第 7 章　实 现

通常把编码和测试统称为实现。

所谓编码就是把软件设计结果翻译成用某种程序设计语言书写的程序。作为软件工程过程的一个阶段，编码是对设计的进一步具体化，因此，程序的质量主要取决于软件设计的质量。但是，所选用的程序设计语言的特点及编码风格也将对程序的可靠性、可读性、可测试性和可维护性产生深远的影响。

无论怎样强调软件测试的重要性和它对软件可靠性的影响都不过分。在开发大型软件系统的漫长过程中，面对着极其错综复杂的问题，人的主观认识不可能完全符合客观现实，与工程密切相关的各类人员之间的通信和配合也不可能完美无缺，因此，在软件生命周期的每个阶段都不可避免地会产生差错。我们力求在每个阶段结束之前通过严格的技术审查，尽可
能早地发现并纠正差错；但是，经验表明审查并不能发现所有差错，此外在编码过程中还不可避免地会引入新的错误。如果在软件投入生产性运行之前，没有发现并纠正软件中的大部分差错，则这些差错迟早会在生产过程中暴露出来，那时不仅改正这些错误的代价更高，而且往往会造成很恶劣的后果。测试的目的就是在软件投入生产性运行之前，尽可能多地发现
软件中的错误。目前软件测试仍然是保证软件质量的关键步骤，它是对软件规格说明、设计和编码的最后复审。

软件测试在软件生命周期中横跨两个阶段。通常在编写出每个模块之后就对它做必要的测试(称为单元测试)，模块的编写者和测试者是同一个人，编码和单元测试属于软件生命周期的同一个阶段。在这个阶段结束之后，对软件系统还应该进行各种综合测试，这是软件生命周期中的另一个独立的阶段，通常由专门的测试人员承担这项工作。

大量统计资料表明，软件测试的工作量往往占软件开发总工作量的 40%以上，在极端情况，测试那种关系人的生命安全的软件所花费的成本，可能相当于软件工程其他开发步骤总成本的 3 倍到 5 倍。因此，必须高度重视软件测试工作，绝不要以为写出程序之后软件开发工作就接近完成 了，实际上，大约还有同样多的开发工作量需要完成。

仅就测试而言，它的目标是发现软件中的错误，但是，发现错误并不是最终目的。软件工程的根本目标是开发出高质量的完全符合用户需要的软件，因此，通过测试发现错误之后还必须诊断并改正错误，这就是调试的目的。调试是测试阶段最困难的工作。

在对测试结果进行收集和评价的时候，软件所达到的可靠性也开始明朗了。软件可靠性模型使用故障率数据，估计软件将来出现故障的情况并预测软件的可靠性。

7.1 编　码

7.1.1 选择程序设计语言

程序设计语言是人和计算机通信的最基本的工具，它的特点必然会影响人的思维和解题方式，会影响人和计算机通信的方式和质量，也会影响其他人阅读和理解程序的难易程度。因此，编码之前的一项重要工作就是选择一种适当的程序设计语言。

适宜的程序设计语言能使根据设计去完成编码时困难最少，可以减少需要的程序测试量，并且可以得出更容易阅读和更容易维护的程序。由于软件系统的绝大部分成本用在生命周期的测试和维护阶段，所以容易测试和容易维护是极端重要的。

使用汇编语言编码需要把软件设计翻译成机器操作的序列，由于这两种表示方法很不相同，因此汇编程序设计既困难又容易出差错。一般说来，高级语言的源程序语句和汇编代码指令之间有一句对多句的对应关系。统计资料表明，程序员在相同时间内可以写出的高级语言语句数和汇编语言指令数大体相同，因此用高级语言写程序比用汇编语言写程序生产率可以提高好几倍。高级语言一般都容许用户给程序变量和子程序赋予含义鲜明的名字，通过名字很容易把程序对象和它们所代表的实体联系起来；此外，高级语言使用的符号和概念更符合人的习惯。因此，用高级语言写的程序容易阅读，容易测试，容易调试，容易维护。

总的说来，高级语言明显优于汇编语言，因此，除了在很特殊的应用领域（例如，对程序执行时间和使用的空间都有很严格限制的情况；需要产生任意的甚至非法的指令序列；体系结构特殊的微处理机，以致在这类机器上通常不能实现高级语言编译程序），或者大型系统中执行时间非常关键的（或直接依赖于硬件的）一小部分代码需要用汇编语言书写之外，其他程序应该一律用高级语言书写。

为了使程序容易测试和维护以减少软件的总成本，所选用的高级语言应该有理想的模块化机制，以及可读性好的控制结构和数据结构；为了便于调试和提高软件可靠性，语言特点应该使编译程序能够尽可能多地发现程序中的错误；为了降低软件开发和维护的成本，选用的高级语言应该有良好的独立编译机制。

上述这些要求是选择程序设计语言的理想标准，但是，在实际选择语言时不能仅仅使用理论上的标准，还必须同时考虑实用方面的各种限制。下面是主要的实用标准：

(1) 系统用户的要求。如果所开发的系统由用户负责维护，用户通常要求用他们熟悉的语言书写程序。

(2) 可以使用的编译程序。运行目标系统的环境中可以提供的编译程序往往限制了可以选用的语言的范围。

(3) 可以得到的软件工具。如果某种语言有支持程序开发的软件工具可以利用，则目标系统的实现和验证都变得比较容易。

(4) 工程规模。如果工程规模很庞大，现有的语言又不完全适用，那么设计

并实现一种供这个工程项目专用的程序设计语言，可能是一个正确的选择。

(5)程序员的知识。虽然对于有经验的程序员来说，学习一种新语言并不困难，但是要完全掌握一种新语言却需要实践。如果和其他标准不矛盾，那么应该选择一种已经为程序员所熟悉的语言。

(6)软件可移植性要求。如果目标系统将在几台不同的计算机上运行，或者预期的使用寿命很长，那么选择一种标准化程度高、程序可移植性好的语言就是很重要的。

(7)软件的应用领域。所谓的通用程序设计语言实际上并不是对所有应用领域都同样适用，例如，FORTRAN 语言特别适合于工程和科学计算，COBOI。语言适合于商业领域应用，c 语言和 Ada 语言适用于系统和实时应用领域，LISP 语言适用于组合问题领域，PROLOG 语言适于表达知识和推理。因此，选择语言时应该充分考虑目标系统的应用范围。

7.1.2 编码风格

源程序代码的逻辑简明清晰、易读易懂是好程序的一个重要标准，为了做到这一点，
应该遵循下述规则。

1.程序内部的文档

所谓程序内部的文档包括恰当的标识符、适当的注解和程序的视觉组织等等。

选取含义鲜明的名字，使它能正确地提示程序对象所代表的实体，这对于帮助阅读者理解程序是很重要的。如果使用缩写，那么缩写规则应该一致，并且应该给每个名字加注解。

注解是程序员和程序读者通信的重要手段，正确的注解非常有助于对程序的理解。通常在每个模块开始处有一段序言性的注解，简要描述模块的功能、主要算法、接口特点、重要数据以及开发简史。插在程序中间与一段程序代码有关的注解，主要解释包含这段代码的必要性。对于用高级语言书写的源程序，不需要用注解的形式把每个语句翻译成自然语言，应该利用注解提供一些额外的信息。应该用空格或空行清楚地区分注解和程序。注解的内容一定要正确，错误的注解不仅对理解程序毫无帮助，反而会妨碍对程序的理解。

程序清单的布局对于程序的可读性也有很大影响，应该利用适当的阶梯形式使程序的层次结构清晰明显。

2.数据说明

虽然在设计期间已经确定了数据结构的组织和复杂程度，然而数据说明的风格却是在写程序时确定的。为了使数据更容易理解和维护，有一些比较简单的原则应该遵循。

数据说明的次序应该标准化(例如，按照数据结构或数据类型确定说明的次序)。有次序就容易查阅，因此能够加速测试、调试和维护的过程。

当多个变量名在一个语句中说明时，应该按字母顺序排列这些变量。

如果设计时使用了一个复杂的数据结构，则应该用注解说明用程序设计语言实现这个数据结构的方法和特点。

3.语句构造

设计期间确定了软件的逻辑结构，然而个别语句的构造却是编写程序的一个主要任务。构造语句时应该遵循的原则是，每个语句都应该简单而直接，不能为了提高效率而使程序变得过分复杂。下述规则有助于使语句简单明了：

·不要为了节省空间而把多个语句写在同一行；
·尽量避免复杂的条件测试；
·尽量减少对“非”条件的测试；
·避免大量使用循环嵌套和条件嵌套；
·利用括号使逻辑表达式或算术表达式的运算次序清晰直观。

4.输入输出

在设计和编写程序时应该考虑下述有关输入输出风格的规则：

·对所有输入数据都进行检验；
·检查输入项重要组合的合法性；
·保持输入格式简单；
·使用数据结束标记，不要要求用户指定数据的数目；
·明确提示交互式输入的请求，详细说明可用的选择或边界数值；
·当程序设计语言对格式有严格要求时，应保持输入格式一致；
·设计良好的输出报表；
·给所有输出数据加标志。

5.效率

效率主要指处理机时间和存储器容量两个方面。虽然值得提出提高效率的要求，但是在进一步讨论这个问题之前应该记住3条原则：首先，效率是性能要求，因此应该在需求分析阶段确定效率方面的要求。软件应该像对它要求的那样有效，而不应该如同人类可能做到的那样有效。其次，效率是靠好设计来提高的。第三，程序的效率和程序的简单程度是一致的，不要牺牲程序的清晰性和可读性来不必要地提高效率。下面从三个方面进一步讨论效率问题。

(1)程序运行时间

源程序的效率直接由详细设计阶段确定的算法的效率决定，但是，写程序的风格也能对程序的执行速度和存储器要求产生影响。在把详细设计结果翻译成程序时，总可以应用下述规则：

·写程序之前先简化算术的和逻辑的表达式；
·仔细研究嵌套的循环，以确定是否有语句可以从内层往外移；
·尽量避免使用多维数组；
·尽量避免使用指针和复杂的表；
·使用执行时间短的算术运算；

·不要混合使用不同的数据类型；
·尽量使用整数运算和布尔表达式。
在效率是决定性因素的应用领域，尽量使用有良好优化特性的编译程序，以自动生成高效目标代码。
(2)存储器效率
在大型计算机中必须考虑操作系统页式调度的特点，一般说来，使用能保持功能域的结构化控制结构，是提高效率的好方法。
在微处理机中如果要求使用最少的存储单元，则应选用有紧缩存储器特性的编译程序，在非常必要时可以使用汇编语言。
提高执行效率的技术通常也能提高存储器效率。提高存储器效率的关键同样是“简单”。
(3)输入输出的效率
如果用户为了给计算机提供输入信息或为了理解计算机输出的信息，所需花费的脑力劳动是经济的，那么人和计算机之间通信的效率就高。因此，简单清晰同样是提高人机通信效率的关键。
硬件之间的通信效率是很复杂的问题，但是，从写程序的角度看，却有些简单的原则可以提高输入输出的效率。例如：
·所有输入输出都应该有缓冲，以减少用于通信的额外开销；
·对二级存储器(如磁盘)应选用最简单的访问方法；
·二级存储器的输入输出应该以信息组为单位进行；
·如果“超高效的”输入输出很难被人理解，则不应采用这种方法。
这些简单原则对于软件工程的设计和编码两个阶段都适用。

7.2 软件测试基础

本节讲述软件测试的基本概念和基础知识。
表面看来，软件测试的目的与软件工程所有其他阶段的目的都相反。软件工程的其他阶段都是“建设性”的：软件工程师力图从抽象的概念出发，逐步设计出具体的软件系统，直到用一种适当的程序设计语言写出可以执行的程序代码。但是，在测试阶段测试人员努力设计出一系列测试方案，目的却是为了“破坏”已经建造好的软件系统——竭力证明程序中有错误不能按照预定要求正确工作。
当然，这种反常仅仅是表面的，或者说是心理上的。暴露问题并不是软件测试的最终目的，发现问题是为了解决问题，测试阶段的根本目标是尽可能多地发现并排除软件中潜藏的错误，最终把一个高质量的软件系统交给用户使用。但是，仅就测试本身而言，它的目标可能和许多人原来设想的很不相同。

7.2.1 软件测试的目标

什么是测试?它的目标是什么?G.Myers 给出了关于测试的一些规则，这些规则也可以看作是测试的目标或定义。
(1)测试是为了发现程序中的错误而执行程序的过程；

(2)好的测试方案是极可能发现迄今为止尚未发现的错误的测试方案;

(3)成功的测试是发现了至今为止尚未发现的错误的测试。

从上述规则可以看出，测试的正确定义是“为了发现程序中的错误而执行程序的过程”。这和某些人通常想象的“测试是为了表明程序是正确的”，“成功的测试是没有发现错误的测试”等等是完全相反的。正确认识测试的目标是十分重要的，测试目标决定了测试方案的设计。如果为了表明程序是正确的而进行测试，就会设计一些不易暴露错误的测试方案；相反，如果测试是为了发现程序中的错误，就会力求设计出最能暴露错误的测试方案。

由于测试的目标是暴露程序中的错误，从心理学角度看，由程序的编写者自己进行测试是不恰当的。因此，在综合测试阶段通常由其他人员组成测试小组来完成测试工作。

此外，应该认识到测试决不能证明程序是正确的。即使经过了最严格的测试之后，仍然可能还有没被发现的错误潜藏在程序中。测试只能查找出程序中的错误，不能证明程序中没有错误。关于这个结论下面还要讨论。

7.2.2 软件测试准则

怎样才能达到软件测试的目标呢?为了能设计出有效的测试方案，软件工程师必须深入理解并正确运用指导软件测试的基本准则。下面讲述主要的测试准则。

(1)所有测试都应该能追溯到用户需求。正如上一小节讲过的，软件测试的目标是发现错误。从用户的角度看，最严重的错误是导致程序不能满足用户需求的那些错误。

(2)应该远在测试开始之前就制定出测试计划。实际上，一旦完成了需求模型就可以着手制定测试计划，在建立了设计模型之后就可以立即开始设计详细的测试方案。因此，在编码之前就可以对所有测试工作进行计划和设计。

(3)把 Pareto 原理应用到软件测试中。Pareto 原理说明，测试发现的错误中的 80%很可能是由程序中 20%的模块造成的。当然，问题是怎样找出这些可疑的模块并彻底地测试它们。

(4)应该从“小规模”测试开始，并逐步进行“大规模”测试。通常，首先重点测试单个程序模块，然后把测试重点转向在集成的模块簇中寻找错误，最后在整个系统中寻找
错误。

