

# 软件工程导论

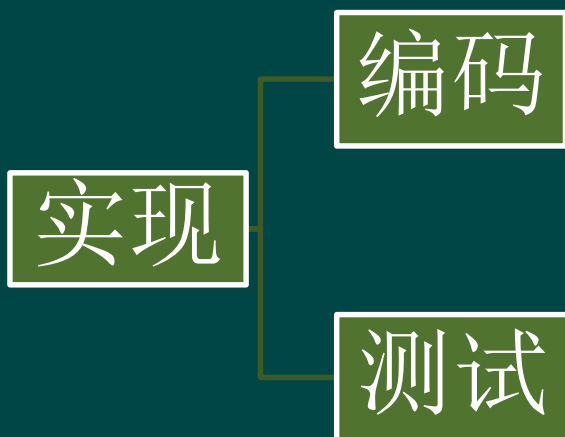


## 第7章 实现





## 第7章 实现



编码和测试统称为实现。

编码：把软件设计结果翻译成程序。

测试：检测程序并改正错误的过程。



# 主要内容

- 7.1 编码
- 7.2 软件测试基础
- 7.3 单元测试
- 7.4 集成测试
- 7.5 确认测试
- 7.6 白盒测试技术
- 7.7 黑盒测试技术
- 7.8 调试
- 7.9 软件可靠性

# 主要内容



7.1 编码

7.2 软件测试基础

7.3 单元测试

7.4 集成测试

7.5 确认测试

7.6 白盒测试技术

7.7 黑盒测试技术

7.8 调试

7.9 软件可靠性



## 7.1 编码

### 7.1.1 选择程序设计语言

- ◆ 计算机程序设计语言基本上可以分为两大类:
  - ◆ 1. 汇编语言;
  - ◆ 2. 高级语言。

## ◆从应用特点看，高级语言可分为：

### 选择适宜的程序设计语言的原因：

- 汇编语言编码需要把软件设计翻译成机器操作的序列，既困难又容易出差错，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；
- 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；
- 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

## ◆选择？

# 7.1 编码



## 理想标准:

- 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;
- 使编译程序能够尽可能多地**发现**程序中的错误;
- 应该有良好的**独立编译**机制。

## 实用标准:

- 系统用户的要求;
- 可以使用的**编译程序**;
- 可以得到的**软件工具**;
- 工程**规模**;
- 程序员的**知识**;
- 软件**可移植性**要求;
- 软件的**应用领域**。

类 型	可选语言
商用数据处理	COBOL
科学工程计算和模拟	FORTRAN PASCAL
实时软件	汇编语言 Ada
系统软件	汇编语言 C Ada
智能软件	LISP PROLOG



# 7.1 编码

## 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是指恰当的标识符、适当的注解和程序的视觉组织等。

- **标识符**：含义清晰、缩写规则一致、为名字加注解；
- **注解**：正确性，简要描述功能、主要算法、接口特点、重要数据以及开发简史或说明包含这段代码的必要性；
- **视觉组织**：适当的阶梯形式使程序的层次结构清晰明显。

清晰  
易读





## 2. 数据说明

数据说明的原则：

- 数据说明的次序应该标准化，**规范标准**
- 当多个变量名在一个语句中说明时，应该**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**文字**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

下述语句构造的原则有**三条**：

- 不要为了节省空间而把多个语句写在一行；
- 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，**简单明了**

## 4. 输入输出

在设计和编写程序时需考虑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；
- 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。

效率

(1) 时间

(2) 存储器效率

(3) 输入输出的效率

## 5. 效率

**效率**主要指处理机时间和存储器容量两个方面。

- 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；
- 效率是靠好设计来提高的；
- 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。



编码产生的源程序，应该正确可靠，简明清晰，而且应具有较高的效率。但是，**清晰和效率却常有矛盾**。

Weinberg 曾作过这样一个实验，他让5个程序员各自编写同一个程序，分别对他们提出了5种不同的编码要求。

结果表明，要求清晰好的程序一般效率比较低，而要求效率高的程序清晰度又不好。对于大多数模块，编码时应该把**简明清晰放在第一位**，如果个别模块要求特别高的效率，就应该把具体要求告诉程序员，以便做特殊的处理。

### Winberg 的程序实验结果

结果 名次 编码要求	评判 项目	清晰性		效率		开发 时间
		程序	输出	内存数	语句数	
程序可读性最佳		1-2	2	3	3	4
输出可读性最佳		1-2	1	5	5	2-3
占内存最小		4	4	1	2	5
语句数最少		5	3	2	1	2-3
开发时间最短		3	3	4	4	1

# ◆ 程序设计工具举例：Visual C++

运用 Visual C++ 开发工具需要掌握：

■ C++语言是在C语言的基础是扩展而成的，两种语言的基本语法和语义是相同。

C++中加入了面向对象程序设计（OOP）的特征：

- 封装性：通过“类”把属性和函数组合在一起。
- 继承性：派生类可从先前定义的基类中继承函数和属性。
- 多态性：一个函数名，由不同的对象解释执行，可得到不同的执行效果。

➤ Windows编程基础；

➤ API是Windows应用程序编程接口；

➤ Win32 API中，核心部分依靠三个主要组件提供Windows的大部分函数，这三个组件分别是：  
**USER32.DLL、GDI32.DLL、KERNEL32.DLL；**

➤ MFC相关知识；

➤ 即Microsoft基本类，封装了SDK（软件开发工具包）结构、功能及应用程序框架内部技术，提供了许多可以重用的类。

➤ Visual C++集成开发工具环境的使用；

➤ 一个win32程序由两大块组成：**程序代码**和**用户接口资源（\*.rc）**：菜单，对话框，图标，光标等；



# 主要内容

7.1 编码



7.2 软件测试基础

7.3 单元测试

7.4 集成测试

7.5 确认测试

7.6 白盒测试技术

7.7 黑盒测试技术

7.8 调试

7.9 软件可靠性



## 7.2 软件测试基础

### 7.2.1. 软件测试的目标

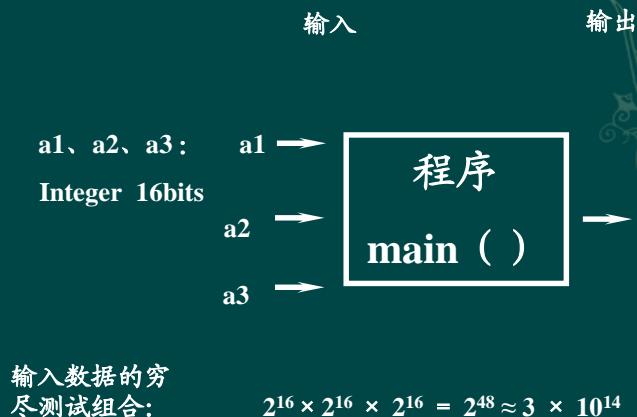
- G. Myers给出的关于测试的一些规则如下：
  - 测试是为了**发现程序中的错误**而执行程序的过程。
  - 好的测试方案是极可能**发现**迄今为止尚未发现的错误的测试方案。
  - 成功的测试是**发现**了至今为止尚未发现的错误的测试。
- 测试的**正确定义**是“为了发现程序中的错误而执行程序的过程”。
- 测试决**不能证明**程序是正确的。
- 即使经过了最严格的测试之后，仍然可能还有没被发现的错误潜藏在程序中。
- 另外，在综合测试阶段通常由**其他人员**组成测试小组来完成测试工作。

## 7.2 软件测试基础

### 7.2.2. 软件测试准则

主要的软件**测试准则**如下：

- 所有测试都应该能**追溯**到用户需求；
- 应该远在测试开始**之前**就制定出测试计划；
- 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）。
- 应该从“小规模”测试开始，并逐步进行“大规模”测试；
- 穷举测试是**不可能**的，比如：
- 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“最佳效果”是指有最大可能性发现错误的测试）。



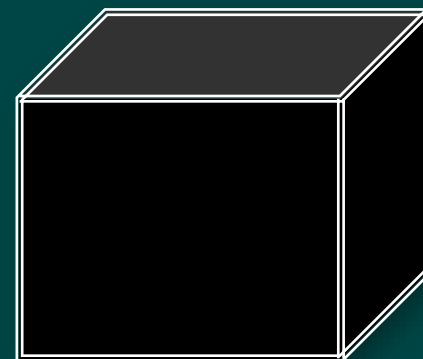


## 7.2 软件测试基础

### 7.2.3. 测试方法

- 测试任何产品都有两种方法：**黑盒测试**和**白盒测试**
- **黑盒测试**（又称功能测试）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。
- 黑盒测试是在**程序接口**进行的测试，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**。

程序接口

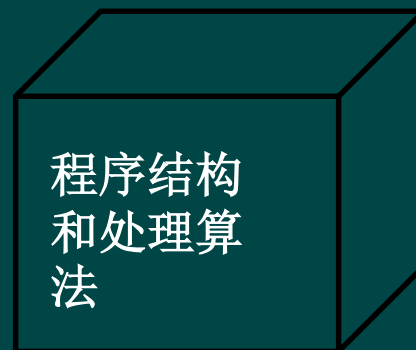






## 7.2 软件测试基础

- **白盒测试**（又称结构测试）是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。





## 7.2 软件测试基础

### 7.2.4. 测试步骤

第4条测试准则是：应该从“小规模”测试开始，并逐步进行“大规模”测试。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**模块测试、子系统测试、系统测试、验收测试和平行运行**等五个步骤组成。



## 7.2 软件测试基础

### 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实体来测试，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。



## 7.2 软件测试基础

### 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。

子系统测试和系统测试，都兼有检测和组装两重含义，通常称为集成测试。



## 7.2 软件测试基础

### 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。

## 7.2 软件测试基础

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。

### 5. 平行运行

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**。



## 7.2 软件测试基础——可能性

比较测试得出的实际结果和预期的结果，如果两者不一致则有可能：

——很可能是程序中有错误。

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，应该进一步仔细测试。

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。

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。

**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。



# 主要内容

7.1 编码

7.2 软件测试基础



7.3 单元测试

7.4 集成测试

7.5 确认测试

7.6 白盒测试技术

7.7 黑盒测试技术

7.8 调试

第7章 7.3 实现 软件可靠性 7.3 单元测试





## 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。
- 单元测试和编码属于软件过程的同一个阶段。
- 在源程序代码通过编译程序的语法检查后，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。
- 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。
- 究竟测什么？
  - ——模块接口，局部数据结构，重要的执行通路，出错处理通路，边界条件

## 7.3 单元测试

### 7.3.1. 测试重点

在单元测试期间着重从以下5个方面对模块进行测试。

#### 1. 模块接口

对模块接口进行测试时主要检查以下几个方面：

- 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。

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。



## 7.3 单元测试

### 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。

## 7.3 单元测试



### 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。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。评价出错处理通路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。



## 7.3 单元测试

### 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 单元测试

### 7.3.2. 代码审查

**代码检查**是指由审查小组正式对源程序进行人工测试。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%~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。**



## 7.3 单元测试

- 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。
- 审查会上需要对照程序设计**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(注：**审查小组的任务是发现错误而不是改正错误**)。
- 审查会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。



## 7.3 单元测试

- 测试方案在代码审查中起一种**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，即：采用代码审查的方法可以**减少系统验证的总工作量**。
- 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 单元测试

