各章中选取不同的主题并了解它们之间的联系，以得到对抽象概念的了解，本章是为了明确一些抽象主题：复杂性、抽象、过程、可读性、重复等等，这些内容对软件开发有较大的影响。

32.1　克服复杂性

计算是唯一可将某一种思想用 1 比特到几百兆比特来度量的领域，其所用比特数比可达 1 到 10^9，这样庞大的数字是令人惊讶的。Edsger Dlikstra 这样表达：“和这样级别的数相比，一般的数学方法是无能为力的。通过激起对更深的概念等级的需求，计算机向我们提出了以前从没遇到过的挑战。”

计算机科学的核心是减少复杂性，尽管谁都希望成为一个英雄并能自如地解决各种计算机问题，但没有人真正能有处理 10^9 级数的能力，计算机科学和软件工程已经开发了许多工具以处理这样的复杂问题，本书的其它部分也或多或少的触及了这样的问题。

减少复杂性的方法

在软件结构级上，问题的复杂性可以通过将系统分成子系统而得到降低，子程序越独立，复杂性就越得以降低，你应仔细定义模块，这样你才能在某一时间集中于某一件事情，而将代码组装成目标有许多好处。

使用全局数据是有害的，因为它可以削弱某一时刻你集中于某事的能力，使用相同的标记符对模块数据是无害的，因为它不会使你的注意力分开。

如果你所设计的子系统间有全局变量共享或粗糙定义的接口，你也会遇到许多复杂的问

题，其结果是冲淡了因将系统分成许多子系统所带来的好处。

复杂性应可通过好的设计得到最大程度的降低，降低复杂性也是促使代码质量提高的动机。将控制结构限制在对 if 和 for 语句的使用，以及使用无循环代码可以降低程序的复杂性。使用 goto 语句也是非常有害的，因为这些语句并不遵循特定的模式，你不能简化运算而降低复杂性，在子程序尽量写短一点。限制循环的嵌套和 if 语句的使用，限制传送给子程序的变量数都是降低复杂性的有效方法。

当你对一个布尔函数进行复杂的测试并抽象化测试的目的时，你就降低了代码的复杂性，当你将复杂的逻辑链用查询表替代时，你同样降低了复杂性，为主要的数据结构编制具体实现细节，从而可简化了工作。

编码约定在某种程度上同样可降低复杂性、标准化程序的格式、循环和变量名时，你就可省下你的精力处理有关程序编制中更富挑战性的有关问题，对代码约定有争议的原因是，其中选择是有限的并且约定有时是武断的。即使是对最细微的情节人们也往往有着最热烈的争论。在使你免受作出并武断决定方面，约定是最有用的。在一些有意义性领域，约定限制的价值有所降低。

使用层次结构对大部分人来说是很自然的，当他们画一个复杂的物体如房子时，他们也是分层画出来的。首先画房子的轮廓，然后是窗户和门，最后是其它细节，他们并不是将房子一块砖一块砖、一块瓦一块瓦、一个钉一个针地画出来的。

在一典型的计算机系统中，将有以下几种递升的层次结构：机器指令、操作系统运行、高级语言程序编制和用户接口操作。对于高级语言程序员，你仅需知道高级编程和用户接口即可。它可使你免受与机器指令打交道以及对最底层的操作系统调用的痛苦。

如果你擅长设计软件系统，除了现成可用的层次之外，你也可以通过编写高级语言程序创建你自己的层次。

抽象和复杂

抽象是另一种通过在不同的层次上处理不同的细节来降低复杂性的。当你使用集合时，你也就使用了抽象，如果你称某物体为房子，而不是玻璃、木材和钉子的混合体时，你也是用的抽象。同样，你将许多房子的集合体称为“小镇”时你也用到了抽象。

抽象是比层次更为一般的概念。层次意味着阶梯式、结构化的组织，抽象并不就是层次的结构化组织。抽象是通过将细节分布在各部分之间而不是层次的各级上来降低复杂性的。

通过对程序的各部分的抽象可以大大提高程序编制的质量。 Fred Brook 曾说过计算机科学取得的最大进步是从机器语言向高级语言的进化——它使程序员从对硬件的很大依赖环境中解放出来，以便于集中精力进行程序开发。子程序的使用也是一个大的进步。DBMS 的使用同样是一个飞跃，因为它们本质是抽象数据，使你能对记录和表而不是单个数据项进行处理。

按作用给变量取相应的名字，正如对计算机的问题是使用“What”而不是使用“how”一样，也可以提高抽象的级别，如果你说：“哦，我正在弹出堆栈，这意味着我最近得到了某个职位。”你当然可以不说：“我正在弹出堆栈”而只需说：“我最近得到了某个职位。”使用命名常量要比直接使用常量更具有抽象性，抽象数据类型也可以降低复杂性因为它们允许你使用和实际数据特性有关的数据类型而不是计算机的数据结构类型。面向对象的程序编制能同时对算法和数据进行一定的抽象——这种类型的抽象是功能分解所不能提供的。

概括地说，软件设计和编码的主要目标是克服复杂性。许多编程风格的目的也就是降低的复杂性。对程序员来说，降低复杂性是一个很关键的问题。

32.2　精选开发过程

本书的第二个主要观点是，软件开发和相应的过程有很大的关系。

对于一个小的项目，程序员的才能对软件质量有着最大的影响。程序员对过程的选择对自身的成功有一定影响。

对有多个程序员的项目，整个组织的特征要比单个程序员的技能更起作用，即使你所在开发组规模很大，集体能力也不是个人力量的简单相加，人们在一起工作的方式，决定了其所在集体力量的大小。整个组的工作过程决定了一个人的工作对组内其它人的影响是有益的还是起阻碍作用。

过程起作用的一个例子是在开始设计和编码之前，没有使要求稳定所产生的后果。如果你不知道自己究竟在创建些什么，你就不能得到一个较好的设计，如果在软件开发过程中，需要对要求和随之而来的设计进行修改，代码也必须作修改，这就势必降低整个系统的质量。