(5)穷举测试是不可能的。所谓穷举测试就是把程序所有可能的执行路径都检查一遍的测试。即使是一个中等规模的程序，其执行路径的排列数也十分庞大，由于受时间、人力和资源的限制，在测试过程中不可能执行每个可能的路径。因此，测试只能证明程序中有错误，不能证明程序中没有错误。但是，精心地设计测试方案，有可能充分覆盖程序逻辑并使程序达到所要求的可靠性。

(6)为了达到最佳的测试效果，应该由独立的第三方从事测试工作。所谓“最佳激果”是指有最大可能性发现错误的测试。由于前面已经讲过的原因，开发软件的软件工程师并不是完成全部测试工作的最佳人选(通常他们主要承担模块测试工作)。

7.2.3 测试方法

测试任何产品都有两种方法：如果已经知道了产品应该具有的功能，可以通过测试来检验是否每个功能都能正常使用；如果知道产品的内部工作过程，可以通过测试来检验产品内部动作是否按照规格说明书的规定正常进行。前一种方法称为黑盒测试，后一种法称为白盒测试。

对于软件测试而言，黑盒测试法把程序看作一个黑盒子，完全不考虑程序的内部结构和处理过程。也就是说，黑盒测试是在程序接口进行的测试，它只检查程序功能是否能按照规格说明书的规定正常使用，程序是否能适当地接收输入数据并产生正确的输出信息，程序运行过程中能否保持外部信息（例如，数据库或文件）的完整性。黑盒测试又称为功能测试。

白盒测试法与黑盒测试法相反，它的前提是可以把程序看成装在一个透明的白盒子里，测试者完全知道程序的结构和处理算法。这种方法按照程序内部的逻辑测试程序，检测程序中的主要执行通路是否都能按预定要求正确工作。白盒测试又称为结构测试。

7.2.4 测试步骤

除非是测试一个小程序，否则一开始就把整个系统作为一个单独的实体来测试是不现实的。根据第 4 条测试准则，测试过程也必须分步骤进行，后一个步骤在逻辑上是前一个步骤的继续。大型软件系统通常由若干个子系统组成，每个子系统又由许多模块组成，因此，大型软件系统的测试过程基本上由下述几个步骤组成。

1.模块测试

在设计得好的软件系统中，每个模块完成一个清晰定义的子功能，而且这个子功能和同级其他模块的功能之间没有相互依赖关系。因此，有可能把每个模块作为一个单独的体来测试，而且通常比较容易设计检验模块正确性的测试方案。模块测试的目的是保证每个模块作为一个单元能正确运行，所以模块测试通常又称为单元测试。在这个测试步骤中所发现的往往是编码和详细设计的错误。

2.子系统测试

子系统测试是把经过单元测试的模块放在一起形成一个子系统来测试。模块相互间的协调和通信是这个测试过程中的主要问题，因此，这个步骤着重测试模块的接口。

3.系统测试

系统测试是把经过测试的子系统装配成一个完整的系统来测试。在这个过程中不仅应该发现设计和编码的错误，还应该验证系统确实能提供需求说明书中指定的功能，而且系统的动态特性也符合预定要求。在这个测试步骤中发现的往往是软件设计中的错误，也可能发现需求说明中的错误。

不论是子系统测试还是系统测试，都兼有检测和组装两重含义，通常称为集成测试。

4.验收测试

验收测试把软件系统作为单一的实体进行测试，测试内容与系统测试基本类似，但是它是在用户积极参与下进行的，而且可能主要使用实际数据(系统将来要处理的信息)进行测试。验收测试的目的是验证系统确实能够满足用户的需要，在这个测试步骤中发现的往往是系统需求说明书中的错误。验收测试也称为确认测试。

5.平行运行

关系重大的软件产品在验收之后往往并不立即投入生产性运行，而是要再经过一段平行运行时间的考验。所谓平行运行就是同时运行新开发出来的系统和将被它取代的旧系统，以便比较新旧两个系统的处理结果。这样做的具体目的有如下几点：

(1)可以在准生产环境中运行新系统而又不冒风险；

(2)用户能有一段熟悉新系统的时间；

(3)可以验证用户指南和使用手册之类的文档；

(4)能够以准生产模式对新系统进行全负荷测试，可以用测试结果验证性能指标。

以上集中讨论了与测试有关的概念，但是，测试作为软件工程的一个阶段，它的根本任务是保证软件的质量，因此除了进行测试之外，还有另外一些与测试密切相关的工作应该完成。这就是下一小节要讨论的内容。

7.2.5 测试阶段的信息流

图 7.1 描绘了测试阶段的信息流，这个阶段的输入信息有两类：(1)软件配置，包括需求说明书、设计说明书和源程序清单等；(2)测试配置，包括测试计划和测试方案。所谓测试方案不仅仅是测试时使用的输入数据(称为测试用例)，还应该包括每组输入数据预定要检验的功能，以及每组输入数据预期应该得到的正确输出。实际上测试配置是软件配置的一个子集，最终交出的软件配置应该包括上述测试配置以及测试的实际结果和
调试的记录。

比较测试得出的实际结果和预期的结果，如果两者不一致则很可能是程序中有错误。设法确定错误的准确位置并且改正它，这就是调试的任务。与测试不同，通常由程序的编写者负责调试。

在对测试结果进行收集和评价的时候，软件可靠性所达到的定性指标也开始明朗了。如果经常出现要求修改设计的严重错误，那么软件的质量和可靠性是值得怀疑的，应该进

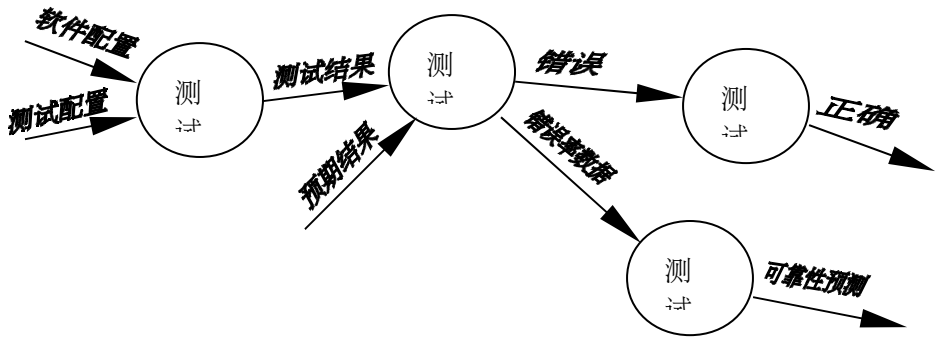

图 7.1　测试阶段的信息流

一步仔细测试。反之，如果看起来软件功能完成得很正常，遇到的错误也很容易改正，则仍然应该考虑两种可能：(1)软件的可靠性是可以接受的；(2)所进行的测试尚不足以发现严重的错误。最后，如果经过测试，一个错误也没有被发现，则很可能是因为对测试配置思考不充分，以致不能暴露软件中潜藏的错误。这些错误最终将被用户发现，而且需要在维护阶段改正它们(但是改正同一个错误需要付出的代价比在开发阶段高出许多倍)。

在测试阶段积累的结果，也可以用更形式化的方法进行评价。软件可靠性模型使用错误率数据估计将来出现错误的情况，并进而对软件可靠性进行预测。

7.3 单元测试

单元测试集中检测软件设计的最小单元——模块。通常，单元测试和编码属于软件过程的同一个阶段。在编写出源程序代码并通过了编译程序的语法检查之后，就可以用详细设计描述作指南，对重要的执行通路进行测试，以便发现模块内部的错误。可以应用人工测试和计算机测试这样两种不同类型的测试方法，完成单元测试工作。这两种测试方法各有所长，互相补充。通常，单元测试主要使用白盒测试技术，而且对多个模块的测试可以并行地进行。

7.3.1 测试重点

在单元测试期间着重从下述5个方面对模块进行测试。

1.模块接口

首先应该对通过模块接口的数据流进行测试，如果数据不能正确地进出，所有其他测试都是不切实际的。

在对模块接口进行测试时主要检查下述几个方面：参数的数目、次序、属性或单位系统与变元是否一致；是否修改了只作输入用的变元；全局变量的定义和用法在各个模块中是否一致。

2.局部数据结构

对于模块来说，局部数据结构是常见的错误来源。应该仔细设计测试方案，以便发现局部数据说明、初始化、默认值等方面的错误。

3.重要的执行通路

由于通常不可能进行穷尽测试，因此，在单元测试期间选择最有代表性、最可能发现错误的执行通路进行测试就是十分关键的。应该设计测试方案用来发现由于错误的计算、不正确的比较或不适当的控制流而造成的错误。

4.出错处理通路

好的设计应该能预见出现错误的条件，并且设置适当的处理错误的通路，以便在真的出现错误时执行相应的出错处理通路或干净地结束处理。不仅应该在程序中包含出错处理通路而且应该认真测试这种通路。当评价出错处理通路时，应该着重测试下述一些可能发生的错误：

(1)对错误的描述是难以理解的；
(2)记下的错误与实际遇到的错误不同；
(3)在对错误进行处理之前，错误条件已经引起系统干预；
(4)对错误的处理不正确；
(5)描述错误的信息不足以帮助确定造成错误的位置。

5.边界条件

边界测试是单元测试中最后的也可能是最重要的任务。软件常常在它的边界上失效，例如，处理 n 元数组的第 n 个元素时，或做到 i 次循环中的第 i 次重复时，往往会发生错误。使用刚好小于、刚好等于和刚好大于最大值或最小值的数据结构、控制量和数据值的测试方案，非常可能发现软件中的错误。

7.3.2 代码审查

人工测试源程序可以由编写者本人非正式地进行，也可以由审查小组正式进行。后者称为代码审查，它是一种非常有效的程序验证技术，对于典型的程序来说，可以查出 30 9，6～70%的逻辑设计错误和编码错误。审查小组最好由下述 4 人组成：

(1)组长，应该是一个很有能力的程序员，而且没有直接参与这项工程；
(2)程序的设计者；
(3)程序的编写者；
(4)程序的测试者。

如果一个人既是程序的设计者又是编写者，或既是编写者又是测试者，则审查小组中应该再增加一个程序员。

审查之前，小组成员应该先研究设计说明书，力求理解这个设计。为了帮助理解，可以先由设计者扼要地介绍他的设计。在审查会上由程序的编写者解释他是怎样用程序代码实现这个设计的，通常是逐个语句地讲述程序的逻辑，小组其他成员仔细倾听他的讲解，并力图发现其中的错误。审查会上进行的另外一项工作，是对照类似于上一小节中介绍的程序设计常见错误清单，分析审查这个

程序。当发现错误时由组长记录下来，审查会继续进行(审查小组的任务是发现错误而不是改正错误)。

审查会还有另外一种常见的进行方法，称为预排：由一个人扮演“测试者”，其他人扮演“计算机”。会前测试者准备好测试方案，会上由扮演计算机的成员模拟计算机执行被测试的程序。当然，由于人执行程序速度极慢，因此测试数据必须简单，测试方案的数目也不能过多。但是，测试方案本身并不十分关键，它只起一种促进思考引起讨论的作用。

在大多数情况下，通过向程序员提出关于他的程序的逻辑和他编写程序时所做的假设的疑问，可以发现的错误比由测试方案直接发现的错误还多。

代码审查比计算机测试优越的是：一次审查会上可以发现许多错误；用计算机测试的方法发现错误之后，通常需要先改正这个错误才能继续测试，因此错误是一个一个地发现并改正的。也就是说，采用代码审查的方法可以减少系统验证的总工作量。

实践表明，对于查找某些类型的错误来说，人工测试比计算机测试更有效；对于其他类型的错误来说则刚好相反。因此，人工测试和计算机测试是互相补充，相辅相成的，缺少其中任何一种方法都会使查找错误的效率降低。

7.3.3 计算机测试

模块并不是一个独立的程序，因此必须为每个单元测试开发驱动软件和(或)存根软件。通常驱动程序也就是一个“主程序”，它接收测试数据，把这些数据传送给被测试的模块，并且印出有关的结果。存根程序代替被测试的模块所调用的模块。因此存根程序也可以称为“虚拟子程序”。它使用被它代替的模块的接口，可能做最少量的数据操作，印出对入口的检验或操作结果，并且把控制归还给调用它的模块。

例如，图 7.2 是一个正文加工系统的部分层次图，假定要测试其中编号为 3.0 的关键模块——正文编辑模块。因为正文编辑模块不是一个独立的程序，所以需要有一个测试驱动程序来调用它。这个驱动程序说明必要的变量，接收测试数据——字符串，并且设置正文编辑模块的编辑功能。因为在原来的软件结构中，正文编辑模块通过调用它的下层

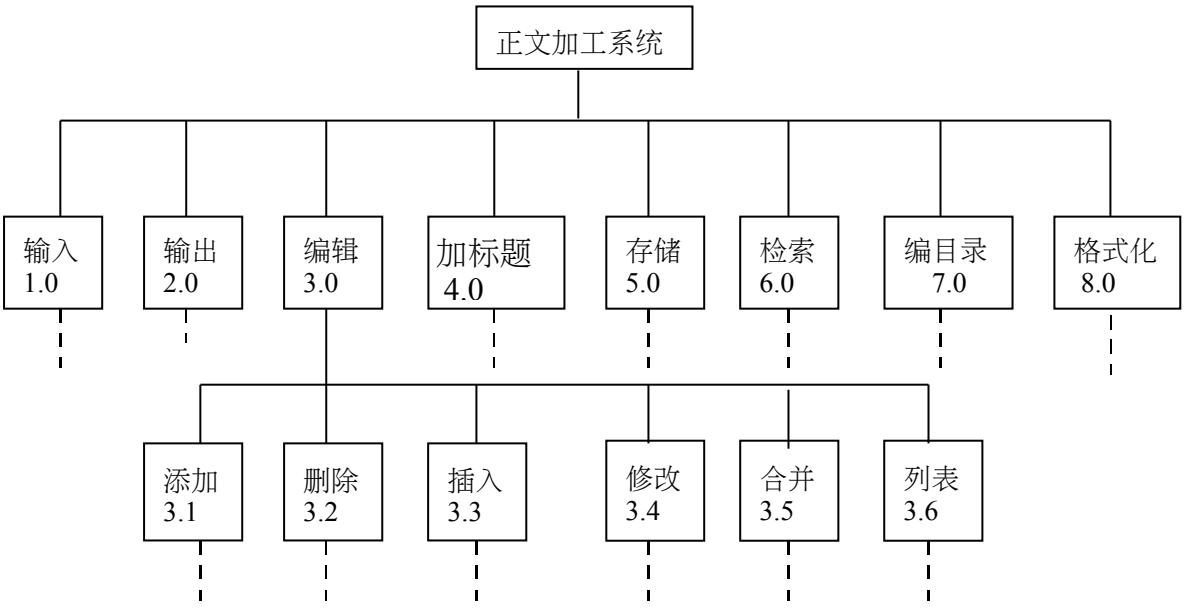

图 7.2　正文加工系统的层次图

模块来完成具体的编辑功能，所以需要有存根程序简化地模拟这些下层模块。为了简单起见，测试时可以设置的编辑功能只有修改(CHANGE)和添加(APPEND)两种，用控制变量 CFUNCT 标记要求的编辑功能，而且只用一个存根程序模拟正文编辑模块的所有
下层模块。下面是用伪码书写的存根程序和驱动程序。

```
I.TEST STUB(*测试正文编辑模块用的存根程序*)
        初始化;
        输出信息“进入了正文编辑程序”;
        输出“输入的控制信息是”CFUNCT;
        输出缓冲区中的字符串;
        IF  CFUNCT=CHANGE
            THEN
            把缓冲区中第二个字改为 * * *
            ELSE
            在缓冲区的尾部加???
         END IF;
        输出缓冲区中的新字符串;
    END TEST STUB
Ⅱ.TEST DRIVER(*测试正文编辑模块用的驱动程序*)
        说明长度为 2 500 个字符的一个缓冲区;
        把 CFUNCT 置为希望测试的状态;
        输入字符串;
        调用正文编辑模块;
        停止或再次初启;
   END TEST DRIVER
```

