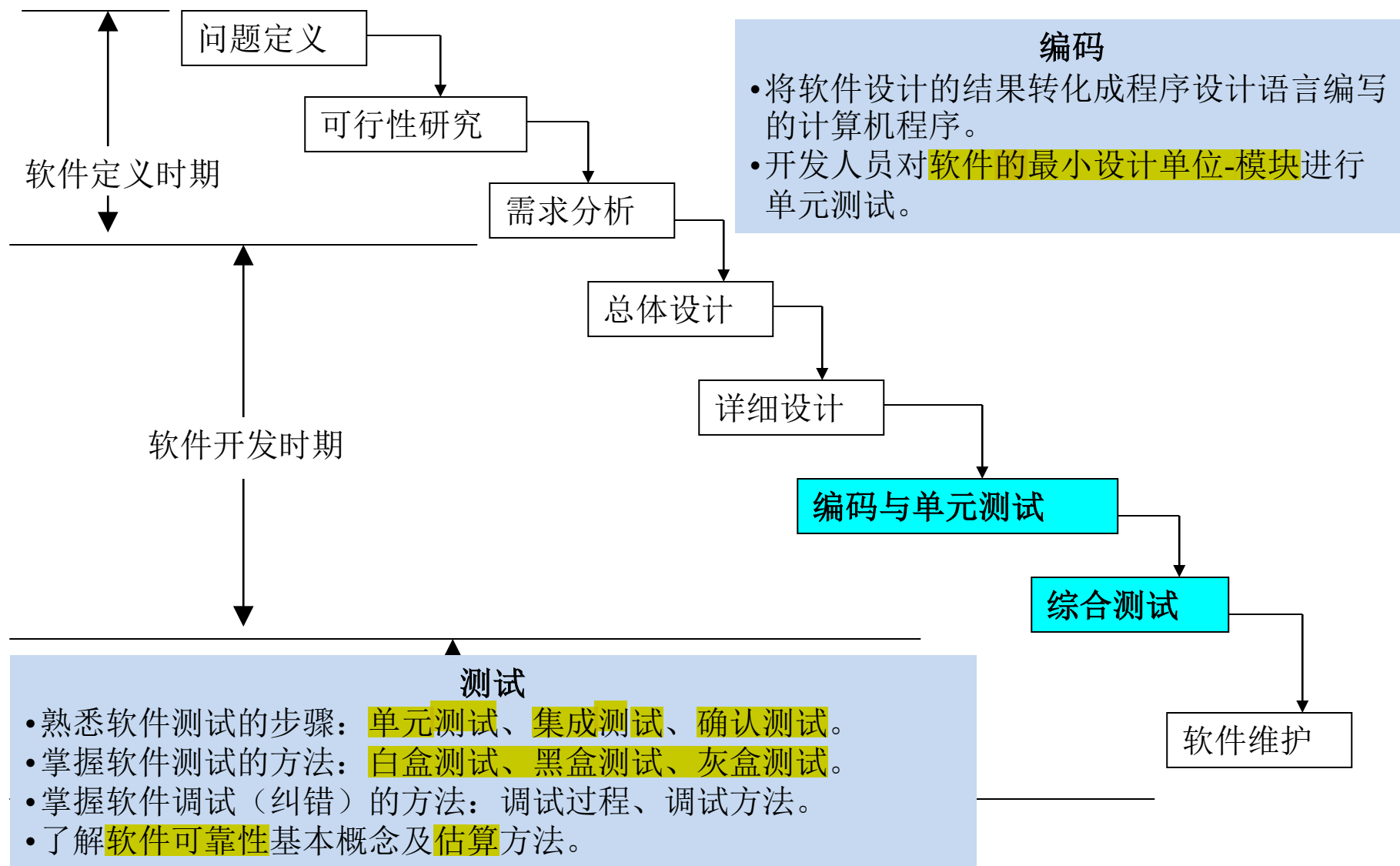


# 第七章 实现

(编码与测试)

Implementation=Coding + Testing

# 实现：编码与测试



# 第七章 实现(编码与测试)

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

# 第七章 实现(编码与测试)

## 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

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。

### 7.1.1 选择程序设计语言

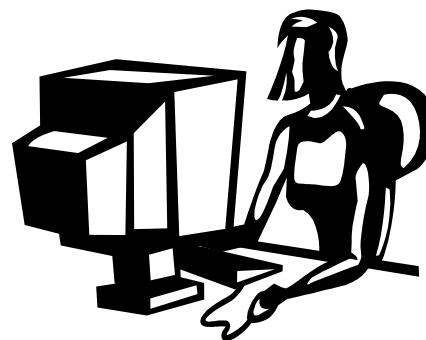
### 7.1.2 编码风格

- (1) 程序内部的文档
- (2) 数据说明
- (3) 语句构造
- (4) 输入输出
- (5) 效率

## 7.1.1 选择程序设计语言

### 实用选择标准：

- 用户对编程语言的要求
- 可以使用的编译程序
- 可以得到的软件工具
- 工程规模
- 程序员的编程语言知识
- 软件可移植性要求
- 软件的应用领域



## 7.1.2 编码风格 —— “好的” 程序应遵循的规则

### (1) 程序文档:

- 源程序文档化, 易于理解的标识符命名、适当的注释、清晰的程序视觉组织等

### (2) 数据说明:

- 易于理解, 便于查阅

### (3) 语句构造:

- 结构化, 简单、直观, 技巧不过份

### (4) 输入输出:

- 遵循人机界面设计准则

### (5) 效率:

- 效率满足需求; 不要为了追求效率而过份使用技巧, 牺牲程序的清晰性、可读性

## 7.1.2 编码风格 —— （1）程序文档

程序中的标识符（名字）

程序中的注释

程序的视觉组织



## 7.1.2 编码风格 —— (1) 程序文档：符号名的命名

### 符号名

- 即标识符，包括模块名、变量名、常量名、标号名、子程序名、数据区名以及缓冲区名等。

### 名字的意义

- 这些名字应能反映它所代表的实际东西，应有一定实际意义。例如，表示次数的量用Times，表示总量的用Total，表示平均值的用Average，表示和的量用Sum等。

### 名字的使用

- 名字不是越长越好，应当选择精炼的意义明确的名字。
- 必要时可使用缩写名字，但这时要注意缩写规则要一致。
- 要给每一个名字加注释。
- 在一个程序中，一个变量只应用于一种用途。

## 7.1.2 编码风格 —— (1) 程序文档：程序的注释

- 注释**决不是可有可无**的，程序文本中，注释行的数量占到整个源程序的**1/3到1/2**，甚至更多。
- 注释分为**序言性注释**和**功能性注释**。

## 7.1.2 编码风格 —— （1）程序文档：程序的注释

- 序言性注释包括：
  - 程序标题；
  - 模块功能和目的的说明；
  - 主要算法说明：算法概要、大意；
  - 接口说明：包括调用形式，参数描述，子程序清单；
  - 数据描述：重要的变量及其用途，约束或限制条件，以及其它有关信息；
  - 模块位置：在哪一个源文件中，或隶属于哪一个软件包；
  - 开发简历：模块设计者，复审者，复审日期，修改日期及有关说明等。

## 7.1.2 编码风格 —— (1) 程序文档：程序的注释

- 功能性注释描述其后的语句或程序段是`在做什么工作`，而不是解释下面怎么做。
- **要点：**
  - 描述`一段`程序，而不是每一个语句；
  - 用`缩进和空行`，使程序与注释容易区别；
- 例如：
  - `/* ADD AMOUNT TO TOTAL */`  
`TOTAL = AMOUNT + TOTAL`
  - 上面注释不清楚，如果注明把月销售额计入年度总额，便使读者理解了下面语句的意图：  
`/* ADD MONTHLY-SALES TO ANNUAL-TOTAL */`  
`TOTAL = AMOUNT + TOTAL`

## 7.1.2 编码风格 —— (1) 程序文档：程序的视觉组织

- **空格**，可以**突出运算的优先性**，避免发生运算的错误。例如，将表达式  
     $(A < -17) \text{ANDNOT} (B < = 49) \text{OR} C$   
写成  $(A < -17) \text{ AND NOT } (B < = 49) \text{ OR } C$
- **空行**，自然的程序段之间可用**空行**隔开；
- **缩进**，是指程序中的各行不必都在左端对齐，都从第一格起排列。这样做使程序完全分不清层次关系。
- 对于**选择语句**和**循环语句**，把其中的程序段语句向右做**阶梯式移行**。使程序的逻辑结构更加清晰。

```
IF (...) THEN
    IF (...) THEN
        .....
    ELSE
        .....
    ENDIF
.....
ELSE
    .....
ENDIF
```

## 单行注释:

```
//设置用户信息
private void btnTest1_Click(object sender, EventArgs e)
{
    string strUser = ""; //定义变量
    strUser = textBox1.Text; //给变量赋值
}
```

////////////////////////////////////

多行注释:

```
///  
/// 获取2个变量的和  
///  
/// <param name="a">变量a</param>  
/// <param name="b">变量b</param>  
/// <returns>返回a和b的和</returns>  
public int getCount(int a, int b)  
{  
    int c = -1;  
    c = a + b;  
    return c;  
}
```

## 长段注释:

```
/*
 * 程序需求:
 * 客户要求本程序可以记录每日的销售额, 并进行按日期统计排序功能:
 * 可打印报表:
 * 可存储报表为excel, 图片等格式.
 */

/*
 * 版本 V1.00 修改时间2014.2.1   修改内容 新增用户变量
 * 版本 V1.01 修改时间2014.2.10  修改内容 新增日期变量
 * 版本 V1.02 修改时间2014.3.4   修改内容 根据客户需求修改界面
 */
public Form1()
{
    InitializeComponent();
}
```



## 7.1.2 编码风格 —— (2) 数据说明

- 数据说明指程序中用到的常量、变量、文件等数据对象的定义。
- 在设计阶段已经确定了数据结构的组织及其复杂性。在编写程序时，则需要注意数据说明的风格。
- 为了使程序中数据说明更易于理解和维护，必须注意以下几点：
  - a. 数据说明的次序应该标准化。有次序易查阅，能加速测试、调试和维护的过程。
  - b. 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。
  - c. 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。

## 7.1.2 编码风格 —— (2) 数据说明

### a. 数据说明的次序:

#### 数据说明

- ① 常量说明
- ② 简单变量类型说明
- ③ 数组说明
- ④ 公用数据块说明
- ⑤ 所有的文件说明

#### 数据类型说明

- ① 整型量说明
- ② 实型量说明
- ③ 字符型量说明
- ④ 逻辑型量说明

## 7.1.2 编码风格 —— (2) 数据说明

b. 当同时有**多个变量名**时，**按字母顺序排列**变量，便于查找。

例如，把

*integer size, length, width, cost, price*

写成

*integer cost, length, price, size, width*

## 7.1.2 编码风格 —— (3) 语句构造

- 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；也不要刻意追求技巧性，使程序编写得过于紧凑。

功能：A[T]与A[I]交换数据值

```
A[I] = A[I] + A[T];  
A[T] = A[I] - A[T];  
A[I] = A[I] - A[T];
```



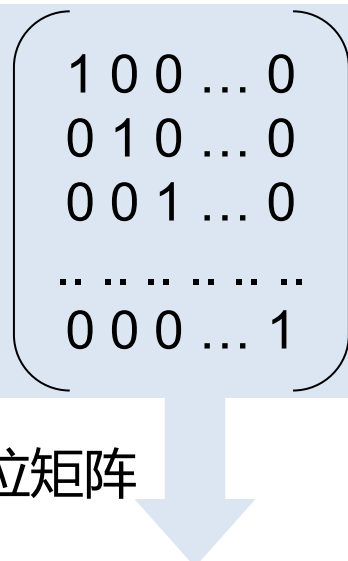
```
WORK = A[T];  
A[T] = A[I];  
A[I] = WORK;
```

未使用中间变量  
难懂

使用中间变量  
清晰

## 7.1.2 编码风格 —— (3) 语句构造

```
int i, j;  
for ( i = 1; i <= n; i++ )  
    for ( j = 1; j <= n; j++ )  
        V[i][j] = ( i / j ) * ( j / i )
```


$$\begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix}$$

功能：初始化对角单位矩阵

```
for ( i = 1; i <= n; i++ )  
    for ( j = 1; j <= n; j++ )  
        if ( i == j )  
            V[i][j] = 1;  
        else  
            V[i][j] = 0;
```

## 7.1.2 编码风格 —— (3) 语句构造

- 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。

例：使用not(!)操作的例子  
将

```
if ( !( char < 0 || char > 9 ) )
```

改成

```
if ( char >= 0 && char <= 9 )
```

不要让人绕弯子想!

## 7.1.2 编码风格 —— (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：

- 对所有输入数据都要进行**检验**，识别错误输入，以保证每个数据的有效性；
- 检查输入项的各种重要组合的**合法性**，必要时报告输入状态信息；
- 使得输入的步骤和操作**尽可能简单**，并保持简单的输入格式；
- 输入数据时，应允许使用**自由格式输入**；
- 应允许**缺省值**；
- 输入一批数据时最好使用**输入结束标志**，而不要由用户指定输入数据数目；
- 在交互式输入输入时，要在屏幕上使用**提示符**明确提示交互输入的请求，指明可使用选择项的种类和取值范围。同时，在数据输入的过程中和输入结束时，也要在屏幕上给出状态信息；
- 当程序设计语言对输入 / 输出格式有严格要求时，应保持输入格式与输入语句的要求的**一致性**；
- 给所有的输出加**注解**，并设计**输出报表**格式。

## 7.1.2 编码风格 —— (5) 程序效率

### 程序效率

- 程序的效率是指程序的执行速度及程序所需占用的内存的存储空间。程序编码是最后提高运行速度和节省存储的机会，因此在此阶段不能不考虑程序的效率。

### 效率准则

- 需求阶段：软件效率以需求为准。
- 设计阶段：好的设计可以提高效率。
- 编码阶段：程序的效率与程序的简单性相关，不要牺牲程序的清晰性和可读性来不必要地提高效率。

### 效率问题

- 程序运行时间效率
- 存储器效率
- 输入输出的效率





## 7.1.2 编码风格 —— (5) 程序效率

在把详细设计结果翻译成程序时，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。

# 第七章 实现(编码与测试)

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

7.2.2 软件测试准则

7.2.3 测试方法

7.2.4 测试步骤

7.2.5 测试阶段的信息流

## 7.2 软件测试基础

### 7.2.1 软件测试的目标

### 7.2.2 软件测试准则

### 7.2.3 测试方法

### 7.2.4 测试步骤

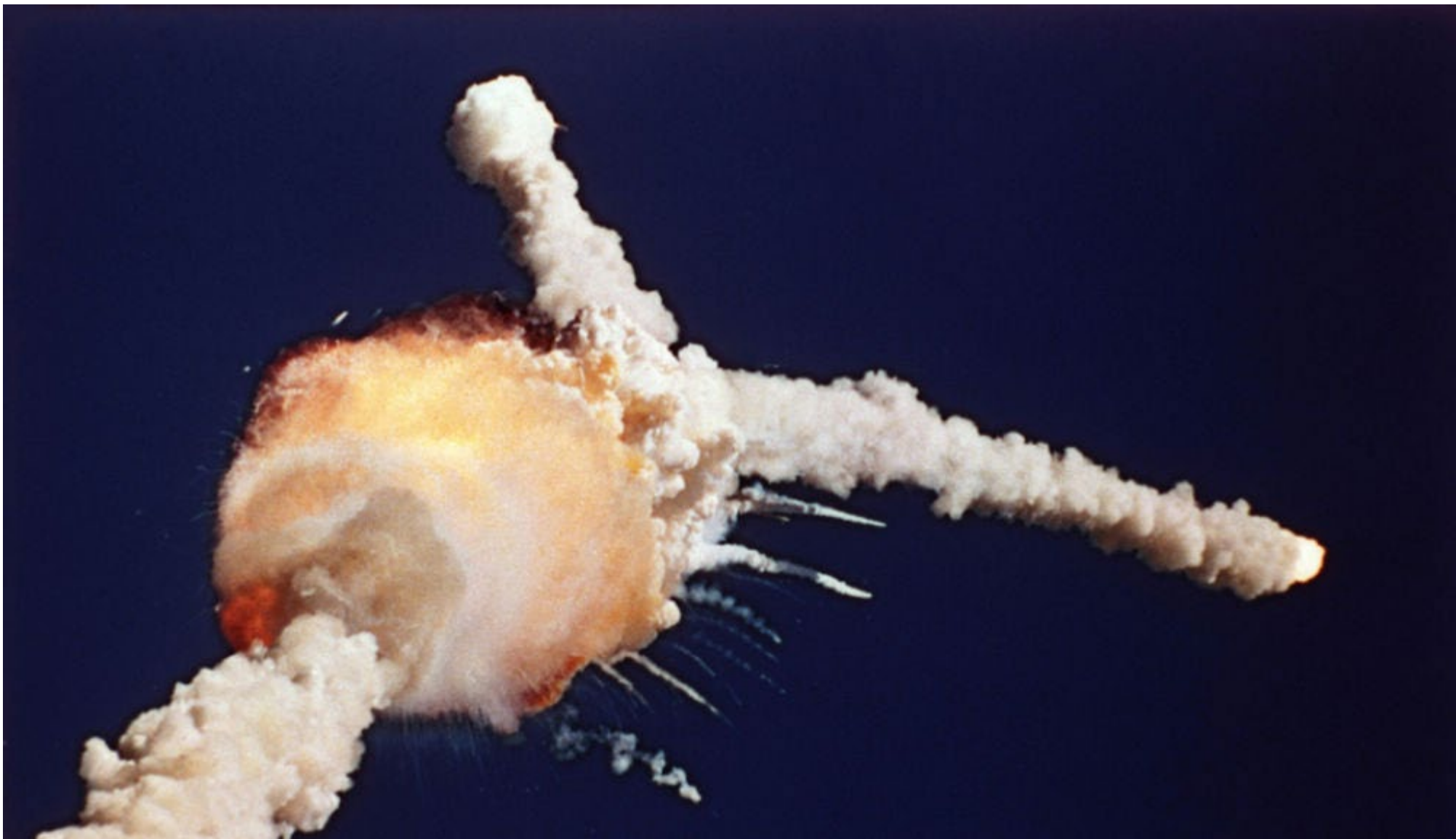
### 7.2.5 测试阶段的信息流















## 7.2.1 软件测试的目的

### 软件错误造成工作的巨大损失

- 案例：1963年, 美国, 飞往火星的火箭爆炸, 损失 \$10 million。原因: FORTRAN循环: `DO 5 I = 1, 3` 误写为 `DO 5 I = 1.3`。

## 7.2.1 软件测试的目的

观点：

- 观点一：软件测试的目的是为了“证明程序正确”。
- 观点二：软件测试的目的是为了“找出程序中的错误”。

定义：

对软件测试目的提出以下观点和定义：

- (1) 软件测试是为了发现错误而执行程序的过程。
- (2) 一个好的测试用例能够发现至今尚未发现的错误。
- (3) 一个成功的测试是发现了至今尚未发现的错误的测试。

测试只能证明程序中有错误，不能证明程序中没有错误

## 7.2.1 软件测试的目的

- 在测试阶段测试人员努力设计出一系列测试方案，目的是为了“破坏”已经建造好的软件系统。
- 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。
- 测试的目标决定了测试方案的设计：
  - 如果测试是为了表明程序是正确的，就会设计一些不易暴露错误的测试方案；
  - 如果测试是为了发现程序中的错误，就会设计出最能暴露错误的测试方案。

测试决不能证明软件是正确的，也不能证明错误的不存在，它只能力求证明错误的存在。

## 7.2.1 软件测试的目的——软件测试的有关问题

软件缺陷是什么？

谁执行测试？

- 开发者？
- 单独的测试人员？
- 两方面人员？

测试什么？

- 每个部分都测试？
- 测试软件中高风险部分？

什么时候测试？

怎样测试？

测试应进行到什么程度？



## 7.2.1 软件测试的目的——软件缺陷

缺点 (defect)

谬误 (fault)

问题 (problem)

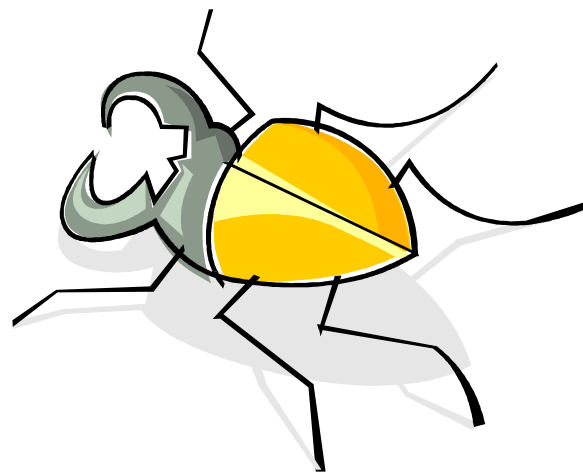
错误 (error)

异常 (anomaly)

偏差 (variance)

失败 (failure)

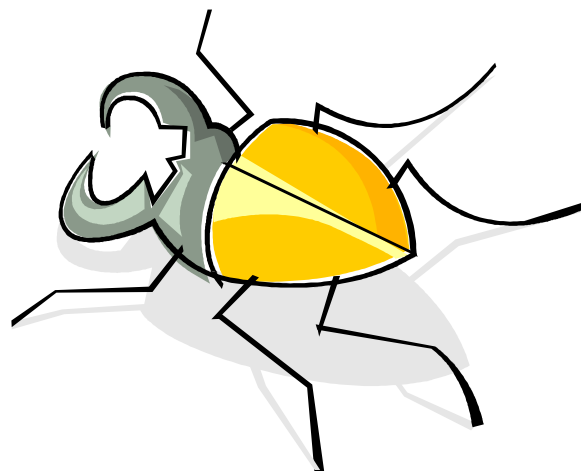
这些缺陷都是描述软件失败的术语! ——Bug



## 7.2.1 软件测试的目的——软件缺陷

一些基本的名词：

- Bug（软件缺陷）
- Test Case（测试用例）
- Test Suite（测试用例集）



Bug可以分解为：

- Symptom（症状）、Fault（程序错误）、根本原因（Root Cause）
- 症状：从用户的角度看，软件除了什么问题
- 程序错误：从代码的角度看，代码的什么错误导致了软件问题
- 根本原因：错误的根源，导致代码错误的根本原因

## 7.2.1 软件测试的目的——软件缺陷

一个关于Bug的完整的例子：

(1) **症状**：用户报告，一个Windows应用程序有的时候会在启动时报错，继而不能运行。

(2) **程序错误**：发现有的时候，一个子窗口的handle为空，导致程序访问了非法内存地址，此为代码错误。

(3) **根本原因**：代码在调用子窗口（OnDraw()）之前，没有确保创建子窗口(CreateSubWindow())，所以子窗口的handle变量有时会在访问时处于为空的状态，导致代码错误。



# 软件测试有关人员

## 测试工具软件开发工程师

(Software Development Engineer in Test, 简称SDE/T)



SDE/T

负责写测试工具代码，并利用测试工具对软件进行测试；或者开发测试工具为软件测试工程师服务。

## 软件测试工程师

(Software Test Engineer, 简称STE)