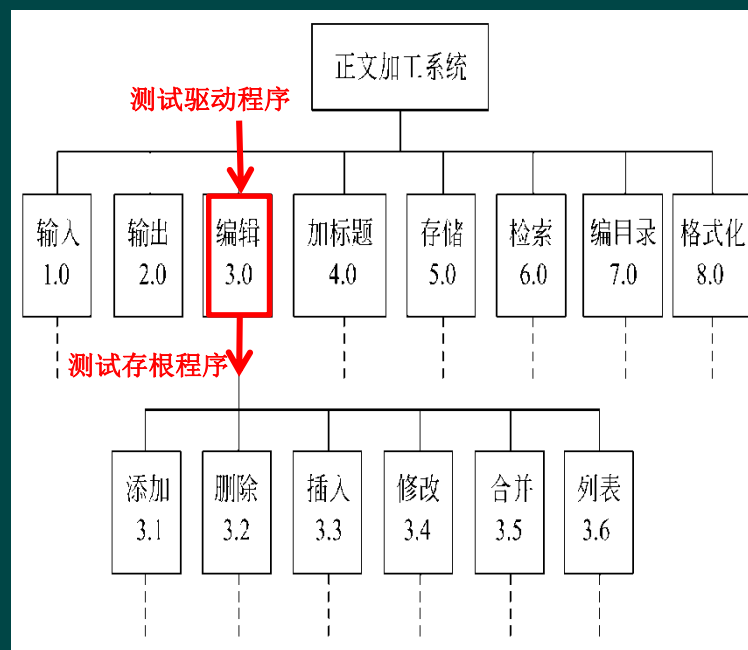
### 7.3.3. 计算机测试

- 模块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。
- 驱动程序是一个“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，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。

## 7.3 单元测试

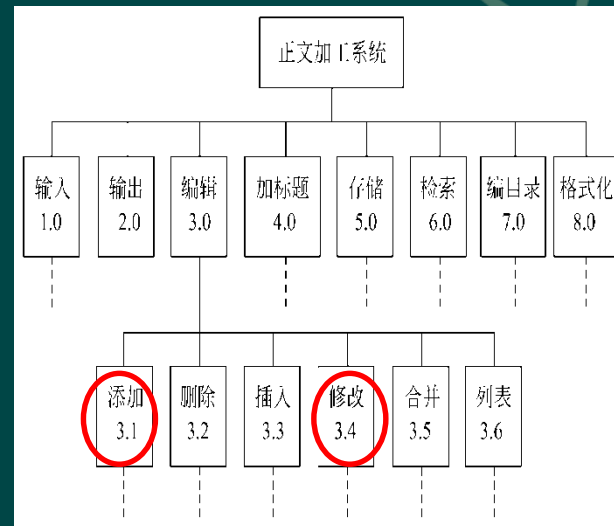
右图是一个正文加工系统的部分层次图，假定要测试编号为3.0的关键模块——正文编辑模块。正文编辑模块不是一个独立的程序，需要有一个**测试驱动程序**来调用它。

这个驱动程序说明必要的变量，接收测试数据——字符串，设置正文编辑模块的编辑功能。并且需要有**存根程序**简化地模拟正文编辑模块的下层模块来完成具体的编辑功能。



## 7.3 单元测试

测试时，设置修改 (CHANGE) 和添加 (APPEND) 两种编辑功能，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。



TEST STUB (\*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

TEST DRIVER (\*驱动程序\*)

说明长度为2500个字符的一个缓冲区;

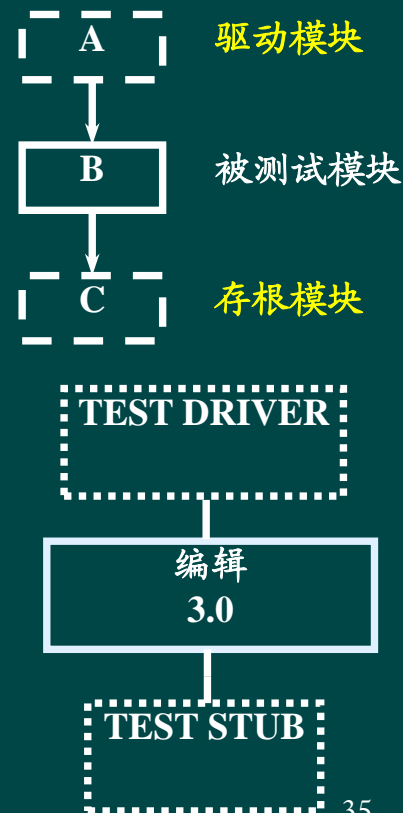
把CFUNCT置为希望测试的状态;

输入字符串;

调用正文编辑块;

停止或再次初启;

END TEST DRIVER





# 主要内容

- 7.1 编码
- 7.2 软件测试基础
- 7.3 单元测试
- ▶ 7.4 集成测试
- 7.5 确认测试
- 7.6 白盒测试技术
- 7.7 黑盒测试技术
- 7.8 调试
- 7.9 软件可靠性



## 7.4 集成测试

集成测试是测试和组装软件的系统化技术。

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。

与接口有关的问题：

- 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。



## 7.4 集成测试

**非渐增式测试**把所有模块放在一起，作为一个整体来测试。测试时会遇到许多的错误，改正错误非常困难，因为在庞大的程序中想要诊断定位一个错误非常困难，而且改正一个错误之后，马上又会遇到新的错误，这个过程会继续下去，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 集成测试

### 7.4.1. 自顶向下集成

- **自顶向下集成方法**是从主控制模块开始，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（及最终附属）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。
- **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。
- **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。

## 7.4 集成测试

如右图，

### 使用深度优先的结合方法：

选取左通路，首先结合模块M1\, M2和M5；

其次，M8或M6(如果为了使M2具有适当功能需要M6)将被结合进来。

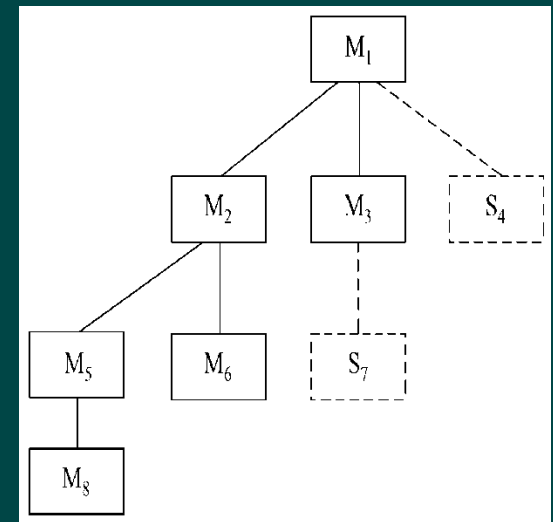
然后构造中央的和右侧的控制通路。

### 使用宽度优先的结合方法：

首先结合模块M2\, M3和M4(代替存根程序S4)，

然后结合下一个控制层次中的模块M5\, M6和M7；

如此继续进行下去，直到所有模块都被结合进来为止。







## 7.4 集成测试

模块结合进软件结构的具体过程由下述4个步骤完成：

- ① 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；
- ② 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）；
- ③ 在结合进一个模块的同时进行测试；
- ④ 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②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。



## 7.4 集成测试

- **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。
- 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**：①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；②从层次系统的底部向上组装软件。
- **自顶向下**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。

## 7.4 集成测试



### 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。

- ① 把低层模块组合成实现某个特定的软件子功能的族；
- ② 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；
- ③ 对由模块组成的子功能族进行测试；
- ④ 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②～④步实质上构成了一个循环。

## 7.4 集成测试

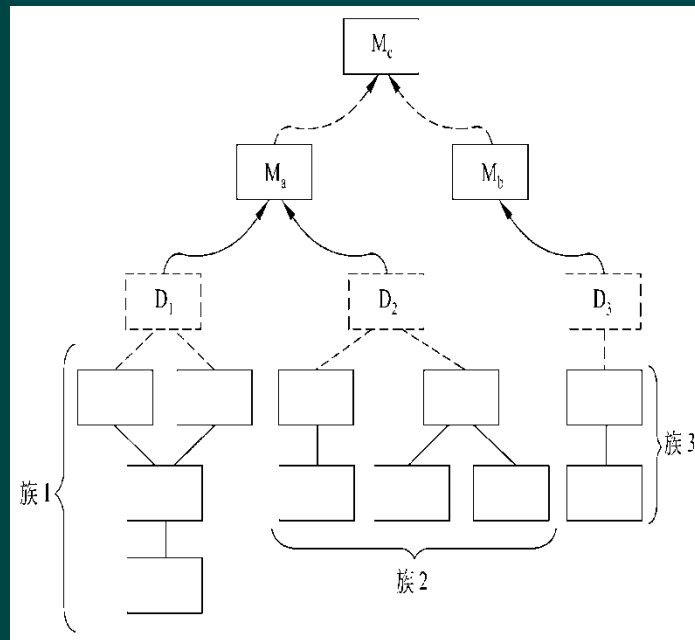


右图描绘了自底向上的结合过程。

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 $M_a$ 和 $M_b$ 这两个模块都与模块 $M_c$ 结合起来。随着结合向上移动，对测试驱动程序的需要减少了。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 集成测试

### 7.4.3. 不同集成测试策略的比较

- **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。
- **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。**

## 7.4 集成测试

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。



# 主要内容

7.1 编码

7.2 软件测试基础

7.3 单元测试

7.4 集成测试



7.5 确认测试

7.6 白盒测试技术

7.7 黑盒测试技术

7.8 调试

7.9 软件可靠性





## 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。
- 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；**确认**指的是为了保证软件确实满足了用户需求而进行的一系列活动。
- **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 确认测试

## 7.5.1. 确认测试的范围

确认测试必须有用户积极参与，或以用户为主进行。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。

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 确认测试

### 7.5.2. 软件配置复查

**软件配置复查**是确认测试的一个重要内容。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 确认测试

### 7.5.3. Alpha和Beta测试

- 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），那么绝大多数软件开发商都使用被称为**Alpha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。
- **Alpha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。
- **Alpha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Alpha测试不同，开发者通常不在Beta测试的现场。
- **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（真实的或想象的），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。



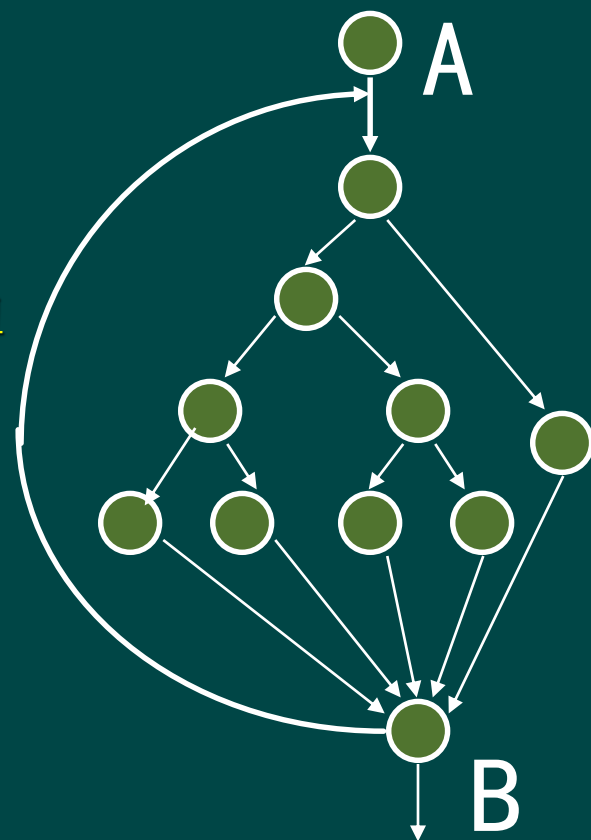
# 主要内容

- 7.1 编码
- 7.2 软件测试基础
- 7.3 单元测试
- 7.4 集成测试
- 7.5 确认测试
- ▶ 7.6 白盒测试技术
- 7.7 黑盒测试技术
- 7.8 调试
- 7.9 软件可靠性