驱动程序和存根程序代表开销，也就是说，为了进行单元测试必须编写测试软件，但是通常并不把它们作为软件产品的一部分交给用户。许多模块不能用简单的测试软件充分测试，为了减少开销可以使用下节将要介绍的渐增式测试方法，在集成测试的过程中同时完成对模块的详尽测试。

模块的内聚程度高可以简化单元测试过程。如果每个模块只完成一种功能，则需要的测试方案数目将明显减少，模块中的错误也更容易预测和发现。

7.4 集成测试

集成测试是测试和组装软件的系统化技术，例如，子系统测试即是在把模块按照设计要求组装起来的同时进行测试，主要目标是发现与接口有关的问题（系统测试与此类似）。例如，数据穿过接口时可能丢失；一个模块对另一个模块可能由于疏忽而造成有害影响；把子功能组合起来可能不产生预期的主功能；个别看来是可以接受的误差可能积累到不能接受的程度；全程数据结构可能有问题等等。不幸的是，可能发生的接口问题多得不胜枚举。

由模块组装成程序时有两种方法。一种方法是先分别测试每个模块，再把所有模块按设计要求放在一起结合成所要的程序，这种方法称为非渐增式测试方法；另一种方法是把下一个要测试的模块同已经测试好的那些模块结合起来进行测试，测试完以后再把下一个应该测试的模块结合进来测试。这种每次增加一个模块的方法称为渐增式测试，这种方法实际上同时完成单元测试和集成测试。这两种方法哪种更好一些呢?下面对比它们的主要优缺点：

非渐增式测试一下子把所有模块放在一起，并把庞大的程序作为一个整体来测试，测试者面对的情况十分复杂。测试时会遇到许许多多的错误，改正错误更是极端困难，因为在庞大的程序中想要诊断定位一个错误是非常困难的。而且一旦改正一个错误之后，马上又会遇到新的错误，这个过程将继续下去，看起来好像永远也没有尽头。

渐增式测试与“一步到位”的非渐增式测试相反，它把程序划分成小段来构造和测试，在这个过程中比较容易定位和改正错误；对接口可以进行更彻底的测试；可以使用系统化的测试方法。因此，目前在进行集成测试时普遍采用渐增式测试方法。

当使用渐增方式把模块结合到程序中去时，有自顶向下和自底向上两种集成策略。

7.4.1 自顶向下集成

自顶向下集成方法是一个日益为人们广泛采用的测试和组装软件的途径。从主控制模块开始，沿着程序的控制层次向下移动，逐渐把各个模块结合起来。在把附属于（及最终附属于）主控制模块的那些模块组装到程序结构中去时，或者使用深度优先的策略，或者使用宽度优先的策略。

参看图 7.3，深度优先的结合方法先组装在软件结构的一条主控制通路上的所有模块。选择一条主控制通路取决于应用的特点，并且有很大任意性。例如，选取左通路，首先结合模块 M_1、M_2 和 M_5；其次，M_8 或 M_6（如果为了使 M 具有适当功能需要 M_6）将被结合进来。然后构造中央的和右侧的控制通路。而宽度优先的结合方法是沿软件结构水平地移动，把处于同一个控制层次上的所有模块组装起来。对于图 7.3 来说，首先结合模块 M_2、M_3 和 M_4（代替存根程序 S_4），然后结合下一个控制层次中的模块 M_5、M_6 和 M_7；如此继续进行下去，直到所有模块都被结合进来为止。

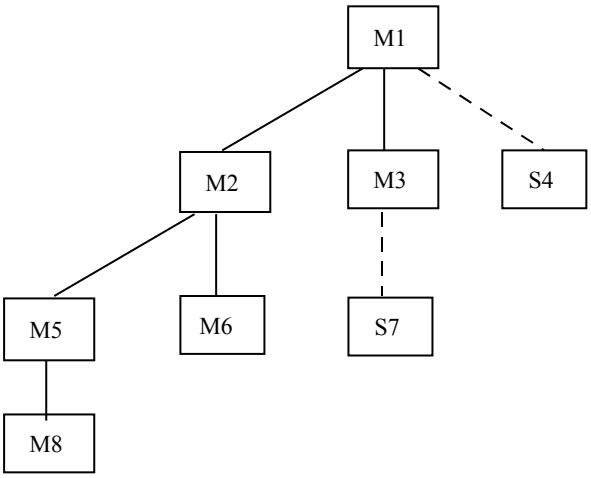

图 7.3　自顶向下结合

把模块结合进软件结构的具体过程由下述 4 个步骤完成:

第一步，对主控制模块进行测试，测试时用存根程序代替所有直接附属于主控制模块的模块;

第二步，根据选定的结合策略(深度优先或宽度优先)，每次用一个实际模块代换一个存根程序(新结合进来的模块往往又需要新的存根程序);

第三步，在结合进一个模块的同时进行测试;

第四步，为了保证加入模块没有引进新的错误，可能需要进行回归测试(即，全部或部分地重复以前做过的测试)。

从第二步开始不断地重复进行上述过程，直到构造起完整的软件结构为止。图 7.3 描绘了这个过程。假设选取深度优先的结合策略，软件结构已经部分地构造起来了，下一步存根程序 S_7，将被模块 M_7，取代。M_7 可能本身又需要存根程序，以后这些存根程序也将被相应的模块所取代。

自顶向下的结合策略能够在测试的早期对主要的控制或关键的抉择进行检验。在一个分解得好的软件结构中，关键的抉择位于层次系统的较上层，因此首先碰到。如果主要控制确实有问题，早期认识到这类问题是很有好处的，可以及早想办法解决。如果选择深度优先的结合方法，可以在早期实现软件的一个完整的功能并且验证这个功能。早期证实软件的一个完整的功能，可以增强开发人员和用户双方的信心。

自顶向下的方法讲起来比较简单，但是实际使用时可能遇到逻辑上的问题。这类问题中最常见的是，为了充分地测试软件系统的较高层次，需要在较低层次上的处理。然而在自顶向下测试的初期，存根程序代替了低层次的模块，因此，在软件结构中没有重要的数据自下往上流。为了解决这个问题，测试人员有两种选择：第一，把许多测试推迟到用真实模块代替了存根程序以后再进行；第二，从层次系统的底部向上组装软件。第一种方法失去了在特定的测试和组装特定的模块之间的精确对应关系，这可能导致在确定错误的位置和原因时发生困难。后一种方法称为自底向上的测试，下面讨论这种方法。

7.4.2 自底向上集成

自底向上测试从“原子”模块(即在软件结构最低层的模块)开始组装和测试。因为是从底部向上结合模块，总能得到所需的下层模块处理功能，所以不需要存根程序。

用下述步骤可以实现自底向上的结合策略：

第一步，把低层模块组合成实现某个特定的软件子功能的族；

第二步，写一个驱动程序(用于测试的控制程序)，协调测试数据的输入和输出；

第三步，对由模块组成的子功能族进行测试；

第四步，去掉驱动程序，沿软件结构自下向上移动，把子功能族组合起来形成更大的子功能族。

上述第二步到第四步实质上构成了一个循环。图 7.4 描绘了自底向上的结合过程。首先把模块组合成族 1、族 2 和族 3，使用驱动程序(图中用虚线方框表示)对每个子功能族进行测试。族 1 和族 2 中的模块附属于模块 M_a，去掉驱动程序 D_1 和 D_2，把这两个族直接同 M_a 连接起来。类似地，在和模块 M_b 结合之前去掉族 3 的驱动程序 D_3。最终 M_8 和 M_b 这两个模块都与模块 M_c 结合起来。

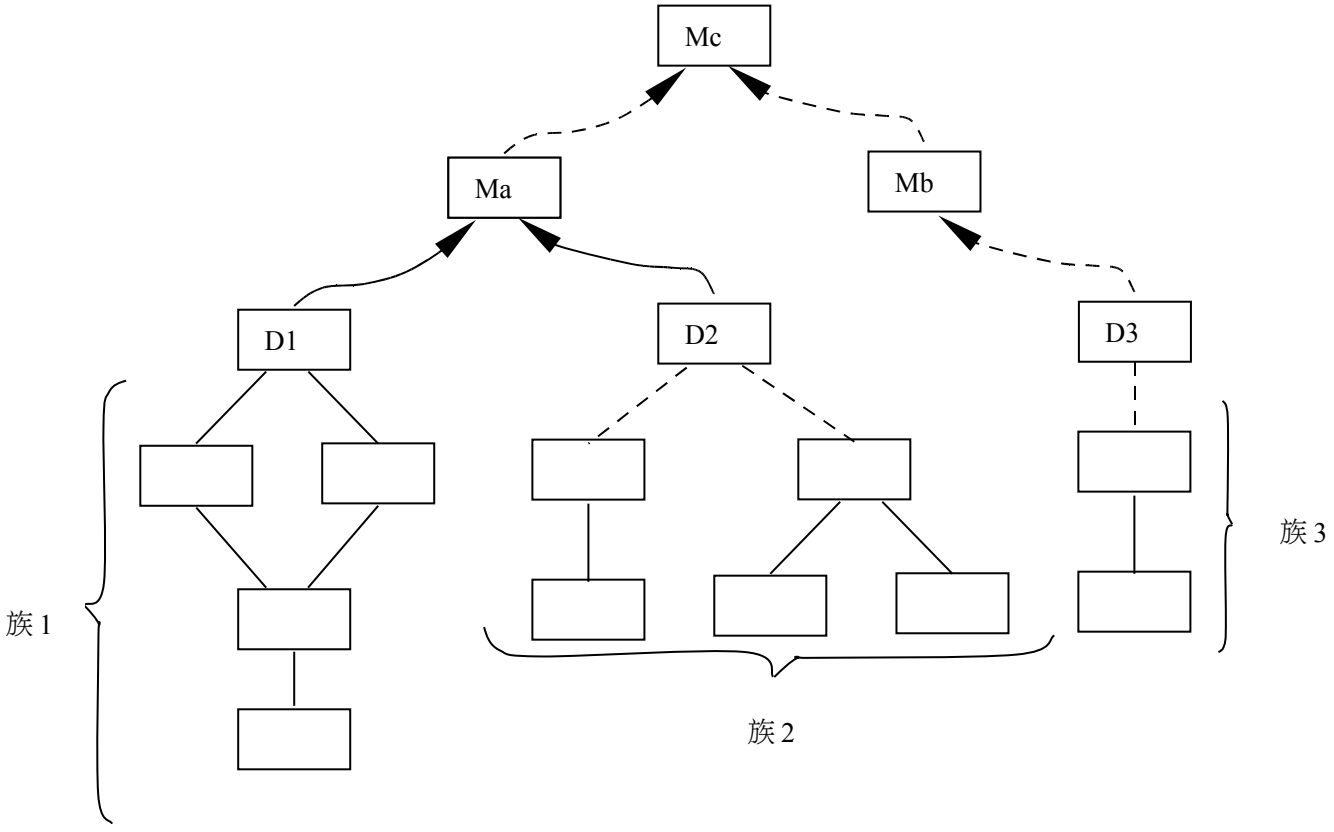

图 7.4　自底向上结

随着结合向上移动，对测试驱动程序的需要也减少了。事实上，如果软件结构的顶部两层用自顶向下的方法组装，可以明显减少驱动程序的数目，而且族的结合也将大大简化。

7.4.3 不同集成测试策略的比较

上面介绍了集成测试的两种策略，到底哪种方法更好一些呢?一般说来，一种方法的优点正好对应于另一种方法的缺点。自顶向下测试方法的主要优点是不需要测试驱动程序，能够在测试阶段的早期实现并验证系统的主要功能，而且能在早期发现上层模块的接口错误。自顶向下测试方法的主要缺点是需要存根程序，可能遇到与此相联系的测试困难，低层关键模块中的错误发现较晚，而且用这种方法在早期不能充分展开人力。可以看出，自底向上测试方法的优缺点与上述自顶向下测试方法的优缺点刚好相反。

在测试实际的软件系统时，应该根据软件的特点以及工程进度安排，选用适当的测试策略。一般说来，纯粹自顶向下或纯粹自底向上的策略可能都不实用，人们在实践中创造出许多混合策略：

(1)改进的自顶向下测试方法。基本上使用自顶向下的测试方法，但是在早期使用自底向上的方法测试软件中的少数关键模块。一般的自顶向下方法所具有的优点在这种方法中也都有，而且能在测试的早期发现关键模块中的错误；但是，它的缺点也比自顶向下方法多一条，即测试关键模块时需要驱动程序。

(2)混合法。对软件结构中较上层使用的自顶向下方法与对软件结构中较下层使用的自底向上方法相结合。这种方法兼有两种方法的优点和缺点，当被测试的软件中关键模块比较多时，这种混合法可能是最好的折衷方法。