STE

负责理解产品的功能要求，然后对其进行测试，检查软件有没有错误(Bug)，决定软件是否具有稳定性，并写出相应的测试规范和测试案例

## 7.2 软件测试基础

7.2.1 软件测试的目标

7.2.2 软件测试准则

7.2.3 测试方法

7.2.4 测试步骤

7.2.5 测试阶段的信息流

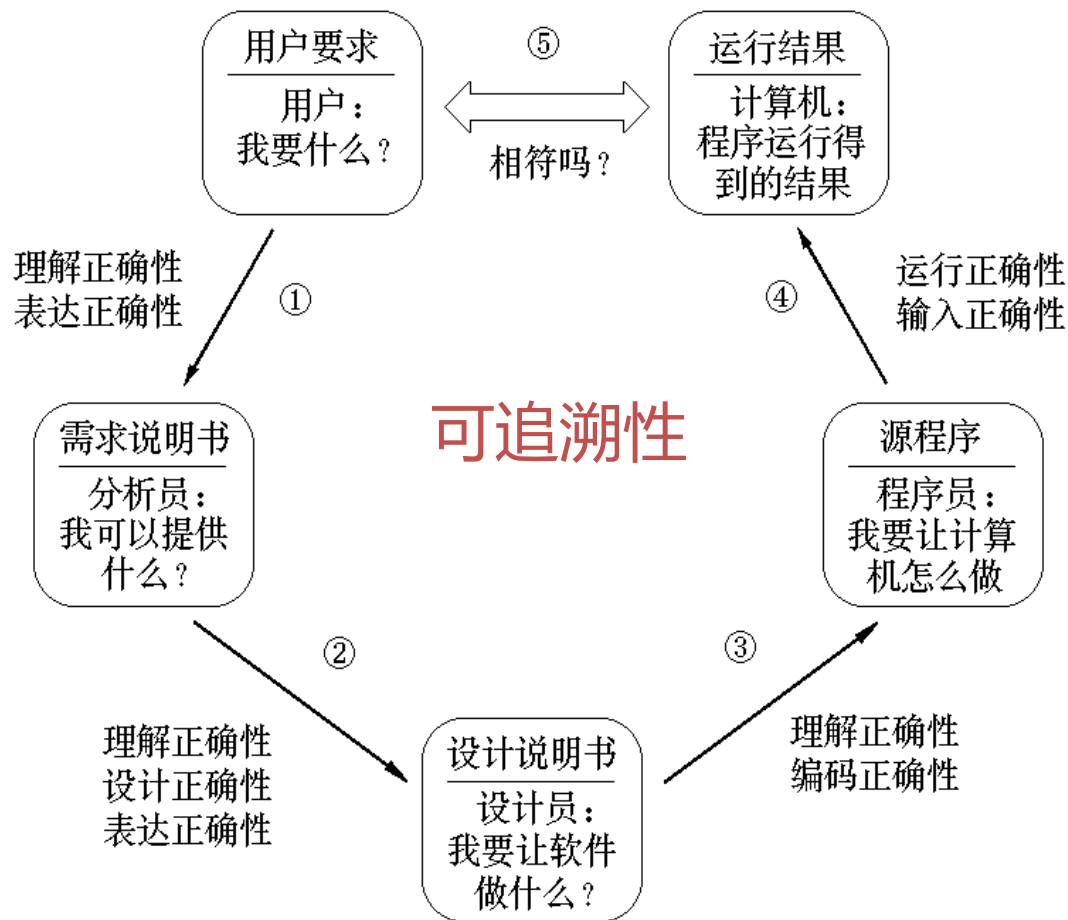
## 7.2.2 软件测试准则

- (1) 所有测试都应该能追溯到用户需求。
- (2) 应“尽早地规划和不断地进行软件测试”。
- (3) 将pareto原则（80%:20%）应用于软件测试。
- (4) 应从“小规模”测试开始，逐步进行“大规模”测试。
- (5) 穷举测试是不可能的。
- (6) 应由独立的第三方从事测试工作。
- (7) 测试用例应由输入数据和预期的输出结果两部分组成。
- (8) 程序修改后要回归测试。
- (9) 应长期保留测试用例，直至系统废弃。

## 7.2.2 软件测试准则

### (1) 测试应该能追溯到用户需求

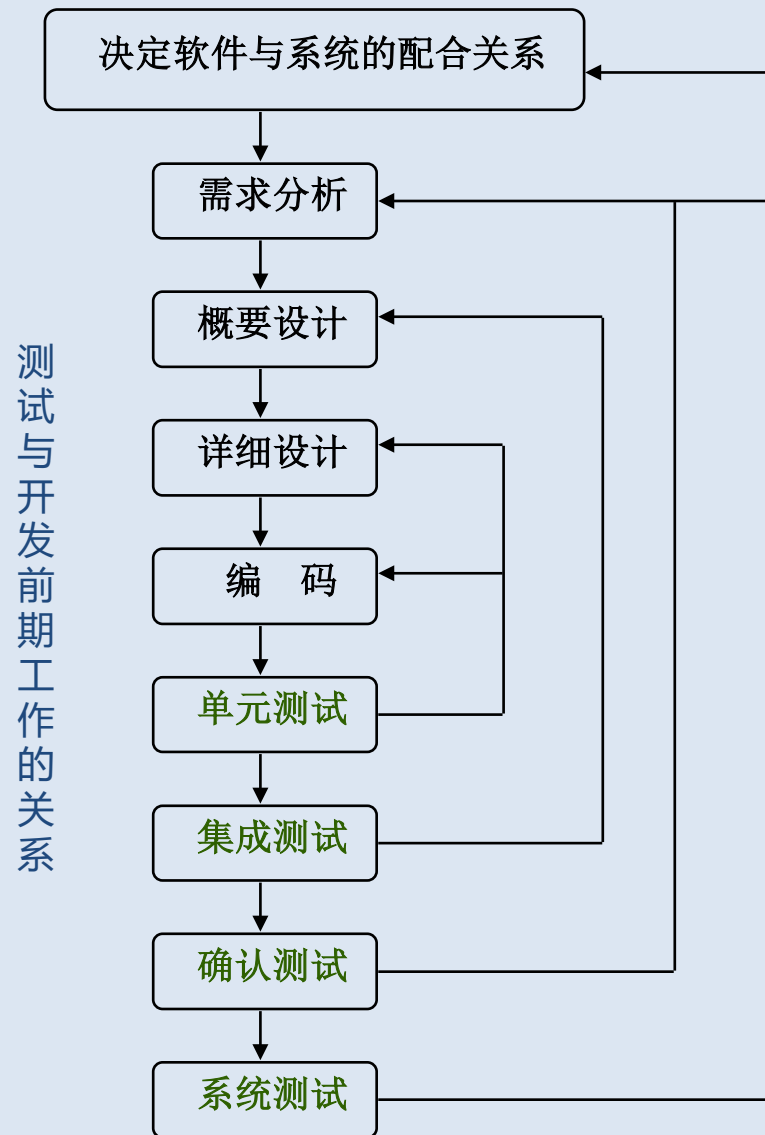
- 软件测试的目标是发现错误。从用户的角度看，最严重的错误是导致程序不能满足用户需求的那些错误。
- 软件中的问题**根源**可能在开发前期的各阶段解决、纠正错误也必须追溯到前期工作。



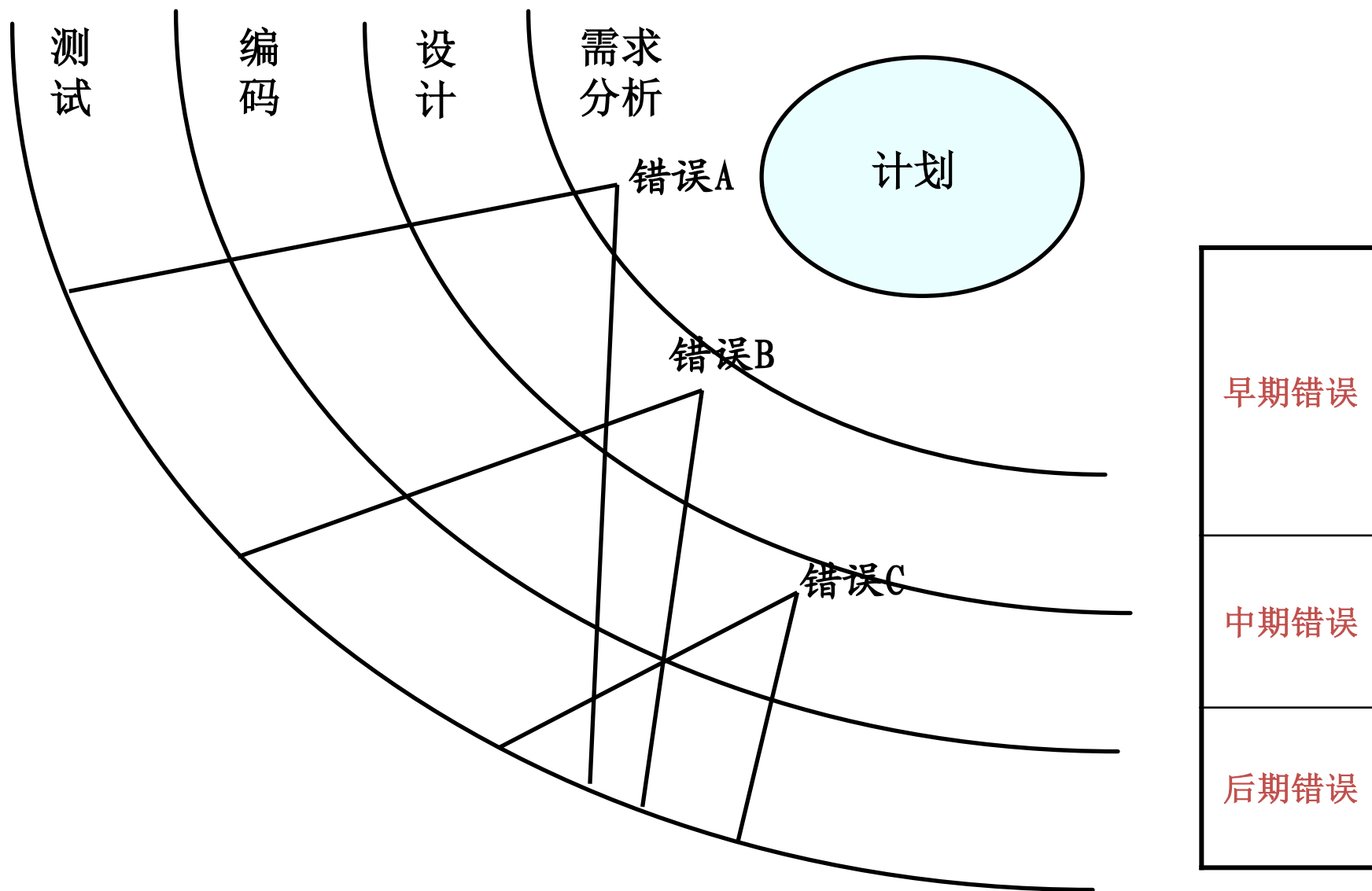
## 7.2.2 软件测试准则

### (2) 尽早地规划和不断地进行软件测试

- 尽早地、不断地进行软件测试。  
概要设计时应完成测试计划，  
详细的测试用例定义可在设计模型确定后开始，  
所有测试可在任何代码被产生之前进行计划和设计。
- 软件测试不等于程序测试。软件测试应贯穿于软件定义与开发的整个期间；
- 程序编写的许多错误是“先天的”。据统计，查出的软件错误中，属于需求分析和软件设计的错误约占 64%，属于程序编写的错误仅占 36%。



## 7.2.2 软件测试准则——错误扩展模型



## 7.2.2 软件测试准则

### (3) 在软件测试中应用pareto原则

- pareto原则：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。问题是怎样找出这些可疑的模块并彻底地测试它们。

### (4) 从“小规模”到“大规模”测试

- 应该从“小规模”测试开始，并逐步进行“大规模”测试。
- 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。

## 7.2.2 软件测试准则

### (5) 注意测试用例的组成

- 测试用例应由输入数据和预期的输出结果两部分组成，并兼顾合理的输入和不合理的输入数据

### (6) 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。由于程序的执行路径庞大（路径组合爆炸），受时间、人力及其他资源的限制，穷举法难以实施。
- 但是，可用穷举法覆盖所有的程序逻辑——遍历程序流图的所有边（即每个语句都被执行一次！），达到所要求的一定的测试可靠性。



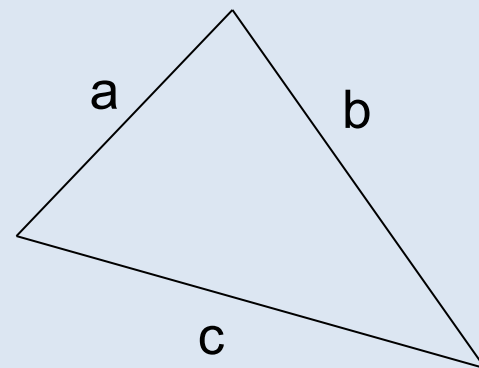
# 穷举测试例

例1:

- 输入三角形的三条边长, 可采用的测试用例数 (设字长16位)为 (次)

$$2^{16} * 2^{16} * 2^{16} \approx 3 * 10^{10} \text{ (次)}$$

- **执行时间:** 设测试一次需1ms, 约需一万年。

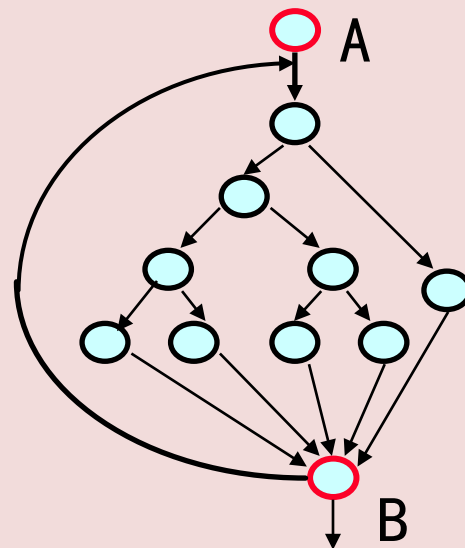


例2:

- 如图, 设程序含4个分支, 循环次数 $\leq 20$ , 从A到B的可能路径有:

$$5^1 + 5^2 + \dots + 5^{20} \approx 10^{14} \text{ (条)}$$

- **执行时间:** 设测试一次需2ms, 则穷举测试需5亿年。



## 7.2.2 软件测试准则

(7) 应该由独立的第三方从事测试工作。

- 由于心理上的原因，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。而其他测试应由第三方承担。

(8) 程序修改后要回归测试

- 程序正常修改或纠错修改之后，可能引入新的错误，因此必须进行**回归测试**。

(9) 应长期保留测试用例，直至系统废弃。

- **保留测试用例**，可用于回归测试、新版本测试。也可在程序出现错误之后，进行分析，寻找出错原因、漏测原因。

## 7.2 软件测试基础

7.2.1 软件测试的目标

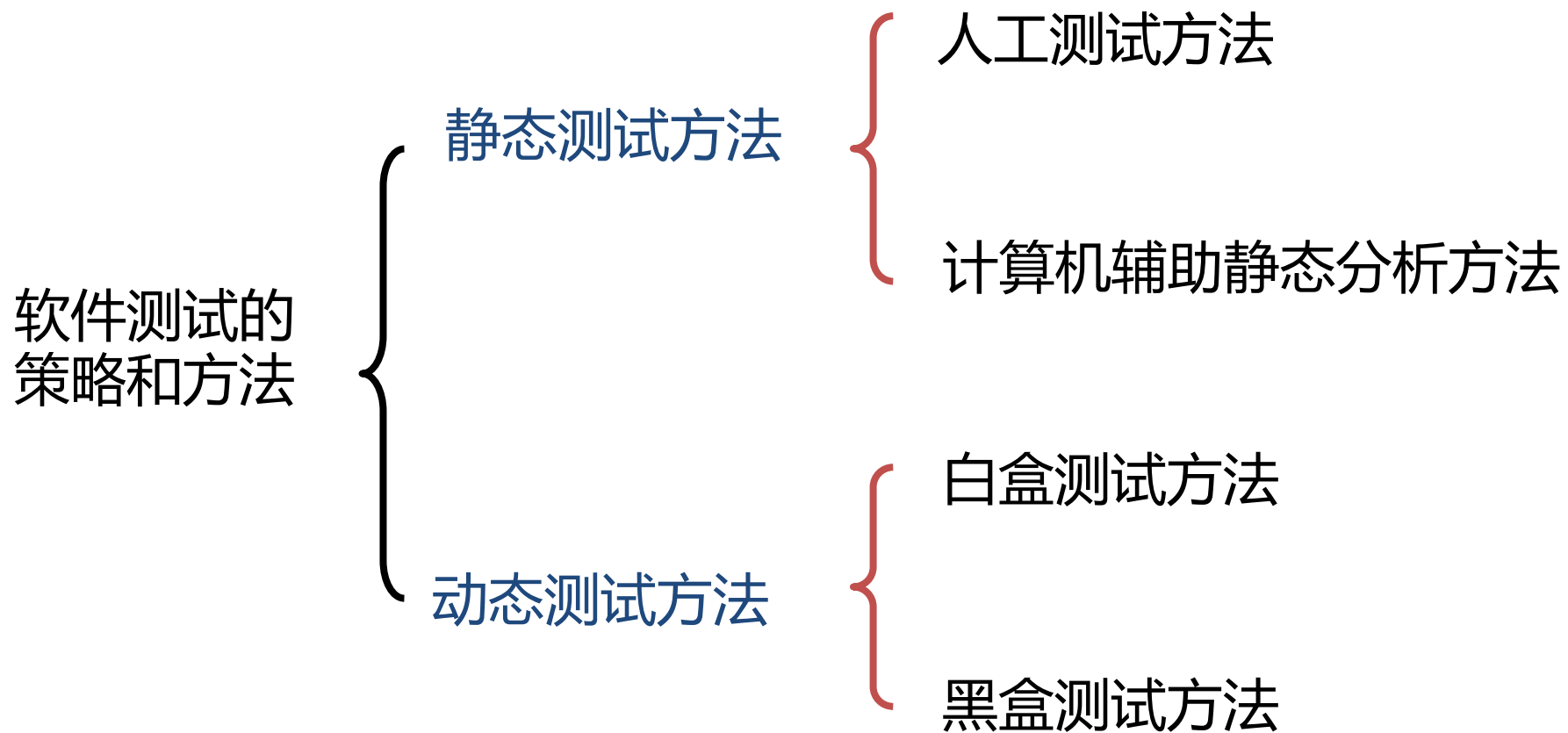
7.2.2 软件测试准则

7.2.3 测试方法

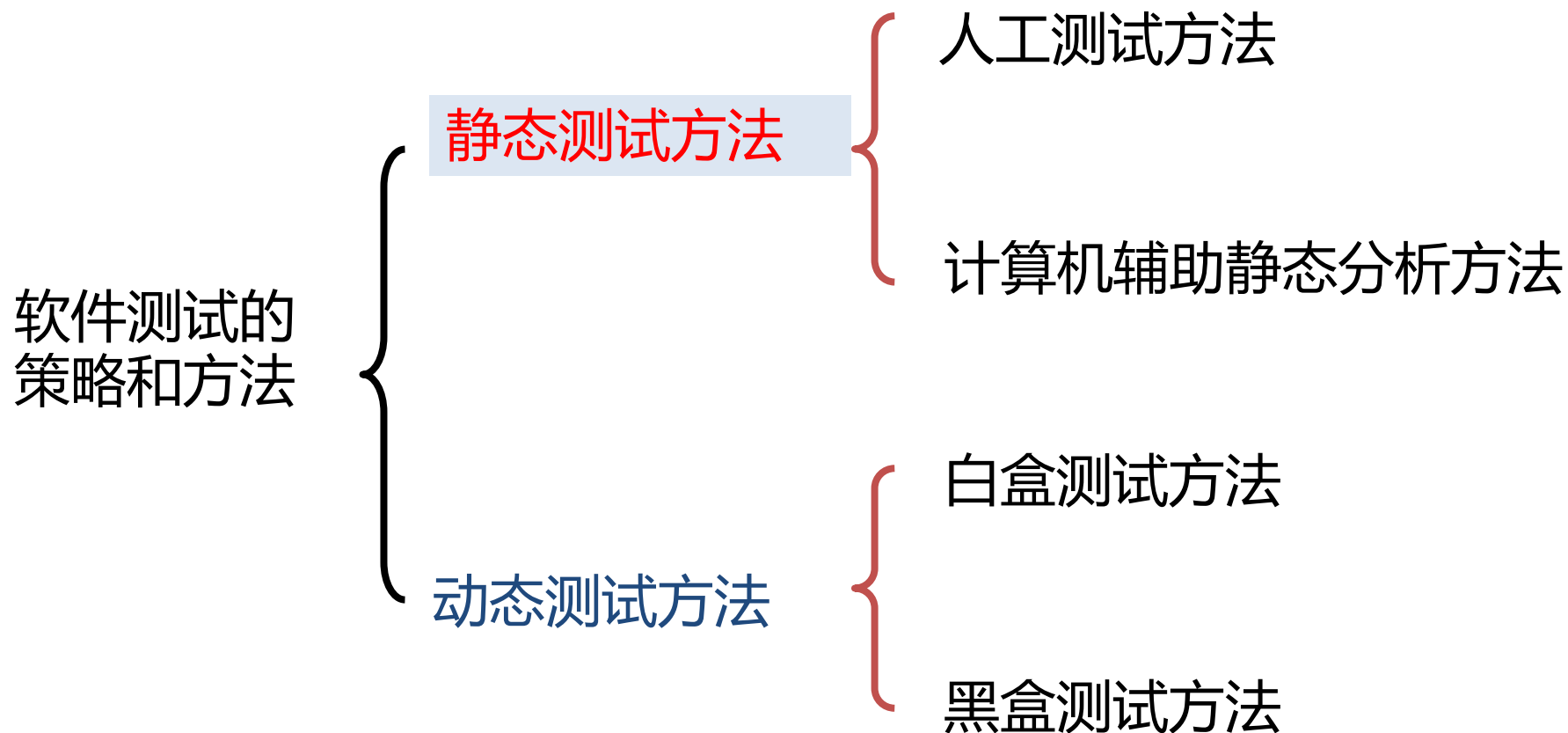
7.2.4 测试步骤

7.2.5 测试阶段的信息流

## 7.2.3 测试方法



## 7.2.3 测试方法——静态测试方法



## 7.2.3 测试方法——静态测试方法

### 静态测试：

- 基本特征是在对软件需求规格说明书、软件设计说明书、源程序进行分析、检查和审阅，**不实际运行被测试的软件**。

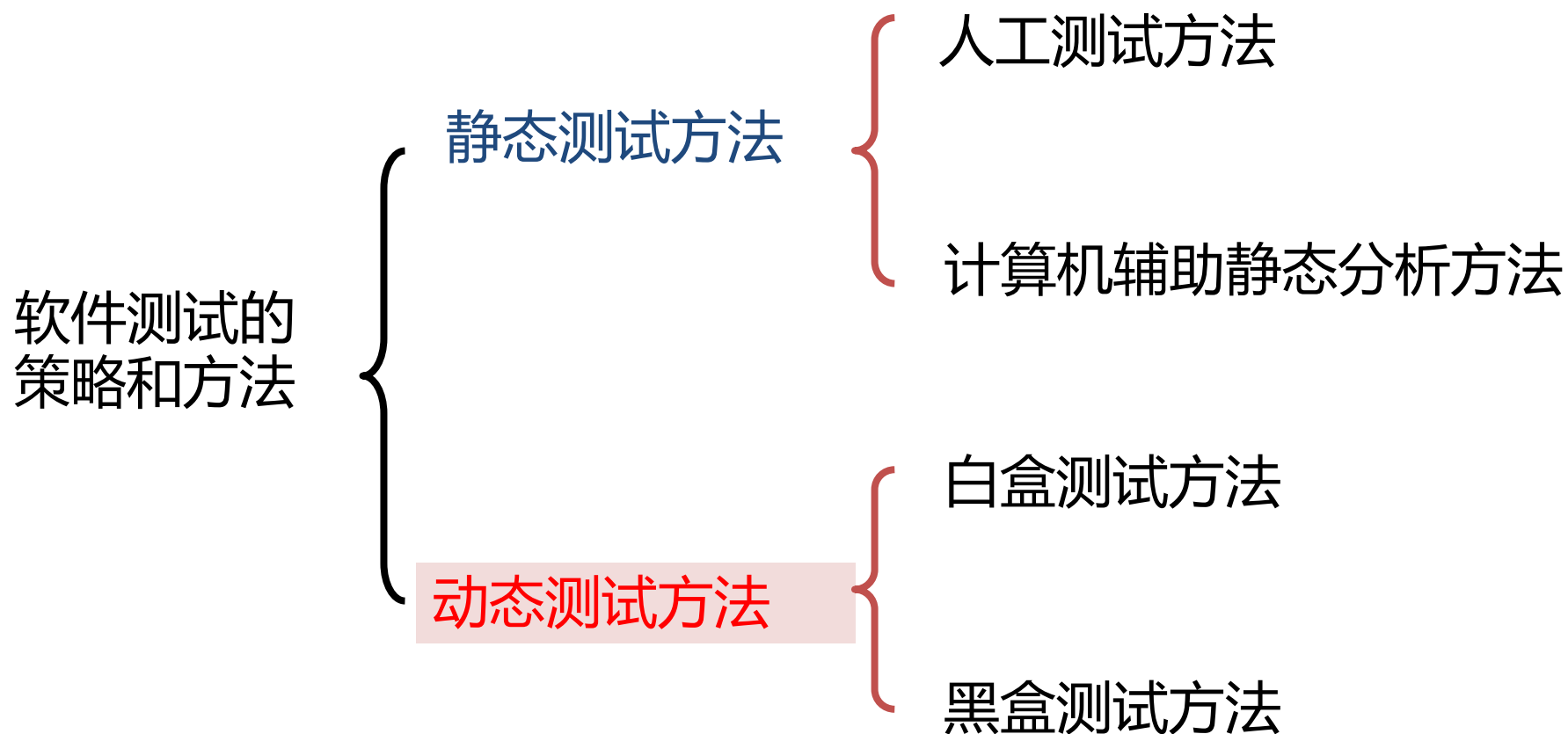
### 分析、检查和审阅的内容：

- 是否符合相关的标准和规范；
- 通过结构分析、流图分析、符号执行等，指出软件缺陷。

### 效果：

- 静态测试约可找出30 ~ 70%的**逻辑设计错误**。

## 7.2.3 测试方法——动态测试方法



## 7.2.3 测试方法——动态测试方法

动态测试：

通过运行软件来检验软件的动态行为和运行结果的正确性。

动态测试的两个基本要素：

- 被测试程序
- 测试数据（测试用例）

测试用例ID

目的

前提

输入

预期输出

后果

执行历史

日期、结果、版本、执行人

动态测试方法：

- (1) 选取定义域有效值,或定义域外无效值;
- (2) 对已选取值决定预期的结果;
- (3) 用选取值执行程序;
- (4) 执行结果与预期的结果相比, 不吻合则程序有错。



## 7.2.3 测试方法——动态测试方法

### 白盒测试法(White Box Testing)

- 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，用于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。
- 如果知道软件的内部工作过程，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。
- 白盒测试又称为结构测试。

### 黑盒测试法(Black Box Testing)

- 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。只检查程序功能是否按照规格说明书的规定正常使用。
- 如果已经知道了软件应该具有的功能，可以通过测试来检验是否每个功能都能正常使用。
- 黑盒测试又称为功能测试。

## 7.2.3 测试方法——动态测试方法：黑盒测试

已知产品的功能设计规格，  
可以进行测试证明每个实现了的  
功能是否符合要求。



黑盒测试  
的内容

Alpha/Beta Testing

菜单/帮助测试

发行测试

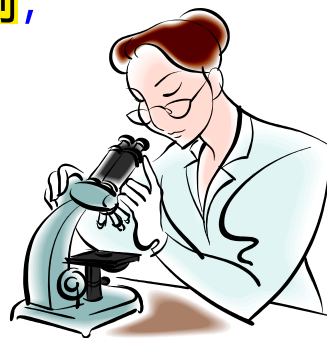
回归测试



## 7.2.3 测试方法——动态测试方法：白盒测试

白盒测试也叫**玻璃盒测试** (Glass Box Testing)，对软件的过程性细节做细致的检查。

这一方法是把测试对象看作一个打开的盒子，它允许测试人员利用程序内部的逻辑结构及有关信息，来设计或选择**测试用例**，对程序所有**逻辑路径**进行测试。



白盒测试  
的内容

对程序模块的所有**独立执行路径**至少测试一次

对所有的**逻辑判定**，取“真”与取“假”的两种情况都能至少测试一次。

在循环的边界和运行边界内执行循环体

测试**内部数据结构**的有效性。

## 7.2 软件测试基础

7.2.1 软件测试的目标

7.2.2 软件测试准则

7.2.3 测试方法

7.2.4 测试步骤

7.2.5 测试阶段的信息流

## 7.2.4 测试步骤

集成  
测试

### 1. 模块测试（单元测试）—— 程序的单元

- 在测试中所发现的往往是编码和详细设计的错误。

### 2. 子系统测试 —— 程序的局部

- 模块相互间的协调和通信是这个测试过程中的主要问题，着重测试模块的接口。

### 3. 系统测试 —— 子系统的集成

- 验证系统能否提供需求说明书中指定的功能，动态特性是否符合预定要求。其中发现的往往是软件设计中的错误，也可能是需求说明中的错误。

### 4. 验收测试（确认测试） —— 用户参与系统整体测试

- 测试内容与系统测试基本类似，但它是在用户积极参与下进行的，而且主要使用实际数据进行测试。目的是验证系统确实能够满足用户的需要，其中发现的往往是系统需求说明书中的错误。

### 5. 平行运行 —— 新旧系统共存

- 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。
- (1) 可以在准生产环境中运行新系统而又不冒风险；(2) 用户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，进一步验证性能指标。

## 7.2 软件测试基础

7.2.1 软件测试的目标

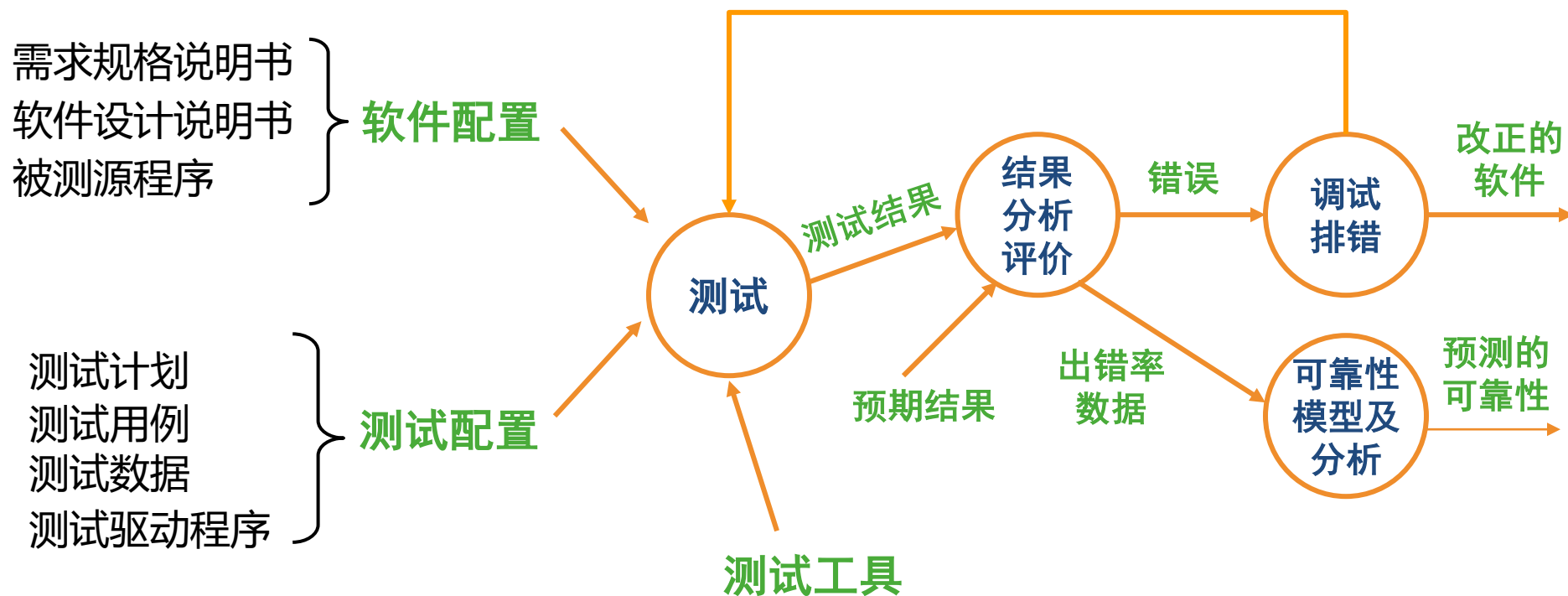
7.2.2 软件测试准则

7.2.3 测试方法

7.2.4 测试步骤

7.2.5 测试阶段的信息流

## 7.2.5 测试阶段的信息流



测试数据自动生成程序、静态分析程序、动态分析程序、测试结果分析程序、以及驱动测试的测试数据库等。

## 7.2.5 测试阶段的信息流

1. 测试阶段的的输入信息有两类：
  - (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。
    - 测试方案包括：测试用例；测试时使用的输入数据； 每组输入数据预定要检验的功能，以及预期应该得到的正确输出。
    - 测试结果和预期结果不一致时，进入调试阶段。
2. 如果经常出现要修改设计的错误，那软件的质量和可靠性是值得怀疑的。
3. 如果软件功能正常，遇到的错误也很容易改正，仍要考虑两种可能：
  - (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。
4. 如果经过测试，一个错误也没有被发现，则很可能是对测试配置思考不充分，以致不能暴露软件中潜藏的错误。



## 7.2.5 测试阶段的信息流——软件测试的对象

软件测试并不等于程序测试。软件测试应贯穿于软件定义与开发的整个期间。因此，需求分析、概要设计、详细设计以及程序编码等所得到的文档资料，包括需求规格说明、概要设计说明、详细设计规格说明以及源程序，都应成为软件测试的对象。

**软件 = 程序 + 文档！**

# 第七章 实现(编码与测试)

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

## 7.3 单元测试

7.3.1 单元测试的重点

7.3.2 代码审查

7.3.3 计算机测试

## 7.3 单元测试

### 7.3.1 单元测试的重点

### 7.3.2 代码审查

### 7.3.3 计算机测试

## 7.3.1 单元测试的重点

### 测试重点1. 模块接口

- 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。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。

### 测试重点2. 局部数据结构

- 对模块来说，**局部数据结构**是常见的错误来源。主要检查内容包括**局部数据说明、初始化、默认值**等方面的错误。

### 测试重点3. 重要的执行通路

- 由于不可能进行穷尽测试，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。

## 7.3.1 单元测试的重点

### 测试重点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。应该着重测试下述一些可能发生的错误：
  - (1) 对错误的描述难以理解；
  - (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。

### 测试重点5. 边界条件

- 边界测试在单元测试中非常重要。软件常常在它的边界上失效。应使用刚好小于、刚好等于和刚好大于最大值或最小值的测试方案，非常可能发现软件中的错误。

## 7.3 单元测试

7.3.1 单元测试的重点

7.3.2 代码审查

7.3.3 计算机测试

## 7.3.2 代码审查 (人工审查)

### 代码审查

由审查小组人工测试源程序称为代码审查。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。

#### 审查小组组成:

- 组长，一个有能力的程序员，且没有直接参与这项工程的发展；
- 程序的设计者；
- 程序的编写者；
- 程序的测试者。

#### 审查的步骤:

- ① 小组成员先研究设计说明书，力求理解这个设计。
- ② 由设计者扼要地介绍他的设计。
- ③ 审查会上程序的编写者逐个语句地解释是怎样用程序代码实现这个设计的。
- ④ 审查会上对照程序设计常见错误，分析审查这个程序。
- ⑤ 当发现时，记录错误，继续审查。



## 7.3.2 代码审查——方法

### 1. 讨论：

- 是由一些有经验的测试人员阅读程序文本及有关文档，对程序的**结构与功能**、**数据的数据**、**接口**、**控制流**以及**语法**进行讨论和分析，从而揭示程序中的错误。

### 2. 走查：

- 是由测试人员用一些测试用例沿程序逻辑运行，并随时记录程序的踪迹；然后进行分析，发现程序中的错误。（**人工模仿计算机执行程序**）

## 7.3 单元测试

7.3.1 单元测试的重点

7.3.2 代码审查

7.3.3 计算机测试

## 7.3.3 计算机测试

### 1. 模块的“计算机测试”

- 在这里是指使用计算机系统对“模块”作为独立测试对象进行测试。
- 由于模块并不是一个独立的程序，要运行它就必须为其开发驱动软件和存根（桩）软件。

### 2. 模块测试的“驱动程序”

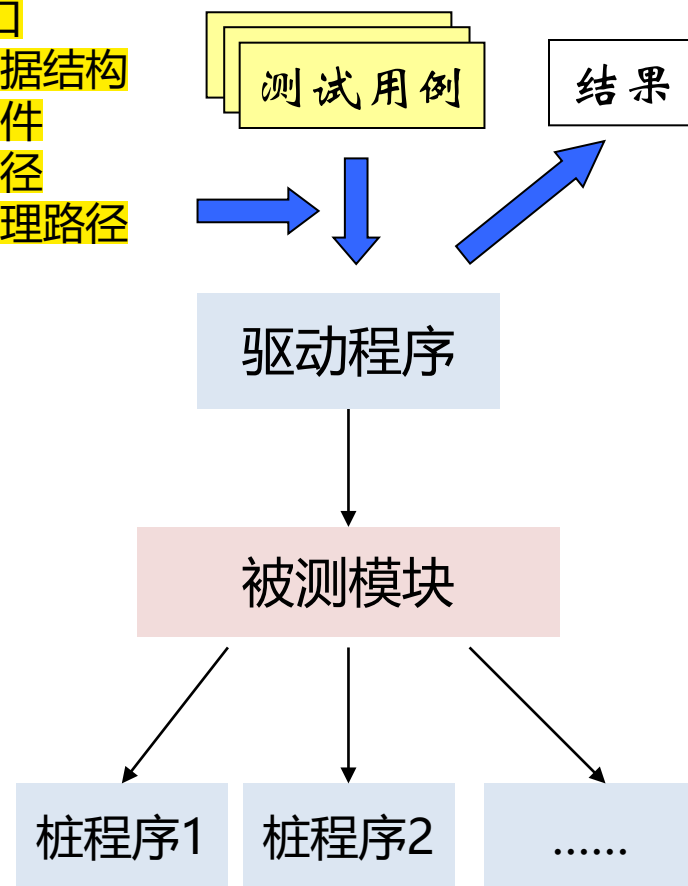
- “驱动程序”就是一个“主程序”，它接收测试数据，把这些数据传送给被测试的模块，并且印出有关的结果。

### 3. 模块测试的“存根程序”

- 存根程序（或称为“桩程序”）代替被测试的模块所调用的模块，也称为“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。

测试内容：

- I/O接口
- 局部数据结构
- 边界条件
- 程序路径
- 错误处理路径



单元测试的环境

### 7.3.3 计算机测试——正文加工系统的模块测试

#### 1. 测试任务:

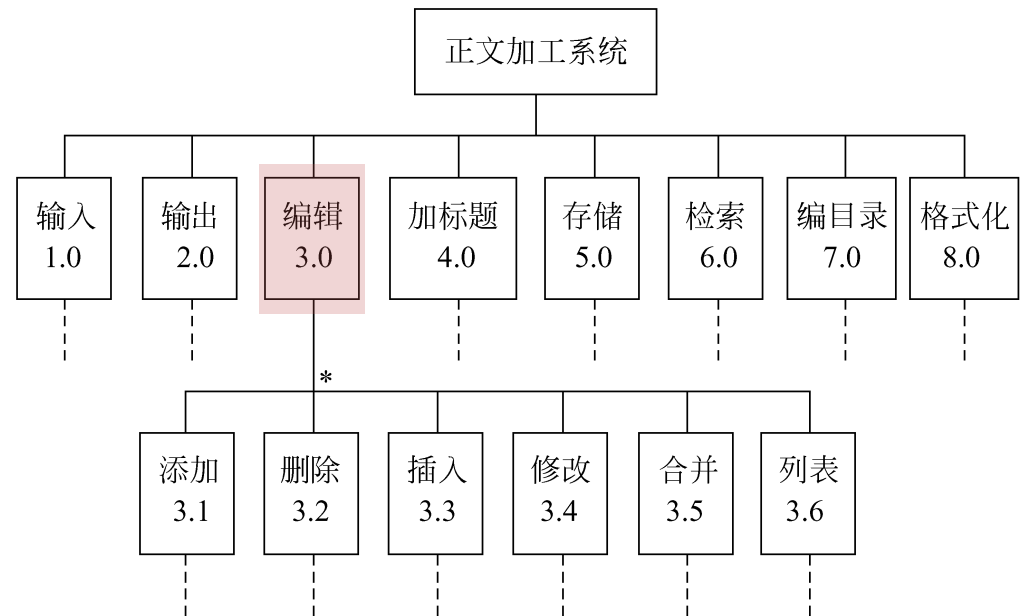
- 假定要测试正文加工系统中编号为3.0的“编辑”模块。

#### 2. 驱动程序: TEST DRIVER

- 说明必要的变量
- 接收测试数据(字符串)
- 调用被测模块“编辑”功能
- 并且设置正文编辑模块的编辑功能(如“修改”、“添加”等)。

#### 3. 存根程序: TEST STUB

- 简化地模拟下层模块, 完成具体的编辑功能(如“修改”、“添加”等)。
- “修改”和“添加”可以做成二个存根程序, 也可只用一个存根程序模拟正文编辑模块的所有下层模块。



正文加工系统的层次图

### 7.3.3 计算机测试

- ① 代码审查是人工测试，一次审查会上可以发现许多错误，但难以发现编码中的出错细节，一般用于模块的初步测试。
- ② 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。
- ③ 计算机测试需要开发驱动程序和存根程序，增加了一些系统开发工作量。可在集成测试中采用增量测试法来同时完成模块测试。
- ④ 人工测试和计算机测试互相补充，相辅相成，缺少其中任何一种方法都会使查找错误的效率降低。

# 第七章 实现(编码与测试)

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

7.4.1 自顶向下集成

7.4.2 自底向上集成

7.4.3 不同集成测试策略的比较

7.4.4 回归测试

## 7.4 集成测试

### 7.4.1 自顶向下集成

### 7.4.2 自底向上集成

### 7.4.3 不同集成测试策略的比较

### 7.4.4 回归测试



## 7.4 集成测试概念及目标

### 集成测试

- 把模块按照设计要求组装起来的同时进行测试，目的是发现模块接口的错误。集成测试是测试和组装软件的系统化技术。

集成测试的其主要目标是发现与接口有关的问题，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。

## 7.4 集成测试——模块组装成系统的方法

### 1. 非渐增式测试方法

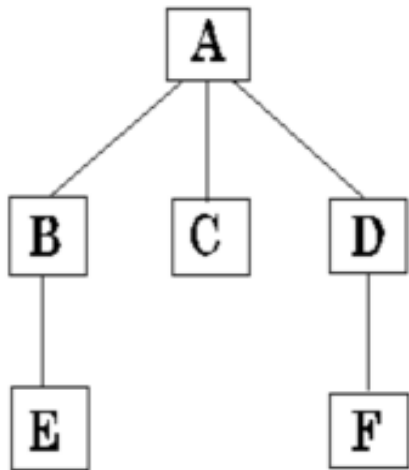
- 非渐增式测试（整体拼装）先分别测试每个模块，再把所有模块按设计要求放在一起结合成所要的程序进行测试，最终得到要求的软件系统。

### 2. 渐增式测试

- 把下一个要测试的模块同已经测试好的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。
- 在组装过程中边连接边测试，以发现连接过程中产生的问题。
- 通过增殖逐步组装成为要求的软件系统。这种每次增加一个模块的方法实际上同时完成单元测试和集成测试。