## 7.6 白盒测试技术

- 通常把测试数据和预期的输出结果称为**测试用例（实际的数据）**。
- 不同的测试数据发现程序错误的**能力差别很大**，因为不可能进行穷尽的测试，为了提高测试效率降低测试成本，应该选用**少量高效**的测试数据，做到尽可能完备。
- 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**。

白盒穷尽测试？



含5个分支, 循环次数 $\leq 20$ , 从A到B的可能路径

$$= 5^1 + 5^2 + \dots + 5^9 + 5^{20} \approx 10^{14}$$

执行时间：设测试一次需2ms，穷举测试需5亿年

## 7.6 白盒测试技术



### 7.6.1. 逻辑覆盖

**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。

- (1) 语句覆盖
- (2) 判定覆盖
- (3) 条件覆盖
- (4) 判定/条件覆盖
- (5) 条件组合覆盖
- (6) 路径覆盖
- (7) 点覆盖
- (8) 边覆盖

## 1. 语句覆盖

语句覆盖的含义是，选择足够的测试数据，使被测程序中每个语句至少执行一次。

全部路径：

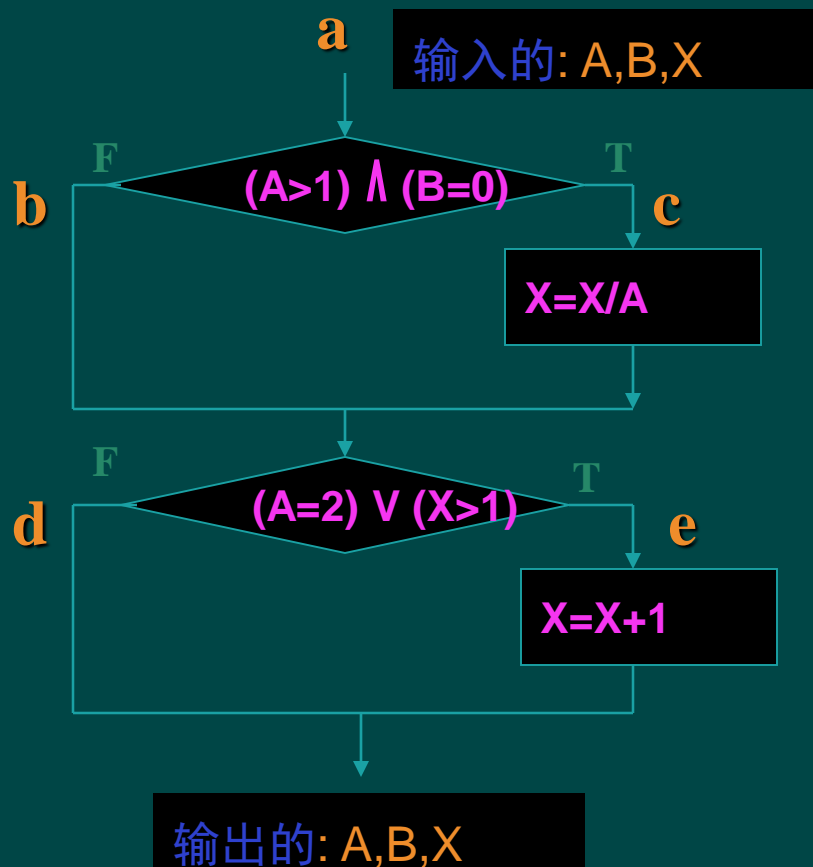
ace (L1)

abd (L2)

abe (L3)

acd (L4)

满足语句覆盖的路径



满足语句覆盖的测试用例

输入2, 0, 4

输出2, 0, 3

a -- c -- e

L1

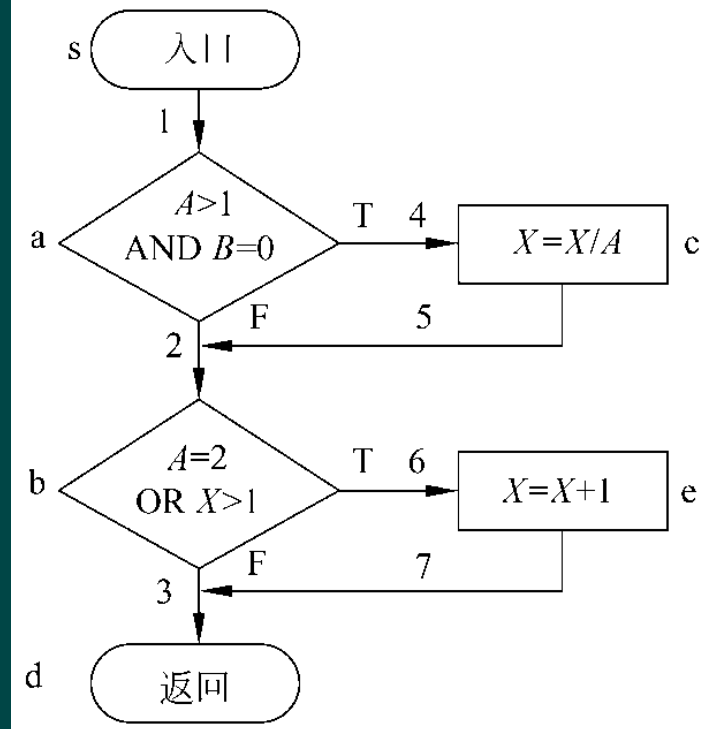


## 7.6 白盒测试技术

### 1. 语句覆盖

语句覆盖对程序的逻辑覆盖很少，在第一个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。

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**。



为了执行sacbed路径，以测试每个语句，只需两个判定表达式  $(A>1) \text{ AND } (B=0)$  和  $(A=2) \text{ OR } (X>1)$  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。