7.4.4 回归测试

在集成测试过程中每当一个新模块结合进来时，程序就发生了变化：建立了新的数据流路径，可能出现了新的 I／O 操作，激活了新的控制逻辑。这些变化有可能使原来工作正常的功能出现问题。在集成测试的范畴中，所谓回归测试是指重新执行已经做过的测试的某个子集，以保证上述这些变化没有带来非预期的副作用。

更广义地说，任何成功的测试都会发现错误，而且错误必须被改正。每当改正软件错误的时候，软件配置的某些成分(程序、文档或数据)也被修改了。回归测试就是用于保证由于调试或其他原因引起的变化，不会导致非预期的软件行为或额外错误的测试活动。

回归测试可以通过重新执行全部测试用例的一个子集人工地进行，也可以使用自动化的捕获回放工具自动进行。利用捕获回放工具，软件工程师能够捕获测试用例和实际运行结果，然后可以回放(即重新执行测试用例)，并且比较软件变化前后所得到的运行结果。

回归测试集(已执行过的测试用例的子集)包括下述 3 类不同的测试用例：

(1)检测软件全部功能的代表性测试用例；

(2)专门针对可能受修改影响的软件功能的附加测试;

(3)针对被修改过的软件成分的测试。

在集成测试过程中，回归测试用例的数量可能变得非常大。因此，应该把回归测试集设计成只包括可以检测程序每个主要功能中的一类或多类错误的那样一些测试用例。一旦修改了软件之后就重新执行检测程序每个功能的全部测试用例，是低效而且不切实际的。

7.5 确认测试

确认测试也称为验收测试，它的目标是验证软件的有效性。

上面这句话中使用了确认(validation)和验证(verification)这样两个不同的术语，为了避免混淆，首先扼要地解释一下这两个术语的含义。通常，验证指的是保证软件正确地实现了某个特定要求的一系列活动，而确认指的是为了保证软件确实满足了用需求而进行的一系列活动。

那么，什么样的软件才是有效的呢?软件有效性的一个简单定义是：如果软件的功能和性能如同用户所合理期待的那样，软件就是有效的。

需求分析阶段产生的软件需求规格说明书，准确地描述了用户对软件的合理期望，因此是软件有效性的标准，也是进行确认测试的基础。

7.5.1 确认测试的范围

确认测试必须有用户积极参与，或者以用户为主进行。用户应该参与设计测试方案，使用用户界面输入测试数据并且分析评价测试的输出结果。为了使得用户能够积极主动地参与确认测试，特别是为了使用户能有效地使用这个系统，通常在验收之前由开发单位对用户进行培训。

确认测试通常使用黑盒测试法。应该仔细设计测试计划和测试过程，测试计划包括要进行的测试的种类及进度安排，测试过程规定了用来检测软件是否与需求一致的测试方案。通过测试和调试要保证软件能满足所有功能要求，能达到每个性能要求，文档资料是准确而完整的，此外，还应该保证软件能满足其他预定的要求(例如，安全性、可移植性、兼容性和可维护性等)。

确认测试有下述两种可能的结果：

(1)功能和性能与用户要求一致，软件是可以接受的;

(2)功能和性能与用户要求有差距。

在这个阶段发现的问题往往和需求分析阶段的差错有关，涉及的面通常比较广，因此解决起来也比较困难。为了制定解决确认测试过程中发现的软件缺陷或错误的策略，通常需要和用户充分协商。

7.5.2 软件配置复查

确认测试的一个重要内容是复查软件配置。复查的目的是保证软件配置的所有成分都齐全，质量符合要求，文档与程序完全一致，具有完成软件维护所必

须的细节，而且已经编好目录。

除了按合同规定的内容和要求，由人工审查软件配置之外，在确认测试过程中还应该严格遵循用户指南及其他操作程序，以便检验这些使用手册的完整性和正确性。必须仔细记录发现的遗漏或错误，并且适当地补充和改正。

7.5.3 AIpha 和 Beta 测试

如果软件是专为某个客户开发的，可以进行一系列验收测试，以便用户确认所有需求都得到满足了。验收测试是由最终用户而不是系统的开发者进行的。事实上，验收测试可以持续几个星期甚至几个月，因此能够发现随着时间流逝可能会降低系统质量的累积错误。

如果一个软件是为许多客户开发的(例如，向大众公开出售的盒装软件产品)，那么，让每个客户都进行正式的验收测试是不现实的。在这种情况下，绝大多数软件开发商都使用被称为 A1pha 测试和 Beta 测试的过程，来发现那些看起来只有最终用户才能发现的错误。

A1pha 测试由用户在开发者的场所进行，并且在开发者对用户的“指导”下进行测试。开发者负责记录发现的错误和使用中遇到的问题。总之，A1pha 测试是在受控的环境中进行的。

Beta 测试由软件的最终用户们在一个或多个客户场所进行。与 A1pha 测试不同，开发者通常不在 Beta 测试的现场，因此，Beta 测试是软件在开发者不能控制的环境中的“真实”应用。用户记录在 Beta 测试过程中遇到的一切问题(真实的或想像的)，并且定期把这些问题报告给开发者。接收到在 Beta 测试期间报告的问题之后，开发者对软件产品进行必要的修改，并准备向全体客户发布最终的软件产品。

7.6 白盒测试技术

设计测试方案是测试阶段的关键技术问题。所谓测试方案包括具体的测试目的(例如，预定要测试的具体功能)，应该输入的测试数据和预期的结果。通常又把测试数据和预期的输出结果称为测试用例。其中最困难的问题是设计测试用的输入数据。

不同的测试数据发现程序错误的能力差别很大，为了提高测试效率降低测试成本，应该选用高效的测试数据。因为不可能进行穷尽的测试，选用少量“最有效的”测试数据，做到尽可能完备的测试就更重要了。

设计测试方案的基本目标是，确定一组最可能发现某个错误或某类错误的测试数据。已经研究出许多设计测试数据的技术，这些技术各有优缺点，没有哪一种是最好的，更没有哪一种可以代替其余所有技术；同一种技术在不同的应用场合效果可能相差很大，因此，通常需要联合使用多种设计测试数据的技术。

本节讲述在用白盒方法测试软件时设计测试数据的典型技术，下一节讲述在用黑盒方法测试软件时设计测试数据的典型技术。

7.6.1 逻辑覆盖

有选择地执行程序中某些最有代表性的通路是对穷尽测试的惟一可行的替代办法。所谓逻辑覆盖是对一系列测试过程的总称，这组测试过程逐渐进行越来越完整的通路测试。测试数据执行(或叫覆盖)程序逻辑的程度可以划分成哪些不同的等级呢?从覆盖源程序语句的详尽程度分析，大致有以下一些不同的覆盖标准。

1.语句覆盖

为了暴露程序中的错误，至少每个语句应该执行一次。语句覆盖的含义是，选择足够多的测试数据，使被测程序中每个语句至少执行一次。例如，图 7.5 所示的程序流程图描绘了一个被测模块的处理算法。

为了使每个语句都执行一次，程序的执行路径应该是 sacbed，为此只需要输入下面的测试数据(实际上 X 可以是任意实数)：

$A=2, B=0, X=4$

语句覆盖对程序的逻辑覆盖很少，在上面例子中两个判定条件都只测试了条
件为真的情况，如果条件为假时处理有错误，显然不能发现。此外，语句覆盖只关心判定表达式的值，而没有分别测试判定表达式中每个条件取不同值时的情况。在上面的例子中，为了执行 sacbed 路径，以测试每个语句，只需两个判定表达式(A>1)AND(B=0)和(A=2)OR(X>1)都取真值，因此使用上述一组测试数据就够了。但是，如果程序中把第一个判定表达式中的逻辑运算符"AND"错写成“OR”，或把第二个判定表达式中的条件“X>1”误写成“X<1”，使用上面的测
试数据并不能查出这些错误。

综上所述，可以看出语句覆盖是很弱的逻辑覆盖标准，为了更充分地测试程序，可采用下述的逻辑覆盖标准。

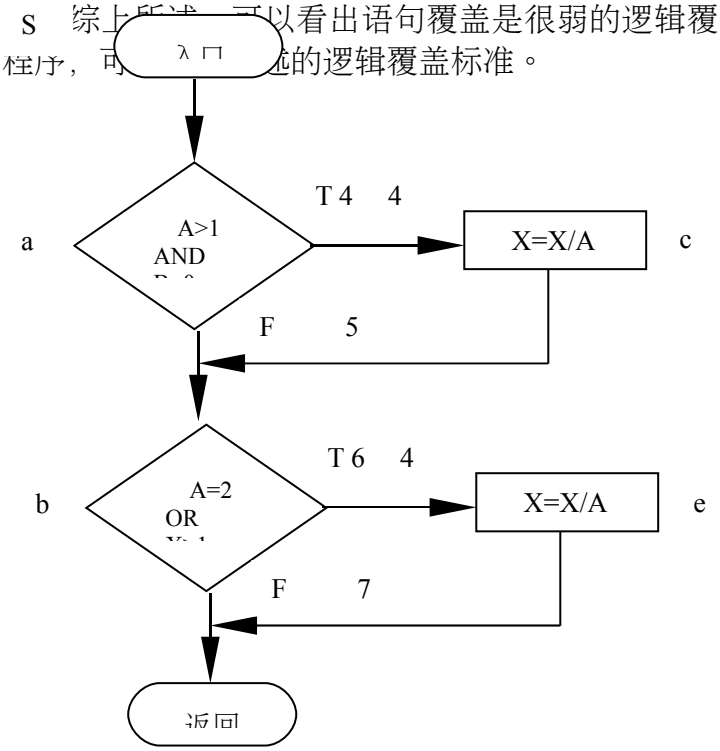

图 7.5 被测试模块的流程图

2.判定覆盖

判定覆盖又叫分支覆盖，它的含义是，不仅每个语句必须至少执行一次，而且每个判定的每种可能的结果都应该至少执行一次，也就是每个判定的每个分支都至少执行一次。

对于上述例子来说，能够分别覆盖路径 sacbed 和 sabd 的两组测试数据，或者可以分别覆盖路径 sacbd 和 sabed 的两组测试数据，都满足判定覆盖标准。例如，用下面两组测试数据就可做到判定覆盖：

Ⅰ．A=3，B=0，X=3 （覆盖 sacbd）

Ⅱ.A=2，B=1，X=1 （覆盖 sabed）

判定覆盖比语句覆盖强，但是对程序逻辑的覆盖程度仍然不高，例如，上面的测试数据只覆盖了程序全部路径的一半。

3.条件覆盖

条件覆盖的含义是，不仅每个语句至少执行一次，而且使判定表达式中的每个条件都取到各种可能的结果。

图 7.5 的例子中共有两个判定表达式，每个表达式中有两个条件，为了做到条件覆盖，应该选取测试数据使得在 a 点有下述各种结果出现：

A>1，A≤1，B=0，B≠0

在 b 点有下述各种结果出现：

A=2，A≠2，X>1，X≤1

只需要使用下面两组测试数据就可以达到上述覆盖标准：

Ⅰ.A=2，B=0，X=4

（满足 A>1，B=0，A=2 和 X>1 的条件，执行路径 sacbed）

Ⅱ.A=1，B=1，X=1

（满足 A≤1，B≠0，A≠2 和 X≤1 的条件，执行路径 sabd）

条件覆盖通常比判定覆盖强，因为它使判定表达式中每个条件都取到了两个不同的结果，判定覆盖却只关心整个判定表达式的值。例如，上面两组测试数据也同时满足判定覆盖标准。但是，也可能有相反的情况：虽然每个条件都取到了两个不同的结果，判定表达式却始终只取一个值。例如，如果使用下面两组测

试数据，则只满足条件覆盖标准并不满足判定覆盖标准(第二个判定表达式的值总为真)：

Ⅰ.A=2，B=0，X=1

(满足 A>1，B一0，A一2和X≤1的条件，执行路径 sacbed)

Ⅱ.A一1，B一1，X一2

(满足 A≤1，B≠0，A≠2和X>I 的条件，执行路径 sabed)

4.判定／条件覆盖

既然判定覆盖不一定包含条件覆盖，条件覆盖也不一定包含判定覆盖，自然会提出一种能同时满足这两种覆盖标准的逻辑覆盖，这就是判定／条件覆盖。它的含义是，选取足够多的测试数据，使得判定表达式中的每个条件都取到各种可能的值，而且每个判定表达式也都取到各种可能的结果。

对于图7.5的例子而言，下述两组测试数据满足判定／条件覆盖标准：

Ⅰ.A=2，B=0，X=4

Ⅱ.A=1，B=1，X=1

但是，这两组测试数据也就是为了满足条件覆盖标准最初选取的两组数据，因此，有时判定／条件覆盖也并不比条件覆盖更强。

5.条件组合覆盖

条件组合覆盖是更强的逻辑覆盖标准，它要求选取足够多的测试数据，使得每个判定表达式中条件的各种可能组合都至少出现一次。

对于图7.5的例子，共有8种可能的条件组合，它们是：

(1)A>1，B—0

(2)A>I，B≠0

(3)A≤1，B=0

(4)A≤1，B≠0

(5)A=2，X>1

(6)A=2，X≤1

(7)A≠2，X>1

(8)A≠2，X≤1

和其他逻辑覆盖标准中的测试数据一样，条件组合(5)～(8)中的X值是指在程序流程图第二个判定框(b点)的X值。

下面的4组测试数据可以使上面列出的8种条件组合每种至少出现一次：

Ⅰ.A=2，B=0，X=4

(针对1，5两种组合，执行路径 sacbed)

Ⅱ.A=2，B=1，X=1

(针对2，6两种组合，执行路径 sabed)

Ⅲ.A=1，B=0，X=2

(针对3，7两种组合，执行路径 sabed)

Ⅳ.A=1，B=1，X=1

(针对4，8两种组合，执行路径 sabd)

显然，满足条件组合覆盖标准的测试数据，也一定满足判定覆盖、条件覆盖和判定／条件覆盖标准。因此，条件组合覆盖是前述几种覆盖标准中最强的。但是，满足条件组合覆盖标准的测试数据并不一定能使程序中的每条路径都执行到，例如，上述4组测试数据都没有测试到路径sacbd。

以上根据测试数据对源程序语句检测的详尽程度，简单讨论了几种逻辑覆盖标准。在上面的分析过程中常常谈到测试数据执行的程序路径，显然，测试数据可以检测的程序路径的多少，也反映了对程序测试的详尽程度。从对程序路径的覆盖程度分析，能够提出下述一些主要的逻辑覆盖标准。