在进行集成测试时普遍采用渐增式测试方法。

## 7.4 集成测试——非渐增式测试



(a)软件结构

## 7.4 集成测试——渐增式测试

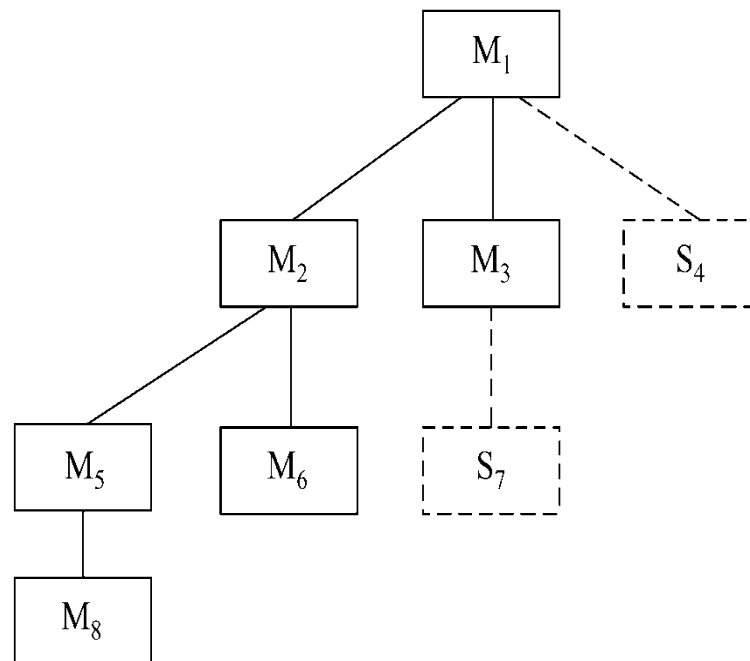
以渐增方式把模块结合到程序中，有两种集成策略：

1. 自顶向下集成
2. 自底向上集成

在实践中常采用混合的策略。

## 7.4.1 渐增式集成——自顶向下集成

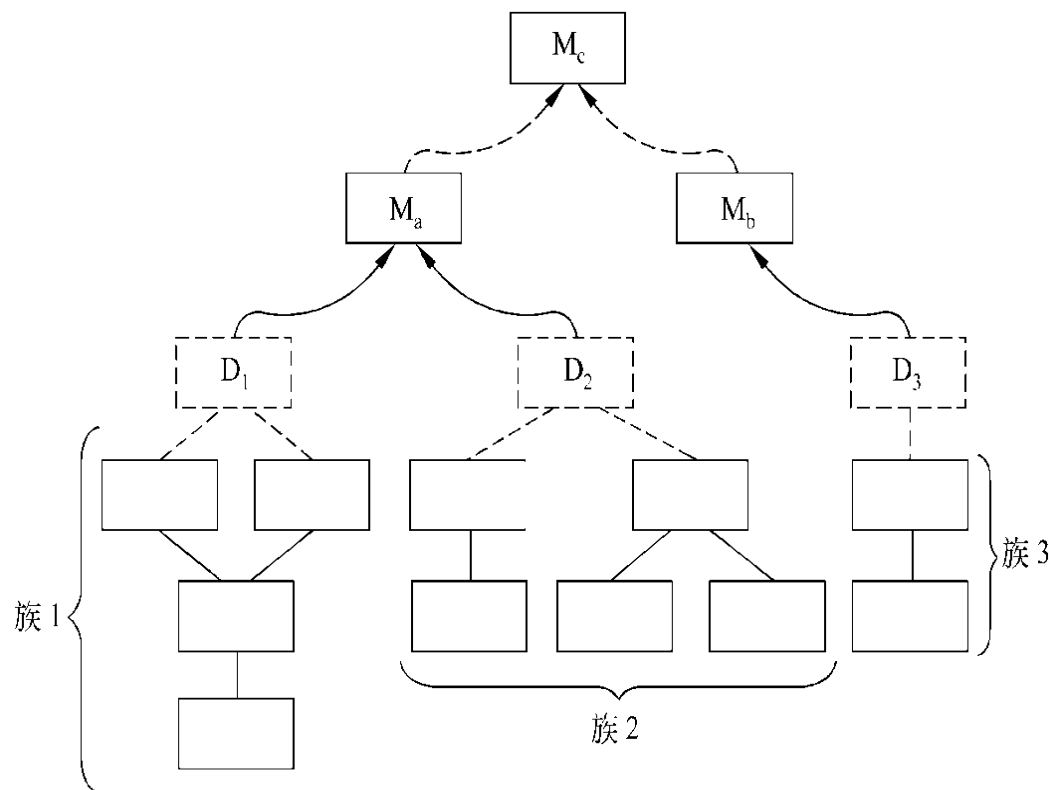
- 自顶向下集成测试：从“主控模块”开始，沿着程序结构的控制层次向下移动，逐渐将各个模块结合（组装）进来，指导所有底层模块。
- 模块组装进来的顺序可有**两种策略**：“深度优先”和“宽度优先”。
- **自顶向下测试需要开发“存根程序”**。



深度（宽度）优先组装，  
需要存根程序

## 7.4.2 渐增式集成——自底向上集成

- 自底向上测试从“底层模块”（即叶子模块）开始，向上移动，逐步将其上层模块组装进来，直到主控模块。
- 同样，模块组装进来的顺序可有二种策略：“深度优先”和“宽度优先”。
- 自底向上测试需要开发“启动模块”



自底向上组装，需要驱动程序

## 7.4.3 回归测试

- ◆ 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。

回归测试是指重新执行已经做过的测试的某个子集，以保证修改变化没有带来非预期的副作用。

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) 针对被修改过的软件成分的测试；
- (3) 专门针对可能受修改影响的软件功能的附加测试。

# 第七章 实现(编码与测试)

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

7.5.1 确认测试的范围

7.5.2 软件配置复查

7.5.3 Alpha和Beta测试

## 7.5 确认测试

### 7.5.1 确认测试的范围

### 7.5.2 软件配置复查

### 7.5.3 Alpha和Beta测试

## 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。
- 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。

## 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**。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**。
- 确认测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）。

## 7.5 确认测试

### 7.5.1 确认测试的范围

### 7.5.2 软件配置复查

### 7.5.3 Alpha和Beta测试

## 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。软件配置指软件需求规格说明、软件设计规格说明、源代码等。

确认测试过程中还应该遵循用户指南及其他操作程序，检验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

7.5.1 确认测试的范围

7.5.2 软件配置复查

7.5.3 Alpha和Beta测试

## 7.5.3 Alpha和Beta测试

### 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。该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。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，并且定期把这些问题报告给开发者。



## 7.5.3 Alpha和Beta测试



# 如何实施测试？

**关键技术** ---- 设计测试方案。

**测试方案** ---- 包括：测试目的，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例的**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很重要。**

## 哪一种测试方案好？

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。

设计测试数据的技术各有优缺点，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。

# 第七章 实现(编码与测试)

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

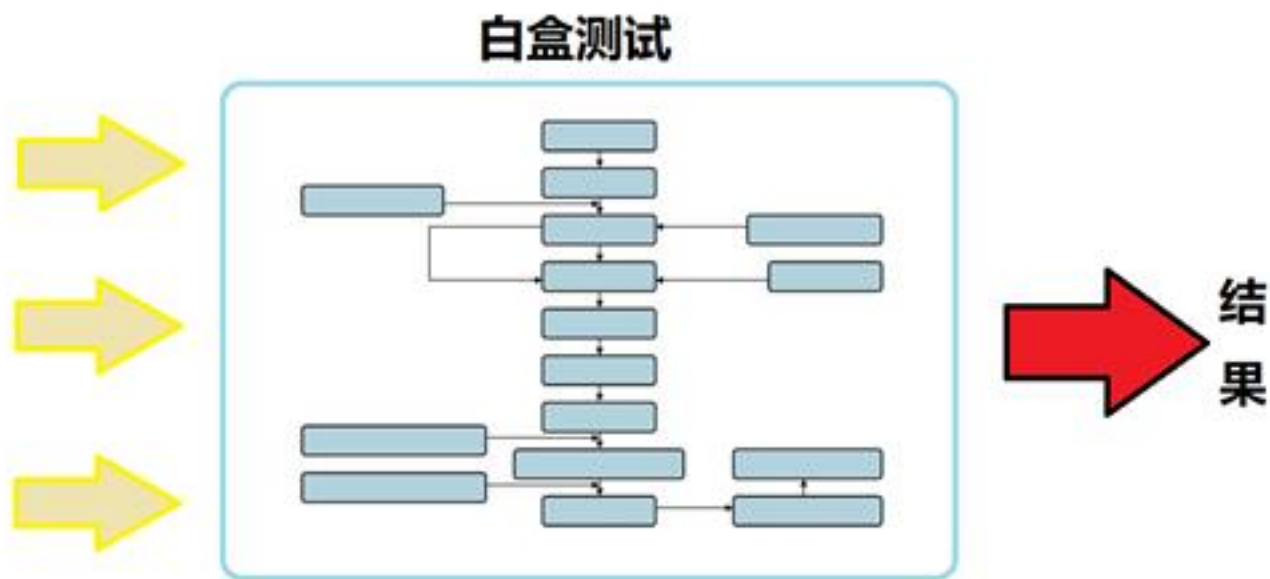
## 7.6 白盒测试技术

### 7.6.1 逻辑覆盖

- 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. 点覆盖
- 7. 边覆盖
- 8. 路径覆盖

### 7.6.2 控制结构测试

- 1. 基本路径测试
- 2. 条件测试
- 3. 循环测试



## 7.6.1 逻辑覆盖

逻辑覆盖：以程序内部的逻辑结构为基础来设计测试用例。

(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) 点覆盖

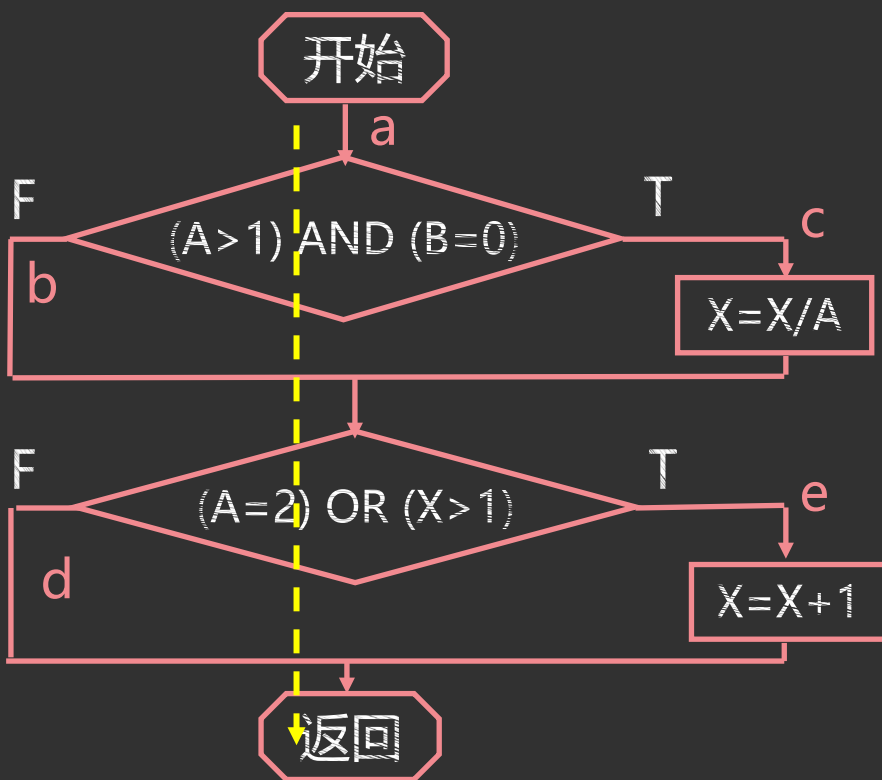
(7) 边覆盖

(8) 路径覆盖



## 7.6 白盒测试技术——逻辑覆盖

(1) 语句覆盖 使程序中每个语句至少执行一次。



只需设计一个测试用例

输入数据: **A=2**

**B=0**

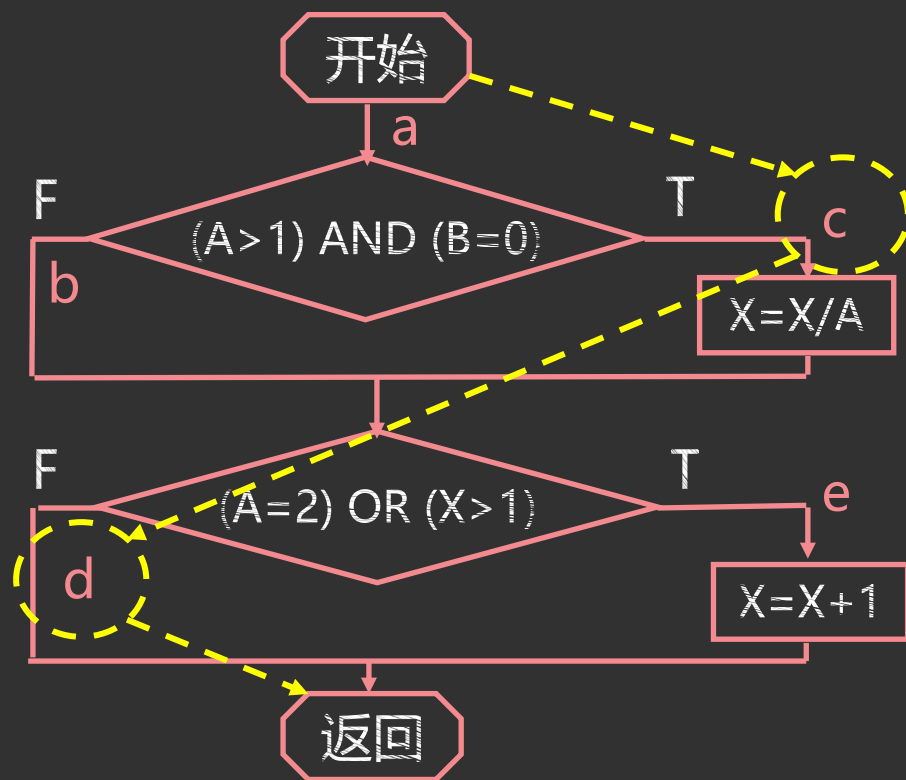
**X=4**

即达到了语句覆盖。

语句覆盖是最弱的逻辑覆盖

## 7.6 白盒测试技术——逻辑覆盖

(2) 判定覆盖 使每个判定的真假分支都至少执行一次。



可设计两组测试用例

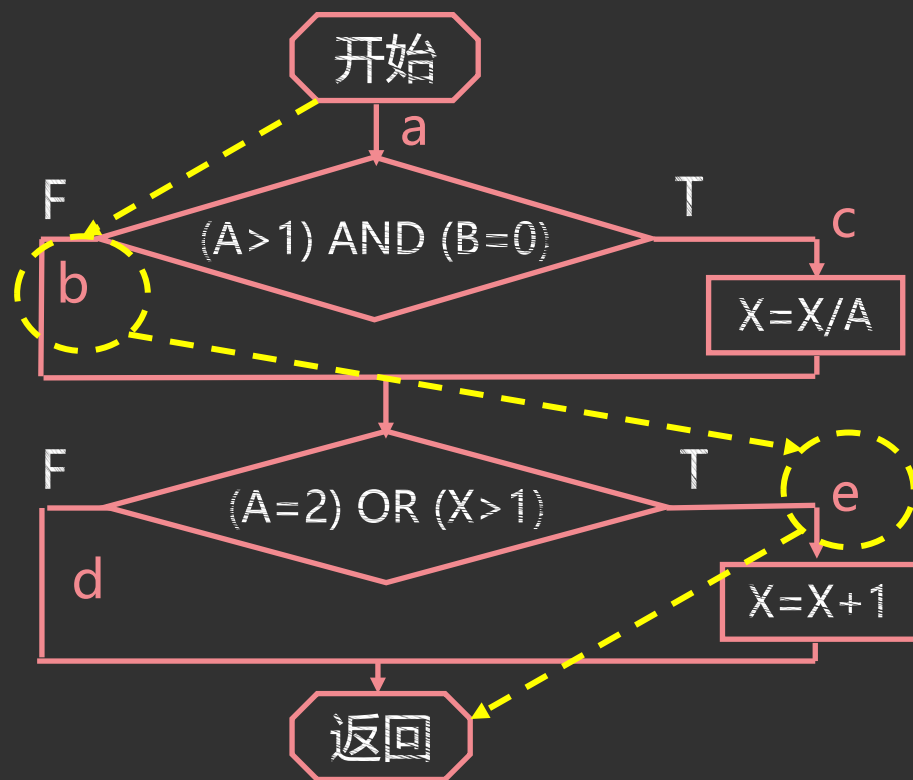
①  $A=3, B=0, X=3$

两组测试用例可覆盖所有判定的真假分支



## 7.6 白盒测试技术——逻辑覆盖

(2) 判定覆盖 使每个判定的真假分支都至少执行一次。



可设计两组测试用例

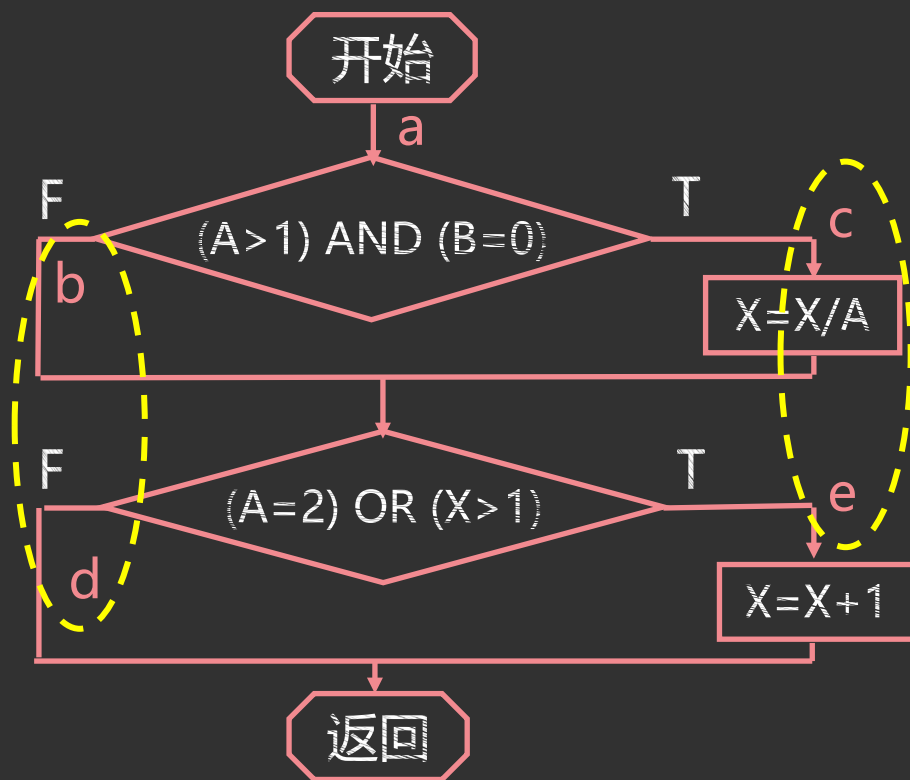
① **A=3, B=0, X=3**

② **A=2, B=1, X=1**

两组测试用例可覆盖所有判定的真假分支

## 7.6 白盒测试技术——逻辑覆盖

(2) 判定覆盖 使每个判定的真假分支都至少执行一次。



可设计两组测试用例

**A=3, B=0, X=3**

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

两组测试用例可覆盖所有判定的真假分支

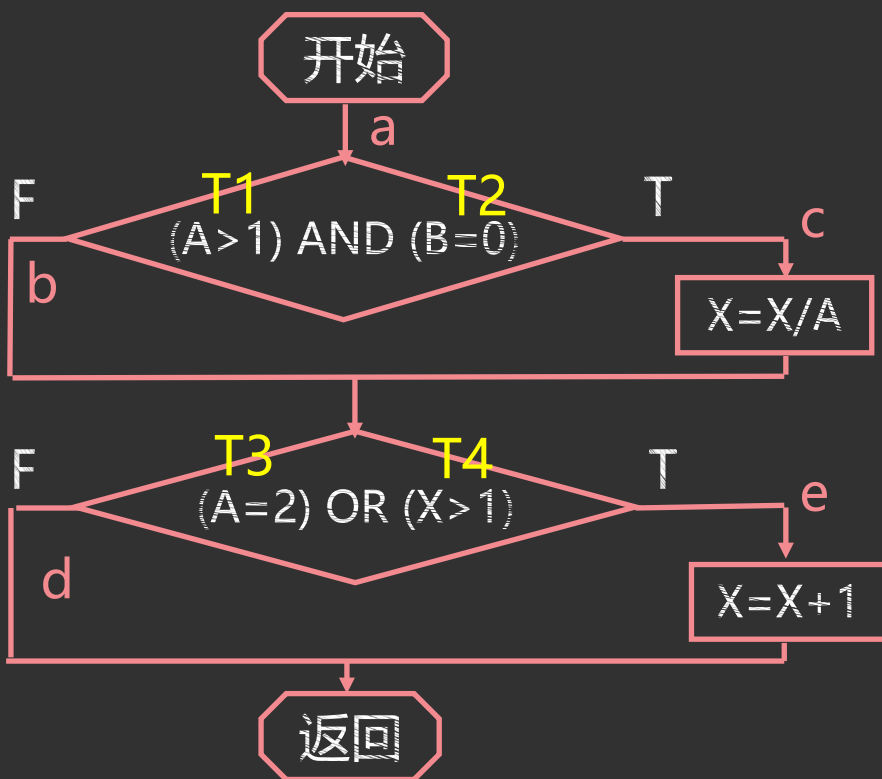
没有覆盖b, d 和 c, e 分支

判定覆盖仍是比较弱的逻辑覆盖, 因为它只覆盖了全部路径的一半。

满足条件:  $T1, \overline{T1},$   
 $T2, \overline{T2}$   
 $T3, \overline{T3}$   
 $T4, \overline{T4}$

## 7.6 白盒测试技术——逻辑覆盖

(3) 条件覆盖 使每个判定的 每个条件的 可能取值 至少执行一次。



第一个判定表达式:

设条件  $A>1$  取真 记为  $T1$

假  $\overline{T1}$

条件  $B=0$  取真 记为  $T2$

假  $\overline{T2}$

第二个判定表达式:

设条件  $A=2$  取真 记为  $T3$

假  $\overline{T3}$

条件  $X>1$  取真 记为  $T4$

假  $\overline{T4}$

满足条件:  $T1, \overline{T1},$   
 $T2, \overline{T2}$   
 $T3, \overline{T3}$   
 $T4, \overline{T4}$