“当然”你说，“在现实中没有真正一成不变的要求，所以这只能分散人的注意力”。反过来，你所采用的过程也决定了要求的稳定性，如果你想对要求更有灵活性，你可以计划分阶段交付软件而不是一次交付完毕。这是你在开发过程中应注意的。你所用过程最终决定了项目的成功或失败，在 3.1 节中的表 3.l 清楚地表明了改正分析错误要比改正设计或代码错误费事得多，所以对项目中某部分的强调也影响总代价和进度。

以上原则对设计同样适用，在你开始建造之间应有一个牢固的基础。如果你在创建完成之前就急着编码的话．以后再对结构进行重大的修改是困难的，人们将会对其进行感情投资，将会继续为自己设计编写代码。一旦你已经开始建造房子后，要想再改变基底是困难的。

过程起作用的主要原因，是由于在软件中从一开始便应牢牢控制质量，有时你可能认为自己可随心所欲地编码，然后再测试以检查出所有错误，这是十分错误的。要知道，测试仅仅告诉你软件中的错误，测试不能使你的程序更有用、运行更快、长度更短、更可读或者更广泛。

不成熟的乐观是另一种类型的处理错误。在有效开发过程中，你在刚开始可能会作出粗糙的调整而在最后会作出好的调查。如果你是一个雕刻师，你可先在对细节进行修饰之前，粗糙地雕刻出大概轮廓。不成熟的乐观是对时间的浪费，因为你花费时间修饰了其实不必修饰的代码。你可能对较短和运行较快的代码段进行了修饰，你也可能修饰了今后将弃之不用的代码，你应常这样想：“我这样作是否正确？改变顺序是否会产生差别?”这样你就能有意味地遵循好的过程。

低级过程同样如此。如果你采用 PDL 流程图，并围绕 PDL 设计代码，你将会因采用自顶向下的设计过程而受益匪浅。你也应及时对代码进行注释。

对大、小过程的观察意味着你停下来留神自己是怎样创建软件的，这是值得你花费时间的。认为“代码才是真正有作用的，你应侧重于代码的质量，而不是某些抽象的过程”是缺乏远见的，它忽视了许多实际的证据。软件开发是一种创造性的活动，如果你并不理解这是一个创造性的过程，你就不能很好地利用你进行软件创建的工具——你的大脑。坏的过程浪费你的脑力，而好的过程则能很好地利用你的脑力。

32.3　首先为人编写程序，其次才是计算机

你的程序好似迷宫一样，你的程序充满了自作聪明的东西和不相关的评论。
我的程序算法精确，结构紧凑，有效率并且注释得体。

Stan Kelly ——Bottle

另外，本书一直强调代码的可读性。和其它人交流是自我检查代码好坏的真正动机。

计算机当然不会在意你的代码是否可读，对计算机来说，它更能“阅读”二进制机器指令而不是高级语言程序，你之所以编写可读的代码是为了帮助别人阅读你的代码，可读性对程序的以下几个方面有着积极的影响：

- 可理解性
- 可检查性
- 错误率
- 调试
- 可修改性
- 开发时间——受以上各种因素的影响
- 外部质量——受以上各种因素的影响

在程序开发的启蒙时期，程序通常被认为是程序员的私人财产。人们对未经同事的允许就查看其程序这类事的看待如同是你私拆别人的情书一样，这也正是程序的实质之所在，程序如同是程序员写给硬件看的情书一样，其中亲密的细节只有当事者自己清楚。于是，程序员们便为亲昵的称呼所笼罩，程序员们对极乐抽象世界的情人是如此着迷，以致于只意识到它们的存在，这样的程序员在局外人看来是不可理解的。

——Michael Marcotty

可读代码的编写并不费时，并且编写可读代码最终对你是有益的，如果你的代码可读性好，你也就能更容易地弄清楚你的代码情况。但是在评审过程中也需阅读代码，在你或别人改正错误时同样需要读代码，而当其它人试图利用你的某部分代码时，他也需要阅读你的代码。

在软件开发过程中代码的可读性并不是可有可无的。你应力争一次就编写出好的代码，而不再费神去阅读较差的代码，否则你可能得返工好几次。

“我所编的代码是为自己而写的，为什么要使它可懂?”，有人说。因为你这一、二周一直在开发另一个程序，“Hey!我上周已编写了这个子程序，现在我需看看这已受过测试和调试的代码以节省一些时间”，如果代码是不可读的，那么就够你受的了。

如果你所编的代码是不可读的，你就好比是本项目中的 Lone Ranger，是令人厌烦的。

你的母亲肯定这样对你说过：“你为何一脸愁情?”习惯影响你的工作，你是不能随意控制习惯的。所以你应确信你应在作的事情有助于你养成良好的习惯。

区分别人所编的代码是否是有争议对你是有益的。Douglas Comer 曾为私人和公用程序提出了一个有用的区分标准：“私人程序”是程序员自己使用的程序，它们不为其它人所使用，也不应被其它人所修改，通常都是琐碎的。“公用程序”是可为其它人而不仅是编写者本人所使用和修改的程序。

公用和私人程序的标准是不同的。私人程序可随意编写并且可充满各种限制——并不影

响其它人而仅影响制作者本人。而公用程序则应该更为仔细地编写：对它们的限制应进行说明，它们应该是可靠的和可修改的，而尾部往往正是这样，在其进入一般流通之前，你应将子程序转换成公用于程序，并且应注意子程序的可读性。

即使你认为只有你自己才阅读代码，实际上却存在其它人需修改你的代码的许多机会。一项研究发现在维护程序员重写某一程序之前往往需仔细研究程序 10 次，维护程序员通常要花费 50％到 60％的时间理解他们将维护的代码，他们真希望你能对其进行有关文档工作。

以前各章提供了使程序可读性好的方法：好的变量名、良好的格式、好的子程序名、小的子程序，在布尔函数中隐含复杂布尔条件测试、在复杂计算中加中间变量等等。没有一种简单的方法能得出可读性较好和不可读程序的差别。