6.点覆盖

图论中点覆盖的概念定义如下：如果连通图G的子图G’是连通的，而且包含G的所有结点，则称G’是G的点覆盖。

在第6.5节中已经讲述了从程序流程图导出流图的方法。在正常情况下流图是连通的有向图。满足点覆盖标准要求选取足够多的测试数据，使得程序执行路径至少经过流图的每个结点一次，由于流图的每个结点与一条或多条语句相对应，显然，点覆盖标准和语句覆盖标准是相同的。

7.边覆盖

图论中边覆盖的定义是：如果连通图G的子图G”是连通的，而且包含G的所有边，则称G”是G的边覆盖。为了满足边覆盖的测试标准，要求选取足够多测试数据，使得程序执行路径至少经过流图中每条边一次。通常边覆盖和判定覆盖是一致的。

8.路径覆盖

路径覆盖的含义是，选取足够多测试数据，使程序的每条可能路径都至少执行一次(如果程序图中有环，则要求每个环至少经过一次)。

作为一个练习，请读者自己设计用路径覆盖标准测试图7.5所示模块的测试数据。

7.6.2 控制结构测试

现有的很多种白盒测试技术，是根据程序的控制结构设计测试数据的技术，下面介绍
几种常用的控制结构测试技术。

1.基本路径测试

基本路径测试是Tom McCabe提出的一种白盒测试技术。使用这种技术设计测试用例时，首先计算程序的环形复杂度，并用该复杂度为指南定义执行路径的

基本集合，从该基本集合导出的测试用例可以保证程序中的每条语句至少执行一次，而且每个条件在执行时都将分别取真、假两种值。

使用基本路径测试技术设计测试用例的步骤如下：

第一步，根据过程设计结果画出相应的流图。

例如，为了用基本路径测试技术测试下列的用 PDL 描述的求平均值过程，首先画出图 7.6 所示的流图。注意，为了正确地画出流图，我们把被映射为流图结点的 PDL 语句编了序号。

```
    PROCEDURE average;
    ／*这个过程计算不超过 100 个在规定值域内的有效数字的平均值;
    同时计算有效数字的总和及个数。  *／
    INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value.minimum, maximum;
    TYPE value[1…100]IS SCALAR ARRAY;
    TYPE average, total.input, total.valid;
         minimum,maximum.sum IS SCALAR;
    TYPE i IS INTEGER;
1:    i=1
      total.input=total.valid=0;
      sum=0;
2:    DO WHILE value[i]<> -999
3:      AND total.input<100
4:    increment total.input by1;
5:    IF value[i]>=minimum
6:    AND value[i]<=maximum
7:      THEN increment total.valid by 1;
      sum=sum+value[i];
8:    ENDIF
      increment i by 1;
9:    ENDDO
10:   IF total.valid>0
11:     THEN average=sum／total.valid;
12:     ELSE average=-999;
13:   ENDIF
    END average
```

步，计算图 7.6 的环形复杂度。

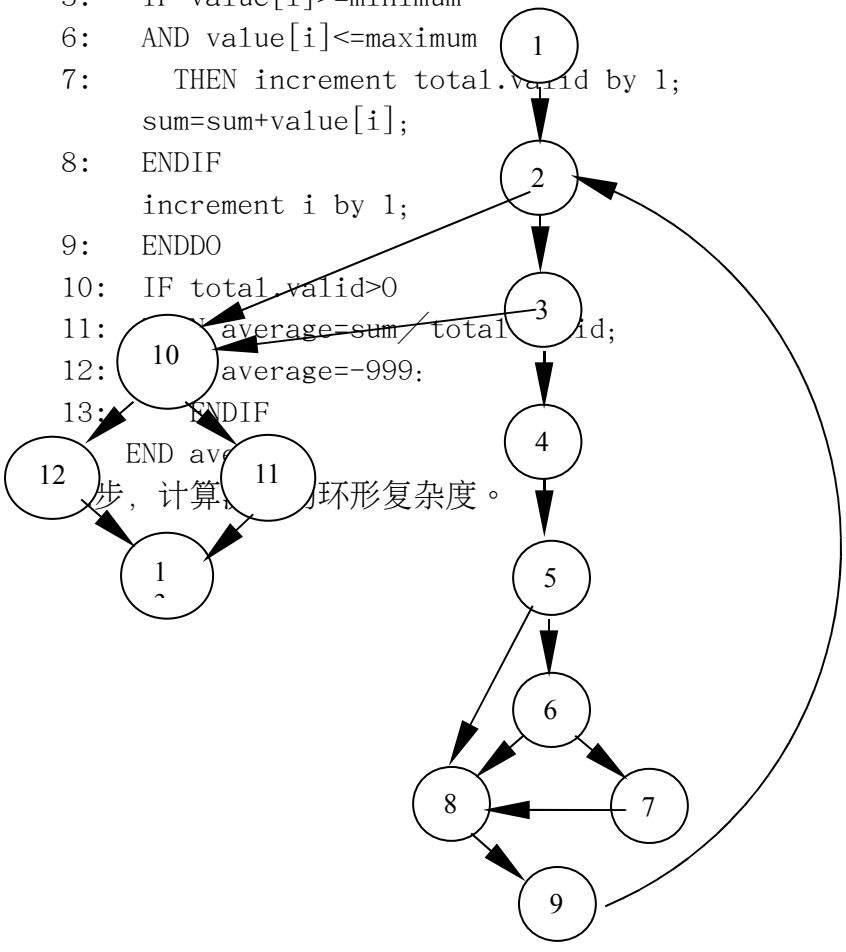

图 7.6　求平均值过程的流图

环形复杂度定量度量程序的逻辑复杂性。有了描绘程序控制流的流图之后，可以用第6.5.1小节讲述的3种方法之一计算环形复杂度。经计算，图7.6所示流图的环形复杂度为6。

第三步，确定线性独立路径的基本集合。

所谓独立路径是指至少引入程序的一个新处理语句集合或一个新条件的路径，用流图术语描述，独立路径至少包含一条在定义该路径之前不曾用过的边。

使用基本路径测试法设计测试用例时，程序的环形复杂度决定了程序中独立路径的数量，而且这个数是确保程序中所有语句至少被执行一次所需的测试数量的上界。

对于图7.6所描述的求平均值过程来说，由于环形复杂度为6，因此共有6条独立路径。例如，下面列出了6条独立路径：

路径1：1—2—10—11—13

路径2：1—2—10—12—13

路径3：1—2—3—10—11—13

路径4：1—2—3—4—5—8—9—2—…

路径5：1—2—3—4—5—6—8—9—2—…

路径6：1—2—3—4—5—6—7—8—9—2—…

路径4、5、6后面的省略号(…)表示，可以后接通过控制结构其余部分的任意路径(例如，
10—11—13)。

通常在设计测试用例时，识别出判定结点是很有必要的。本例中结点2、3、5、6和10是判定结点。

第四步，设计可强制执行基本集合中每条路径的测试用例。

应该选取测试数据使得在测试每条路径时都适当地设置好了各个判定结点的条件。例如，可以测试上一步得出的基本集合的测试用例如下：

路径1的测试用例：

value[k]一有效输入值，其中k<i(i的定义在下面)

value[i]=-999，其中2≤i≤100

预期结果：基于是的正确平均值和总数

注意，路径1无法独立测试，必须作为路径4、5和6的一部分来测试。

路径2的测试用例：

value[1]=-999

预期结果：average=-999，其他都保持初始值

路径3的测试用例：

试图处理101个或更多个值

前100个数值应该是有效输入值

预期结果：前100个数的平均值，总数为100

注意，路径3也无法独立测试，必须作为路径4、5和6的一部分来测试。

路径4的测试用例：

value[i]=有效输入值，其中i<100

value[k]<minimum，其中k<i

预期结果：基于是的正确平均值和总数

路径5的测试用例：

value[i]=有效输入值，其中i<100

value[k]>maximum，其中k<i

预期结果：基于是的正确平均值和总数

路径6的测试用例：

value[i]=有效输入值，其中i<100

预期结果：正确的平均值和总数

在测试过程中，执行每个测试用例并把实际输出结果与预期结果相比较。一旦执行完所有测试用例，就可以确保程序中所有语句都至少被执行了一次，而且每个条件都分别取过true值和false值。

应该注意，某些独立路径(例如，本例中的路径1和路径3)不能以独立的方式测试，也就是说，程序的正常流程不能形成独立执行该路径所需要的数据组合(例如，为了执行本例中的路径1，需要满足条件total.valid>0)。在这种情况下，这些路径必须作为另一个路径的一部分来测试。

2.条件测试

尽管基本路径测试技术简单而且高效，但是仅有这种技术还不够，还需要使用其他控制结构测试技术，才能进一步提高自盒测试的质量。

用条件测试技术设计出的测试用例，能够检查程序模块中包含的逻辑条件。一个简单条件是一个布尔变量或一个关系表达式，在布尔变量或关系表达式之前还可能有一个NOT("┐")算符。关系表达式的形式如下：

E_1 〈关系算符> E_2

其中，E_1和 E_2是算术表达式，而<关系算符>是下列算符之一：“<”，“≤”，“=”，“≠”，“>”或“≥”。复合条件由两个或多个简单条件、布尔算符和括弧组成。布尔算符有 OR(“|”)，AND(“&”)和 NOT("┐")。不包含关系表达式的条件称为布尔表达式。

因此，条件成分的类型包括布尔算符、布尔变量、布尔括弧(括住简单条件或复合条件)、关系算符及算术表达式。

如果条件不正确，则至少条件的一个成分不正确。因此，条件错误的类型如下：

·布尔算符错(布尔算符不正确，遗漏布尔算符或有多余的布尔算符)
·布尔变量错
·布尔括弧错
·关系算符错
·算术表达式错

条件测试方法着重测试程序中的每个条件。本节下面将讲述的条件测试策略有两个优点：①容易度量条件的测试覆盖率；②程序内条件的测试覆盖率可指导附加测试的
设计。

条件测试的目的不仅是检测程序条件中的错误，而且是检测程序中的其他错误。如果程序 P 的测试集能有效地检测 P 中条件的错误，则它很可能也可以有效地检测 P 中的其他错误。此外，如果一个测试策略对检测条件错误是有效的，则很可能该策略对检测程序的其他错误也是有效的。

人们已经提出了许多条件测试策略。分支测试可能是最简单的条件测试策略：对于复合条件 C 来说，C 的真分支和假分支以及 C 中的每个简单条件，都应该至少执行一次。

域测试要求对一个关系表达式执行 3 个或 4 个测试。对于形式为

E_1<关系算符>E_2

的关系表达式来说，需要 3 个测试分别使E_1的值大于、等于或小于E_2的值。如果<关系算符>错误而E_1和E_2正确，则这 3 个测试能够发现关系算符的错误。为了发现E_1和 E2 中的错误，让E_1值大于或小于E_2值的测试数据应该使这两个值之间的差别尽可能小。

包含 n 个变量的布尔表达式需要2^n个(每个变量分别取真或假这两个可能值的组合数)测试。这个策略可以发现布尔算符、变量和括弧的错误，但是，该策略仅在 n 很小时才是实用的。

在上述种种条件测试技术的基础上，K.C.Tai 提出了一种被称为 BRO(branch andrelational operator)测试的条件测试策略。如果在条件中所有布尔变量和关系算符都只出现一次而且没有公共变量，则 BRO 测试保证能发现该条件中的分支错和关系算符错。

BRO 测试利用条件 C 的条件约束来设计测试用例。包含 n 个简单条件的条件 C 的条件约束定义为 $(D_1,D_2,\cdots,Dn)$，其中 Di(0<i≤n)表示条件 C 中第 i 个简单条件的输出约束。如果在条件 C 的一次执行过程中，C 中每个简单条件的输出都满足 D 中对应的约束，则称 C 的这次执行覆盖了 C 的条件约束 D。

对于布尔变量 B 来说，B 的输出约束指出，B 必须是真(t)或假(f)。类似地，对于关系表达式来说，用符号>，=和<指定表达式的输出约束。

作为第一个例子，考虑下列条件

C_1：B_1 & B_2

其中，B_1和B_2是布尔变量。C_1的条件约束形式为(D_1，D_2)，其中D_1和D_2中的每一个都是“t”或“f”。值(t，f)是C_1的一个条件约束，并由使B_1值为真B_2值为假的测试所覆盖。BRO测试策略要求，约束集{(t，t)，(f，t)，(t，f)}被C_1的执行所覆盖。如果C_1因布尔算符错误而不正确，则至少上述约束集中的一个约束将迫使C_1失败。

作为第二个例子，考虑下列条件

C_2：B_1&(E_3=E_4)

其中，B_1是布尔变量，E_3和E_4是算术表达式。C_2的条件约束形式为(D_1，D_2)，其中D_1是“t”或“f”，D_2是>，=或<。除了C_2的第二个简单条件是关系表达式之外，C_2和C_1相同，因此，可以通过修改C_1的约束集{(t，t)，(f，t)，(t，f)}得出C_2的约束集。注意，对于(E_3=E_4)来说，“t”意味“=”，而“f”意味着“<”或“>”，因此，分别用(t，=)和(f，=)替换(t，t)和(f，t)，并用(t，<)和(t，>)替换(t，f)，就得到C。的约束集{(t，=)，(f，=)，(t，<)，(t，>)}。覆盖上述条件约束集的测试，保证可以发现C_2中布尔算符和关系算符的错误。

作为第三个例子，考虑下列条件

C_3：(E_1>E_2)＆(E_2=E_4)

其中，E_1、E_2、E_3和E_4是算术表达式。C_3的条件约束形式为(D_1，D_2)，而D_1和D_2的每一个都是>，=或<。除了C_3的第一个简单条件是关系表达式之外，C_3和C_2相同，因此，可以通过修改C_2的约束集得到C_3的约束集，结果为：

{(>，=)，(=，=)，(<，=)，(>，<)，(>，>))

覆盖上述条件约束集的测试，保证可以发现C。中关系算符的错误。

3．循环测试

循环是绝大多数软件算法的基础，但是，在测试软件时却往往未对循环结构进行足够的测试。

循环测试是一种白盒测试技术，它专注于测试循环结构的有效性。在结构化的程序中通常只有3种循环，即简单循环、串接循环和嵌套循环，如图7．7所示。下面分别讨论这3种循环的测试方法。