## 7.6 白盒测试技术——逻辑覆盖

### (3) 条件覆盖

测试用例	通过路径	满足的条件	覆盖分支
------	------	-------	------

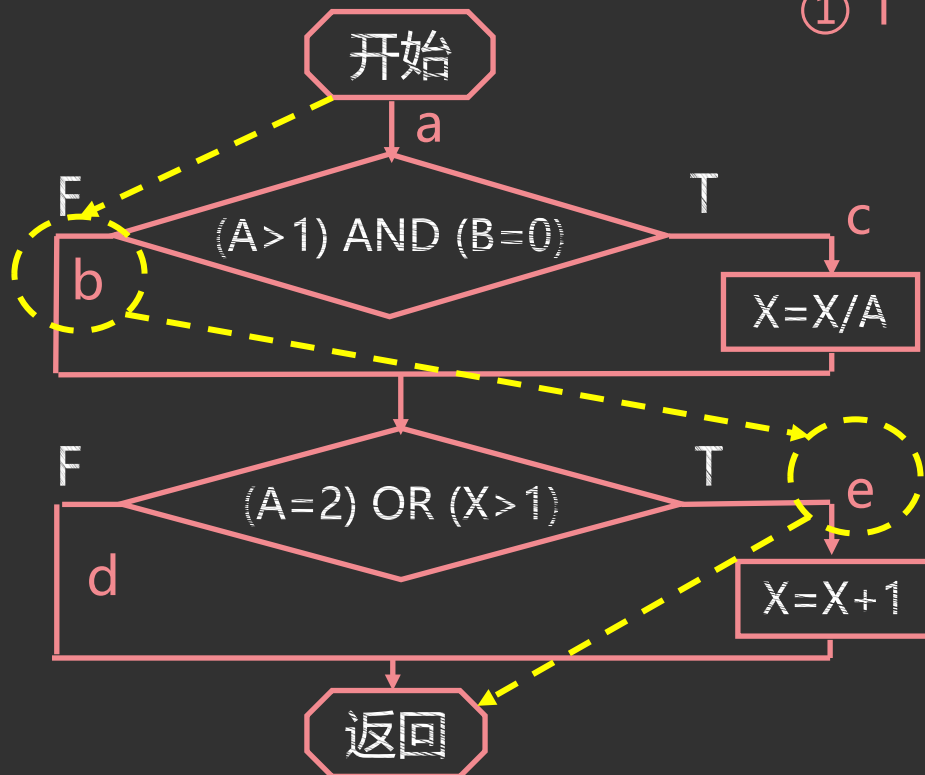
A B X

① 1 0 3

a b e

$\overline{T1}, T2, \overline{T3}, T4$

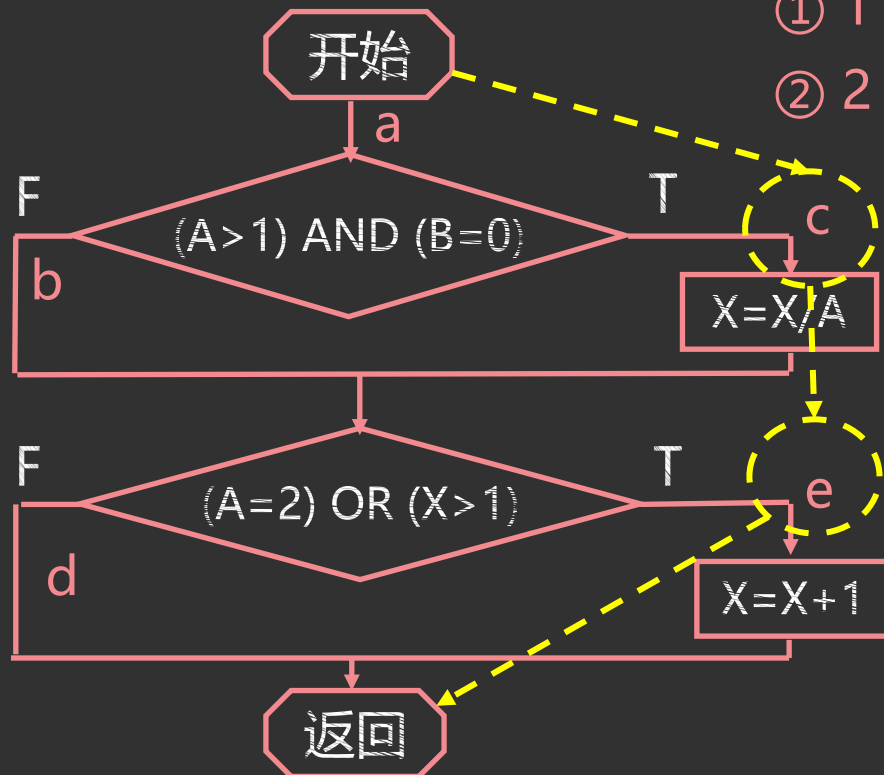
b, e



满足条件:  $T1, \overline{T1}, T2, \overline{T2}, T3, \overline{T3}, T4, \overline{T4}$

## 7.6 白盒测试技术——逻辑覆盖

### (3) 条件覆盖



测试用例	通过路径	满足的条件	覆盖分支
A B X			
① 1 0 3	a b e	$\overline{T1}, T2, \overline{T3}, T4$	b, e
② 2 1 1	a c e	$T1, \overline{T2}, T3, \overline{T4}$	c, e

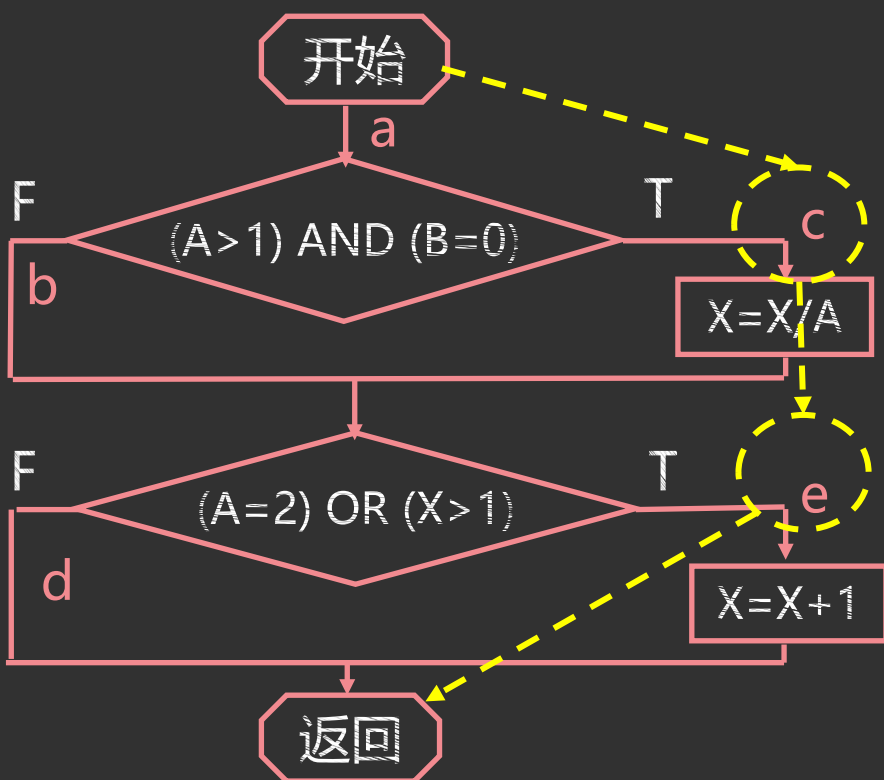
条件覆盖 不一定包含判定覆盖  
判定覆盖 也不一定包含条件覆盖

上述两个测试用例  
覆盖了4个条件, 8种可能取值  
但是没有覆盖 c、d 分支,  
所以不满足判定覆盖的要求.

## 7.6 白盒测试技术——逻辑覆盖

满足条件:  $T1, \overline{T1},$   
 $T2, \overline{T2}$   
 $T3, \overline{T3}$   
 $T4, \overline{T4}$

(4) 判定/条件覆盖 能同时满足判定、条件两种覆盖标准的取值

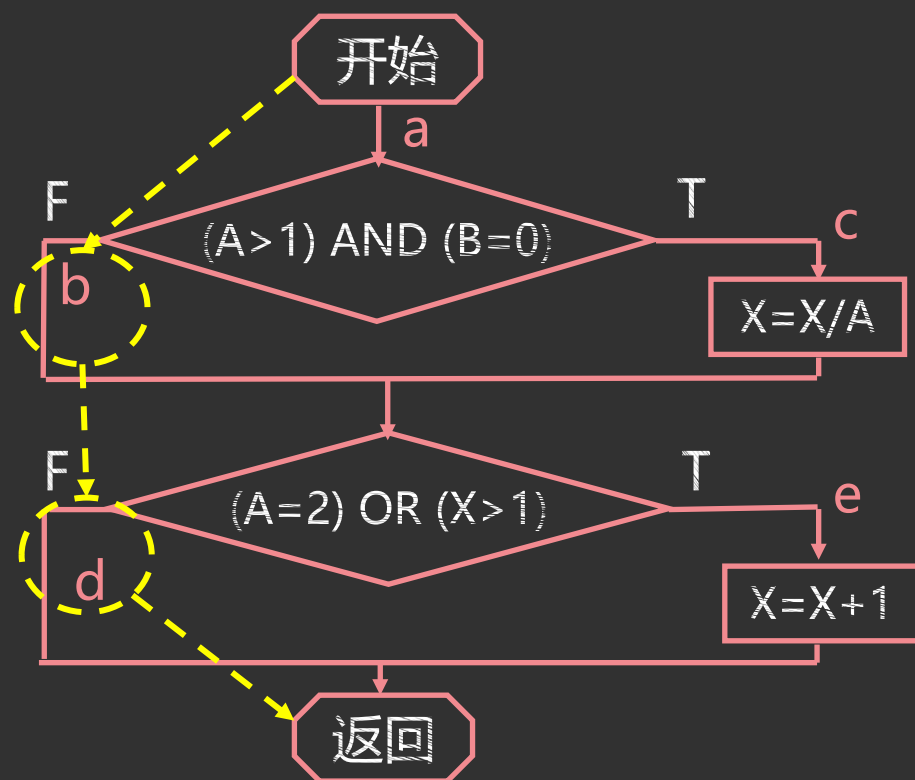


测试用例	通过路径	满足的条件	覆盖分支
A B X			
① 2 0 4	a c e	T1, T2, T3, T4	c, e

## 7.6 白盒测试技术——逻辑覆盖

满足条件:  $T1, \overline{T1},$   
 $T2, \overline{T2}$   
 $T3, \overline{T3}$   
 $T4, \overline{T4}$

(4) 判定/条件覆盖 能同时满足判定、条件两种覆盖标准的取值



测试用例	通过路径	满足的条件	覆盖分支
A B X			
① 2 0 4	a c e	$T1, T2, T3, T4$	c, e
② 1 1 1	a b d	$\overline{T1}, \overline{T2}, \overline{T3}, \overline{T4}$	b, d

没有覆盖 b, e 和 c, d 分支  
 所以判定/条件覆盖 并不比条件覆盖更强

## 7.6 白盒测试技术——逻辑覆盖

(5) 条件组合覆盖 所有可能的条件取值组合至少执行一次

$A > 1, B = 0$

$A > 1, B \neq 0$

$A \neq 1, B = 0$

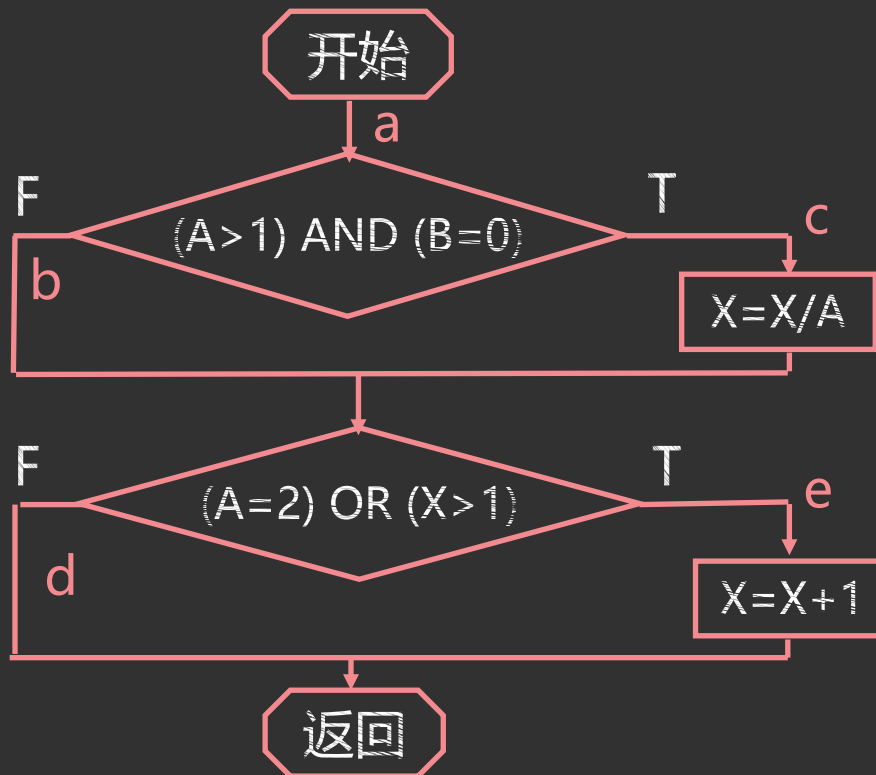
$A \neq 1, B \neq 0$

$A = 2, X > 1$

$A = 2, X \neq 1$

$A \neq 2, X > 1$

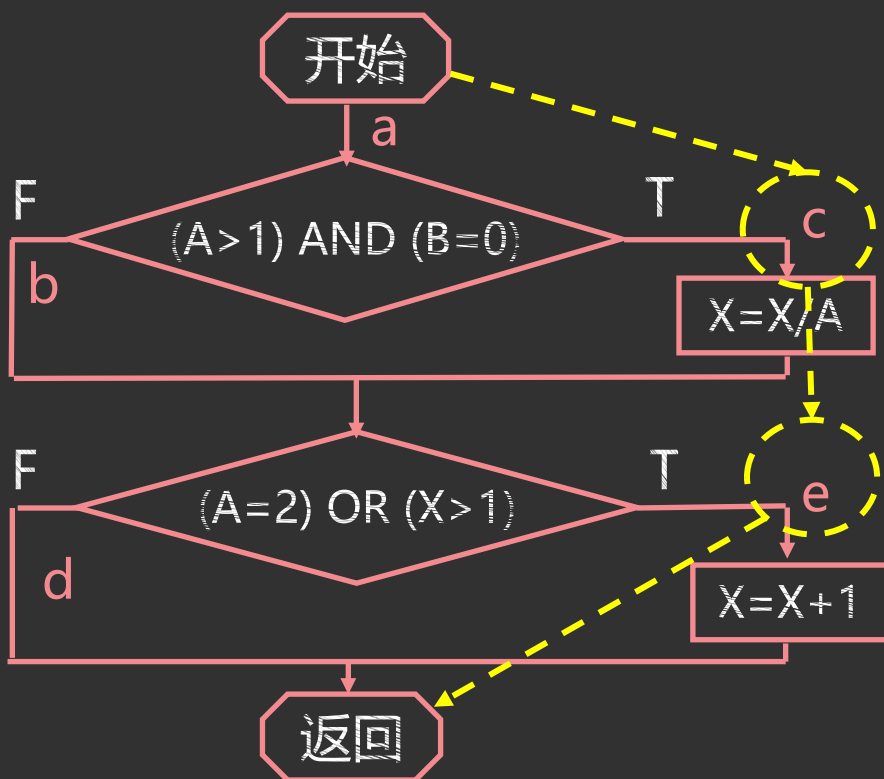
$A \neq 2, X \neq 1$





## 7.6 白盒测试技术——逻辑覆盖

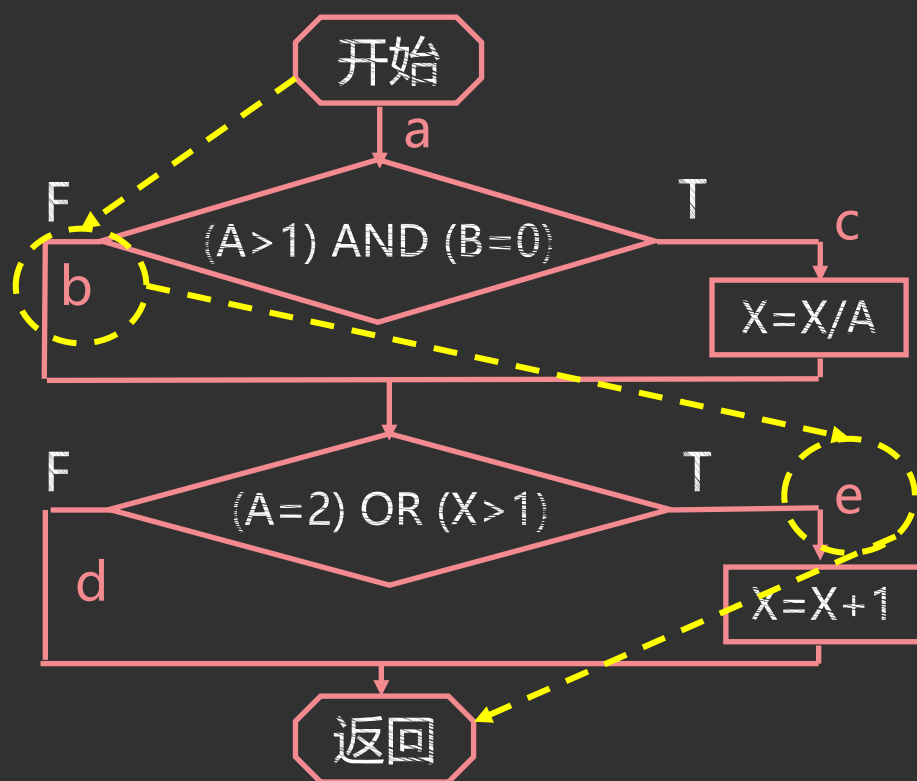
(5) 条件组合覆盖 所有可能的条件取值组合至少执行一次



测试用例	通过路径	满足的条件	覆盖分支
①	2 0 4 a c e	T1, T2, T3, T4	c, e

## 7.6 白盒测试技术——逻辑覆盖

(5) 条件组合覆盖 所有可能的条件取值组合至少执行一次



测试用例 通过路径

A B X

① 2 0 4 a c e

② 2 1 1 a b e

满足的条件

T1, T2, T3, T4

T1,  $\overline{T2}$ , T3,  $\overline{T4}$

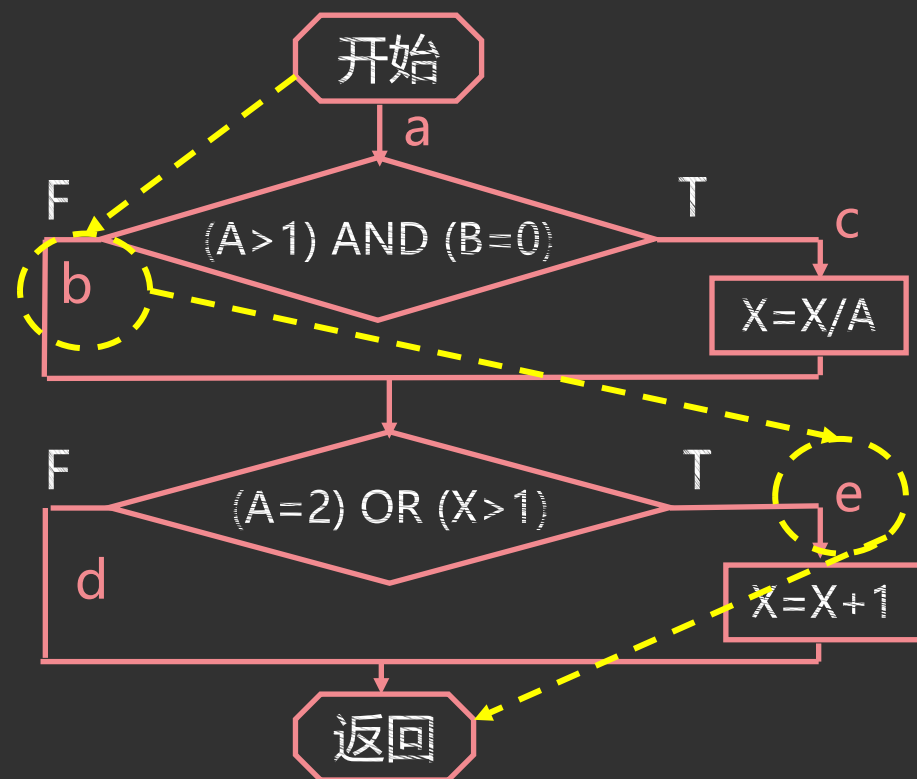
覆盖分支

c, e

b, e

## 7.6 白盒测试技术——逻辑覆盖

(5) 条件组合覆盖 所有可能的条件取值组合至少执行一次



测试用例 通过路径

A B X

① 2 0 4 a c e

② 2 1 1 a b e

③ 1 0 2 a b e

满足的条件

T1, T2, T3, T4

T1,  $\overline{T2}$ , T3,  $\overline{T4}$

$\overline{T1}$ , T2,  $\overline{T3}$ , T4

覆盖分支

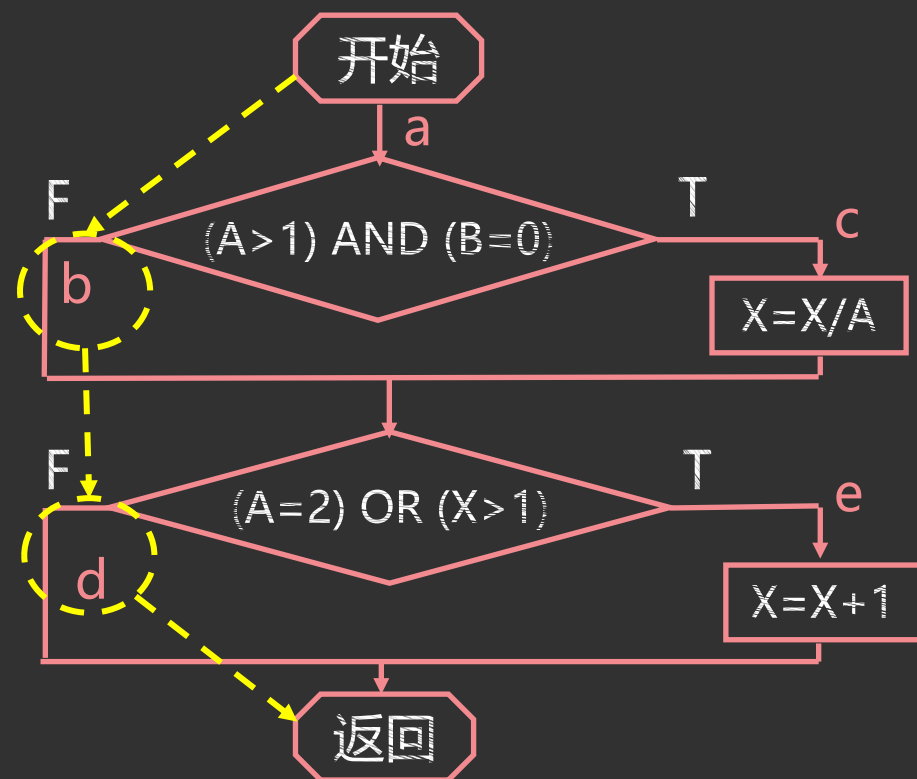
c, e

b, e

b, e

## 7.6 白盒测试技术——逻辑覆盖

(5) 条件组合覆盖 所有可能的条件取值组合至少执行一次



测试用例 通过路径

A B X

① 2 0 4 a c e

② 2 1 1 a b e

③ 1 0 2 a b d

④ 1 1 1 a b d

满足的条件

T1, T2, T3, T4

T1,  $\overline{T2}$ , T3,  $\overline{T4}$

$\overline{T1}$ , T2,  $\overline{T3}$ , T4

$\overline{T1}$ ,  $\overline{T2}$ ,  $\overline{T3}$ ,  $\overline{T4}$

覆盖分支

c, e


b, e

b, e

b, d

## 7.6 白盒测试技术——逻辑覆盖

对源程序语句检测的详尽程度，5种标准

发现错误的 能力	标 准	含 义
	语句覆盖	每条语句至少执行一次
	判定覆盖	每一判定的每个分支至少执行一次
	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
	判定/条件覆盖	同时满足判定覆盖和条件覆盖的要求
	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

### (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。

我们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。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。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)。