如果你认为你没有必要使你的代码可懂，因为没有任何人读你的代码，你应确信你没想错。

32.4　注意约定使用

约定是复杂性管理的一种有效工具，以前各章曾提到了一些特殊的约定。本节使用许多例子讨论约定所带来的好处。

程序开发中的许多细节在某种程度上是武断的，在一循环中你应缩进多少格？你怎样使用注释格式？你怎样安排子程序变量？以上大部分问题都有几种正确的回答，约定可使你免于回答这个问题，它是反复使用相同的武断决定，对有许多程序员的项目来说，使用约定可以防止由于不同的程序员的决定不同所引起的混淆。

约定能精确地传递重要的信息，在命名约定中，单一字符可区分局部和全局变量、模块等；字母的大小写可以准确地区分排序型、命名常量和变量。缩进约定可精确地显示程序的逻辑结构，对齐约定可准确地表明相关的语句。

约定可防止一些已知的危害，你可通过建立约定排除危险的习惯，限制危险习惯在一定场合的使用，或者弥补其带来的危害，例如，通过禁止全局变量和在一行中写多条语句的使用，排除这些并不正确的用法。你也可以限制 goto 语句只在其转移到子程序末尾时才能使用，你还可要求复杂表达式中使用括号，或要求在一指针被释放后马上将其清除掉以防止悬挂指针的出现。

约定可提高低级任务的预测性。对处理存储器请求、错误处理、输入 / 输出和子程序接口使用约定可使你的代码结构良好，并使其它程序员易于理解你的代码——只要这些程序员了解你的约定。正如前几章中所指出的那样，不用 goto 语句使你能排除一种非约定的控制结构。想来读者都对 for、if 或 case 语句有一个粗略的了解吧。但是你很难知道 goto 语句是向上或向下跳转。goto 语句的使用增加了读者的不确定感。有了好的约定，你和你的读者就能想到一起了。需优化的细节有所减少，这反过来会提高程序的可读性。

约定可以弥补语言的不足，对于不支持命名常量的语言，约定可以区分可读写变量和只读变量，限制 goto 语句和指针的使用也是约定弥补语言缺点的例子。

大项目的程序员有时对约定似乎是使用过分了，他们建立了许多标准和原则以致要想记住它们是要费不少力气的，但是小项目的程序员又似乎对约定使用不够，他们并没有完全认识到约定所带来的益处。你应能了解约定的真正价值并能正确地利用它们，使用约定以提供所需

要的结构。

32.5　根据问题范围编程

处理复杂性的一个特定的方法是在最高可能的抽象级上工作。而在最高抽象上编程的一种方法是根据问题而不是计算机进行编程。

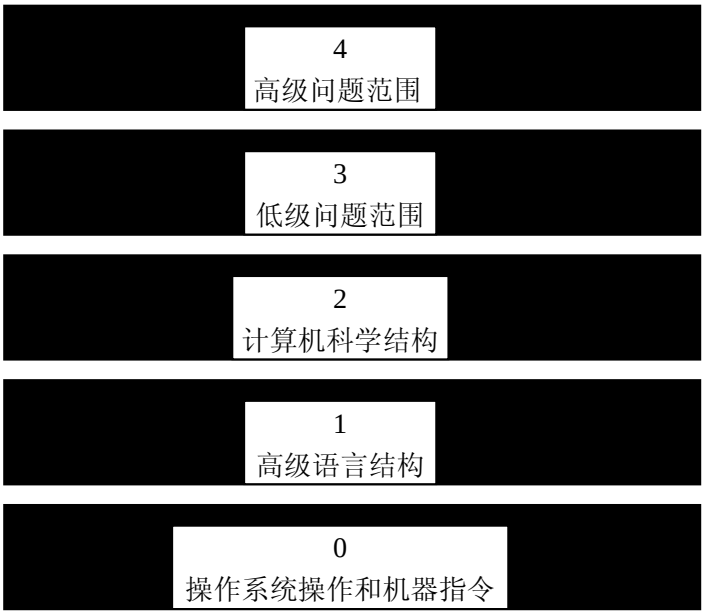
高级代码不应被以下细节所填满:文件、堆栈、队列、数组和一些字符如 i、j、k 等。高级代码应是对要解决问题的描述，它应由确切表明程序功能的描述性子程序封装而成，而不是充斥，诸如打开一个文件之类的细节。高级代码也不应包含如：i 在此处代表雇员记录变量，以后它将用作顾客帐目文件变量…这类注释。

以上都是笨拙的编程作法。在程序的顶层，你无需知道雇员数据是来自记录或作为文件存储。在这种级上的信息应是隐含的。在最高级上你不必知道数据是如何存储的。你也无需阅读一条解释 i 在这儿是什么意思和 i 用作两个目的注释。如果 i 被用作了两个目的，你就应看到为这两个目的而设置的两个不同变量，它们应是两个有特色的名字：EmployeeIdx 和 ClientIdx。

将问题分解成不同的抽象级

很明显，你不得不在计算机科学范围的某些级上工作，但是你可将你的程序分为计算机科学范围和问题范围这两方面的工作。

如果你使用高级语言编程,你无需担心最低级——你的高级语言会自动地考虑这个问题的。你应考虑较高级的情况，但是大部分的程序员没有做到这点，如果你在设计一个程序，你应



至少将其分成以下抽象级：

第一级：高级语言结构

高级语言结构是原始数据类型，控制结构等等，你使用高级语言是很自然的，因为没有它们你就无法编写高级语言程序。许多程序员从来不在此抽象级之上工作，这就使他们遇到不少困难。

第二级：计算机科学结构

计算机科学结构级别比高级语言结构级稍高一些。它们往往是你在大学课程中所学到过

的操作和数据类型——堆钱、队列、链表、树、索引文件、顺序文件、排序法、搜索算法等等。虽然本级比高级语言结构级要高，仍有许多的细节工作要作，以克服复杂性。

第三级：低级问题领域

在本级中，你需要对问题的原始描述。为了在这级上编写代码，你应清楚问题的专业术语和积木块，以便能解决问题。在本级中，子程序还不能直接解决问题，但是它们可以用来生成更高级的子程序以便直接解决问题。