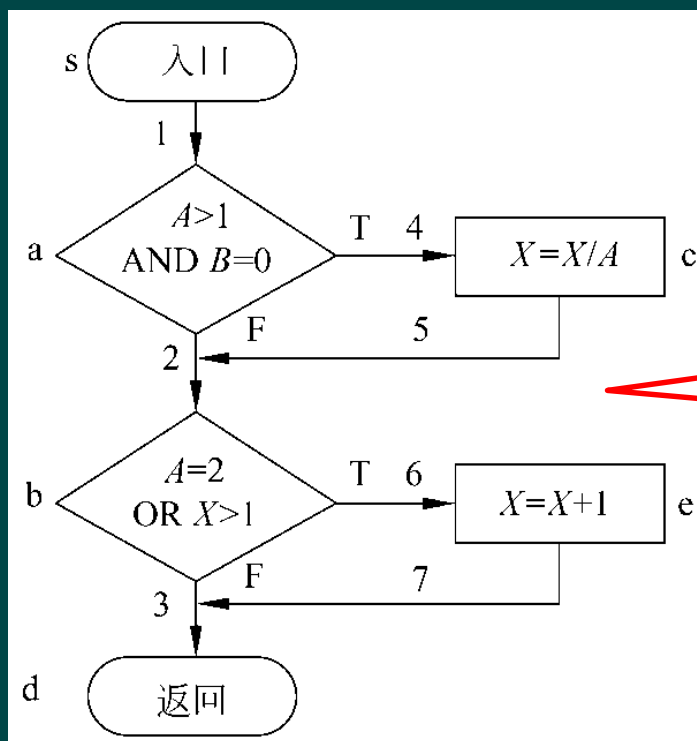
综上所述，可以看出语句覆盖是很弱的逻辑覆盖标准。



## 7.6 白盒测试技术

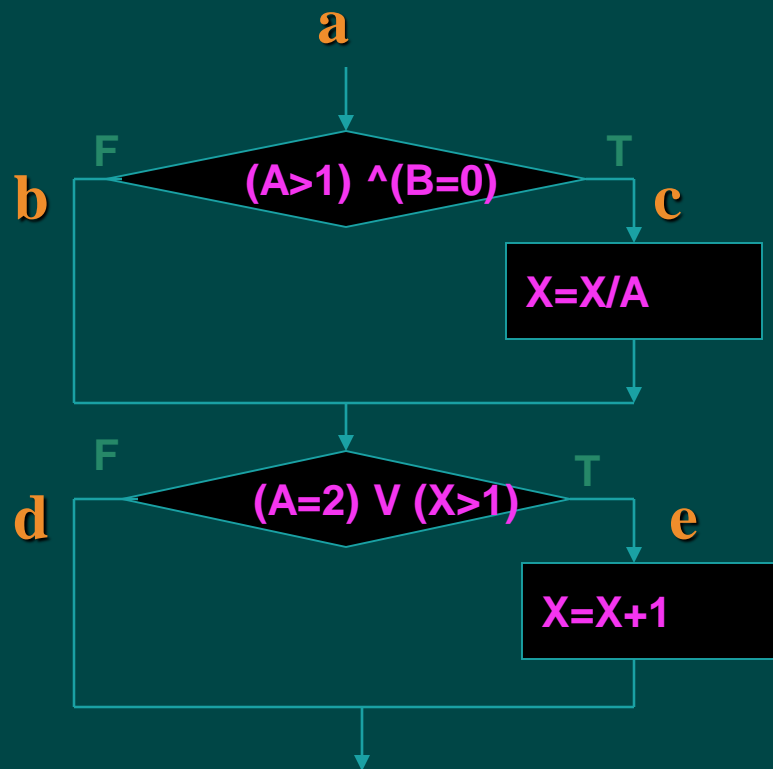
### 2. 判定覆盖

**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（真分支，假分支）都至少执行一次。



思考：要完成该程序的判定覆盖，需要几组测试用例？

## 2. 判定覆盖（一种情况）



a	--	c	--	e	L1
a	--	b	--	d	L2

取“真”分支  
测试用例如下

A,B,X输入: 2, 0, 4

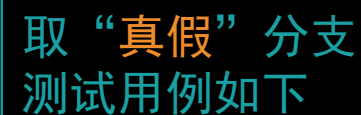
A,B,X结果: 2, 0, 3

A,B,X输入: 1, 1, 1

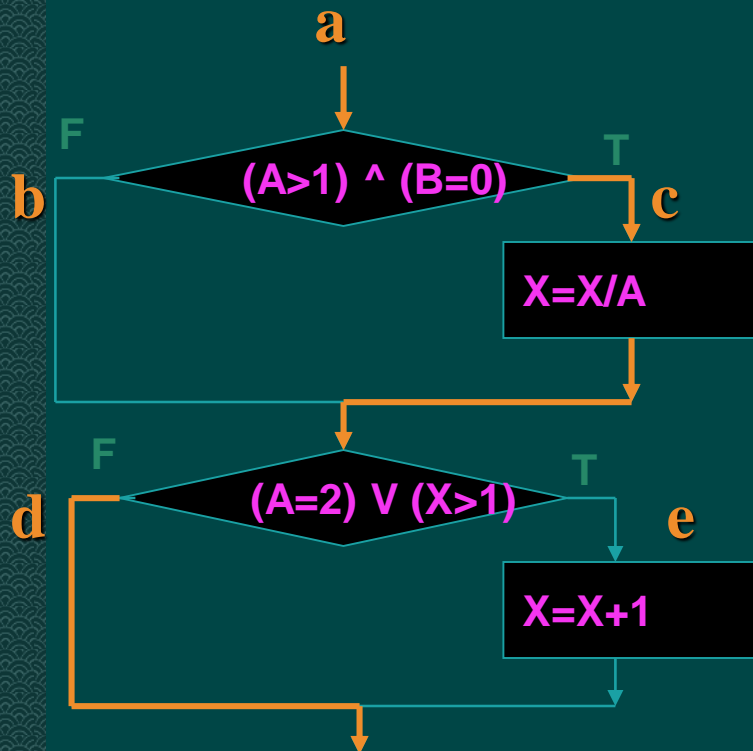
A,B,X结果: 1, 1, 1

取“假”分支  
测试用例

问题：这种情况是判定覆盖的唯一方式吗？


$$\begin{array}{ccc} 2, & 1, & 1 \\ 2, & 1, & 2 \end{array}$$

## 2. 判定覆盖（另外一种情况）



取“真假”分支  
测试用例如下

3, 0, 3
3, 0, 1

a -- c -- d L4

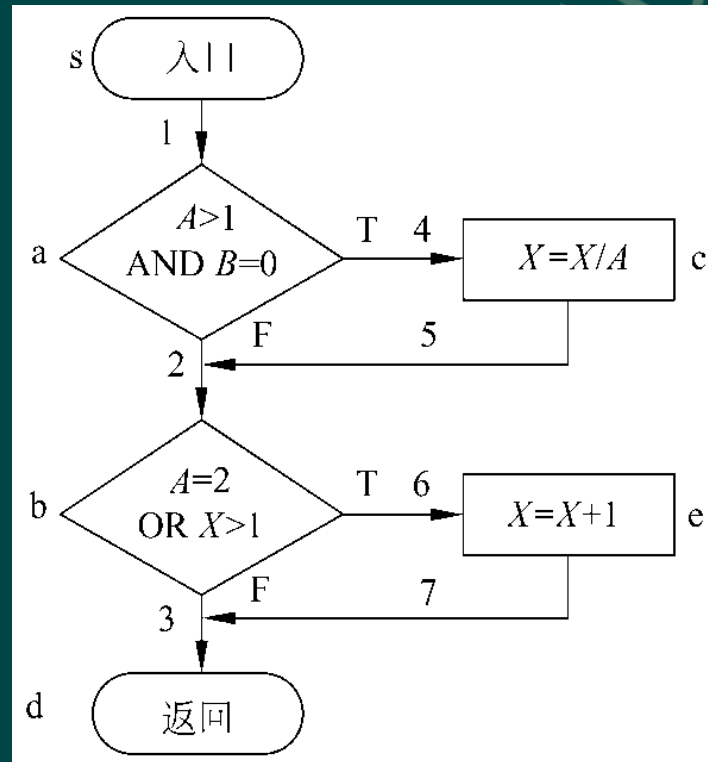
- 将L4的判定路径的测试用例数据定为输入 **(3,0,3)**，输出 **(3,0,1)** 则它将覆盖 **L4 a-c-d**
- 所以，测试用例的取法 **并不是唯一的**。
- **注意**有例外情形：例如，若把上图中第二个判断中的条件  $X > 1$  错写成  $X < 1$  那么再利用上面的两组测试用例，仍能得到同样的结果。
- 这表明，只要判定覆盖，还不能保证一定能查出在判断的条件中存在的错误。
- 因此，还需要**更强的**逻辑覆盖准则来检验判断内部的条件。

## 7.6 白盒测试技术

### 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所有可能的结果。想要做到条件覆盖？

上例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，应该选取测试数据满足下面的要求（**注意一定要编号**）



在a点有下述各种结果出现：

① $A > 1$ , ② $A \leq 1$ , ③ $B = 0$ , ④ $B \neq 0$ ;

在b点有下述各种结果出现：

⑤ $A = 2$ , ⑥ $A \neq 2$ , ⑦ $X > 1$ , ⑧ $X \leq 1$ ;

测试用例？

寻找**合适的**（**尽可能少的**）测试用例，使之能够覆盖情况①~~⑧

## 7.6 白盒测试技术

### 3. 条件覆盖

只需要使用下面两组测试数据就可以达到上述覆盖标准:

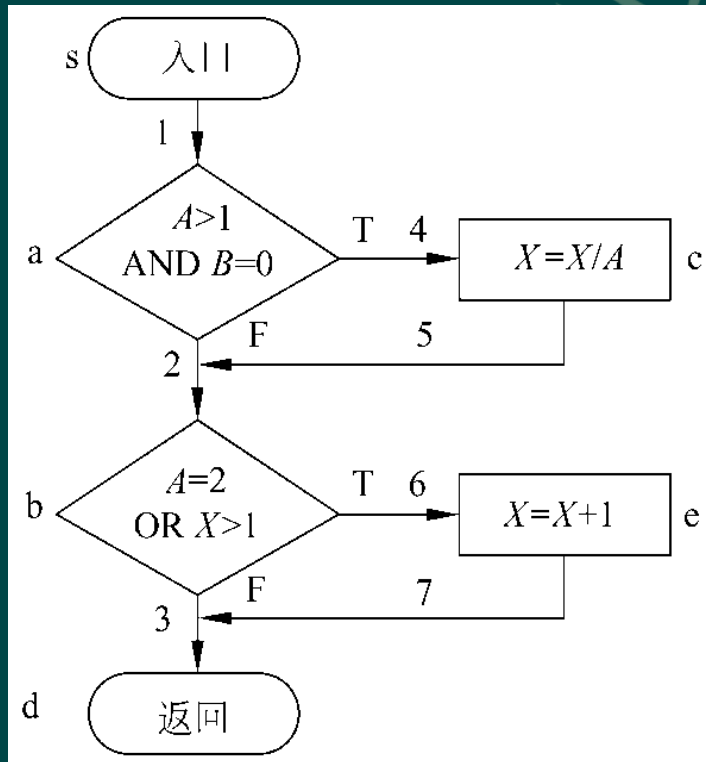
- ①  $A=2, B=0, X=4$  (满足 $A>1, B=0, A=2$ 和 $X>1$  (覆盖1、3、5和7), 执行路径sacbed)
- ②  $A=1, B=1, X=1$  (满足 $A\leq 1, B\neq 0, A\neq 2$ 和 $X\leq 1$  (覆盖2、4、6和8), 执行路径sabdd)

**问题: 满足条件覆盖是否一定满足判定覆盖?**

条件覆盖通常比判定覆盖强, 但满足条件覆盖的测试数据不一定满足判定覆盖。

例如, 上面两组测试数据也同时满足判定覆盖标准。但是, 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):

- ①  $A=2, B=0, X=1$  (满足 $A>1, B=0, A=2$ 和 $X\leq 1$ , 覆盖1、3、5和8, 执行路径sacbed左右)
- ②  $A=1, B=1, X=2$  (满足 $A\leq 1, B\neq 0, A\neq 2$ 和 $X>1$ , 覆盖2、4、6和7, 执行路径sabed左右)



## 7.6 白盒测试技术

### 4. 判定/条件覆盖

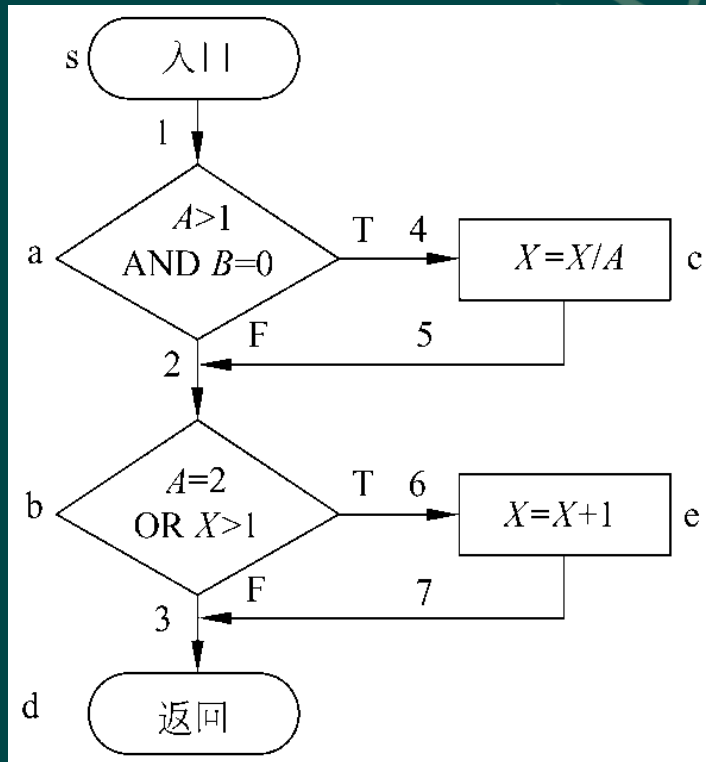
**判定/条件覆盖**是一种能**同时**满足判定覆盖和条件覆盖的逻辑覆盖，它的含义是，选取足够多的测试数据，使得判定表达式中的**每个条件**都取到各种可能的值，而且**每个判定表达式**也都取到所有可能的结果。**测试用例一定比判定或者条件覆盖多吗？**

对于上例而言，下述两组测试数据满足判定/条件覆盖标准：

- ①  $A=2, B=0, X=4$
- ②  $A=1, B=1, X=1$

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。

**问题：更强的覆盖？**



# 7.6 白盒测试技术

## 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。

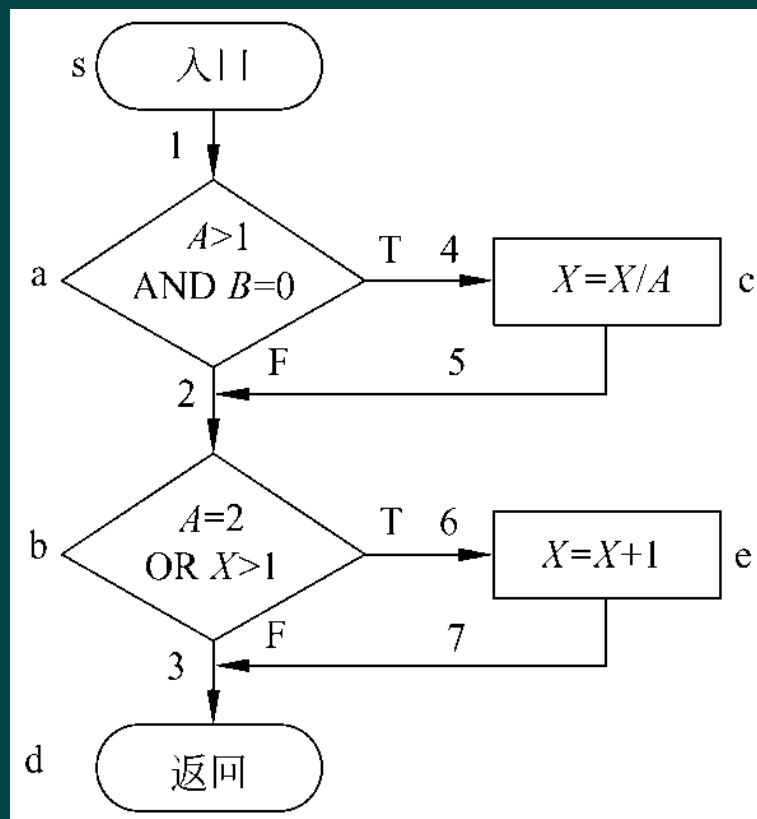
对于条件  $A > 1$ ,  $B = 0$ ,  $A = 2$ ,  $X > 1$

**所有的可能性** (  $A > 1$ ,  $A \leq 1$ ,  $B = 0$ ,  $A > 1, B \neq 0$  和  $A = 2$ ,  $A \neq 2$ ,  $X > 1$ ,  $X \leq 1$ ) :

共有8种可能的**条件组合**，它们分别是：

- |                       |                          |
|-----------------------|--------------------------|
| (1) $A > 1, B = 0$    | (2) $A > 1, B \neq 0$    |
| (3) $A \leq 1, B = 0$ | (4) $A \leq 1, B \neq 0$ |
| (5) $A = 2, X > 1$    | (6) $A = 2, X \leq 1$    |
| (7) $A \neq 2, X > 1$ | (8) $A \neq 2, X \leq 1$ |