(1)简单循环。应该使用下列测试集来测试简单循环，其中n是允许通过循环的最大次数。

·跳过循环。

·只通过循环一次。

·通过循环两次。

·通过循环m次，其中研<n－1。

·通过循环，z－1，n，n＋1次。

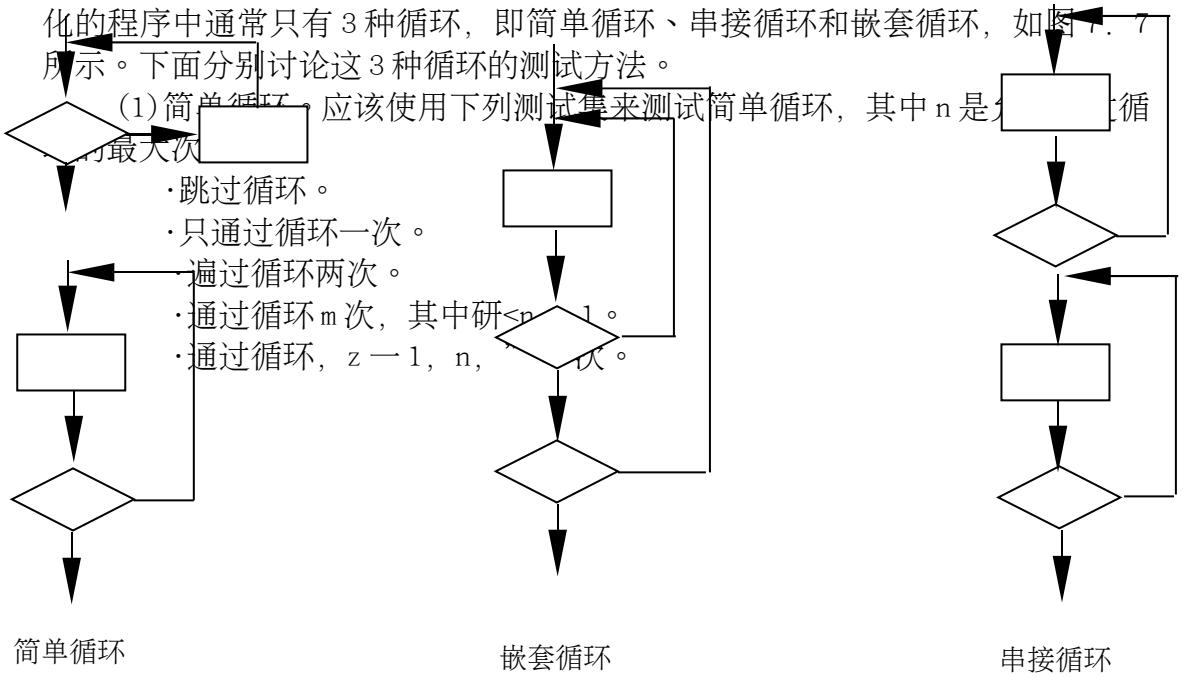

图7.7　3种循环

(2)嵌套循环。如果把简单循环的测试方法直接应用到嵌套循环，可能的测试数就会随嵌套层数的增加按几何级数增长，这会导致不切实际的测试数目。B.Beizer 提出了一种能减少测试数的方法：

·从最内层循环开始测试，把所有其他循环都设置为最小值。

·对最内层循环使用简单循环测试方法，而使外层循环的迭代参数(例如，循环计数

器)取最小值，并为越界值或非法值增加一些额外的测试。

·由内向外，对下一个循环进行测试，但保持所有其他外层循环为最小值，其他嵌套

循环为“典型”值。

·继续进行下去，直到测试完所有循环。

(3)串接循环。如果串接循环的各个循环都彼此独立，则可以使用前述的测试简单循环的方法来测试串接循环。但是，如果两个循环串接，而且第一个循环的循环计数器值是第二个循环的初始值，则这两个循环并不是独立的。当循环不独立时，建议使用测试嵌

7.7 黑盒测试技术

黑盒测试着重测试软件功能。黑盒测试并不能取代白盒测试，它是与白盒测试互补的测试方法，它很可能发现白盒测试不易发现的其他类型的错误。

黑盒测试力图发现下述类型的错误：①功能不正确或遗漏了功能；②界面错误；③数据结构错误或外部数据库访问错误；④性能错误；⑤初始化和终止

错误。

白盒测试在测试过程的早期阶段进行，而黑盒测试主要用于测试过程的后期。设计
黑盒测试方案时，应该考虑下述问题：

(1)怎样测试功能的有效性?

(2)哪些类型的输入可构成好测试用例?

(3)系统是否对特定的输入值特别敏感?

(4)怎样划定数据类的边界?

(5)系统能够承受什么样的数据率和数据量?

(6)数据的特定组合将对系统运行产生什么影响?

应用黑盒测试技术，能够设计出满足下述标准的测试用例集：

(1)所设计出的测试用例能够减少为达到合理测试所需要设计的测试用例的总数’

(2)所设计出的测试用例能够告诉我们，是否存在某些类型的错误，而不是仅仅指出
与特定测试相关的错误是否存在。

7.7.1 等价划分

等价划分是一种黑盒测试技术，这种技术把程序的输入域划分成若干个数据类，据此导出测试用例。一个理想的测试用例能独自发现一类错误(例如，对所有负整数的处理都
不正确)。

以前曾经讲过，穷尽的黑盒测试(即用所有有效的和无效的输入数据来测试程序)通常是不现实的。因此，只能选取少量最有代表性的输入数据作为测试数据，以期用较小的代价暴露出较多的程序错误。等价划分法力图设计出能发现若干类程序错误的测试用例，从而减少必须设计的测试用例的数目。

如果把所有可能的输入数据(有效的和无效的)划分成若干个等价类，则可以合理地做出下述假定：每类中的一个典型值在测试中的作用与这一类中所有其他值的作用相同。因此，可以从每个等价类中只取一组数据作为测试数据。这样选取的测试数据最有代表性，最可能发现程序中的错误。

使用等价划分法设计测试方案首先需要划分输入数据的等价类，为此需要研究程序的功能说明，从而确定输入数据的有效等价类和无效等价类。在确定输入数据的等价类时常常还需要分析输出数据的等价类，以便根据输出数据的等价类导出对应的输入数据等价类。

划分等价类需要经验，下述几条启发式规则可能有助于等价类的划分：

(1)如果规定了输入值的范围，则可划分出一个有效的等价类(输入值在此范围内)，两个无效的等价类(输入值小于最小值或大于最大值)；

(2)如果规定了输人数据的个数，则类似地也可以划分出一个有效的等价类和两个无效的等价类；

(3)如果规定了输入数据的一组值，而且程序对不同输入值做不同处理，则每个允许的输入值是一个有效的等价类，此外还有一个无效的等价类(任一个不

允许的输入值)’

(4)如果规定了输入数据必须遵循的规则，则可以划分出一个有效的等价类(符合规则)和若干个无效的等价类(从各种不同角度违反规则)；

(5)如果规定了输入数据为整型，则可以划分出正整数、零和负整数等 3 个有效类；

(6)如果程序的处理对象是表格，则应该使用空表，以及含一项或多项的表。

以上列出的启发式规则只是测试时可能遇到的情况中的很小一部分，实际情况千变万化，根本无法一一列出 。为了正确划分等价类，一是要注意积累经验，二是要正确分析被测程序的功能 。此外，在划分无效的等价类时还必须考虑编译程序的检错功能，一般说来，不需要设计测试数据用来暴露编译程序肯定能发现的错误 。最后说明一点，上面列出的启发式规则虽然都是针对输入数据说的，但是其中绝大部分也同样适用于输出数据。

划分出等价类以后，根据等价类设计测试方案时主要使用下面两个步骤:

(1)设计一个新的测试方案以尽可能多地覆盖尚未被覆盖的有效等价类，重复这一步骤直到所有有效等价类都被覆盖为止；

(2)设计一个新的测试方案，使它覆盖一个而且只覆盖一个尚未被覆盖的无效等价类，重复这一步骤直到所有无效等价类都被覆盖为止。

注意，通常程序发现一类错误后就不再检查是否还有其他错误，因此，应该使每个测试方案只覆盖一个无效的等价类。

下面用等价划分法设计一个简单程序的测试方案。

假设有一个把数字串转变成整数的函数。运行程序的计算机字长 16 位，用二进制补码表示整数。这个函数是用 Pasca1 语言编写的，它的说明如下:

```
function strtoint(dstr: shortstr): integer。
```

函数的参数类型是 shortstr，它的说明是:

```
type shortstr=array[1..6]of char;
```

被处理的数字串是右对齐的，也就是说，如果数字串比 6 个字符短，则在它的左边补空格。如果数字串是负的，则负号和最高位数字紧相邻(负号在最高位数字左边一位)。

考虑到 Pasca1 编译程序固有的检错功能，测试时不需要使用长度不等于 6 的数组做实在参数，更不需要使用任何非字符数组类型的实在参数。

分析这个程序的规格说明，可以划分出如下等价类:

有效输入的等价类有

(1)1～6 个数字字符组成的数字串(最高位数字不是零)；

(2)最高位数字是零的数字串；

(3)最高位数字左邻是负号的数字串；

无效输入的等价类有

(4)空字符串(全是空格)；

(5)左部填充的字符既不是零也不是空格；

(6)最高位数字右面由数字和空格混合组成；

(7)最高位数字右面由数字和其他字符混合组成;
(8)负号与最高位数字之间有空格;

合法输出的等价类有

(9)在计算机能表示的最小负整数和零之间的负整数;
(10)零;
(11)在零和计算机能表示的最大正整数之间的正整数;
非法输出的等价类有
(12)比计算机能表示的最小负整数还小的负整数;
(13)比计算机能表示的最大正整数还大的正整数。
因为所用的计算机字长 16 位，用二进制补码表示整数，所以能表示的最小负整数是
-32 768，能表示的最大正整数是 32 767。
根据上面划分出的等价类，可以设计出下述测试方案(注意，每个测试方案由 3 部分
内容组成):
(1)1～6 个数字组成的数字串，输出是合法的正整数。
输入: ‘ 1’
预期的输出: 1
(2)最高位数字是零的数字串，输出是合法的正整数。
输入: ‘000001’
预期的输出: 1
(3)负号与最高位数字紧相邻，输出合法的负整数。
输入: ‘-00001’
预期的输出: -1
(4)最高位数字是零，输出也是零。
输入: ‘000000’
预期的输出: 0
(5)太小的负整数。
输入: ‘-47561’
预期的输出: “错误——无效输入”
(6)太大的正整数。
输入: ‘132767’
预期的输出: “错误——无效输入”
(7)空字符串。
输入: ‘ ’
预期的输出: “错误——没有数字”
(8)字符串左部字符既不是零也不是空格。
输入: ‘×××××1’
预期的输出: “错误——填充错”.
(9)最高位数字后面有空格。
输入: ‘ 1 2’

预期的输出：“错误——无效输人”
(10)最高位数字后面有其他字符。
输入：‘　　1××2’
预期的输出：“错误——无效输入”
(11)负号和最高位数字之间有空格。
输入：‘　一　　12’
预期的输出：“错误——负号位置错”

7.7.2 边界值分析

经验表明，处理边界情况时程序最容易发生错误。例如，许多程序错误出现在下标、纯量、数据结构和循环等等的边界附近。因此，设计使程序运行在边界情况附近的测试方案，暴露出程序错误的可能性更大一些。

使用边界值分析方法设计测试方案首先应该确定边界情况，这需要经验和创造性，通常输入等价类和输出等价类的边界，就是应该着重测试的程序边界情况。选取的测试数据应该刚好等于、刚刚小于和刚刚大于边界值。也就是说，按照边界值分析法，应该选取刚好等于、稍小于和稍大于等价类边界值的数据作为测试数据，而不是选取每个等价类内的典型值或任意值作为测试数据。

通常设计测试方案时总是联合使用等价划分和边界值分析两种技术。例如，为了测试前述的把数字串转变成整数的程序，除了上一小节已经用等价划分法设计出的测试方案外，还应该用边界值分析法再补充下述测试方案：

(12)使输出刚好等于最小的负整数。
输入：‘-32768’
预期的输出为：-32768
(13)使输出刚好等于最大的正整数。
输入：‘　32767’
预期的输出：32767
原来用等价划分法设计出来的测试方案5最好改为：
(14)使输出刚刚小于最小的负整数。
输入：‘-32769’
预期的输出：“错误——无效输入”
原来的测试方案6最好改为：
(15)使输出刚刚大于最大的正整数。
输入：‘　32768’
预期的输出：“错误——无效输入”

此外，根据边界值分析方法的要求，应该分别使用长度为0，1和6的数字串作为测试数据。上一小节中设计的测试方案1，2，3，4和7已经包含了这些边界情况。

7.7.3 错误推测

使用边界值分析和等价划分技术，有助于设计出具有代表性的、因而也就容易暴露程序错误的测试方案。但是，不同类型不同特点的程序通常又有一些特殊

的容易出错的情况。此外，有时分别使用每组测试数据时程序都能正常工作，这些输入数据的组合却可能检测出程序的错误。一般说来，即使是一个比较小的程序，可能的输入组合数也往往十分巨大，因此必须依靠测试人员的经验和直觉，从各种可能的测试方案中选出一些最可能引起程序出错的方案。对于程序中可能存在哪类错误的推测，是挑选测试方案时的一个重要因素。

错误推测法在很大程度上靠直觉和经验进行。它的基本想法是列举出程序中可能有的错误和容易发生错误的特殊情况，并且根据它们选择测试方案。第 7.3 节列出了模块中一些常见错误的清单，这些是模块测试经验的总结。对于程序中容易出错的情况也有一些经验总结出来，例如，输入数据为零或输出数据为零往往容易发生错误；如果输入或输出的数目允许变化(例如，被检索的或生成的表的项数)，则输入或输出的数目为 0 和 1 的情况(例如，表为空或只有一项)是容易出错的情况。还应该仔细分析程序规格说明书，注意找出其中遗漏或省略的部分，以便设计相应的测试方案，检测程序员对这些部分的处理是否正确。

此外，经验表明，在一段程序中已经发现的错误数目往往和尚未发现的错误数成正比。例如，在 IBM OS／370 操作系统中，用户发现的全部错误的 47％只与该系统 4％的模块有关。因此，在进一步测试时要着重测试那些已发现了较多错误的程序段。

等价划分法和边界值分析法都只孤立地考虑各个输入数据的测试功效，而没有考虑多个输人数据的组合效应，可能会遗漏了输入数据易于出错的组合情况。选择输入组合的一个有效途径是利用判定表或判定树为工具，列出输入数据各种组合与程序应作的动作(及相应的输出结果)之间的对应关系，然后为判定表的每一列至少设计一个测试用例。