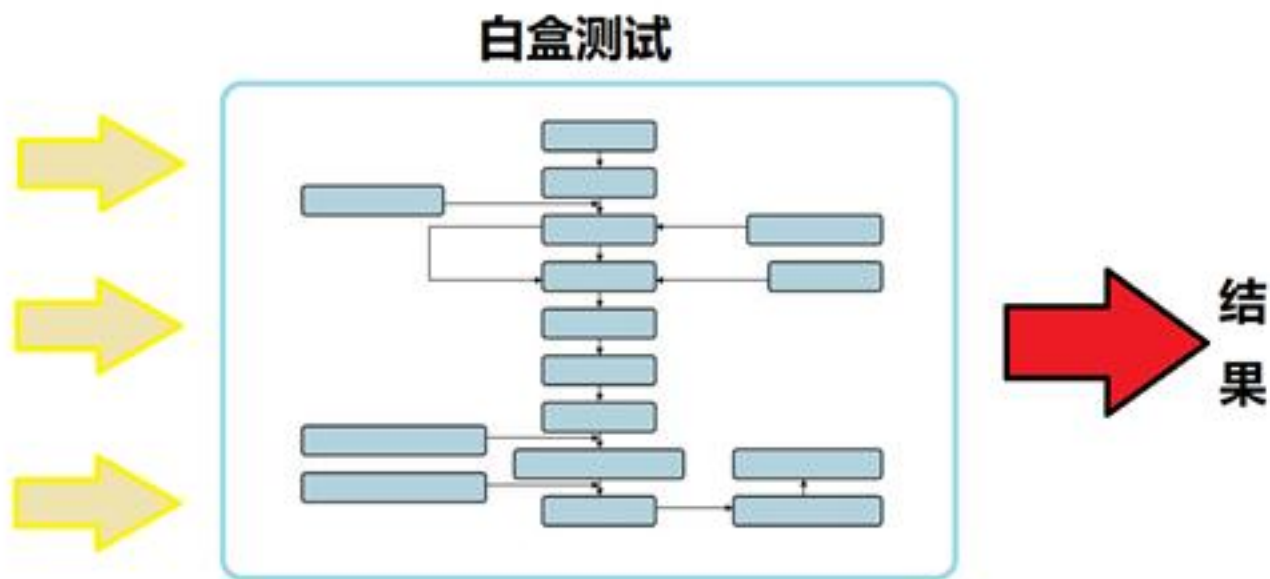
## 7.6 白盒测试技术

### 7.6.1 逻辑覆盖

- 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. 点覆盖
- 7. 边覆盖
- 8. 路径覆盖

### 7.6.2 控制结构测试

- 1. 基本路径测试
- 2. 条件测试
- 3. 循环测试



## 7.6.2 控制结构测试

1. 基本路径测试
2. 条件测试
3. 循环测试

## 7.6.2 控制结构测试

1. 基本路径测试

2. 条件测试

3. 循环测试

# 1.基本路径测试

## 方法概要

- “基本路径测试”是McCabe提出的基于“程序环形复杂度”来确定程序中路径进行测试的一种控制结构测试方法，属于白盒测试技术。
- 设计测试用例时，先计算程序的环形复杂度，以环形复杂度为指南，定义执行路径的基本集合，然后从该基本集合导出测试用例。

## 覆盖能力

- 所设计出的测试用例可保证程序中的每一条语句至少执行一次（语句覆盖），且每个条件语句在执行时分别取真、假值至少一次（判定覆盖、或条件覆盖（如果拆成元条件））。

## 7.6.2 控制结构测试——基本路径测试步骤

**STEP1:** 根据过程设计结果转化成程序相应的流图

**STEP2:** 计算程序流图的环路复杂度

- 区域数目
- 边、节点数目
- 判定节点数目

**STEP3:** 确定线性独立路径的基本集合

- 按照图论，一条独立路径至少包含有一条在其他独立路径中从未有过的边。
- 在程序流图中，一条独立路径至少包含有一条在其他独立路径中从未出现过的  
新语句或新条件。

**STEP4:** 设计可强制执行基本基本集合中每条路径的测试用例

- 根据判断结点给出的条件，选择适当的数据以保证每一条路径可以被测试到——  
用逻辑覆盖（语句覆盖、判定覆盖、条件覆盖、等……）的方法。

# 第一步：画流图

## PROCEDURE average

/\*该过程计算不超过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和个数；\*/

INTERFACE RETURNS average, total.input, total.valid;

INTERFACE ACCEPTS value, minimum, maximum;

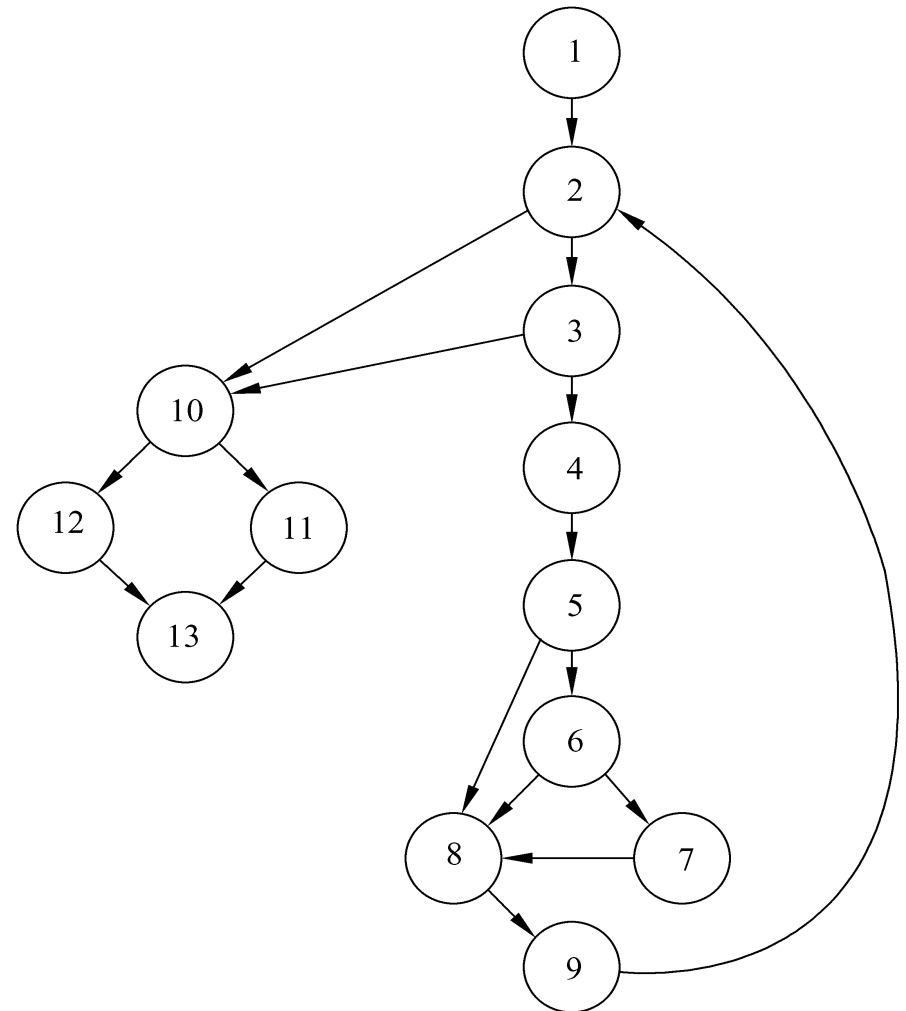
TYPE value[1...100] IS SCALAR ARRAY;

TYPE average, total.input, total.valid;

minimum, maximum, sum IS SCALAR;

TYPE i IS INTERGE;

```
1  i=1;
   total.input=total.valid=0;
   sum=0;
2,3 DO WHILE value[i]<>-999 AND total.input<100
4   increment total.input by 1;
5,6 IF value[i]>minimum 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
```



求平均值过程的PDL  $\longrightarrow$  求平均值过程的流图

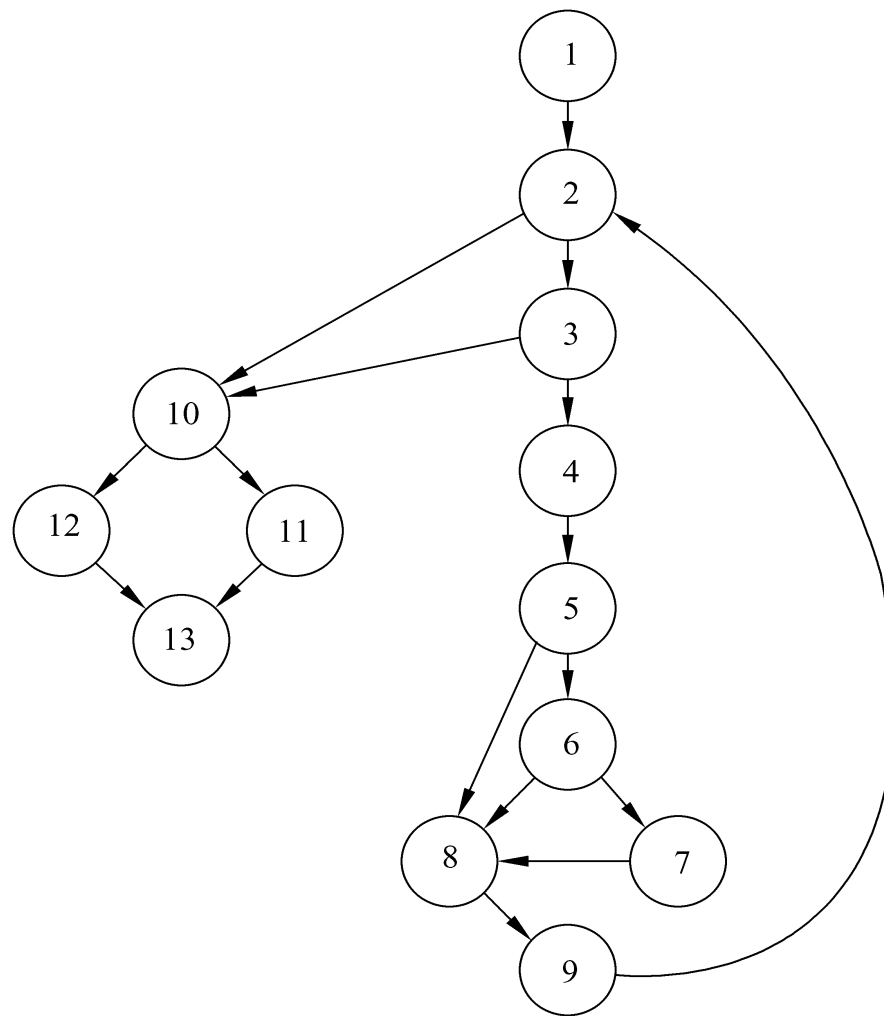
## 第二步:求环形复杂度

环形复杂度

= 边数-节点数+2

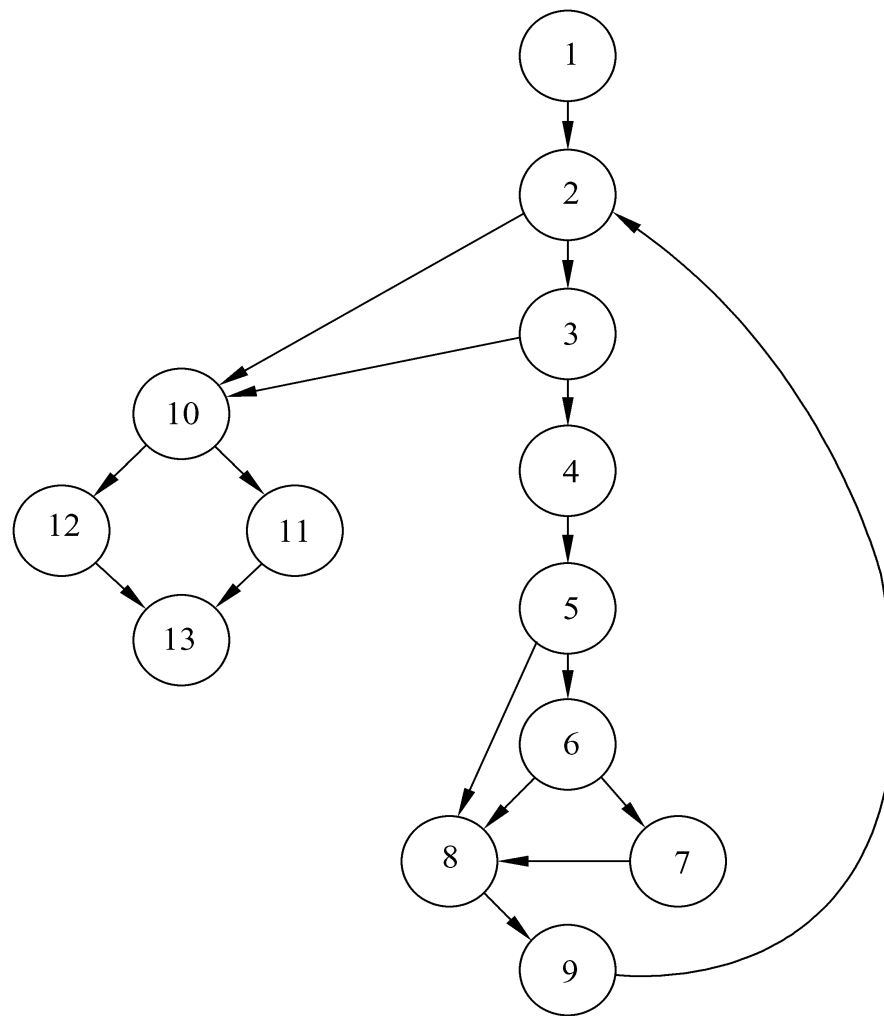
= 17-13+2

= 6



## 第三步:求独立路径集

由于环形复杂度为6，因此最多有6条独立路径：



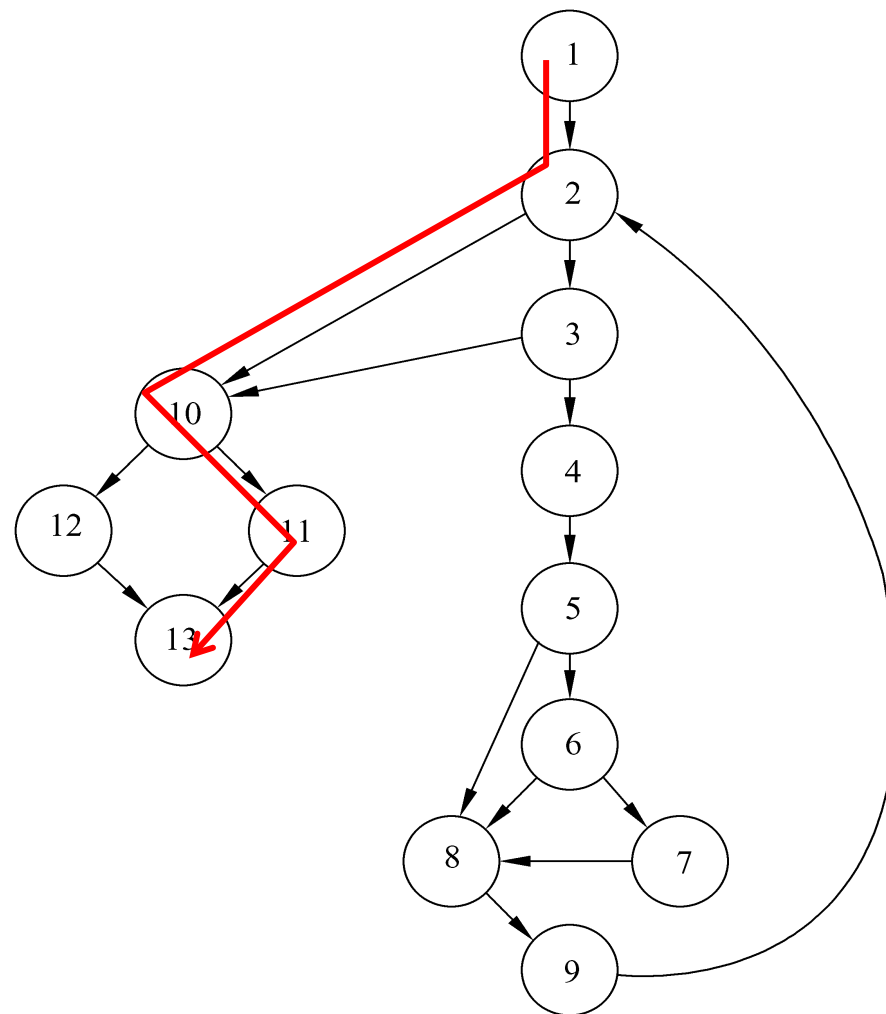
注意：下一条路径应至少包含前面路径中没有出现的一条新边。



## 第三步:求独立路径集

由于环形复杂度为6, 因此最多有6条独立路径:

路径1: 1-2-10-11-13



注意: 下一条路径应至少包含前面路径中没有出现的一条新边。

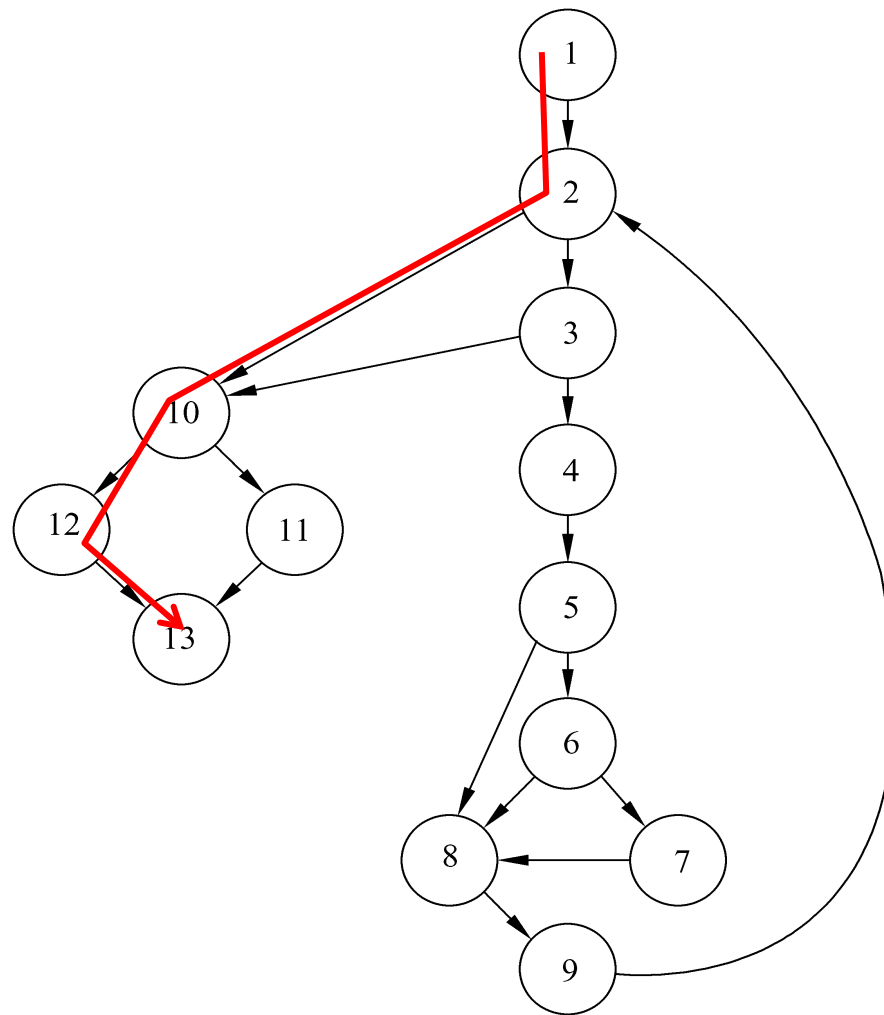
## 第三步:求独立路径集

由于环形复杂度为6，因此最多有6条独立路径：

路径1：1-2-10-11-13

路径2：1-2-10-12-13

注意：下一条路径应至少包含前面路径中没有出现的一条新边。



## 第三步:求独立路径集

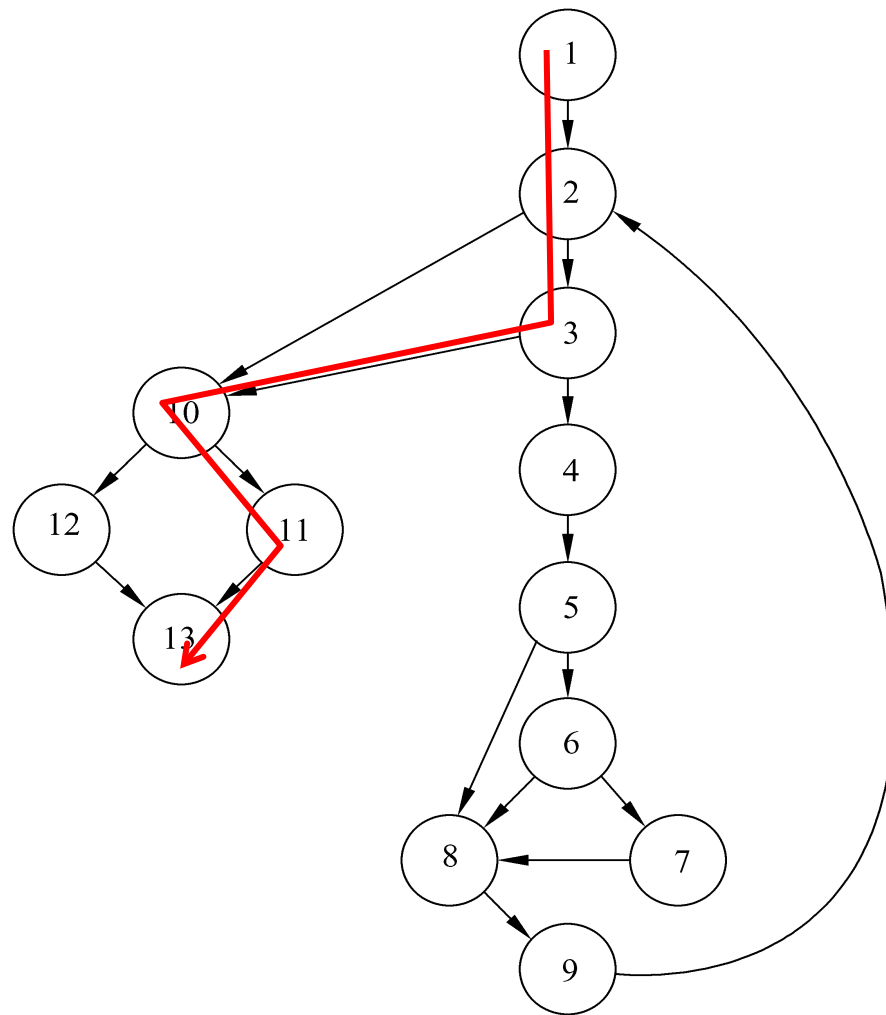
由于环形复杂度为6，因此最多有6条独立路径：

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

注意：下一条路径应至少包含前面路径中没有出现的一条**新边**。



## 第三步:求独立路径集

由于环形复杂度为6，因此最多有6条独立路径：

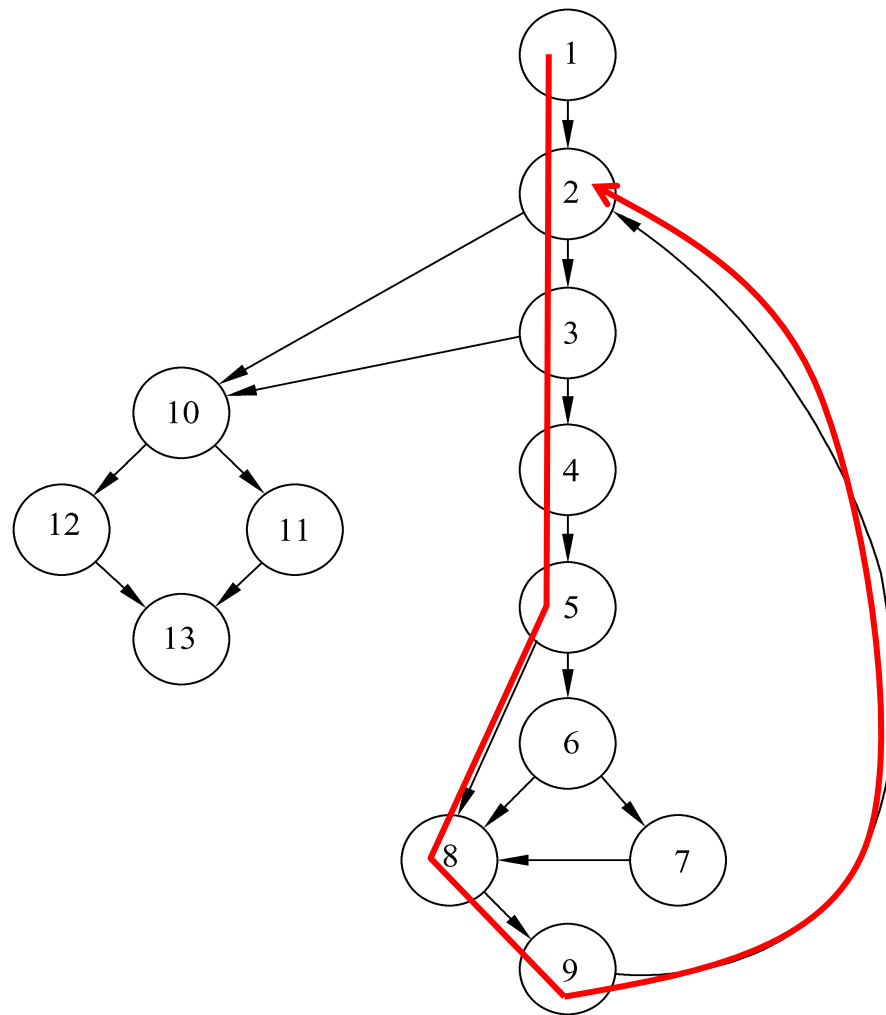
路径1：1-2-10-11-13

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-...