测试用例？





## 7.6 白盒测试技术

### 5. 条件组合覆盖

下面的4组测试数据使上面列出的8种条件组合每种至少出现一次：

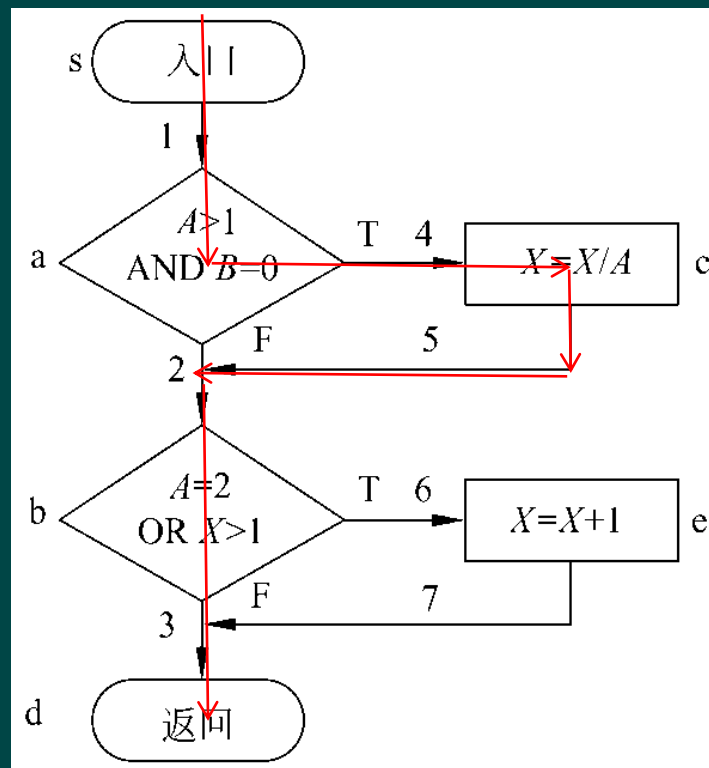
- ①  $A=2, B=0, X=4$  (覆盖 (1) 和 (5) , 执行路径sacbed)
- ②  $A=2, B=1, X=1$  (覆盖 (2) 和 (6) , 执行路径sabed)
- ③  $A=1, B=0, X=2$  (覆盖 (3) 和 (7) , 执行路径sabed)
- ④  $A=1, B=1, X=1$  (覆盖 (4) 和 (8) , 执行路径sabd)

是否满足之前的覆盖？

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。

**问题：是否还有其他覆盖方式？**

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。

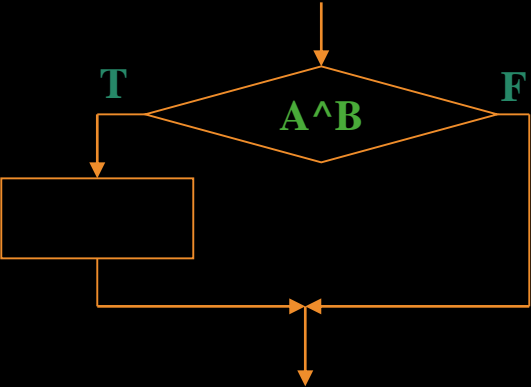
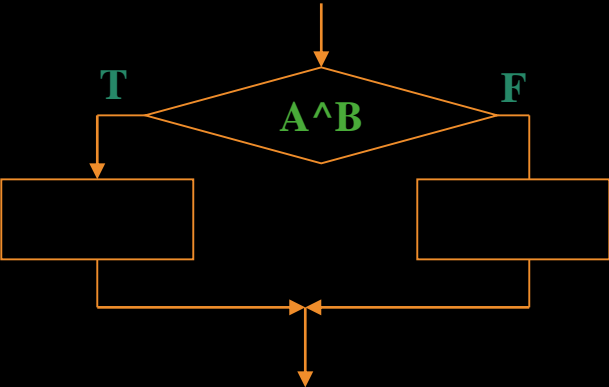




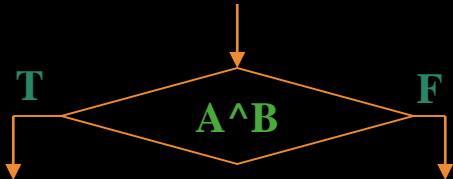
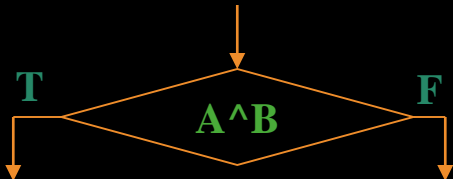
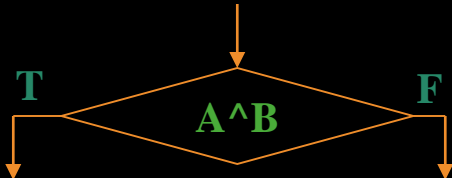
## 总结：逻辑覆盖测试的5种标准

发现错误的 能力	标 准	含 义
1(弱)	语句覆盖	每条语句至少执行一次
2	判定覆盖	每一判定的每个分支至少执行一次
3	条件覆盖	每一判定中的每个条件，分别按“真”、“假”至少各执行一次
4	判定/条件覆盖	同时满足判定覆盖和条件覆盖的要求
5 (强)	条件组合覆盖	求出判定中所有条件的各种可能组合值，每一可能的条件组合至少执行一次



覆盖标准	程序结构举例	测试用例应满足的条件
语句覆盖		$A \wedge B = .T.$
判定覆盖		$A \wedge B = .T.$ $A \wedge B = .F.$



覆盖标准	程序结构举例	测试用例应满足的条件
条件覆盖		$A=.T.$ $A=.F.$ $B=.T.$ $B=.F.$
判定/条件覆盖		$A^B=.T.$ , $A^B=.F.$ $A=.T.$ $A=.F.$ $B=.T.$ $B=.F.$
条件组合覆盖		$A=.T.$ $\wedge$ $B=.T.$ $A=.T.$ $\wedge$ $B=.F.$ $A=.F.$ $\wedge$ $B=.T.$ $A=.F.$ $\wedge$ $B=.F.$

## 7.6 白盒测试技术

### 6. 点覆盖、边覆盖和路径覆盖

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。

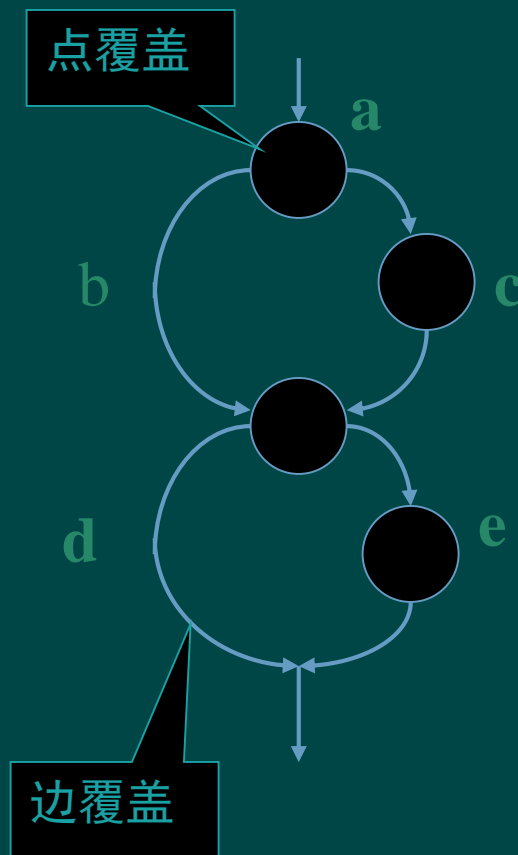
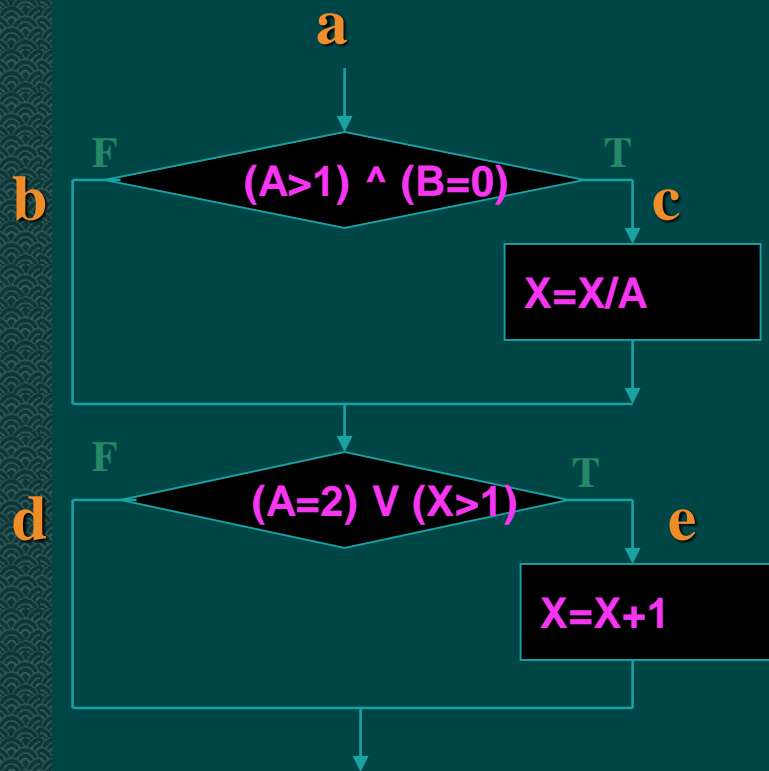
图论中**点覆盖**的定义如下：如果连通图 $G$ 的子图 $G'$ 是连通的，而且包含 $G$ 的所有结点，则称 $G'$ 是 $G$ 的点覆盖。

在前面已经讲述了从**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**。

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**。

**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)。

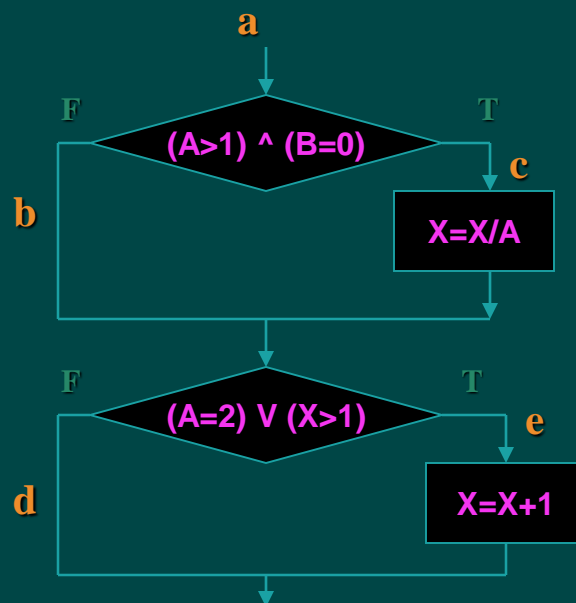
从上述分析情况来看，路径测试的特征是，它不但包含了语句覆盖同时也满足了判定覆盖（在程序图上分别称为点覆盖与边覆盖）。换句话说，只要满足了路径覆盖，也就必然满足语句和判定覆盖。



点覆盖

边覆盖

a-c-e L1  
a-b-d L2  
a-b-e L3  
a-c-d L4



判断	条件	取真值	取假值
判断 (一)	A>1	T1	$\overline{T1}$
	B=0	T2	$\overline{T2}$
判断 (二)	A=2	T3	$\overline{T3}$
	X>1	T4	$\overline{T4}$