第四级：高级问题领域

本级提供了在本问题上的抽象能力。你在本级上的代码在某种程度上应是可读的——一般的应用人员即可读懂。本级代码并不完全依赖于你编程所用高级语言的某些特征，因为你创建了自己处理问题的工具。因此，在本级上你的代码更依赖于工具而不是你使用语言的能力。

细节的实现隐藏在两层外壳下面，在计算机科学结构层中，硬件或操作系统的变化对其不会有什么影响，本级上也包含用户的见解。因为当程序变动时，它将接受用户的见解而作修改的。问题范围的变化对这层有较大影响，但是通过编制问题领域的积木块，此变化应是容易协调的。

在问题领域采用低级方法

即使没有一完整的有组织的方法能适用于问题领域，你可以使用本书中的有关方法以解决实际问题而不是对计算机科学问题。

- 在问题领域使用抽象数据类型以实现有实际意义的结构。
- 隐含有关计算机结构和实现细节的有关信息。
- 在面向对象的程序中，设计和问题有直接关系的成员，压缩问题领域中的有关信息。
- 对有意义的字符串和数字使用命名常量。
- 对中间计算结果使用中间变量。
- 使用布尔函数以净化复杂布尔测试。

32.6　当心飞来之祸

程序编制既不是完全的一门艺术也不是一门科学，更不是二者的神圣的结合。在某种程度上它是介于二者之间的一门技艺。在创建某一软件产品过程中，它仍然需要大量的个人判断，在程序编制中，作出好的判断，也要求微妙问题的警告信息作用反应。警告信息提醒你程序中的可能错误，但是它不会像路标上那样醒目地提示：“当心飞来之祸”。

当你或其它人说到“这真是一个棘手的程序”时，这通常是对不好的代码的警告信号。“棘手的程序”是“坏程序”的代名词，如果你认为某子程序是棘手的，你可以考虑重写它。

比一般子程序含更多错误的子程序是一种警告信号，少数一些出错的子程序含有更多的错误，你可能还会出现错误，此时你应重写它。

如果程序编制是一门科学，每条错误信息就表明一特定明确的改错方法，因为程序编制只是一门技艺，警告信息仅指出了你应考虑的问题，这并不意味着你一定要重写子程序才能提高其质量。

正如某子程序所出现的反常错误数，警告你子程序的质量低，而程序中所出现的反常错误

数则说明你的整个过程中存在问题的。好的开发过程中不允许出现有错误的代码，在结构评审后应对结构进行检查和平衡，设计评审后应修改设计，代码评审后应进行代码修改。但代码接受测试之前，绝大多数代码应得到检查，除了工作努力以外，你还应认真仔细对项目检查多次调试意味着欠缺认真，在一天中编写许多代码然后花费二周的时间调试也是不细心认真的表现。

你也可将设计度量用作一种警告，大多数设计度量在给出设计质量上颇有启发性的：某一占二页多长打印纸的子程序并不就意味着设计得不好，但是它说明子程序是复杂的，同样，在子程序中有 10 个以上判决点、三层以上逻辑嵌套、不寻常的变量数目和其它子程序的异常耦合或内部代码欠紧凑也都是警告标志；但是这并不意味着子程序编得不好。只是其出现会使你以怀疑的眼光看待你的子程序。

任何警告信息都可能使你怀疑你的子程序的质量。正如 Charle Saunder 所说的：“怀疑使我们不舒服和不满意，从而难以自我解脱”。将警告信息视为对“怀疑的激励”这样就促使你追求更完美的境界。

如果你发现自己老在编制相同的代码或作相似的修改，你将会“不自在和不满意”，并怀疑子程序或宏所使用的控制是否合适，如果你因不能自如地实现对子程序的调用，而发现要创建测试事例的骨架是困难的，你将会产生怀疑并问自己本子程序是否和其它子程序耦合过紧，当你因子程序的缺乏独立性而不能重新利用其它程序中的代码时，这也是子程序耦合过于紧密的警告信号。

当出现警告信息时，你应对此有所注意，因为意味着你的设计并不是很好，以致不能自如地编码，当你在编写注释、命名变量、分解问题将其各方面分给良好定义的子程序等方面存在困难时，这就说明在编码之前你需更深入地考虑设计，平凡的子程序命名和使用一行注释发生困难也同样是不好的信号，当你心中对设计有一个清晰的了解时，在低级细节上进行编码是很容易的。

如果你的程序让人难以理解的话，你对此应有所警觉，任何不安都是暗示。如果你自己都难以理解，别的程序员就不用提了。他们是希望你努力提高可理解性的。如果你不是通过阅读来理解代码的，这也是太复杂了。真是这样的话，你应简化它。

如果你想充分利用警告信息，你应在程序编制中创建自己的错误警告。这是相当有用的，因为即使你知道此警告，你也很易忽视它。 Glenford Myers 在错误改正的研究中发现，没发现错误的最常见的原因只是因为忽视了它们，这些错误在测试输出中是可见的，但是未引起注意。

在你编程时不应忽视有关问题。其中的一个例子是在释放指针后将其置为 NULL 或 NIL，否则你误用它们后可导致糟糕的问题。即使在释放后，指针也可能会指向有效的存储单元。将其设置为 NULL 或 NIL 可确保它指向一无效的存储单元，这样可避免错误。

编译警告是文字警告，它们也往往易被忽视，如果你的程序出现了警告或错误，你应改正它们，当你对印有“WARNING”这类警告信息都忽视时，你不大可能注意到其它微妙的错误。