注意：下一条路径应至少包含前面路径中没有出现的一条**新边**。



## 第三步:求独立路径集

由于环形复杂度为6，因此最多有6条独立路径：

路径1：1-2-10-11-13

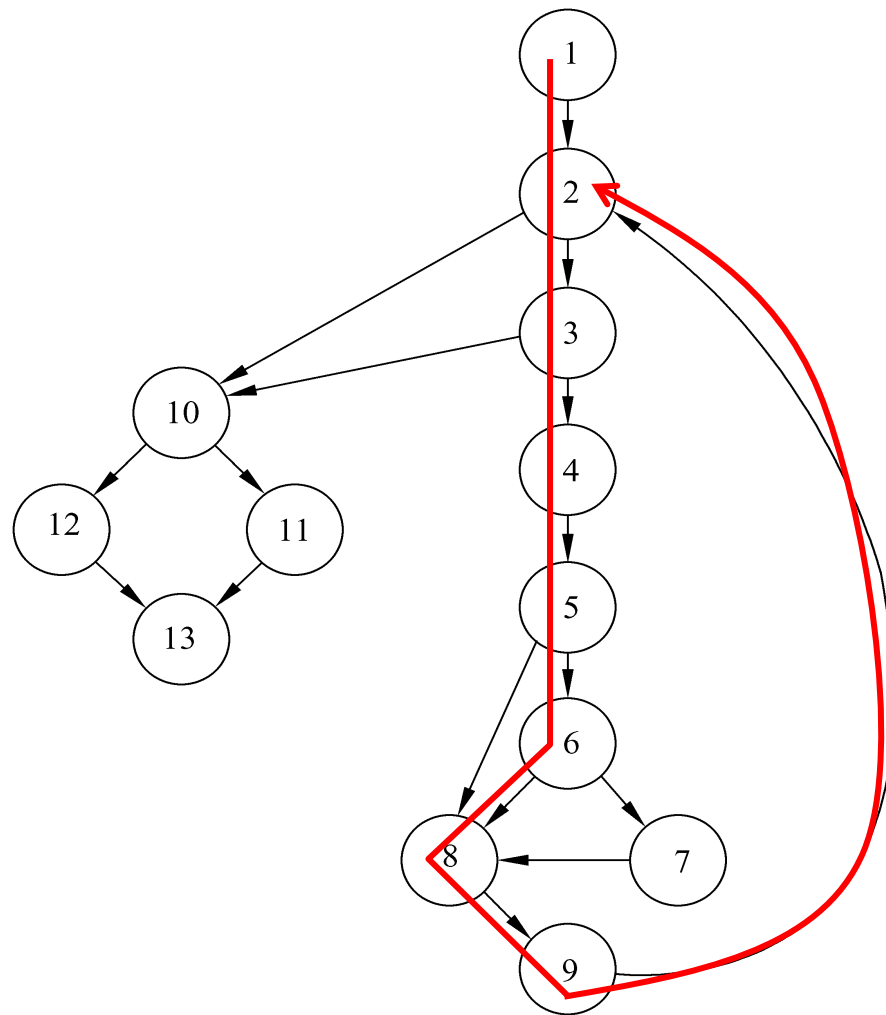
路径2：1-2-10-12-13

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-...

注意：下一条路径应至少包含前面路径中没有出现的一条**新边**。



## 第三步:求独立路径集

由于环形复杂度为6，因此最多有6条独立路径：

路径1：1-2-10-11-13

路径2：1-2-10-12-13

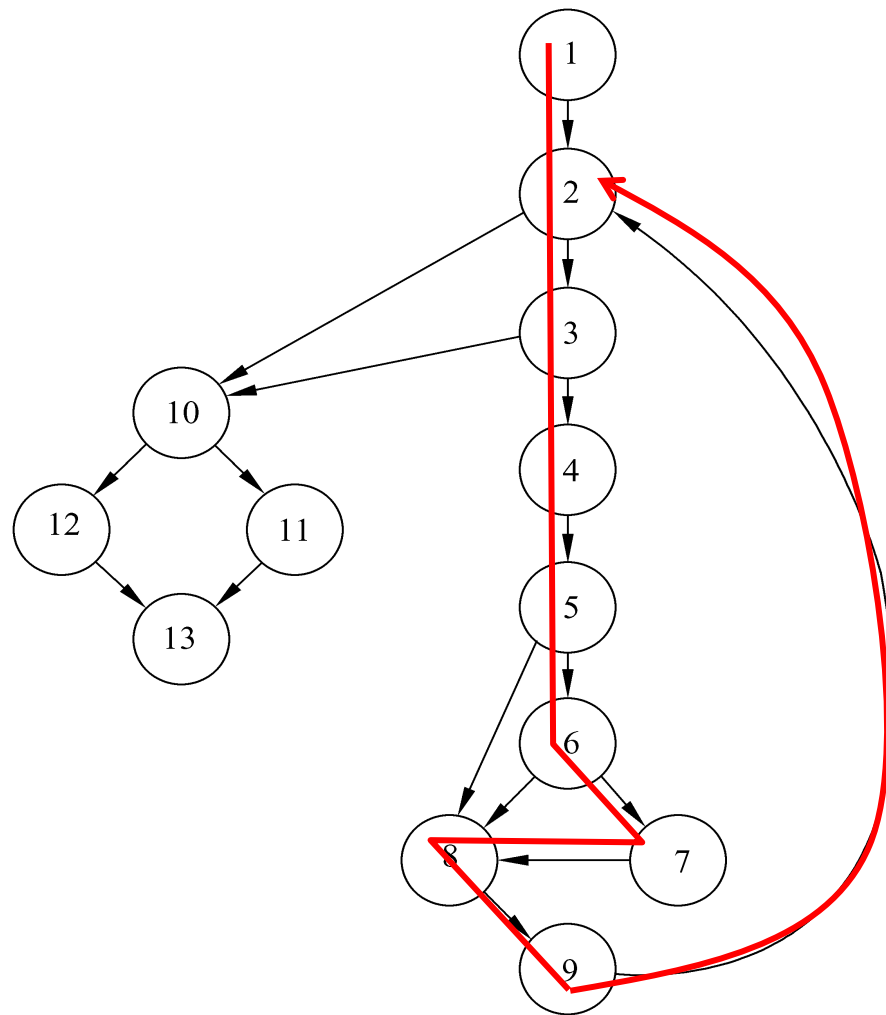
路径3：1-2-3-10-11-13

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-...

注意：下一条路径应至少包含前面路径中没有出现的一条**新边**。



## 第四步：设计测试用例

路径1：1-2-10-11-13

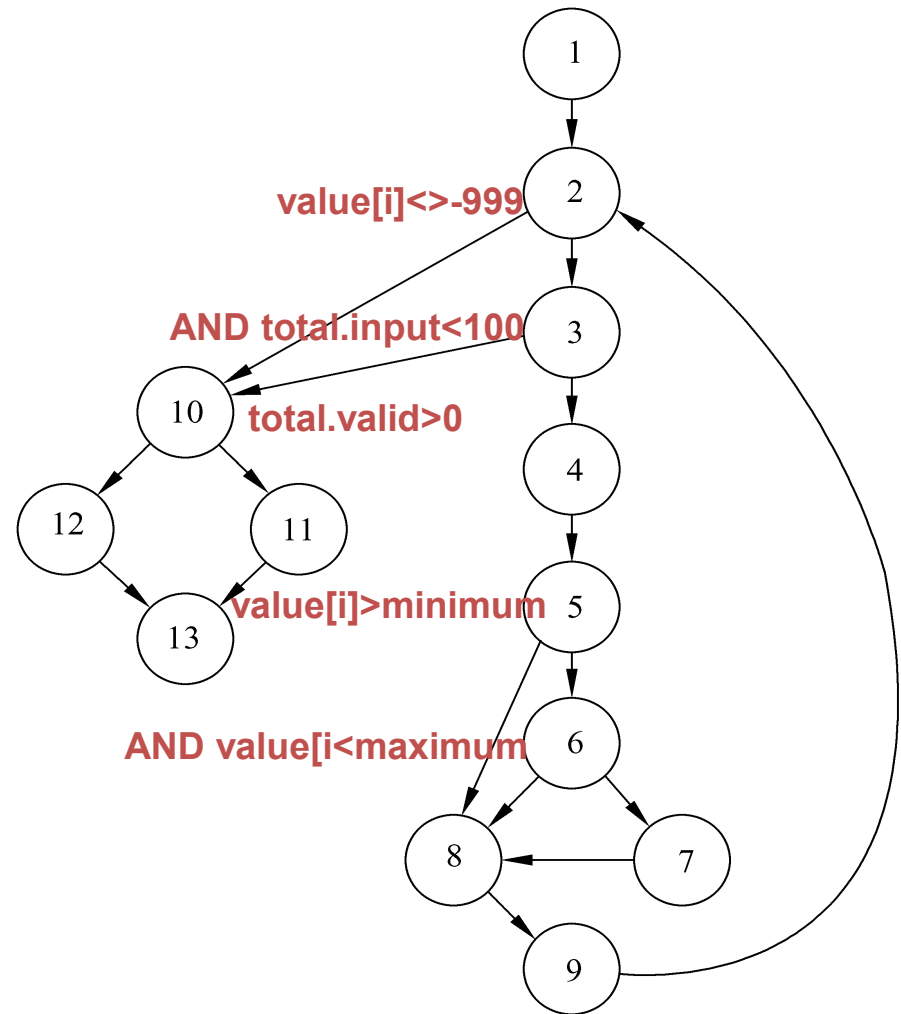
- 输入：  $\text{value}[k]=\text{有效输入}, k < i$ ;  
 $\text{value}[i]=-999$  且  $2 \leq i \leq 100$ .
- 预期结果： **k个有效输入的正确的平均值和总和。**

路径2：1-2-10-12-13

- 输入：  $\text{value}[1]=-999$ , 即 **0个有效输入值。**
- 预期结果：  $\text{average}=-999$ , 其他保持初始值.

路径3：1-2-3-10-11-13

- 输入:  $\text{value}[i]=\text{有效输入} (k \leq 100), 1 \leq i \leq k; k > 100$ , 即有效输入多于100个.
- 预期结果： **前100个有效输入的正确的平均值和总和。**



## 第四步：设计测试用例

路径4：1-2-3-4-5-8-9-2-...

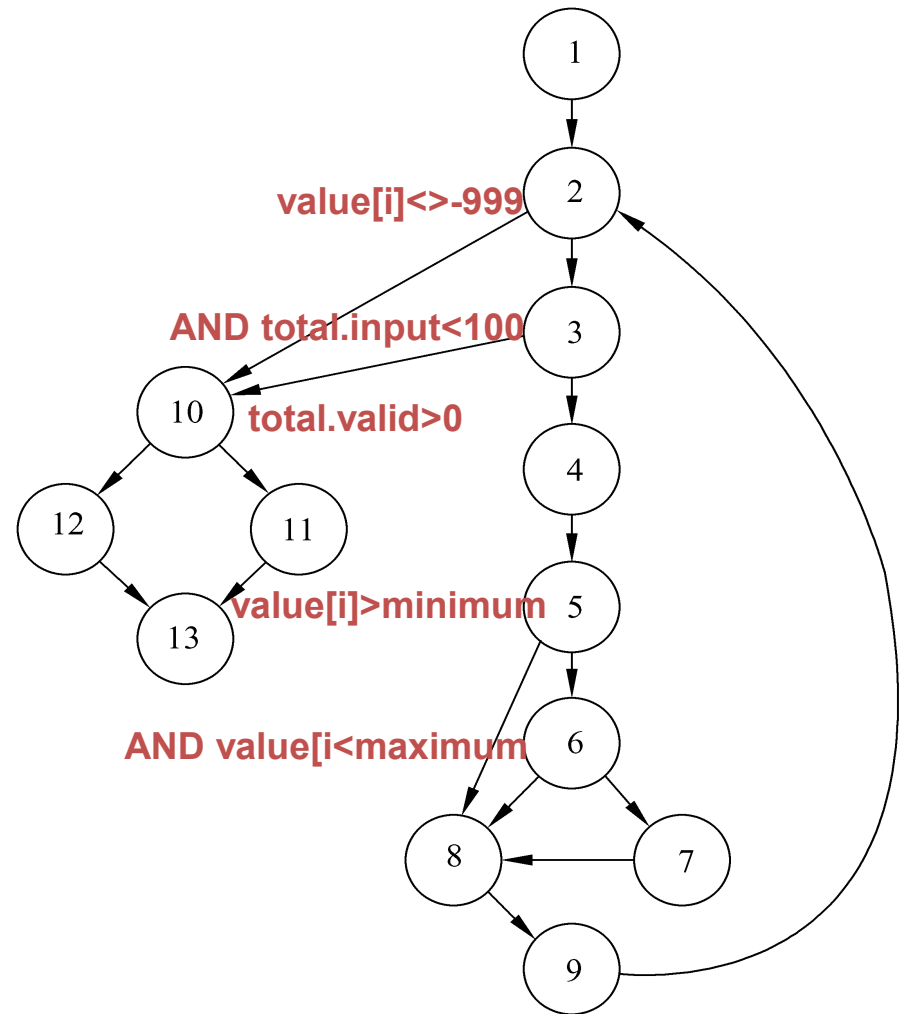
- 输入：value[i]=有效输入( $k \geq \min$ ),  $i < 100$ ;  
value[k]<minimum,  $k < i$ 。即有值过小的无效输入。
- 预期结果：滤除了过小值的正确的平均值和总和

路径5：1-2-3-4-5-6-8-9-2-...

- 输入：value[i]=有效输入( $k \leq \max$ ,  $i < 100$ ;  
value[k]>maximum,  $k < i$ 。即有值过大的无效输入。
- 预期结果滤除了过大值的正确的平均值和总和。

路径6：1-2-3-4-5-6-7-8-9-2-...

- 输入：value[i]=有效输入, ( $\min < k < \max$ ),  
 $i < 100$ ;
- 预期结果：正确的平均值和总和。





## 2. 条件测试

### 方法概述

- 条件测试是基于程序中的条件来设计测试用例的一种控制结构测试方法，是对基本路径测试技术的补充。
- 该方法首先找出被测程序中的所有逻辑条件（分支语句、循环语句中的条件），在将复合条件分解成简单条件（布尔变量、关系表达式），然后设计测试用例覆盖所有简单条件的取值及可能的构成复合条件的组合值。

### 覆盖能力

- 条件测试的着重点是测试程序中的每个条件，测试用例应覆盖每个条件的各种取值和组成复合条件的简单条件的各种组合取值，可实现判定/条件覆盖、条件组合覆盖。

## 7.6.2 控制结构测试

1. 基本路径测试

2. 条件测试

3. 循环测试

## 7.6.2 条件测试——条件表达式概念

### 条件表达式

- 条件表达式由布尔变量、关系表达式、布尔运算符、括号（改变计算优先序）组成，其计算结果的取值为“真”或“假”。又称为逻辑表达式。

### 布尔表达式

- 由布尔变量和布尔运算符构成的表达式（不包含关系表达式），其计算结果的取值为“真”或“假”。
- 布尔常量：“真”（T）、“假”（F）；
- 布尔变量：取值为“真”或“假”的变量。
- 布尔运算符：OR(|)、AND(&)、NOT(¬)等。

### 关系表达式

- 由算术量（常量、变量）、算术表达式、关系运算符构成的表达式，其计算结果的取值为“真”或“假”。
- 关系表达式的一般形式为： $E1 <\text{关系运算符}> E2$ ，其中E1、E2为算术表达式。
- 关系运算符： $<$ 、 $\leq$ 、 $=$ 、 $\neq$ 、 $\geq$ 、 $>$ 。

## 7.6.2 条件测试——条件错误的类型

- 布尔算符错
- 布尔变量错
- 关系算符错
- 算术表达式错
- 括号错

## 7.6.2 条件测试——条件测试策略

### 1. 分支测试

- 对于复合条件C，C的真分支、假分支、C中每个简单条件都至少执行一次。

### 2. 域测试

- 对于每个关系表达式，至少执行3个测试：E1比E2大、E1比E2小、E1与E2相同
- 对于  $n$  个变量的布尔表达式，测试  $n$  个变量的各种取值的可能组合，共有  $2^n$  种。（注意可行性， $n$  太大将导致组合爆炸！）

### 3. BRO测试（Branch and relational operator）

- 分支与关系运算符（BRO）测试对于条件C，分解出简单条件（假定为  $n$  个），针对每个简单条件的输出约束（ $D1$ 、 $D2$ 、... $Dn$ ），设计相应的测试用例。

# BRO测试举例

例1: **C1**:  $B1 \& B2$

- $B1, B2$ 为布尔变量。
- $C1$ 的条件约束形如  $(D1, D2)$  , 其中 $D1$ 和 $D2$ 的值是 $t$ 或 $f$ 。BRO测试策略要求约束集  $\{(t,t), (f,t), (t,f)\}$  由 $C1$ 的执行所覆盖, 如果 $C1$ 由于布尔运算符错误而不正确, 该约束集中至少有一个约束强制 $C1$ 失败。

例2: **C2**:  $B1 \& (E3 = E4)$

- $B1$ 为布尔变量,  $E3, E4$ 为算术运算符。
- $C2$ 的条件约束为  $(D1, D2)$  , 其中 $D1$ 约束是  $T$  或  $F$  ,  $D2$ 约束为关系运算符。因此约束集为  $\{(t,=), (f,=), (t,<), (t,>)\}$  , 此约束集的覆盖率将保证检测 $C2$ 的布尔运算符和关系运算符错误。

例3: **C3**:  $(E1 > E2) \& (E3 = E4)$

- $C3$ 的条件约束为  $(D1, D2)$  , 约束集为  $\{(>,=), (=,=), (<=), (>,<), (>,>)\}$ 。

## 7.6.2 控制结构测试

1. 基本路径测试
2. 条件测试
3. 循环测试

## 7.6.2 控制结构测试——循环测试

### 方法概述

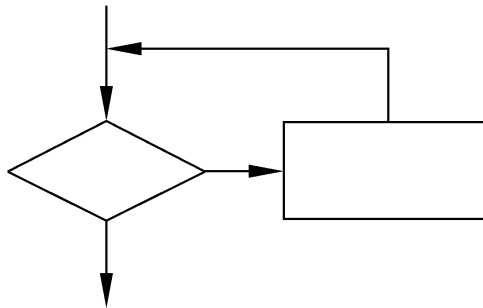
- 循环测试是专注于测试循环结构的有效性的一种白盒测试技术。测试的要点是发现循环次数控制中可能的错误。

### 三种循环组合

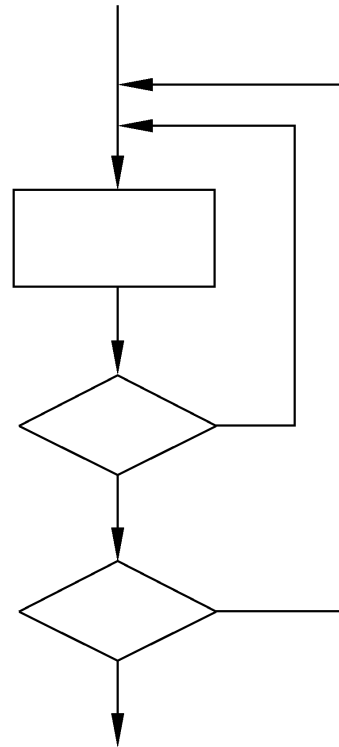
- 简单循环：单个循环
- 嵌套循环：循环中有循环
- 串接循环：一个循环接着又一个循环



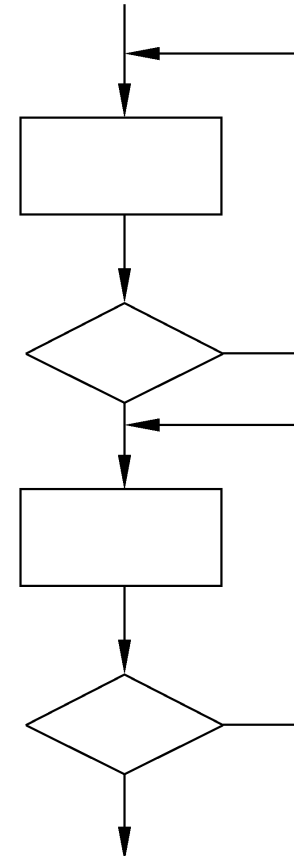
# 循环测试——三种循环



(a) 简单循环



(b) 嵌套循环



(c) 串接循环

# 循环测试——（1）简单循环测试方法

设计测试用例时考虑如下执行情况（假设n为循环的最多次数）：

- 通过循环0次（跳过循环）
- 通过循环1次
- 通过循环2次
- 通过循环m次， $1 < m < n-1$
- 通过循环n-1次
- 通过循环n次
- 通过循环n+1次
- 注意可能的死循环！