路径测试可选取的 **测试用例** 如下表

测试用例	通过路径	条件取值
(2,0,4),(2,0,3)	ace <b>L1</b>	T1 T2 T3 T4
(1,1,1),(1,1,1)	abd <b>L2</b>	$\overline{T1}$ $\overline{T2}$ $\overline{T3}$ $\overline{T4}$
(1,1,2),(1,1,3)	abe <b>L3</b>	$\overline{T1}$ $\overline{T2}$ $\overline{T3}$ T4
(3,0,3),(3,0,1)	acd <b>L4</b>	T1 T2 $\overline{T3}$ $\overline{T4}$



对于程序图，可以选取如下测试用例完成路径覆盖：

I. **A=1, B=1, X=1**

(执行路径1—2—3)

II. **A=1, B=1, X=2**

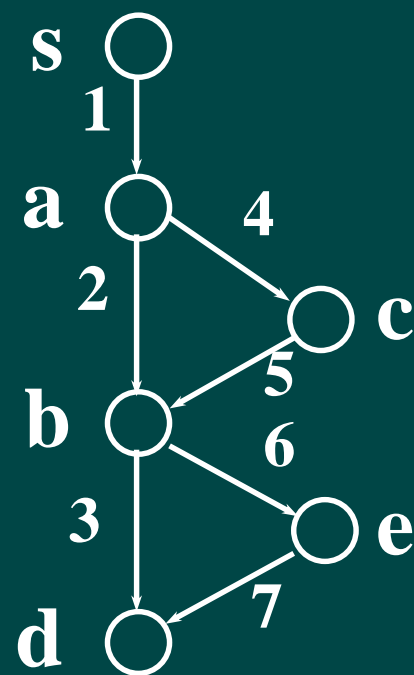
(执行路径1—2—6—7)

III. **A=3, B=0, X=1**

(执行路径1—4—5—3)

IV. **A=2, B=0, X=4**

(执行路径1—4—5—6—7)



程序图



## 更强的测试数据：满足路径覆盖和条件组合覆盖

路径: sacbd

◇ 1)  $A=3, B=0, X=1$

◇ 2)  $A=2, B=0, X=4$

◇ 3)  $A=2, B=1, X=1$

◇ 4)  $A=1, B=0, X=2$

◇ 5)  $A=1, B=1, X=1$

2) - 5) 满足条件组合覆盖, 执行  
路径分别是: sacbed、sabed、sabed、  
sabd

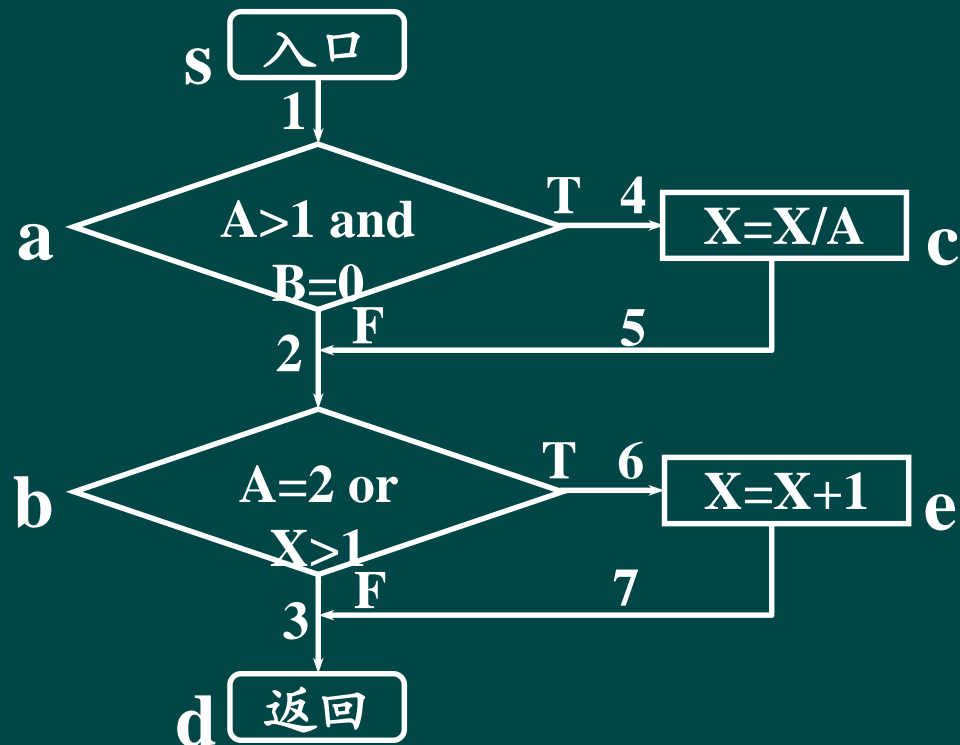
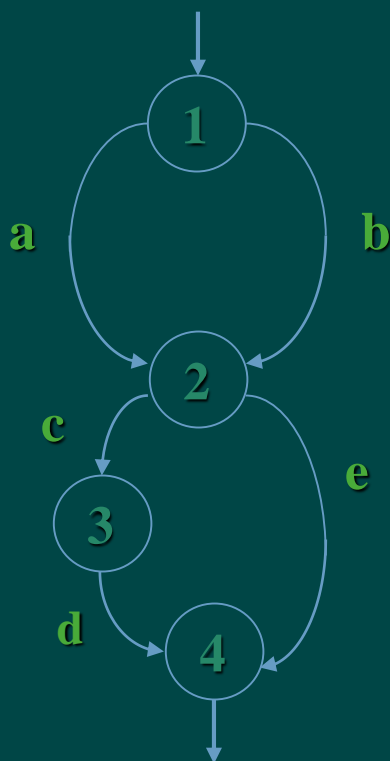


图 被测试模块的流程图



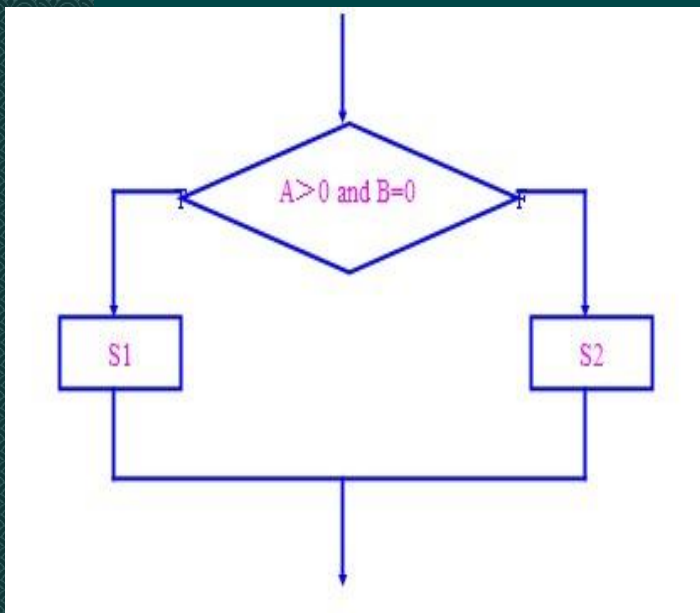
根据左侧给出的程序图，请填写下列表



测试路径	覆盖结点/边	覆盖标准
<b>a-c-d</b>	①②③④	点覆盖
<b>a-c-d, b-e</b>	<b>a,b,c,d,e</b>	边覆盖
<b>a-e-d, b-e</b> <b>a-e, b-c-d</b>	①②③④ <b>a,b,c,d,e</b>	路径覆盖

## 课堂练习——白盒测试

如下图显示某程序的逻辑结构。按照样例填写下表，试为它设计足够的测试用例，分别实现对程序的语句覆盖、判定覆盖、条件覆盖和条件组合覆盖。



覆盖种类	需满足的条件	测试数据	期望结果
语句覆盖	样例 $A > 0, B = 0$	$A = 2, B = 0$	执行S1
	.....	.....	.....
判定覆盖	.....	.....	.....
条件覆盖	.....	.....	.....
条件组合覆盖	.....	.....	.....

## 7.6 白盒测试技术

### 7.6.2. 控制结构测试\*

**基本路径测试**是Tom McCabe提出的一种白盒测试技术。使用基本路径测试设计测试用例时，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。

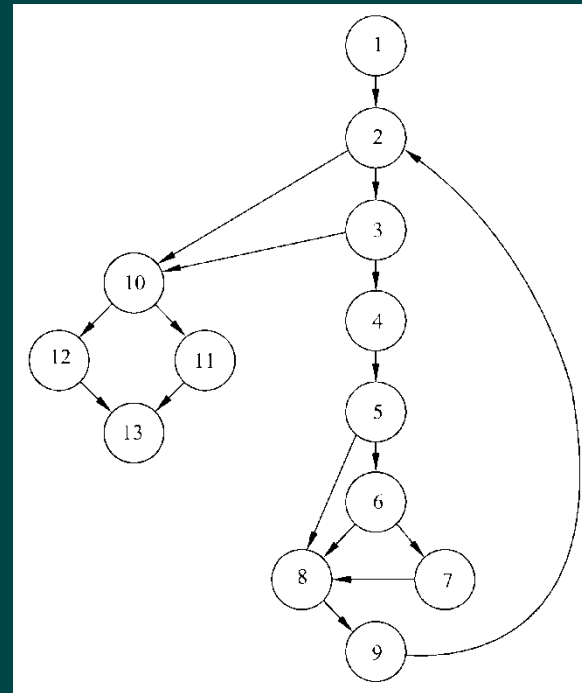
#### ① 根据过程设计结果画出相应的流图。

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，首先画出下图所示的流图。注意，为了正确地画出流图，这里把被映射为流图结点的PDL语句编了序号。

# 7.6 白盒测试技术

## 1. 基本路径测试

```
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 by 1;
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
```



## 7.6 白盒测试技术

### 1. 基本路径测试

#### ② 计算流图的环形复杂度。

环形复杂度定量度量程序的逻辑复杂性。使用前面讲述的3种方法之一计算环形复杂度。经计算，流图的环形复杂度为6。

#### ③ 确定线性独立路径的基本集合。

**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，即独立路径至少包含一条在定义该路径之前不曾用过的边。

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。

上述程序的环形复杂度为6，因此共有6条独立路径。

路径1: 1-2-10-11-13

路径2: 1-2-10-12-13

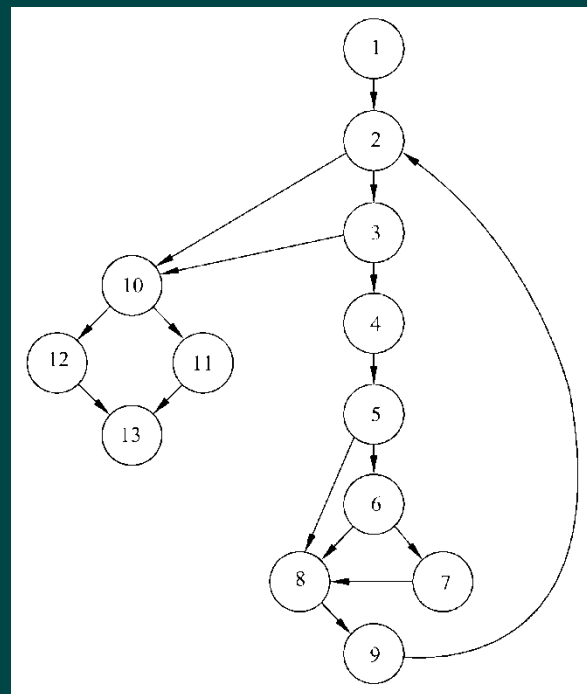
路径3: 1-2-3-10-11-13

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-...