选择输入组合的另一个有效途径是把计算机测试和人工检查代码结合起来。例如，通过代码检查发现程序中两个模块使用并修改某些共享的变量，如果一个模块对这些变量的修改不正确，则会引起另一个模块出错，因此这是程序发生错误的一个可能的原因。应该设计测试方案，在程序的一次运行中同时检测这两个模块，特别要着重检测一个模块修改了共享变量后，另一个模块能否像预期的那样正常使用这些变量。反之，如果两个模块相互独立，则没有必要测试它们的输入组合情况。通过代码检查也能发现模块相互依赖的关系，例如，某个算术函数的输入是数字字符串，调用 7.7.1 节例子中的“strtoint”函数，把输入的数字串转变成内部形式的整数。在这种情况下，不仅必须测试这个转换函数，还应该测试调用它的算术函数在转换函数接收到无效输入时的响应。

7.8 调　试

调试(也称为纠错)作为成功测试的后果出现，也就是说，调试是在测试发现错误之后排除错误的过程。虽然调试应该而且可以是一个有序过程，但是，目前它在很大程度上仍然是一项技巧。软件工程师在评估测试结果时，往往仅面对着软件错误的症状，也就是说，软件错误的外部表现和它的内在原因之间可能并没有明显的联系。调试就是把症状和原因联系起来的尚未被人深入认识的智力过程。

7.8.1 调试过程

调试不是测试，但是它总是发生在测试之后。如图 7.8 所示，调试过程从执行一个测试用例开始，评估测试结果，如果发现实际结果与预期结果不一致，则这种不一致就是一个症状，它表明在软件中存在着隐藏的问题。调试过程试图找出产生症状的原因，以便改正错误。

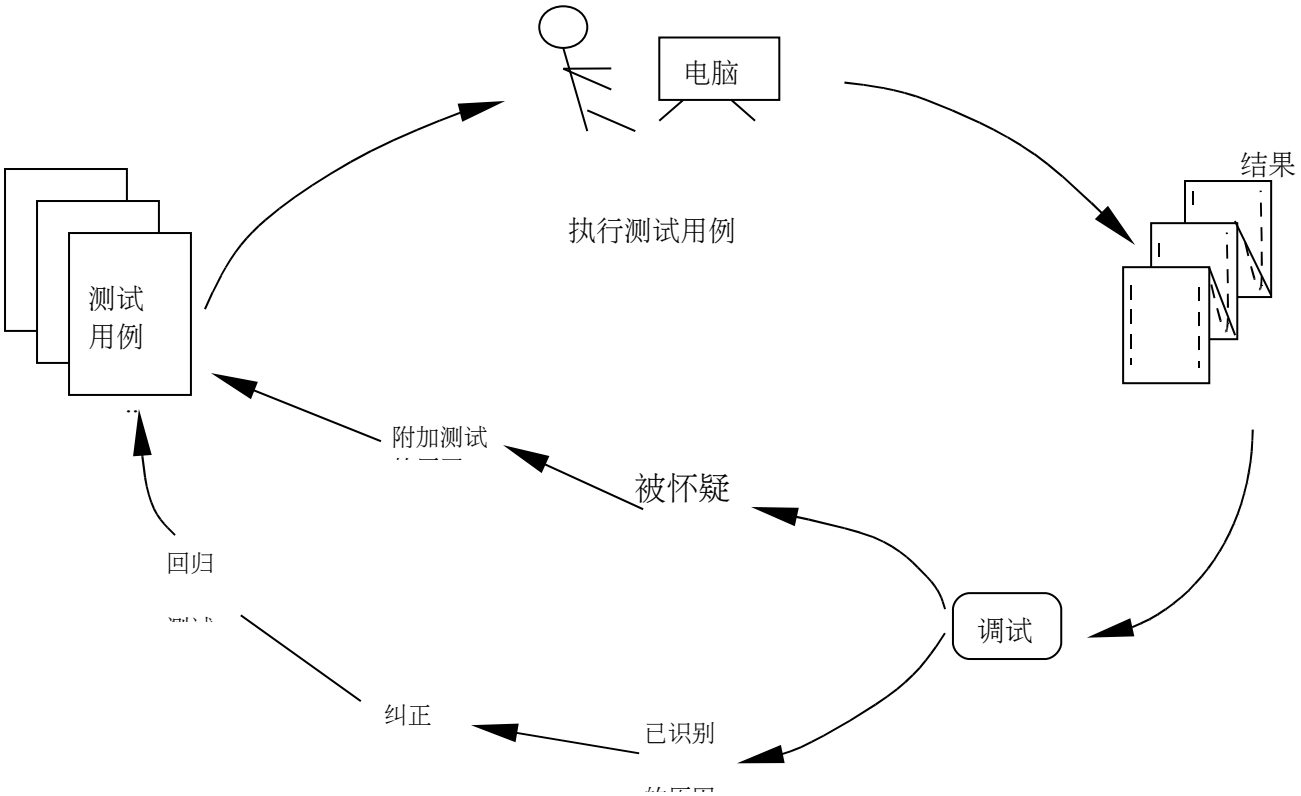

图 7.8 测试过程

调试过程总会有以下两种结果：①找到问题的原因并把问题改正和排除掉了；②没找出问题的原因。在后一种情况下，调试人员可以猜想一个原因，并设计测试用例来验证这个假设，重复此过程直至找到原因并改正了错误。

调试是软件开发过程中最艰巨的脑力劳动。调试工作如此困难，可能心理方面的原因多于技术方面的原因，但是，软件错误的下述特征也是相当重要的原因：

(1)症状和产生症状的原因可能在程序中相距甚远，也就是说，症状可能出现在程序的一个部分，而实际的原因可能在与之相距很远的另一部分。紧耦合的程序结构更加剧了这种情况。

(2)当改正了另一个错误之后，症状可能暂时消失了。

(3)症状可能实际上并不是由错误引起的(例如，舍入误差)。

(4)症状可能是由不易跟踪的人为错误引起的。

(5)症状可能是由定时问题而不是由处理问题引起的。

(6)可能很难重新产生完全一样的输入条件(例如，输入顺序不确定的实时

应用系统)。

(7)症状可能时有时无，这种情况在硬件和软件紧密地耦合在一起的嵌入式系统中特别常见。

(8)症状可能是由分布在许多任务中的原因引起的，这些任务运行在不同的处理机上。

在调试过程中会遇到从恼人的小错误(例如，不正确的输出格式)到灾难性的大错误(例如，系统失效导致严重的经济损失)等各种不同的错误。错误的后果越严重，查找错误原因的压力也越大。通常，这种压力会导致软件开发人员在改正一个错误的同时引入两个甚至更多个错误。

7.8.2 调试途径

无论采用什么方法，调试的目标都是寻找软件错误的原因并改正错误。通常需要把系统地分析、直觉和运气组合起来，才能实现上述目标。一般说来，有下列3种调试途径
可以采用：

1.蛮干法

蛮干法可能是寻找软件错误原因的最低效的方法。仅当所有其他方法都失败了的情况下，才应该使用这种方法。按照“让计算机自己寻找错误”的策略，这种方法印出内存的内容，激活对运行过程的跟踪，并在程序中到处都写上wRITE(输出)语句，希望在这样生成的信息海洋的某个地方发现错误原因的线索。虽然所生成的大量信息也可能最终导致调试成功，但是，在更多情况下这样做只会浪费时间和精力。在使用任何一种调试方法之前，必须首先进行周密的思考，必须有明确的目的，应该尽量减少无关信息的数量。

2.回溯法

回溯是一种相当常用的调试方法，当调试小程序时这种方法是有效的。具体做法是，从发现症状的地方开始，人工沿程序的控制流往回追踪分析源程序代码，直到找出错误原因为止。但是，随着程序规模扩大，应该回溯的路径数目也变得越来越大，以至彻底回溯变成完全不可能了。

3.原因排除法

对分查找法、归纳法和演绎法都属于原因排除法。

对分查找法的基本思路是，如果已经知道每个变量在程序内若干个关键点的正确值，则可以用赋值语句或输入语句在程序中点附近“注入”这些变量的正确值，然后运行程序并检查所得到的输出。如果输出结果是正确的，则错误原因在程序的前半部分；反之，错误原因在程序的后半部分。对错误原因所在的那部分再重复使用这个方法，直到把出错范围缩小到容易诊断的程度为止。

归纳法是从个别现象推断出一般性结论的思维方法。使用这种方法调试程序时，首先把和错误有关的数据组织起来进行分析，以便发现可能的错误原因。然后导出对错误原因的一个或多个假设，并利用已有的数据来证明或排除这些假设。当然，如果已有的数据尚不足以证明或排除这些假设，则需设计并执行一些新的测试用例，以获得更多的数据。

演绎法从一般原理或前提出发，经过排除和精化的过程推导出结论。采用这种方法调试程序时，首先设想出所有可能的出错原因，然后试图用测试来排除每一个假设的原因。如果测试表明某个假设的原因可能是真的原因，则对数据进行细化以准确定位错误。

上述3种调试途径都可以使用调试工具辅助完成，但是工具并不能代替对全部设计文档和源程序的仔细分析与评估。

如果用遍了各种调试方法和调试工具却仍然找不出错误原因，则应该向同行求助。把遇到的问题向同行陈述并一起分析讨论，往往能开阔思路，较快找出错误原因。

一旦找到错误就必须改正它，但是，改正一个错误可能引入更多的其他错误，以至“得不偿失”。因此，在动手改正错误之前，软件工程师应该仔细考虑下述3个问题：

(1)是否同样的错误也在程序其他地方存在?在许多情况下，一个程序错误是由错误的逻辑思维模式造成的，而这种逻辑思维模式也可能用在别的地方。仔细分析这种逻辑模式，有可能发现其他错误。

(2)将要进行的修改可能会引入的“下一个错误”是什么?在改正错误之前应该仔细研究源程序(最好也研究设计文档)，以评估逻辑和数据结构的耦合程度。如果所要做的修改位于程序的高耦合段中，则修改时必须特别小心谨慎。

(3)为防止今后出现类似的错误，应该做什么?如果不仅修改了软件产品还改进了开发软件产品的软件过程，则不仅排除了现有程序中的错误，还避免了今后在程序中可能出现的错误。

7.9 软件可靠性

测试阶段的根本目标是消除错误，保证软件的可靠性。读者可能会问，什么是软件的可靠性呢?应该进行多少测试，软件才能达到所要求的可靠程度呢?这些正是本节要着重讨论的问题。

7.9.1 基本概念

1.软件可靠性的定义

对于软件可靠性有许多不同的定义，其中多数人承认的一个定义是：软件可靠性是程序在给定的时间间隔内，按照规格说明书的规定成功地运行的概率。

在上述定义中包含的随机变量是时间间隔。显然，随着运行时间的增加，运行时出现程序故障的概率也将增加，即可靠性随着给定的时间间隔的加大而减

少。

按照 IEEE 的规定，术语“错误”的含义是由开发人员造成的软件差错(bug)，而术语“故障”的含义是由错误引起的软件的不正确行为。在下面的论述中，将按照 IEEE 规定的含义使用这两个术语。

2.软件的可用性

通常用户也很关注软件系统可以使用的程度。一般说来，对于任何其故障是可以修复的系统，都应该同时使用可靠性和可用性衡量它的优劣程度。

软件可用性的一个定义是：软件可用性是程序在给定的时间点，按照规格说明书的规定，成功地运行的概率。

可靠性和可用性之间的主要差别是，可靠性意味着在 0 到£这段时间间隔内系统没有失效，而可用性只意味着在时刻 f，系统是正常运行的。因此，如果在时刻£系统是可用的，则有下述种种可能：在 0 到£这段时间内，系统一直没失效(可靠)；在这段时间内失效了一次，但是又修复了；在这段时间内失效了两次修复了两次；……

如果在一段时间内，软件系统故障停机时间分别为£。，，£。。，…，正常运行时间分别为

$$A = \frac{T_{up}}{T_{up} + T_{down}} \qquad (7.1)$$

其中 $T_{up} = \sum t_{ui}$ $T_{down} = \sum t_{di}$

如果引入系统平均无故障时间 MTTF 和平均维修时间 MTTR 的概念，则(7.1)式可
以变成

$$A = \frac{MTTF}{MTTF + MTTR} \qquad (7.2)$$

平均维修时间 MTTR 是修复一个故障平均需要用的时间，它取决于维护人员的技术水平和对系统的熟悉程度，也和系统的可维护性有重要关系，第 8 章将讨论软件维护问题。平均无故障时间 MTTF 是系统按规格说明书规定成功地运行的平均时间，它主要取决于系统中潜伏的错误的数目，因此和测试的关系十分密切。

7.9.2 估算平均无故障时间的方法

软件的平均无故障时间 MTTF 是一个重要的质量指标，往往作为对软件的一项要求，由用户提出来。为了估算 MTTF，首先引入一些有关的量。

1.符号

在估算MTTF的过程中使用下述符号表示有关的数量：

E_r——测试之前程序中错误总数；

I_r——程序长度(机器指令总数)；

τ——测试(包括调试)时间；

$E_d(\tau)$——在0至τ期间发现的错误数；

$E_c(\tau)$——在0至τ期间改正的错误数。

2.基本假定

根据经验数据，可以作出下述假定。

(1)在类似的程序中，单位长度里的错误数Er／J，近似为常数。美国的一些统计数字表明，通常

$$0.5\times10^{-2}\leqslant E_T/I_T\leqslant 2\times10^{-2}$$

也就是说，在测试之前每1 000条指令中大约有5～20个错误。

(2)失效率正比于软件中剩余的(潜藏的)错误数，而平均无故障时间MTTF与剩余的错误数成反比。

(3)此外，为了简化讨论，假设发现的每一个错误都立即正确地改正了(即，调试过程没有引入新的错误)。因此

$$E_c(\tau)=E_d(\tau)$$

剩余的错误数为

$$E_r(\tau)=E_r-E_c(\tau) \qquad (7.3)$$

单位长度程序中剩余的错误数为

$$\varepsilon_r(\tau)=E_T/I_T-E_c(\tau)/I_T \qquad (7.4)$$

3.估算平均无故障时间

经验表明，平均无故障时间与单位长度程序中剩余的错误数成反比，即

$$MTTF=\frac{1}{K(E_T/I_T-E_c(\tau)/I_T)} \qquad (7.5)$$

其中K为常数，它的值应该根据经验选取。美国的一些统计数字表明，K的典型值是200。

估算平均无故障时间的公式，可以评价软件测试的进展情况。此外，由(7.5)式可得

$$E_c=E_T-\frac{I_T}{K\times MTTF} \qquad (7.6)$$

因此，也可以根据对软件平均无故障时间的要求，估计需要改正多少个错误之

后，测试工作才能结束。

4.估计错误总数的方法

程序中潜藏的错误的数目是一个十分重要的量，它既直接标志软件的可靠程度，又是计算软件平均无故障时间的重要参数。显然，程序中的错误总数 E_T 与程序规模、类型、开发环境、开发方法论、开发人员的技术水平和管理水平等都有密切关系。下面介绍估计 E_T 的两个方法。