为何注意警告错误在软件开发中这般重要？你在程序开发中的仔细程度在很大程度上决定了程序的质量，所以对警告信息的注意程度对最终产品有所影响。

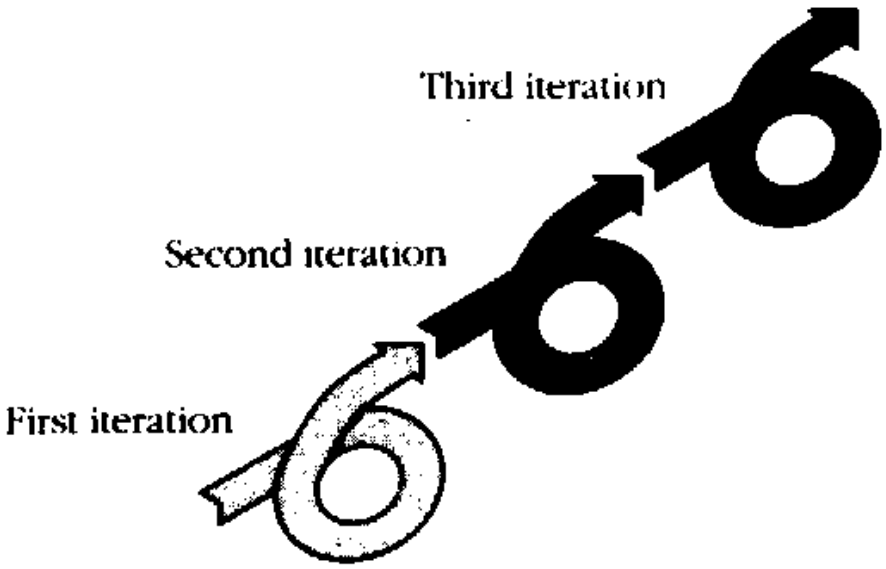
32.7　重复

在许多软件开发活动中，重复是合适的。当你开始系统的开发时，你会同用户商量需求的各种形式直到你满意为止，这就是一个重复的过程，如果你想通过升级获取更大的灵活性，你将分阶段交付系统，这也是一个重复过程。如果在制作最终产品之前你迅速而又廉价地开发出了几种不同的方法，这也是另外一种重复，在最初阶段，重复可能同开发过程的其它任何方面一样重要，在开始研究不同的方法之前，局限于某种解决方法，项目就会失败，在你开始创建之前，重复可使你对产品有所了解。

正如第二十二章对创建管理的讨论所指出的那样，在初始计划阶段，由于所使用方法的不同，进度评估的结果会有很大差别，使用重复方法比单一方法更能得出精确的估计。

软件设计和其它所有开发过程一样，是要经过逐步修改和提高的，软件往往是被确证有效而不是被证明，这意味着它将被重复测试和开发直到能正确地回答问题为止。高级和低级设计都应被重复，第一次尝试所产生的结果也许是可行的，但并不就是最好的。使用不同的方法重复进行就能加深对问题的了解。

重复方法对代码调整同样有益。一旦软件是可运行的，你可重写小部分代码以提高整个系统的性能。然而，许多优化并不会对原来的代码有益而只会降低其性能。这并不是一个凭直觉就能把握的过程。有些方法看起来使系统较小和运行速度较快，而实际上使系统更大和运行速度更慢。对任何优化方法的效果不确定性往往要求反复地调整和度量。如果它是整个系统性能的瓶颈，你应对代码进行多次调整，你最终的几次尝试可能比最初的几次更为成功。



至少，重复有助于提高产品的描述、计划、设计、代码质量和其性能。

评审可使开发过程少走弯路，在评审过程的任何阶段都可插入重复，评审的目的是检查在特定时间内的工作质量，如果产品在评审中未通过的话，它将会返工。如果通过了，也就不需要重复了。当然如果你走向重复的极端的话，你将如 Fred Brook 所说的那样，创建一个你将废弃不用的东西，工程学的一个任务，是其它人用 1 美元来完成的，你用一角钱来完成它。将整个系统废弃不用就是别人用一美元能完成的事你得用二美元才能完成。在软件工程中，你应尽可能迅速地将要报废的事情处理掉。

32.8　不要固执己见

偏执在软件开发过程中有着各种变种：顽固地地坚持某种设计方法，执着于某种特定格式或注释风格，一味地不用 goto 语句，它都是不正确的。

判断

不幸的是，一些专业优秀人员往往更容易偏执。在新方法开始流行之前应得到充分的证实，研究结果向最初者的传播称为“技术转移”，它对提高软件开发的最初状况有着重要的影响，忧而传播一种新的方法和卖狗皮育是不同的两回事。人们对于一种方法的传播往往是这样理解的。叫卖者向人们兜售自己的东西时，极力鼓吹他的东西是如何灵验、试图以此来说服别人、这些叫卖者要人们忘掉过去已学的一切，因为本方法能将你的效率提高 100% 。

当你碰到这样的人时，应问问他使用新方法工作了多长时间，他可能会说。“很好我已经使用了 15 年，许多客户都已证明了我这种方法的有效性”。废话，他不会说许多人采用他的成果而浪费了 800 美元。你可继续追问他自己亲自使用了本方法有多久、通常地不仅没有使用此新方法开发某一系统，甚至也没有用他提出要抛弃的旧方法开发过系统！

你不应对流行的时尚充耳不闻。应取其之长，补己之短，但是应保留好原来的一切。

选择

对某种方法的盲目迷信，会阻碍你对编程问题最有效解答，如果软件开发是一个确定的。精确过程，你可按一套僵化的方法以寻求问题的解决。实际上，软件开发是靠经验的，此时僵化的过程是不合适的，其成功的希望更是微乎其微。例如，在设计中，有时自顶向下分解法有效而有时面向对象的方法，自底向上分解法或数据结构方法更有效。你应有尝试几种途径的愿望，知道有时会成功或失败，但是在你对其尝试之前是不会知道结果的。

对某一种方法的固执己见也是有害的因为它只能使你勉为其难地解决问题。如果你在充分了解问题之前，就得出解决问题的方法的话，这是不明智的，你因此而限制了各种可能的解决方法并且有可能排除了最为有效的方法。