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。



#### ④ 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各个判定结点的条件。  
测试第③步得出的基本集合的测试用例如下。

路径1的测试用例： 1-2-10-11-13

value [k] =有效输入值, 其中 $k < i$  (i的定义在下面)

value [i] =-999, 其中 $2 \leq i \leq 100$

预期结果：基于k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或5或6的一部分来测试**

路径2的测试用例： 1-2-10-12-13

value [1] =-999

预期结果： average=-999, 其他都保持初始值

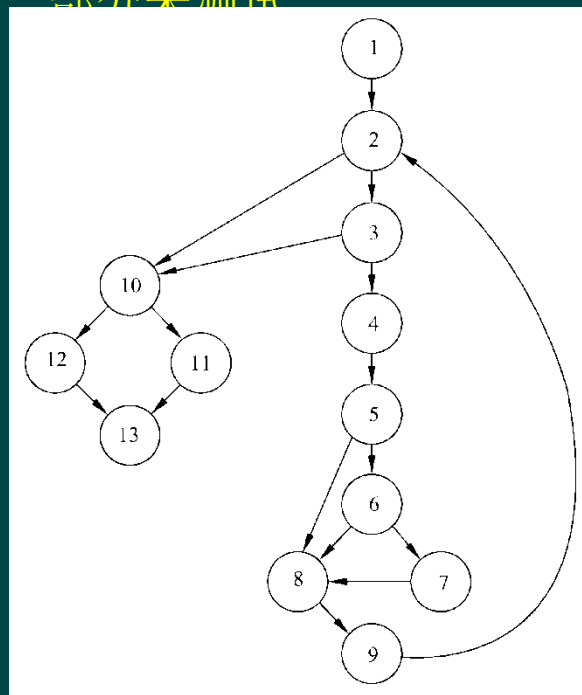
路径3的测试用例： 1-2-3-10-11-13

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无法独立测试，必须作为路径4或5或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k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k的正确平均值和总数

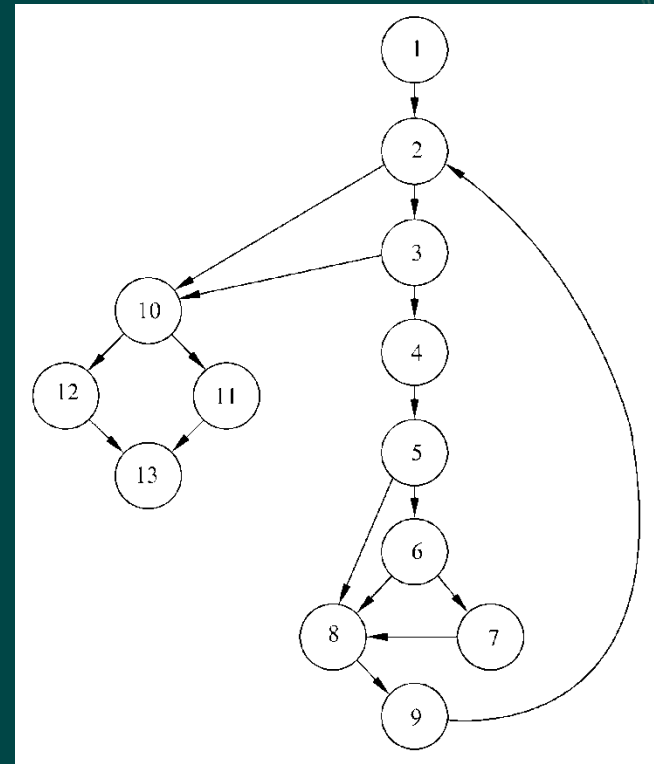
路径6的测试用例：

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。

**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，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。





# 7.6 白盒测试技术

## 2. 条件测试\*\*

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 ( $\neg$ ) 算符。关系表达式的形式如下：

$$E1 < \text{关系算符} > E2$$

其中，E1和E2是算术表达式，而<关系算符>是下列算符之一：<，≤，=，≠，>或≥。布尔算符有OR (|)，AND (&) 和NOT ( $\neg$ )。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，条件错误的类型有：布尔算符错、布尔变量错、布尔括弧错、关系算符错、算术表达式错。

# 7.6 白盒测试技术

## 2. 条件测试\*\*

条件测试方法着重测试程序中的每个条件。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。

在分支测试、域测试等条件测试技术的基础上，K. C. Tai提出了一种被称为**BR0(branch and relational 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BR0测试保证能发现该条件中的分支错和关系算符错。

## 7.6 白盒测试技术

### 2. 条件测试\*\*

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 $(D_1, D_2, \dots, D_n)$ ，其中 $D_i (0 < i \leq n)$ 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：C1: B1 & B2**

其中，B1和B2是布尔变量。C1的条件约束形式为 $(D_1, D_2)$ ，其中D1和D2中的每一个都是t或f。值 $(t, f)$ 是C1的一个条件约束，并由使B1值为真B2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C1的执行所覆盖。如果C1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C1失败。



## 作为第二个例子，考虑下列条件

C2:  $B1 \ \& \ (E3=E4)$

其中， $B1$ 是布尔变量， $E3$ 和 $E4$ 是算术表达式。C2的条件约束形式为 $(D1, D2)$ ，其中 $D1$ 是t或f， $D2$ 是 $>$ ， $=$ 或 $<$ 。除了C2的第二个简单条件是关系表达式之外，C2和C1相同，因此，可以通过修改C1的约束集 $\{(t, t), (f, t), (t, f)\}$ 得出C2的约束集。

注意，对于 $(E3=E4)$ 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C2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C2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

C3:  $(E1>E2) \ \& \ (E3=E4)$

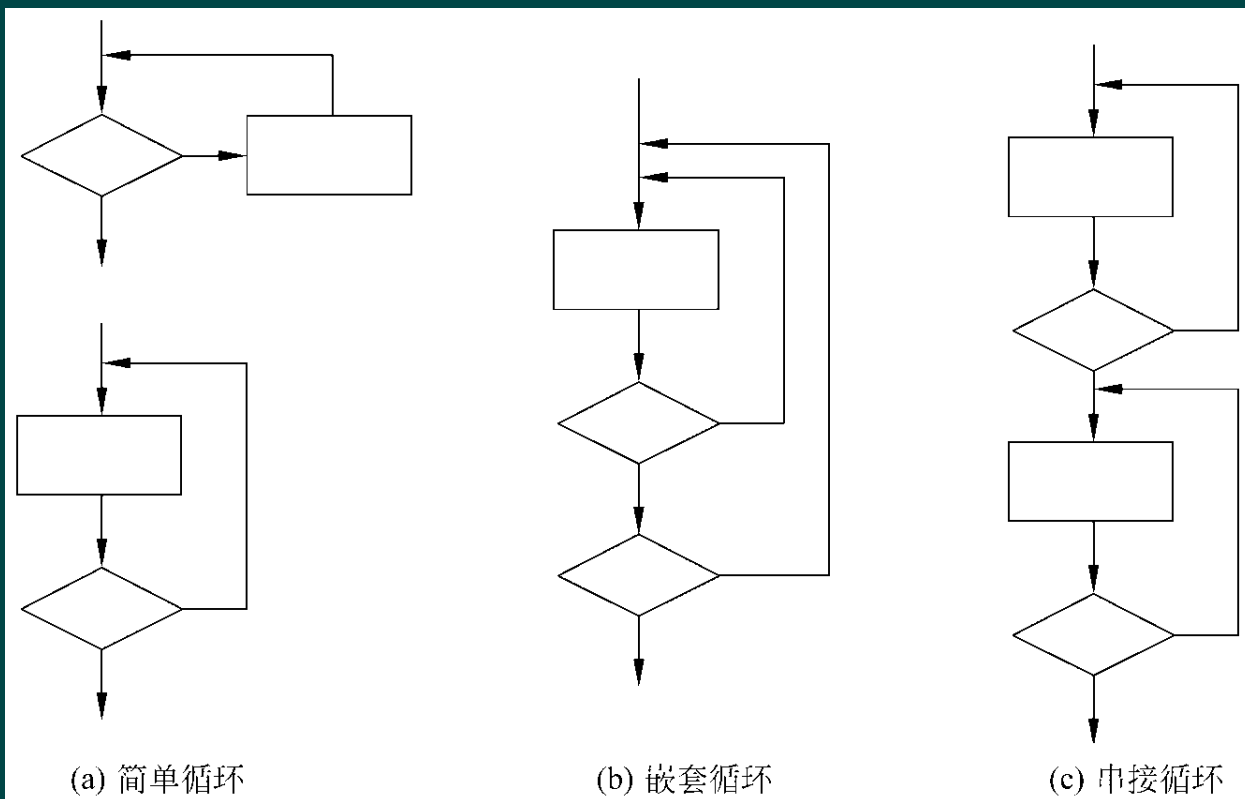
其中， $E1$ 、 $E2$ 、 $E3$ 和 $E4$ 是算术表达式。C3的条件约束形式为 $(D1, D2)$ ，而 $D1$ 和 $D2$ 的每一个都是 $>$ ， $=$ 或 $<$ 。除了C3的第一个简单条件是关系表达式之外，C3和C2相同，因此，可以通过修改C2的约束集得到C3的约束集，结果为：

$\{(>, =), (=, =), (<, =), (>, <), (>, >)\}$ 覆盖上述条件约束集的测试，保证可以发现C3中关系算符的错误。

## 7.6 白盒测试技术

### 3. 循环测试

**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。



## 7.6 白盒测试技术



### 3. 循环测试

#### (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 $m < n-1$ 。
- 通过循环 $n-1, n, n+1$ 次。

#### (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，测试数就会随嵌套层数的增加按几何级数增长，B. Beizer提出了一种能减少测试数的方法。跳过循环。

- 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，建议使用测试嵌套循环的方法来测试串接循环。



# 主要内容

- 7.1 编码
- 7.2 软件测试基础
- 7.3 单元测试
- 7.4 集成测试
- 7.5 确认测试
- 7.6 白盒测试技术
- ▶ 7.7 黑盒测试技术
- 7.8 调试
- 7.9 软件可靠性





## 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。



## 7.7 黑盒测试技术



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人们，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 黑盒测试技术

### 7.7.1 等价划分（等价类划分）

- ◇ 等价类划分是一种黑盒测试技术。
- ◇ **穷尽**的黑盒测试需要使用所有可能的输入数据（有效的和无效的）进行测试，通常是不现实的。因此产生了等价类划分。



◆等价类划分的思想（测试无法穷尽）：

◆ 如果将所有可能的输入数据（有效的和无效的）划分为若干个等价类，就可以假定用每一个等价类中的代表值作为测试用例来进行测试时，等价于用该类中所有值进行了测试。



◇ 用等价类划分设计测试用例时，主要分两步：  
划分等价类、确定测试用例。

◇ 1) 等价类划分

◇ 划分等价类需要经验，以下给出一些规则：

A. 如果某输入条件规定了输入的范围，那么可以划分为一个有效的等价类和两个无效的等价类。

如X的值的输入范围是[1, 99]，那么测试X时，可以这样划分：有效等价类为[1, 99]，无效等价类为 $(-\infty, 1)$ 和 $(99, +\infty)$ 。



- ◆ B. 如果某个输入条件规定了一组可能的值，且程序可以对不同的值作出不同的处理，那么可以为每种值确定一个有效的等价类，同时还有一个无效等价类。

如，“职称”这个量可能的值是：教授、副教授、讲师、助教。那么可以这样划分：四类有效等价类分别为教授、副教授、讲师、助教，无效等价类为四种职称以外的所有值。

## ◆2) 确定测试用例

- ◆ A. 给每个等价类规定一个唯一的编号;
- ◆ B. 设计一个新的测试用例, 使其**尽可能多地**覆盖未被覆盖过的有效等价类。重复此步, 直至所有有效等价类被覆盖;

- ◆ C. 设计一个新的测试用例, 使其覆盖而且只覆盖一个尚未覆盖的无效等价类。重复此步, 直至所有

**注意: 有效等价类和无效等价类的覆盖区别?**

- ◆ 通过设计测试用例, 检查其它类错, 侧, 覆盖一个无效等价类。

## 7.7 黑盒测试技术

划分等价类需要经验，下述的**启发式规则**可能有助于等价类划分。

- (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**输入数据的个数**，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等价类(符合规则)和若干个无效等价类(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。