(1)植入错误法

使用这种估计方法，在测试之前由专人在程序中随机地植入一些错误，测试之后，根据测试小组发现的错误中原有的和植入的两种错误的比例，来估计程序原有错误的总数 E_T。

假设人为地植入的错误数为 N_s，经过一段时间的测试之后发现 n_s，个植入的错误，此外还发现了 n 个原有的错误。如果可以认为测试方案发现植入错误和发现原有错误的能力相同，则能够估计出程序中原有错误的总数为

$$\hat{N} = \frac{n}{n_s} N_s \qquad (7.7)$$

其中 $\hat{N}$ 即是错误总数 E_T，的估计值。

(2)分别测试法

植入错误法的基本假定是所用的测试方案发现植入错误和发现原有错误的概率相同。但是，人为地植入的错误和程序中原有的错误可能性质很不相同，发现它们的难易程
自然也不相同，因此，上述基本假定可能有时和事实不完全一致。

如果有办法随机地把程序中一部分原有的错误加上标记，然后根据测试过程中发现的有标记错误和无标记错误的比例，估计程序中的错误总数，则这样得出的结果比用植入错误法得到的结果更可信一些。

为了随机地给一部分错误加标记，分别测试法使用两个测试员(或测试小组)，彼此独立地测试同一个程序的两个副本，把其中一个测试员发现的错误作为有标记的错误。具体做法是，在测试过程的早期阶段，由测试员甲和测试员乙分别测试同一个程序的两个副本，由另一名分析员分析他们的测试结果。用 r 表示测试时间，假设

- τ =0 时错误总数为 B_0;
- $\tau = \tau_1$，时测试员甲发现的错误数为 B_1;
- $\tau = \tau_1$，时测试员乙发现的错误数为 B_2;
- $\tau = \tau_1$ 时两个测试员发现的相同错误数为 b_c。

如果认为测试员甲发现的错误是有标记的，即程序中有标记的错误总数为 B_1，则测试员乙发现的 B_2 个错误中有 b_c 个是有标记的。假定测试员乙发现有标记错误和发现无标记错误的概率相同，则可以估计出测试前程序中的错误总数为

$$\hat{B}_0 = \frac{B_2}{b_c} B_1 \qquad (7.8)$$

使用分别测试法，在测试阶段的早期，每隔一段时间分析员分析两名测试员的测试结果，并且用(7.8)式计算 $\hat{B}_0$ 如果几次估算的结果相差不多，则可用 $\hat{B}_0$ 的平均值作为 E_T 的估计值。.此后一名测试员可以改做其他工作，由余下的一名测试员继续完成测试工作，因为他可以继承另一名测试员的测试结果，所以分别测试法增加的测试成本并不太多。

7.10 小 结

实现包括编码和测试两个阶段。

按照传统的软件工程方法学，编码是在对软件进行了总体设计和详细设计之后进行的，它只不过是把软件设计的结果翻译成用某种程序设计语言书写的程序，因此，程序的质量基本上取决于设计的质量。但是，编码使用的语言，特别是写程序的风格，也对程序质量有相当大的影响。

大量实践结果表明，高级程序设计语言较汇编语言有很多优点。因此，除非在非常必要的场合，一般不要使用汇编语言写程序。至于具体选用哪种高级程序设计语言，则不仅要考虑语言本身的特点，还应该考虑使用环境等一系列实际因素。

程序内部的良好文档资料，有规律的数据说明格式，简单清晰的语句构造和输入输出格式等等，都对提高程序的可读性有很大作用，也在相当大的程度上改进了程序的可维护性。

目前软件测试仍然是保证软件可靠性的主要手段。测试阶段的根本任务是发现并改正软件中的错误。

软件测试是软件开发过程中最艰巨最繁重的任务，大型软件的测试应该分阶段地进行，通常至少分为单元测试、集成测试和验收测试 3 个基本阶段。

设计测试方案是测试阶段的关键技术问题，基本目标是选用最少量的高效测试数据，做到尽可能完善的测试，从而尽可能多地发现软件中的问题。

应该认识到，软件测试不仅仅指利用计算机进行的测试，还包括人工进行的测试(例如，代码审查)。两种测试途径各有优缺点，互相补充，缺一不可。

白盒测试和黑盒测试是软件测试的两类基本方法，这两类方法各有所长，相互补充。

通常，在测试过程的早期阶段主要使用白盒方法，而在测试过程的后期阶段主

要使用黑盒方法。为了设计出有效的测试方案，软件工程师应该深入理解并坚持运用关于软件测试的基本准则。

设计白盒测试方案的技术主要有，逻辑覆盖和控制结构测试；设计黑盒测试方案的技术主要有，等价划分、边界值分析和错误推测。

在测试过程中发现的软件错误必须及时改正，这就是调试的任务。为了改正错误，首先必须确定错误的准确位置，这是调试过程中最困难的工作，需要审慎周密的思考和推理。为了改正错误往往需要修正原来的设计，必须通盘考虑统筹兼顾，而不能“头疼医头、脚疼医脚”，应该尽量避免在调试过程中引进新错误。

测试和调试是软件测试阶段中的两个关系非常密切的过程，它们往往交替进行。

程序中潜藏的错误的数目，直接决定了软件的可靠性。通过测试可以估算出程序中剩余的错误数。根据测试和调试过程中已经发现和改正的错误数，可以估算软件的平均无故障时间；反之，根据要求达到的软件平均无故障时间，可以估算出应该改正的错误数，从而能够判断测试阶段何时可以结束。

习题7

1.下面给出的伪码中有一个错误。请仔细阅读这段伪码，说明该伪码的语法特点，
找出并改正伪码中的错误。字频统计程序的伪码如下：

```
INITIALIZE the Program
READ the first text record
DO WHILE there are more words in the text record
   DO WHILE there are more words in the text record
      EXTRACT the next text word
      SEARCH the word_table for the extracted word
      IF the extracted word is found
         INCREMENT the word's occurrence count
       ELSE
         INSERT the extracted word into the table
      END IF
      INCREMENT the words_processed count
   END DO at the end of the text record
   READ the next text record
END DO when aI】text records have heen read
PRINT the table and summary information
TERMINATE the program
```

2.研究下面给出的伪码程序，要求：

(1)画出它的程序流程图。

(2)它是结构化的还是非结构化的?说明理由。

(3)若是非结构化的，则

(a)把它改造成仅用3种控制结构的结构化程序;
(b)写出这个结构化设计的伪码;
(c)用盒图表示这个结构化程序。
(4)找出并改正程序逻辑中的错误。

```
COMMENT: PROGRAM SEARCHES FOR FIRST N REFERENCES
         TO A TOPIC IN AN INFORMAT10N RETRIEVAL
         SYSTEM WITH T TOTAI, EINTRIES

         INPUTN
         INPUT KEYWORD(S)FOR TOPIC
         I=0
         MATCH=O
         DO WHILE I≤T
             I=I+1
             IF WORD=KEYWORD
                THEN MATCH=MATCH+1
                     STORE IN BUFFER
              END
              IF MATCH=N
                 THEN GOTO OUTPUT
              END
          END
          IF N=O
             THEN PRINT ”NO MATCH”
OUTPUT:   ELSE CALL SUBROUTINE TO PRINT BUFFER
          INFORMAT10N
          END
```

3.在第2题的设计中若输入的N值或KEYWORD不合理，会发生问题。
(1)给出这些变量的不合理值的例子。
(2)将这些不合理值输入程序会有什么后果?
(3)怎样在程序中加入防错措施，以防止出现这些问题?
4.回答下列问题:
(1)什么是模块测试和集成测试?它们各有什么特点?
(2)假设有一个由1 000行FORTRAN语句构成的程序(经编译后大约有5 000条机器指令)，你估计在对它进行测试期间将发现多少个错误?为什么?
(3)设计下列伪码程序的语句覆盖和路径覆盖测试用例:

```
START
INPUT(A, B, C)
IF A>5
  THEN X=10
  ELSE X=1
END IF
IF B>10
  THEN Y=20
```

```
  ELSE Y=2
END IF
IF C>15
  THEN Z=30
  ELSE Z=3
END IF
PR1NT(X, Y, Z)
STOP
```

5.某图书馆有一个使用cRT终端的信息检索系统，该系统有下列4个基本检索
命令：

名　称	语法	操　　作
BROWSE (浏览)	b(关键字)	系统搜索给出的关键字，找出字母排列与此关键字最相近的 字。然后在屏幕上显示约20个加了行号的字，与给出的关键 字完全相同的字约在中央
SELECT (选取)	s(屏幕上的行号)	系统创建一个文件保存含有由行号指定的关键字的全部图书 的索引，这些索引都有编号(第一个索引的编号为1，第二个为 2……依此类推)
DISPLAY (显示)	d(索引号)	系统在屏幕上显示与给定的索引号有关的信息，这些信息与通 常在图书馆的目录卡片上给出的信息相同。这条命令接在 BROWSE／SELECT或FIND命令后面用，以显示文件中的索 引信息
FIND (查找)	f(作者姓名	系统搜索指定的作者姓名，并在屏幕上显示该作者的著作的索 引号，同时把这些索引存人文件

要求：
(1)设计测试数据以全面测试系统的正常操作；
(2)设计测试数据以测试系统的非正常操作。

6.航空公司A向软件公司B订购了一个规划飞行路线的程序。假设你是软件公司C的软件工程师，A公司已雇用你所在的公司对上述程序进行验收测试。你的任务是，根据下述事实设计验收测试的输入数据，解释你选取这些数据的理由。

领航员向程序输入出发点和目的地，以及根据天气和飞机型号而初步确定

的飞行高度。程序读入途中的风向风力等数据，并且制定出 3 套飞行计划(高度，速度，方向及途中 的 5 个位置校核点)。所制定的飞行计划应做到燃料消耗和飞行时间都最少。

7.严格说来，有两种不同的路径覆盖测试，分别为程序路径覆盖和程序图路径覆盖。这两种测试可分别称为程序的自然执行和强迫执行。所谓自然执行是指测试者(人或计算机)读入程序中的条件表达式，根据程序变量的当前值计算该条件表达式的值(真或假)，并相应地分支。强迫执行是在用程序图作为程序的抽象模型时产生的一个人为的概念，它可以简化测试问题。强迫执行的含义是，一旦遇到条件表达式，测试者就强迫程序分两种情况(条件表达式的值为真和为假)执行。显然，强迫执行将遍历程序图的所有路径，然而由于各个条件表达式之间存在相互依赖的关系，这些路径中的某一些在自然执行时可能永远也不会进入。

为了使强迫执行的概念在实际工作中有用，它简化测试工具的好处应该超过它使用额外的不可能达到的测试用例所带来的坏处。在绝大多数情况下，强迫执行的测试数并不比自然执行的测试数大很多，此外，对强迫执行的定义实际上包含了一种技术，能够缩短在测试含有循环的程序时所需要的运行时间。

程序的大部分执行时间通常用于重复执行程序中的 DO 循环，特别是嵌套的循环。因此必须发明一种技术，使得每个 DO 循环只执行一遍。这样做并不会降低测试的功效，因为经验表明第一次或最后一次执行循环时最容易出错。

Laemme1 教授提出的自动测试每条路径的技术如下:

当编写程序时每个 DO 循环应该写成一种包含测试变量 T 和模式变量 M 的特殊形式，因此

```
DO I=1 TO 38
```

应变成

```
DO I=1 TO M*38+(1-M)*T
```

可见，当 M=0 时处于测试模式，而 M=1 时处于正常运行模式。当处于测试模式时，令 T=0 则该循环一次也不执行，令 T=1 则该循环只执行一次。

类似地应该使用模式变量和测试变量改写 IF 语句，例如

```
IF X+Y>0 THEN Z=X
         ELSE Z=Y
```

应变成

```
IF M*(X+Y)+T>0 THEN Z=X
                EISE Z=Y
```

正常运行时令 M=1 和 T=0，测试期间令 M=0，为测试 THEN 部分需令 T=+1，测试 ELSE 部分则令 T=-1。

要求:

(1)选取一个包含循环和 1F 语句的程序，用 Laemme1 技术修改这个程序，上机实际测试这个程序并解释所得到的结果。

(2)设计一个程序按照 Laemme1 技术自动修改待测试的程序。利用这个测试工具修改上一问中人工修改的程序，两次修改得到的结果一致吗?

(3)怎样把 Laemme1 技术推广到包含 WHILE DO 和 REPEAT UNTIL 语句的程序?

(4)试分析 Laemme1 技术的优缺点并提出改进意见。

8.对一个包含 10 000 条机器指令的程序进行一个月集成测试后，总共改正了 15 个错误，此时 MTTF=10h；经过两个月测试后，总共改正了 25 个错误(第二

个月改正了 10 个错误)，MTTF=15h。

要求：

(1)根据上述数据确定 MTTF 与测试时间之间的函数关系，画出 MTTF 与测试时间 τ 的关系曲线。在画这条曲线时做了什么假设?

(2)为做到 MTTF=100h，必须进行多长时间的集成测试?当集成测试结束时总共改正了多少个错误，还有多少个错误潜伏在程序中?

9.如对一个长度为 100 000 条指令的程序进行集成测试期间记录下下面的数据：

(a)7 月 1 日：集成测试开始，没有发现错误。

(b)8 月 2 日：总共改正 100 个错误，此时 MTTF=0.4h

(c)9 月 1 日：总共改正 300 个错误，此时，MTTF=2h

根据上列数据完成下列各题：

(1)估计程序中的错误总数;

(2)为使 MTTF 达到 10h，必须测试和调试这个程序多长时间?

(3)画出 MTTF 和测试时间 τ 之间的函数关系曲线。

10.在测试一个长度为 24 000 条指令的程序时，第一个月由甲、乙两名测试员各自独立测试这个程序。经一个月测试后，甲发现并改正 20 个错误，使 MTTF 达到 10h。与此同时，乙发现 24 个错误，其中 6 个甲也发现了。以后由甲一个人继续测试这个程序。问：

(1)刚开始测试时程序中总共有多少个潜藏的错误?

(2)为使 MTTF 达到 60h，必须再改正多少个错误?还需用多长测试时间?

(3)画出 MTTF 与集成测试时间 τ 之间的函数关系曲线。