刚开始你可能对任何新方法感到不安，让你不要过于偏执的建议并不是让你在用新方法解决问题遇到困难时就停止采用新方法。参考本书和其它途径所提供的各种方法你也应抱有选择的态度、对几方面问题的讨论已经得出了许多你不能同时使用的方法。每一特定问题你应选择一种或几种方法。你也应将方法视为工具盒中的工具，并在工作中利用自己的判断选择最好的工具。在大部分时间内工具的选择是无关紧要的。而有些场合，工具的选择是重要的，所以你应仔细地作出选择，工程学在某种程度上是对各种方法作出综合评估，如果你早早就将自己的选择限制在单一工具上，你就无法作出衡量。

工具箱比喻是有用的，因为它使具体的选择抽象化了。在高级层次上，你有几种选择方法，你可选择几种不同的数据结构之一以代表任何给足的设计。而在更具体的级上，你可在格式、释、代码、命名变量、子程序参数传送方面选择几种不同方案。

顽固的态度和可选择软件创建工具箱方法是矛盾的，也和创建高质量软件所需的态度是不相容的。

实验

选择和实验有着密切的联系，在整个开发过程你应坚持实验，但是偏执可能会阻止你这样作，为了有效地进行实验，你应能随机应变。否则，实验只徒然浪费你的时间。

在软件开发过程中，许多人的偏执往往是由于害怕发生错误，试图避免发生错误是最大的错误。设计正是一个仔细地计划小错误而达到避免大错误的过程。软件开发中的实验是创建测试的过程，这样你便能知道一种方法是失败或成功——实验本身就是一个成功地解决问题的过程。

在许多级上，实验和选择一样合适，在你准备作出选择的每一级上，你都能提出对应的实验，以决定哪一种方法最佳。在结构设计级上，实验包括使用不同的方法勾画出软件的结构。在细节设计级上，实验可能包括使用不同的低级设计方法，遵循高级结构的有关规定。在编程语言级上，实验包括编写一短的实验性程序，以检查你并不很熟悉的语言的部分功能。这种实验可能会调试代码并度量它，以确证它是真正变短了或运算速度加快了。在软件开发过程中，实验可能是收集质量和效率数据，以了解检查是否比普查要能发现更多的错误。

重要的是你应对软件开发的各方面都保待开放的思想，这样你就能在开发过程中学到不少技术。开发性实验和对某种方法的顽固坚持是不相容的。

32.9　小　结

- 程序编制的一个主要目的复杂性管理。
- 编程过程对最终产品的影响比人们想象中的要大。
- 合作程序开发要求各成员之间进行广泛的交流，其次才是和计算机的交流，而个人则主要是和你自己而不是和计算机交流。
- 当被乱用时，约定好比是雪上加霜，而使用得当的话，约定可增加开发环境的有用结构，并有助于管理复杂性和交流。
- 面向问题而不是解答的编程有助于对复杂性的管理。
- 注意警告信息是相当重要的，因为编程几乎是纯智力活动。
- 在开发过程重复越多，产品质量也就越高。
- 武断的方法和高质量软件开发是不相容的。你应知道各种程序编制的方法，并能从中挑选出适合你工作的方法。

第三十三章　从何处获取更多的信息

目录

当你读到这里后，你应对软件开发过程很有了一些了解。人们可以获取很多信息，也许，你现在所遇到的错误是别人早期碰到过的。除非你是想自讨苦吃，你应阅读他们所写的书以避免他们所犯过的错误并找出解决问题的新方法。

33.1　软件领域的资料库

由于本书涉及了许多文章和其它书籍，想知道先读什么是困难的。

软件开发资料库由几类资料组成：解释有效编程基本概念的书，解释在程序开发中技术管理、知识背景的书，还有关于语言、操作系统、环境和硬件等一些对特定项目较为有用的参考书籍。

在所有其它书籍中，你有一些集中讨论主要软件开发活动的书籍，它们主要是关于分析、设计、创建、管理和测试等方面的书籍。

作者推荐的 10 种书

以下一些书理构成了软件开发活动中基本的资料库：

1．The Psychology of Computer Programming（1971 年）Gerald Weingerg，本书包含许多程序开发中的趣闻轶事。

2．Programming Pearls（1986）作者是 Jon Bentley。本书对程序开发进行了生动有趣的讲义，其对各种原因的生动阐述会使许多人觉得程序编制是一件有趣的事情。

3．Classics in Software Engineering （1979）作者是 Ed Yourdon，本书收集了软件工程的许多研究论文。

4. Principles of Software Engineering Mangemment（1988）作者是 Tom Glib，本书主要讨论管理，并有助于你加强对项目的管理。

5．Structured desigen（1979）作者是 Yourdon 和 Constantine。，本书主要讨论结构设计的有关方法和思想。

6. The Art of Software Testing（1979）作者是 Glenford。

7. 一本关于需求分析的书。

8. 项目计划和效率进行宣分析的书。
9. 本关于数据结构和算法分析的书。
10.一本从整体上讨论软件开发过程的书。

33.2　软件创建信息

作者之所以写这本书是因为作者当时还没有发现一本对软件创建进行详细讨论的书，当作者正在编写此书时，Prentice Hall 出版了 Michael Marcotty 所编的 Software Implememtation 一书（1919 年）。Marcotty 侧重于抽象、复杂性、可读性和修改，对创建进行了讨论。读者也可看 Prentice Hall 的 Practical Software Engineering 一书。

33.3　创建之外的主题

以下是似乎和软件创建无关的一些书籍。

软件创律的一般书籍

l. Classics in Software Engineering（1979），Ed Yourdon
2. Writings of the Revolution：selected Readings on Software Engineering（1982），Ed Yourdon
3. Tutorial：Programming Producitivity：Issues for the Eighties（1986），Capers Jone
4. Software Conflict: Essays on the Art and science of Software Engineering（1991）Robert L．Glass