# 循环测试——（2）嵌套循环测试方法

## 嵌套循环测试的困难

- 当循环嵌套的层数较多时，采用简单循环测试方法，测试数按几何级数增长，将产生“组合爆炸”，使测试不可行。
- 例如，3层循环至少测试 $7^3=343$ 次。

## 一种减少测试次数的方法

- 从最内层循环开始测试，所有外层循环都设置成最小值（训练1次）；
- 按照简单循环方法对内层循环进行测试，必要时增加一些其他额外测试（如非法值、越界值）；
- 由内向外，对下一个循环进行测试。将该循环的所有外层循环都设置成最小值（循环1次），内层循环都设置成典型值（循环次数为1与最大值之间），
- 重复上述操作，直到最外层循环。

# 循环测试——（3）串接循环测试方法

根据上、下串接的循环之间的关系，采用不同的策略：

- 上、下循环之间没有关系，彼此独立，则各循环各自按照简单循环测试方式进行测试。
- 上一循环的计数器（或控制条件）影响下一循环的计数器（或控制条件），则采用嵌套循环测试方法进行测试。

# 第七章 实现(编码与测试)

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

### 7.7.1 等价划分

### 7.7.2 边界值分析

### 7.7.3 错误推测

- 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。

## 7.7 黑盒测试技术——黑盒测试力图发现的错误类型

1. 功能不正确或遗漏了的功能;
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。

# 设计黑盒测试方案时应考虑的问题

- 怎样测试功能的有效性？
- 哪些类型的输入可构成好的测试用例？
- 系统是否对特定的输入值特别敏感？
- 怎样划定数据类的边界？
- 系统能够承受什么样的数据率和数据量？
- 数据的特定组合将对系统运行产生什么影响？



# 黑盒测试技术测试用例的设计原则

## 减少测试用例数量

- 所设计出来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。

## 测出某些类型的错误

- 所设计出来测试用例能够告知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 黑盒测试的主要技术

7.7.1 等价划分

7.7.2 边界值分析

7.7.3 错误推测

## 7.7 黑盒测试的主要技术

### 7.7.1 等价划分

### 7.7.2 边界值分析

### 7.7.3 错误推测

## 7.7.1 等价划分法

### 等价划分测试技术

- 等价划分测试技术把所有可能的输入数据(有效的和无效的)划分成若干个等价的子集, 使得每个子集中的一个典型值作为测试用例。在测试中的作用与这一子集中所有其它值的作用相同。

### 什么是等价类?

- 等价类别或等价区间是指测试相同目标或者暴露相同软件缺陷的一组测试用例。
- 发现若干类程序错误, 从而减少必须设计的测试用例的数目。

## 7.7.1 等价划分法

### 划分等价类——划分的输入数据的等价类

- 分析程序功能划分输入数据的分等价类：有效等价类和无效等价类。
- 分析程序的输出数据的等价类，反推出输入数据的等价类。

### 划分等价类的基本标准

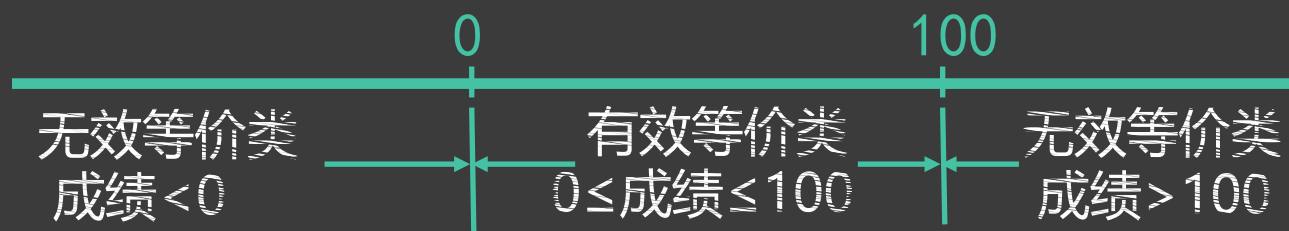
- 全覆盖
- 不相交
- 代表性

## 7.7 黑盒测试技术——等价划分法

### 划分等价类的例子

- (1) 如果输入条件规定了取值范围，那么可以定义一个有效等价类（在取值范围内）和两个无效等价类（小于最小值、大于最大值）。

例：输入值是学生成绩，范围是0 ~ 100



### 划分等价类的例子

- (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**，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等价类，并有一个无效等价类(所有不允许的输入值的集合)。

例：输入条件说明学历可为：**专科、本科、硕士、博士四种之一**，则分别取这四个值作为**四个有效等价类**，另外把**四种学历之外的任何学历**作为无效等价类。

## 7.7 黑盒测试技术——等价划分法

### 等价类划分法设计测试用例的步骤：

#### ■ 第一步：构建等价类表

- 每一个等价类（包括有效等价类和无效等价类）规定一个唯一的编号。

#### ■ 第二步：为有效等价类设计测试方案（测试用例）

- 设计一个新的测试用例，使其尽可能多地覆盖尚未覆盖的有效等价类，重复这一步骤，直到所有有效等价类均被测试用例所覆盖；

#### ■ 第三步：为无效等价类设计测试方案（测试用例）

- 设计一个新的测试用例，使其只覆盖一个无效等价类，重复这一步骤直到所有无效等价类均被覆盖；



## 7.7 黑盒测试技术——等价划分法

### 等价类划分法设计测试用例的案例：

#### 例：“输入日期”的测试用例

- 某报表处理系统要求用户输入处理报表的日期，日期限制在2003年1月至2008年12月，即系统只能对该段期间内的报表进行处理，如日期不在此范围内，则显示输入错误信息。系统日期规定由年、月的6位数字字符组成，前四位代表年，后两位代表月。
- 如何用等价类划分法设计测试用例，来测试程序的日期检查功能？

## 7.7 黑盒测试技术——等价划分法

### 第一步：等价类划分

“报表日期”输入条件的等价类表

输入条件	有效等价类	无效等价类
报表日期的类型及长度	6位数字字符 (1)	有非数字字符 (4) 少于6个数字字符 (5) 多于6个数字字符 (6)
年份范围	在2003 ~ 2008之间 (2)	小于2003 (7) 大于2008 (8)
月份范围	在1 ~ 12之间 (3)	小于1 (9) 大于12 (10)

## 7.7 黑盒测试技术——等价划分法

### 第二步：为有效等价类设计测试用例

对表中编号为1, 2, 3 的3个有效等价类用一个测试用例覆盖

测试数据	期望结果	覆盖范围
200306	输入有效	等价类(1)(2)(3) <ul style="list-style-type: none"><li>(1) 6位数字字符</li><li>(2) 年在2003 ~ 2008之间</li><li>(3) 月在1 ~ 12之间</li></ul>

## 7.7 黑盒测试技术——等价划分法

第三步：为每一个无效等价类至少设计一个测试用例

测试数据	期望结果	覆盖范围
003MAY	输入无效	等价类(4)非数字
20035	输入无效	等价类(5)少于6位
2003005	输入无效	等价类(6)多于6位
200105	输入无效	等价类(7)小于2003
200905	输入无效	等价类(8)大于2003
200300	输入无效	等价类(9)小于1
200313	输入无效	等价类(10)大于12

## 7.7 黑盒测试技术——等价划分法

等价类划分即把输入空间分解成一系列子域，  
软件在一个子域内的行为应是等价的。



软件错误分为两类：计算错误和域错误



- 针对计算错误的测试方法
- 针对域错误的测试方法:测试域边界划定的正确性

## 7.7 黑盒测试的主要技术

7.7.1 等价划分

7.7.2 边界值分析

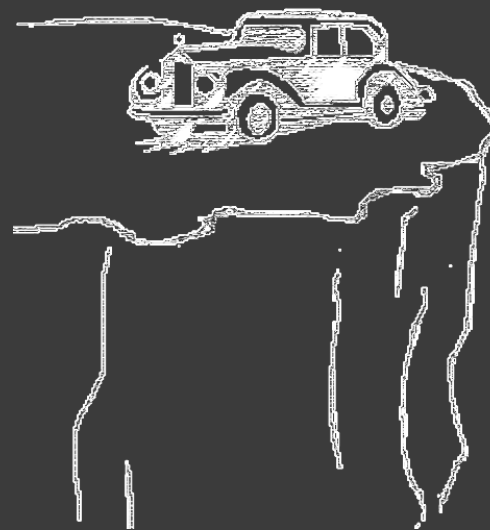
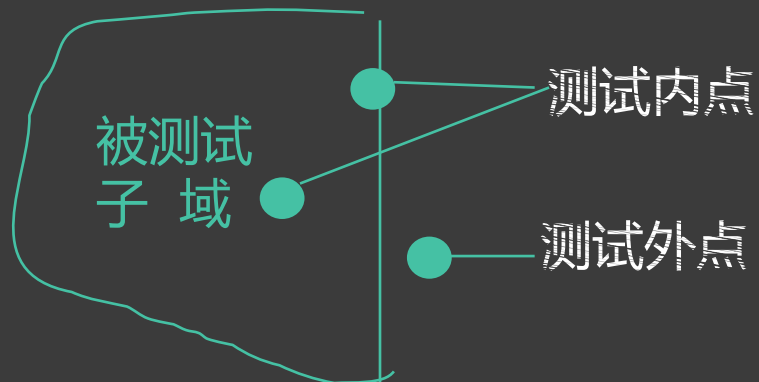
7.7.3 错误推测

## 7.7 黑盒测试技术——边界值分析

### 7.7.2 边界值分析

边界值分析法与等价类划分法区别：

- 边界值分析不是从某等价类中随便挑一个作为代表，而是使这个等价类的每个边界都要作为测试条件。
- 边界值分析不仅考虑输入条，还要考虑输出空间产生的测试情况



## 7.7 黑盒测试技术——边界值分析

### “报表日期（6位数字字符）”边界值分析法测试用例

输入条件	测试用例说明	测试数据	期望结果	选取理由
报表日期的类型及长度	1个数字字符 5个数字字符 7个数字字符 有1个非数字字符 全部是非数字字符 6个数字字符	5 20035 2003005 2003.5 MAY--- 200305	显示出错 显示出错 显示出错 显示出错 显示出错 输入有效	仅有1个合法字符 比有效长度少1 比有效长度多1 只有1个非法字符 6个非法字符 类型及长度均有效
日期范围	在有效范围边界上 选取数据	200301 200812 200300 200813	输入有效 输入有效 显示出错 显示出错	最小日期 最大日期 刚好小于最小日期 刚好大于最大日期
月份范围	月份为1月 月份为12月 月份<1 月份>12	200301 200312 200300 200313	输入有效 输入有效 显示出错 显示出错	最小月份 最大月份 刚好小于最小月份 刚好大于最大月份



## 7.7 黑盒测试技术——边界值分析

有效等价类和用来测试 `getNumDaysInMonth()` 方法所选的有效输入：

有效等价类	月份输入值	年份输入值
一个月有31天,非闰年	7(七月)	1901
一个月有31天, 闰年	7(七月)	1904
一个月有30天,非闰年	6(六月)	1901
一个月有30天, 闰年	6(六月)	1904
一个月为28或29天,非闰年	2(二月)	1901
一个月为28或29天, 闰年	2(二月)	1904

## 7.7 黑盒测试技术——边界值分析

用来测试getNumDaysInMonth()方法的附加边界值

等价类	月份输入值	年份输入值
可以被400整除的闰年	2(二月)	2000
可以被100整除的非闰年	2(二月)	1900
非正数无效月份	0	1291
正数无效月份	13	1315

## 7.7 黑盒测试的主要技术

7.7.1 等价划分

7.7.2 边界值分析

7.7.3 错误推测

## 7.7.3 错误推测法

### 错误推测法

- 根据经验、直觉和预感来进行测试。
- 基本思想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，并根据他们设计测试用例。

### 错误推测用例选取

- 在直觉认为容易出错的地方建立测试等价类：缺省值、空白、空值、零值、空表、只有一项的表、无输入条件等
- 在已经找到软件缺陷的地方，根据其出错特点设计测试用例。

# 第七章 实现(编码与测试)

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

7.8.1 调试过程

7.8.2 调试途径

7.8.3 纠错问题

## 7.8 调试

### 7.8.1 调试过程

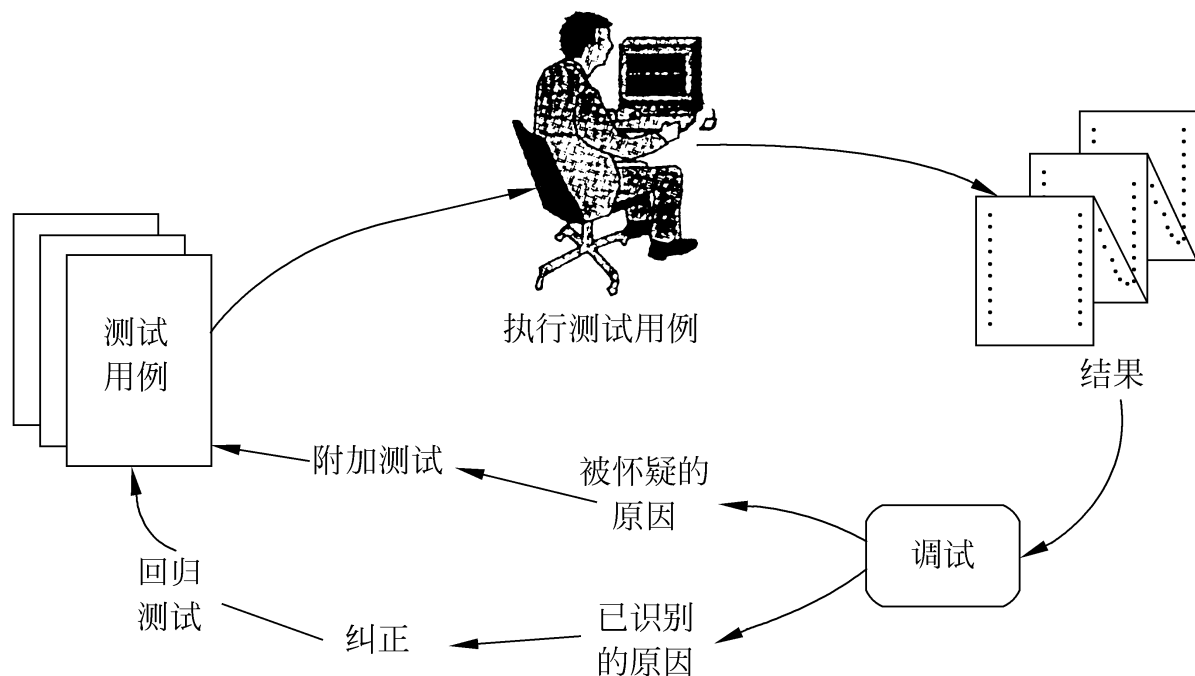
### 7.8.2 调试途径

### 7.8.3 纠错问题

## 7.8.1 调试过程

调试过程：

- ① 测试
- ② 分析测试结果
- ③ 调试
- ④ 识别出错误原因：  
改正错误，进行回归测试；
- ⑤ 未识别出错误原因：  
设计附加测试用例，进行测试。





## 7.8 调试

7.8.1 调试过程

7.8.2 调试途径

7.8.3 纠错问题

## 7.8.2 调试途径

3种调试途径：

- 蛮干法
- 回溯法
- 原因排除法

## 7.8.2 调试途径

### 3种调试途径：

- 蛮干法 --- 逐点（单步）跟踪
- 回溯法
- 原因排除法

- 这种方法的操作与出错症状没关联，目的不明确，效率低下。
- 建议仅当其他调试方法都失败时，才采用本方法。

## 7.8.2 调试途径

### 3种调试途径：

- 蛮干法 --- 逐点（单步）跟踪
- 回溯法 --- 从出错处向上追溯
- 原因排除法

■ 比较适用于小程序的调试。因为当程序很大的时候，控制结构非常复杂，回溯的路径数目太大，不好操作。

## 7.8.2 调试途径

3种调试途径：

- 蛮干法 --- 逐点（单步）跟踪
- 回溯法 --- 从出错处向上追溯
- 原因排除法 --- 对分查找法、归纳法和演绎法

# 原因排除法

## ① 对分查找法

- 对分查找法每次将程序中可能出错的部分缩小约一半，直到确定出错位置。
- 如果知道若干个关键点正确值，那么程序的输出正确，则错误原因在程序的前半部，否则在后半部。

## ② 归纳法（从个别现象到一般结论）

- 在调试中，将出现错误有关的数据组织起来进行分析，试图发现可能出错规律和错误原因。
- 然后导出对出错原因的假设，在利用测试数据（测试用例）证明或排除这些假设。

## ③ 演绎法（从一般原理到特定结论）

- 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特定的结论。
- 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存在，则设计测试数据进行精化测试，以确定更准确的错误原因和出错位置。

## 7.8 调试

7.8.1 调试过程

7.8.2 调试途径

7.8.3 纠错问题

## 7.8.3 纠错问题

- 经过测试（暴露和发现错误）、调试（找到出错原因和定位出错位置）之后，下一步就是改正错误（即纠错）。
- 改正已知错误的同时，也可能引入新的错误（**因为模块是强内聚的**）。因此在动手纠错之前，应考虑如下有关问题：
  - ① 同样的错误是否也在程序的其他地方存在？（程序员的思维模式的定式、惯性，所谓本性难改）
  - ② 将要进行的纠错**可能会引入的下一个错误**是什么？（要特别注意程序逻辑及数据结构**高耦合部位**）
  - ③ 如何防止今后犯同类型的错误？（从失败和错误中学习，引以为戒。）



# 第七章 实现(编码与测试)

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

- 软件可靠性的定义
- 软件可用性的定义

### 7.9.2 估算平均无故障时间的方法

- 符号
- 基本假定
- 估算平均无故障时间
- 估计错误总数的方法

## 7.9.1 基本概念

### 1. 软件可靠性的定义

- 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。
- 定义中包含的随机变量是“时间间隔”，随着运行时间的增加，程序出现故障的概率也增加，即软件的可靠性会降低。

### 2. 软件的可用性的定义

- 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。
- 软件的可用性=软件系统可以使用的程度。

### 3. 术语“错误”与“故障”的定义（IEEE）

- 错误：由开发人员造成的软件差错（bug）。（因）
- 故障：由错误引起的软件的不正确行为。（果）

# 可靠性与可用性区别

## 可靠性

- 系统在时间间隔 $[0..t]$ 内是可靠的，意味着从0到t这段时间内系统没有失效（没有出现任何故障）。
- 可靠性是“时间区间”上的概率。

## 可用性

- 系统在时刻 $t$ 是可用的，意味著两种可能：①从0到 $t$ 这段时间内系统没有失效（可靠），②从0到 $t$ 这段时间内系统有失效（1次或多次）但已经修复。
- 可用性是“时间点”上的概率。

# 可用性的计算

- 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：

$$A_{ss} = T_{up} / (T_{up} + T_{down})$$

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

- 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，则上式可以变成

$$A_{ss} = MTTF / (MTTF + MTTR)$$

## 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) 是软件产品的一项重要指标。
- 估算MTTF的方法：
  - ① 用到的有关符号
  - ② 基本假定
  - ③ 估算平均无故障时间
  - ④ 估计错误总数的方法

## ① 估算MTTF的方法——符号

$E_T$ : 测试之前程序中错误总数

$I_T$ : 程序长度(机器指令总数)

$T$ : 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

根据经验数据，可以作出下述假定：

(1) 单位长度里的错误数 $E_T/I_T$ 近似为常数。一些统计数字表明，通常 $0.5 \times 10^{-2} \leq E_T/I_T \leq 2 \times 10^{-2}$ 。也就是说，在测试之前每1000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) 假设发现的每一个错误都立即正确地改正了，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

- 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

$$MTTF = 1/[K(E_T/I_T - E_C(\tau)/I_T)]$$

- 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

$$E_C = E_T - I_T/(K \times MTTF)$$

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

### 错误总数估算问题

- 程序中隐藏的潜在错误数数目是一个十分重要的量，直接反映了程序的可靠程度，又是计算软件平均无故障时间MTTF的重要参数。
- 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、开发人员的水平和管理人员的水平密切相关。因此，错误数的估算或计算是一件复杂的工作。
- $MTTF = 1/[K(E_T/I_T - E_c(\tau)/I_T)]$

### 二种简单的估算的程序中的错误总数 $E_T$ 方法：

- (1) 植入错误法
- (2) 分别测试法

# (1) 植入错误法

## 基本原理

- 在测试前由专人（第三方，非编码、测试人员）在程序中随机地植入一些错误。
- 测试人员按常规测试程序，根据发现的错误中原有的和植入错误的比例，来估计程序中原有错误的总数 $E_T$ 。

## 估算错误数

- 假设人为地植入的错误数为 $N_s$ ，经测试之后发现 $n_s$ 个植入的错误， $n$ 个原有（固有）的错误。如果认为测试方案发现植入错误和发现原有错误的能力相同，则能估出程序中原有错误的总数为

$$\hat{N} = (n/n_s) \times N_s$$

其中 $\hat{N}$ 即是错误总数 $E_T$ 的估计值。

# (1) 植入错误法存在的问题

## 问题

- 人为地植入的错误和原有错误可能性质很不相同，发现它们的难易程度也不相同，因此，上述基本假定可能有时和事实不完全一致。

## 解决方法

- 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。
- 这就是后面的“分别测试法”

## (2) 分别测试法

### 基本原理

- 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看作有标记的错误。
- 由另一名分析员分析他们的测试结果, 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。

### 估算错误数

- 假定测试员甲发现的错误作为有标记错误。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 $b_c$ 。

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, 则可估计出测试前程序中的错误总数为

$$\hat{B}_0 = (B_2/b_c)B_1$$

# 本章小结

## 【实现】

- 包括编码和测试两个阶段。

## 【编码】

- 编码是将设计结果翻译成程序(某种程序语言编写)。
- 编码风格对软件的可读性、可维护性、可靠性、可用性很重要。
- 编码风格包括：良好的文档（注释）、数据说明、语句构造、输入输出和效率保障等。

## 【测试】

- 测试是主要任务是发现错误并改正错误，保证软件的可靠性。
- 两类基本测试方法：白盒测试和黑盒测试。
- 测试的过程包括：单元测试（模块）、子系统测试（子系统集成）、系统测试（确认需求）、验收测试（用户参与）、平行运行（试运行）。
- 白盒测试技术：逻辑覆盖测试（语句、判定、条件、判定/条件、条件组合；点、边、路径）、控制结构测试（基本路径、条件、循环）。
- 黑盒测试技术：等价划分、边值分析、错误推测。

## 【调试】

- 寻找错误原因和确定出错位置：蛮干法、回溯法、原因排除法（对分法、归纳法、演绎法）。

## 【软件可靠性】

- 软件可靠性定义
- 软件可用性定义
- 稳态可用性Ass估算
- 平均无故障时间MTTF估算
- 错误数估算方法：植入错误法、分别测试法（标记错误）