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**。





## 7.7 黑盒测试技术

实例：一个把数字串变成整数的函数。

计算机字长：16 bits，函数由PASCAL语言编写。

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

```
type shortstr = array[1..6] of char; /字符串6位/
```

( 16位字长能表示的整型数范围是 $[-2^{15}, 2^{15}-1]$ ，即 $[-32768, 32767]$  )



## 7.7 黑盒测试技术

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。

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)。如: [ 0 , 999999 ]
- (2) 最高位数字是零的数字串。 如: “012345”
- (3) 最高位数字左邻是负号的数字串。 如: “- 12345”

- 无效输入的等价类有

- (1) 空字符串(全是空格)。如 “ ” ;
- (2) 左部填充的字符既不是零也不是空格。如: “A12345”
- (3) 最高位数字右面由数字和空格混合组成。 如: “123 45”
- (4) 最高位数字右面由数字和其他字符混合组成。如: “12A345”
- (5) 负号与最高位数字之间有空格。如: “- 1234”

。 。 。 。 。 。

## 7.7 黑盒测试技术

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。

- 合法输出的等价类有

- (1) 在计算机能表示的最小负整数和零之间的负整数。如：[ -32768 , 0 )
- (2) 零。
- (3) 在零和计算机能表示的最大正整数之间的正整数。如： ( 0 , 32767 ]

- 非法输出的等价类有

- (1) 比计算机能表示的最小负整数还小的负整数。如： “-32769”
- (2) 比计算机能表示的最大正整数还大的正整数。如： “123456”

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。

## 7.7 黑盒测试技术

根据划分出的等价类，可以设计出下述测试方案如下：

编号	描述	输入	预期输出
1	1~6个数字组成的数字串，输出是合法的正整数	‘1’	1
2	最高位数字是零的数字串，输出是合法的正整数	‘000001’	1
3	负号与最高位数字紧相邻，输出合法的负整数	‘-00001’	-1
4	最高位数字是零，输出也是零	‘000000’	0
5	太小的负整数	‘-47561’	错误—无效输入
6	太大的正整数	‘132767’	错误—无效输入
7	空字符串	‘ ’	错误—没有数字
8	字符串左部字符既不是零也不是空格	‘×××××1’	错误—填充错
9	最高位数字后面有空格	‘12’	错误—无效输入
10	最高位数字后面有其他字符	‘1××2’	错误—无效输入
11	负号和最高位数字之间有空格	‘-12’	错误—负号位置错

## 7.7 黑盒测试技术

### 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，通常输入等价类和输出等价类的边界。选取的测试数据应该**刚好等于、刚好小于和刚好大于**边界值。

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。

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。

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。

编号	描述	输入	预期输出
1	使输出刚好等于最小的负整数	‘-32768’	-32768
2	使输出刚好等于最大的正整数	‘32767’	32767
3	使输出刚刚小于最小的负整数	‘-32769’	错误—无效输入
4	使输出刚刚大于最大的正整数	‘32768’	错误—无效输入

## 7.7 黑盒测试技术

### 7.7.3. 错误推测

**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。

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。

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。

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。



## 7.7.4 实用测试策略

◇ 对软件系统进行实际测试时，应该联合使用各种设计测试方案的方法，形成一种综合策略。具体可以使用如下策略：

- ◇ 1) 在任何情况下都应该进行**边界值分析**；
- ◇ 2) 必要时用**等价划分法**补充测试方案；
- ◇ 3) 必要时再用**错误推测法**补充测试方案；
- ◇ 4) 对照程序逻辑，检查已经设计出的测试方案。可以根据对程序**可靠性的要求**采用不同的逻辑覆盖标准





## 7.7 黑盒测试技术

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，而没有考虑多个输入数据的组合效应，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。

# 课堂练习——黑盒



- ◆ 某城市电话号码由三部分组成。它们的名称和内容分别是：
- ◆ 地区码：空白或三位数字；
- ◆ 前 缀：非 ‘0’ 或 ‘1’ 的三位数字；
- ◆ 后 缀：4位数字。
- ◆ 假定被测程序能接受一切符合上述规定的电话号码，拒绝所有不符合规定的电话号码。
- ◆ 根据该程序的规格说明，在表1作等价类的划分（注意编号），在表2对应内容给出不同方案的测试用例。

表1

输入条件	有效等价类	无效等价类
地区码	1. 2. ....	
前缀		
后缀		

表2

方案	内容（来自表1）			输入（用例）	预期输出
	地区码	前缀	后缀		
1					有效/无效





# 主要内容

- 7.1 编码
- 7.2 软件测试基础
- 7.3 单元测试
- 7.4 集成测试
- 7.5 确认测试
- 7.6 白盒测试技术
- 7.7 黑盒测试技术
- 7.8 调试
- 7.9 软件可靠性



## 7.8 调试

- **调试**（也称为纠错）作为成功测试的后果出现，即调试是在测试发现错误之后排除错误的过程。
- 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\*

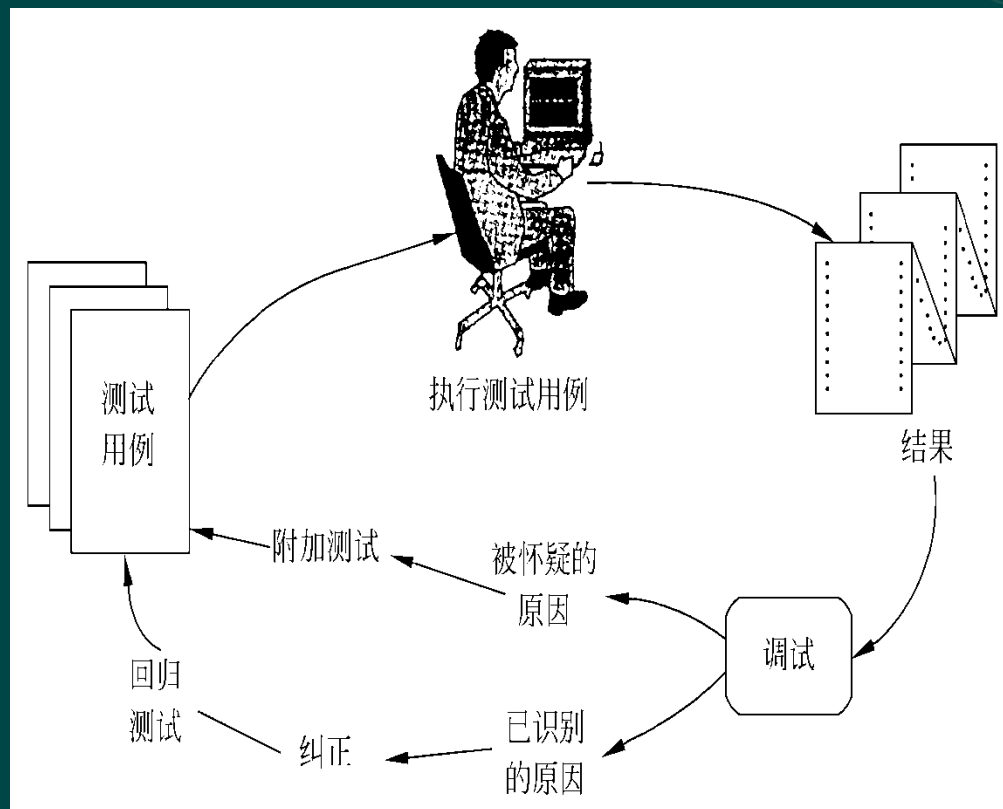
- 调试不是测试。
- 调试过程从执行一个测试用例开始，评估测试结果，如果发现实际结果与预期结果不一致，则这种不一致就是一个症状，它表明在软件中存在着隐藏的问题。调试过程试图找出产生症状的原因，以便改正错误。

## 7.8 调试

调试过程总会有以下两种结果之一：

- ①找到了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。



## 7.8 调试

调试工作如此困难，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。

## 7.8 调试



### 7.8.2. 调试途径

#### 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。
- **蛮干法**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。
- 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。



## 7.8 调试

### 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：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。
- 随着程序规模的扩大，应该回溯的路径数目变得越来越大，回溯法不适用于这种规模的程序。

### 3. 原因排错法

对分查找法、归纳法和演绎法都属于原因排除法。

## 7.8 调试\*

**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 midpoint 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。

**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。

**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。

# 主要内容



- 7.1 编码
- 7.2 软件测试基础
- 7.3 单元测试
- 7.4 集成测试
- 7.5 确认测试
- 7.6 白盒测试技术
- 7.7 黑盒测试技术
- 7.8 调试



7.9 软件可靠性





## 7.9 软件可靠性

### 7.9.1. 基本概念

**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。软件可靠性随着给定的时间间隔的加大而减少。

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**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 $t$ 这段时间间隔内系统没有失效，而可用性只意味着在时刻 $t$ ，系统是正常运行的。



## 7.9 软件可靠性

如果在一段时间内，软件系统故障停机时间分别为 $t_{d1}, t_{d2}, \dots$ ，正常运行时间分别为 $t_{u1}, t_{u2}, \dots$ ，则系统的稳态可用性为：

$$A_{ss} = \frac{T_{up}}{T_{up} + T_{down}}$$

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，则上式变为：

$$A_{ss} = \frac{MTTF}{MTTF + MTTR}$$

平均维修时间MTTR是修复一个故障平均需要的时间，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。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。



## 7.9 软件可靠性

### 7.9.2. 估算平均无故障时间的方法

#### 1. 符号

在估算MTTF的过程中使用下述符号表示有关的数量。

$E_T$ ——测试之前程序中错误总数；

$I_T$ ——程序长度(机器指令总数)；

$\tau$ ——测试(包括调试)时间；

$E_d(\tau)$ ——在0至 $\tau$ 期间发现的错误数；

$E_c(\tau)$ ——在0至 $\tau$ 期间改正的错误数。

#### 2. 基本假定

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

$$0.5 \times 10^{-2} \leq ET/IT \leq 2 \times 10^{-2}$$

## 7.9 软件可靠性

(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) 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 $E_r(\tau)=E_T - E_c(\tau)$ , 单位长度程序中剩余的错误数为 $\varepsilon_r(\tau)=E_T/I_T - E_c(\tau)/I_T$ 。

### 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:

$$MTTF = \frac{1}{K(ET/IT - EC(\tau)/IT)}$$

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。

可以根据估算平均无故障时间的公式, 得出计算 $E_c$ , 则可以估计需要改正多少个错误之后, 测试工作才能结束。



## 7.9 软件可靠性

### 4. 符号

#### (1) 植入错误法

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，来估计程序中原有错误的总数 $E_T$ 。

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 
$$\hat{N} = \frac{n}{n_s} N_s$$

其中， $\hat{N}$  即是错误总数 $E_T$ 的估计值。

## 7.9 软件可靠性



### 4. 符号

#### (2) 分别测试法

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  $\tau$  表示测试时间，假设

- $\tau=0$ 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。



## 7.9 软件可靠性

### 4. 符号

#### (2) 分别测试法

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

$$\hat{B}_0 = \frac{B_2}{b_c} B_1$$

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，并且使用上面的公式计算 $\hat{B}_0$ 。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。



# 本章小结



1. 实现包括编码和测试两个阶段。
2. 高级程序设计语言较汇编语言有很多优点。
3. 通常软件测试至少分为单元测试、集成测试和验收测试3个基本阶段。
4. 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)。
5. 白盒测试和黑盒测试是软件测试的两类基本方法，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。
6. 及时改正测试过程中发现的软件错误就是调试的任务。
7. 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。