软件工程总览

计算机程序员和软件工程师都应有一本关于软件工程的高级参考书。这样的一本书应是对基本方法的总览，而不是对具体细节的讨论了。它们应是对有效软件工程的概括，并总结了软件工程的有关技术。以下是有关书籍。

l. Software Engineering（1989 年），Ian Sommerville
2. Software Engineering：A practitioner's Apporoach（1987），Roger S. Pressman
3. Practical Handbook for Software Development(1985)，N.D.Birrell 和 M．A．Ould
4. Making Software Engineering Happen: A Guide for Installing the Technology（1988），Roger Pressman

用户界面设计

这类书籍主要讨论用户界面设计中颜色、响应时间，命令行结构、菜单设计等有关工程原则、参考书籍如下：

1. Design the User Interface: strategies for Effective human－Computer Interaction（1987）Ben Shneiderman
2．The Elements of Frieldly Software Design：the Mew Edition（1991），Paul Hechel
3．The Art of Human－Computer InterfaceDesign（1990），Brenda Laurel

4．The Psycholoss of Everyday things（1988），Donald A. Norman

数据库设计

不管你的程序的是大还是小，是否使用数据库，你知道数据库都是有益的，如果你的程序中用到了文件，你就是在用数据库。以下是二本参考书籍。

1．An Introduction to Database System（1977），Chris Date

2. Fundamentals of Database Systems（1989），Ramez Elmasri 和 Shamkant B. Navathe

正规的方法

正规的方法是通过逻辑变量而不是运行测试证明程序的正确性。正规方法还只被应用在少数项目中，但是其中一些已在质量和效率方面给人留下了难忘的印象，以下是一些参考书籍。

1. Software Engineering Mathematics（1988），Jim Woodcock 和 Martin Loomes。

2. Structured Programming: Theory and practice（1979），Richard C. linger，Harlan D, Mills 和 Bernard I. Witt

3. Social Process and proofs of Theorems and programs（1979），Richard A．DeMillo，Richard J. Lipton 和 Alan J. Perlis。

4. Program Verification：The Very Idea（1988），James H. Fetzer

33.4　期刊

初级程序员杂志

以下杂志可在本地书报摊下发现：

BYTE

Computer Language

Datamation

Dr Dobb's Journal

高级程序员杂志

这类杂志在当地书报排难以买到，你可去大学图书馆去查询或预订它们；

IEEE Software

IEEE Computer

Communication of the Acm

精装出版物

American Programmer

The Software Practitioner

Software Practice and Experience

专题出版物

1. 专业出版物
可向 ACM 或 IEEE 等组织去信联系
2. 通俗出版物
The C Users Journal
DBMS：Developing corporate Applications
Embedded systems Programming
LAN Techndogy：The Technical Resource for Network Specialists
LAN：The Local Area Network Magazine
MacWorld
PC Techniques
Personal Workstation：Configuring Windowed Networked Computers
UNIX Review
Software Management News
Windows Tech Journal

33.5　参加专业组织

学习程序编制的最好方法是和其它程序员进行接触和交流。你可加入本地有关组织。你也可加入 ACM 和 IEEE 等国际性专业组织。

代码大全中文电子版的诞生来源于 4 个月来下面列表中近 100 位网友出色的志愿工作。

全书扫描：Sequoia　项目组织：Bear

OCR 及校对进程 (2001-04-05 18:42:00 100% 完成)：

序号	志愿者	校对范围	送出日期	送回日期	状态	备注
0	Bear	序言目录	01-01-04	01-01-09	OK	
		360--369	01-01-10	01-01-13	OK	
1	Wu Ke	001--010	01-01-05	01-01-08	OK	Good!!
		210--219	01-01-09	01-01-12	OK	Good!
2	differ	011--020	01-01-05	01-01-07	OK	Good!!!
		250--259	01-01-09	01-01-11	OK	Thanks.
		260--269	01-01-09	01-01-11	OK	Thanks.
		270--279	01-01-09	01-01-11	OK	Thanks.
3	Li Yuan	021--030	01-01-05	01-01-10	OK	Good!
4	蚊子	031--040	01-01-05	01-02-01	OK	Good!
5	Jerry Xing	041--050	01-01-05	01-01-09	OK	Good!!
		400--409	01-01-10	01-01-14	OK	Good!
6	stephen	051--060	01-01-06	01-02-26	OK	Good! So late :)
7	Leeseon	061--070	01-01-06	01-01-08	OK	Good!
8	王晋隆	071--080	01-01-06	01-01-15	OK	Good!
9	meijg	081--089	01-01-06	01-01-08	OK	Good!!
		200--209	01-01-08	01-01-16	OK	Good!
10	Jimi	090--099	01-01-07	01-01-21	OK	Good!
11	Xuym	100--109	01-01-07	01-01-09	OK	Good!!
		300--309	01-01-10	01-01-22	OK	Good!!
12	longman	110--119	01-01-07	01-01-09	OK	Good!
13	mingnan	120--129	01-01-07	01-01-08	OK	Good!
		340--349	01-01-10	01-01-15	OK	Good!
14	minqiangshang	130--134	01-01-07	01-01-12	OK	Good!
15	深夜清风	140--149	01-01-08	01-01-08	OK	Good!!
		420--429	01-01-10	O1-01-11	OK	Good!
16	雨轩	150--159	01-01-08	01-01-10	OK	Good!!
		520--525	01-01-11	01-01-12	OK	Good!
17	eighth	160--169	01-01-08	01-01-10	OK	Good!
18	风中之烛	170--179	01-01-08	01-01-10	OK	Good!!
		330--339	01-01-10	01-01-16	OK	Good!
19	Zhangbin_$$$_	180--189	01-01-08	01-02-03	OK	Good!

20	benjamin	190--199	01-01-08	01-01-16	OK	Good!
21	kimfeng	220--229	01-01-09			wangzhe new
22	大海	230--239	01-01-09	01-01-11	OK	Good!
23	lij	240--249	01-01-09	01-01-12	OK	Good!
24	xafgs	280--289	01-01-10			xinjian new
25	javylee	290--299	01-01-10			liangyx new
26	LECON	310--319	01-01-10	01-01-10	OK	So quickly
27	喻强	320--329	01-01-10			无法下载
28	Mr Jiang	350--359	01-01-10			陈立峰
29	chencqs	370--379	01-01-10	01-01-10	OK	So quickly
30	张岩	380--389	01-01-10	01-01-10	OK	So quickly
31	萧萧	390--399	01-01-10	01-01-12	OK	Good!
32	laojiu	410--419	01-01-10	01-01-21	OK	Good!
33	freewei	430--439	01-01-10	01-01-11	OK	Good!!
34	HugeAnt	440--449	01-01-11			Shen_Ray
35	liangyx	450--459	01-01-11	01-01-17	OK	Good!
36	yunannan	460--469	01-01-11	01-01-11	OK	So quickly
37	RichardYu	470--479	01-01-12	01-01-17	OK	Good!
38	Cao	480--489	01-01-12			出差 zhangjinyu new
39	王晓初	490--499	01-01-12	01-01-15	OK	Good!
40	冰狐	500--509	01-01-12	01-01-12		Good!!
41	tecman	510--519	01-01-12	01-01-13	OK	Good!
42	杨立波	135--139	01-01-13		...	已寄去 on 01-02-13
43	dxx	090--099	01-01-16	01-01-21	OK	Good!
44	Brandon Wang	220--229	01-02-12	01-02-20	OK	VeryGood! kimfeng old
45	xinjian	280--289	01-02-13	01-02-27	OK	VeryGood! xafgs old
46	vdgame	320--329	01-02-14	01-02-16	OK	VeryGood! 喻强 old
47	Shen_Ray	440--449	01-02-16	01-02-23	OK	很棒！ HugeAnt old
48	liangyx	290--299	01-02-16	01-02-26	OK	Good! javylee old
49	陈立峰	350--359	01-02-19	01-02-20	OK	So quickly!
50	zhangjinyu	480--489	01-02-20	01-02-28	OK	很棒！Cao old
51	mayongzhen	135--139	01-03-20		...	给一个非 263 邮箱给我！
52	dongfang7	135--139	01-04-04	01-04-05	OK	Very Good!

格式化进程（2001-04-20 09:09:00 100%完成）

章	志愿者	送出日期	送回日期	状态	备注
1-6	Bear	01-01-04	01-01-09	OK	
7	Meijq	01-02-14	01-02-24	OK	虽页多，却很棒！
8	Libinbin	01-02-14	01-02-18	OK	Good!
9	barrycqj	01-02-09	barrycqj	OK-	差 135-139 ！
10	JSL	01-02-07	01-02-11	OK	Good!
11	Fadey	01-02-14	01-02-22	OK	很棒
12	杨立波	01-02-08	01-02-14	OK	Good!
13	Gyb	01-02-07	01-02-10	OK	Good!
14	getit911	01-02-09	01-03-19	OK	Good!
15	Gyb	01-03-20	01-03-21	OK	So Quickly! free51 has gone , wai_chow lodgue old !
16	ctx	01-02-09	01-03-01	OK	NoDoc 一个字一个字地敲上去的！
17	jiang chun yi	01-02-09		...	!
17	huzhiyan	01-03-21		...	联系不上，263！
17	Bear	01-03-30	01-03-31	OK	
18	dongfang7	01-02-10	01-02-15	OK	缺 280-299；但自己 OCR 了。 页虽多，却很漂亮！
19	mingnan	01-02-19	01-02-28	OK	Very Good!
20	SimonLee	01-03-05	01-03-20	OK	Very Good!金添 old
21	彭晖	01-02-20	01-03-01	OK	Very Good!
22	Brandon Wang	01-02-20	01-03-08	OK	Very Good!
23	Benjamin	01-02-12	01-02-14	OK	非常棒
24	锡山市堰桥中学	01-02-13	01-02-23	OK	Very Good! Steelim old for 出差了
25	lij	01-03-02	01-03-12	OK	Perfect !
26	winsun	01-02-12		...	01-03-04 重发
26	xieye	01-03-21	01-03-22	OK	Perfect ! is shcoking
27	vdgame	01-02-20	01-03-04	OK	Wonderful!
28	Russell Feng	01-02-20	...	...	2001-04-08 之前
28	Dongfang7	01-04-09	01-04-19	OK	Perfect!
29A	DeepBlue Eye	01-02-20	.	...	459-469
29A	Bear	01-04-09	01-04-09	OK	
29B	杨涛	01-02-28		...	470-486 过两天
29B	abc	01-03-27	.	...	Gone off on 01-04-09
29B	郭力	01-04-09	01-04-09	OK	Perfect!
30	libinbin	01-02-19	01-02-23	OK	Perfect!
31	JSL	01-02-20	01-02-25	OK	Perfect!

32	darkstar13	01-02-13	01-03-07	OK	Very Good!
33	vdgame	01-02-13	01-02-14	OK	Very Good!

统校进程 (2001-04-21 21:09:00 100%完成)：

章	志愿者	送出日期	送回日期	状态	备注
01-05	Bear	01-03-05	01-03-31	OK	终于完成了！
06-10	dongfang7	01-03-06	01-03-20	OK	Perfect! 完成 135-139
11-15	Yu Youling	01-03-06		...	给一个非 263 邮箱给我！
11-15	kingofwang	01-03-29	01-04-10	OK	Perfect!
16-20	zigzeg	01-03-06	01-04-19	OK	Good!
21-25	wslee	01-03-06	01-03-16	OK	Perfect!!!
26-30	hyj	01-03-08	01-04-10	OK	Perfect!
31-33	bgfish	01-03-06	01-03-09	OK	Perfect!

全书整合进程 (2001-04-29 0:15:00 100%完成)

章	志愿者	开始日期	结束日期	状态	备注
1-33	Bear	01-04-21	01-04-29	OK	Perfect!

One for all, all for one!

www.delphidevelopers.com