第10章 软件测试方法

- 软件测试的基本概念
- 白盒测试的测试用例设计
- 基本路径覆盖
- 黑盒测试的测试用例设计
- 软件测试的策略
- 人工测试
- 调试

10.1 软件测试的基本概念

- 什么是软件测试

- **软件测试是在软件投入生产性运行之前，对软件需求分析、设计规格说明和编码的最终复审，是软件质量控制的关键步骤。**
- **软件测试是为了发现错误而执行程序的过程。**
- **或者说，软件测试是根据软件开发各阶段的规格说明和程序的内部结构而精心设计一批测试用例（即输入数据及其预期的输出结果），并利用这些测试用例去运行程序，以发现程序错误的过程。**

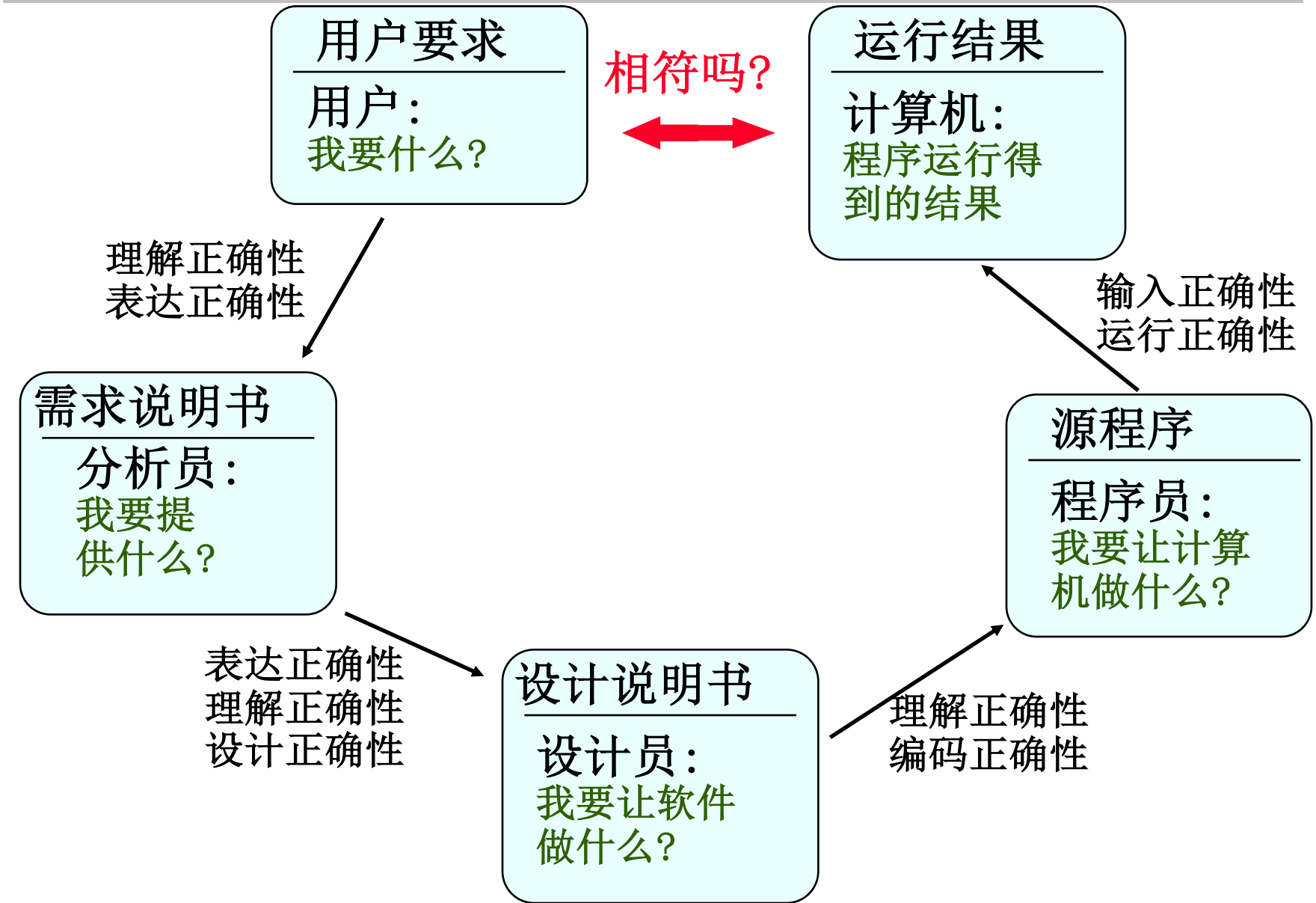
10.1 软件测试的基本概念

- 软件测试的目的和原则

  基于不同的立场，存在着两种完全不同的测试目的。

  - **从用户的角度出发，普遍希望通过软件测试检验软件中隐藏的错误和缺陷，以考虑是否可以接受该产品。**
  - **从软件开发者的角度出发，则希望测试成为表明软件产品中不存在错误的过程，验证该软件已正确地实现了用户的要求，确立人们对软件质量的信心。**

软件生存期各阶段间需保持的正确性



10.1 软件测试的基本概念

Grenford J.Myers就**软件测试目的**提出了以下观点。

- **测试**是程序的执行过程，目的在于发现错误。
- **好的测试用例**在于能发现至今未发现的错误。
- **成功的测试**是发现了至今未发现的错误的测试。

10.1 软件测试的基本概念

根据以上测试目的，软件测试的原则如下：

（1）应当把“尽早地和不断地进行软件测试”作为软件开发者的座右铭。

（2）测试用例应由测试输入数据和与之对应的预期输出结果这两部分组成。

（3）程序员应避免检查自己的程序。

（4）在设计测试用例时，应当包括合理的输入条件和不合理的输入条件。

10.1 软件测试的基本概念

（5）充分注意测试中的群集现象。把Pareto原理应用于软件测试。

Pareto原理：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。

（6）严格执行测试计划，排除测试的随意性。

（7）应当对每一个测试结果作全面检查。

（8）妥善保存测试计划、测试用例、出错统计和最终分析报告，为维护提供方便。

10.1 软件测试的基本概念

测试工作量和测试人员：

- **在整个软件开发中，测试工作量一般占30%～40%，甚至≥50%。**
- **在人命关天的软件(如飞机控制、核反应堆等）中，测试所花费的时间往往是其它软件工程活动时间之和的三到五倍。**

Windows2000开发人员中：

- **项目经理约250人**
- **开发人员约1700人**
- **测试人员约3200人**

10.1 软件测试的基本概念

- 软件测试的对象

软件测试应贯穿于软件定义与开发的整个期间。需求分析、概要设计、详细设计、程序编码等各阶段所得到的文档资料，包括需求规格说明、概要设计规格说明、详细设计规格说明以及源程序，都应成为软件测试的对象。

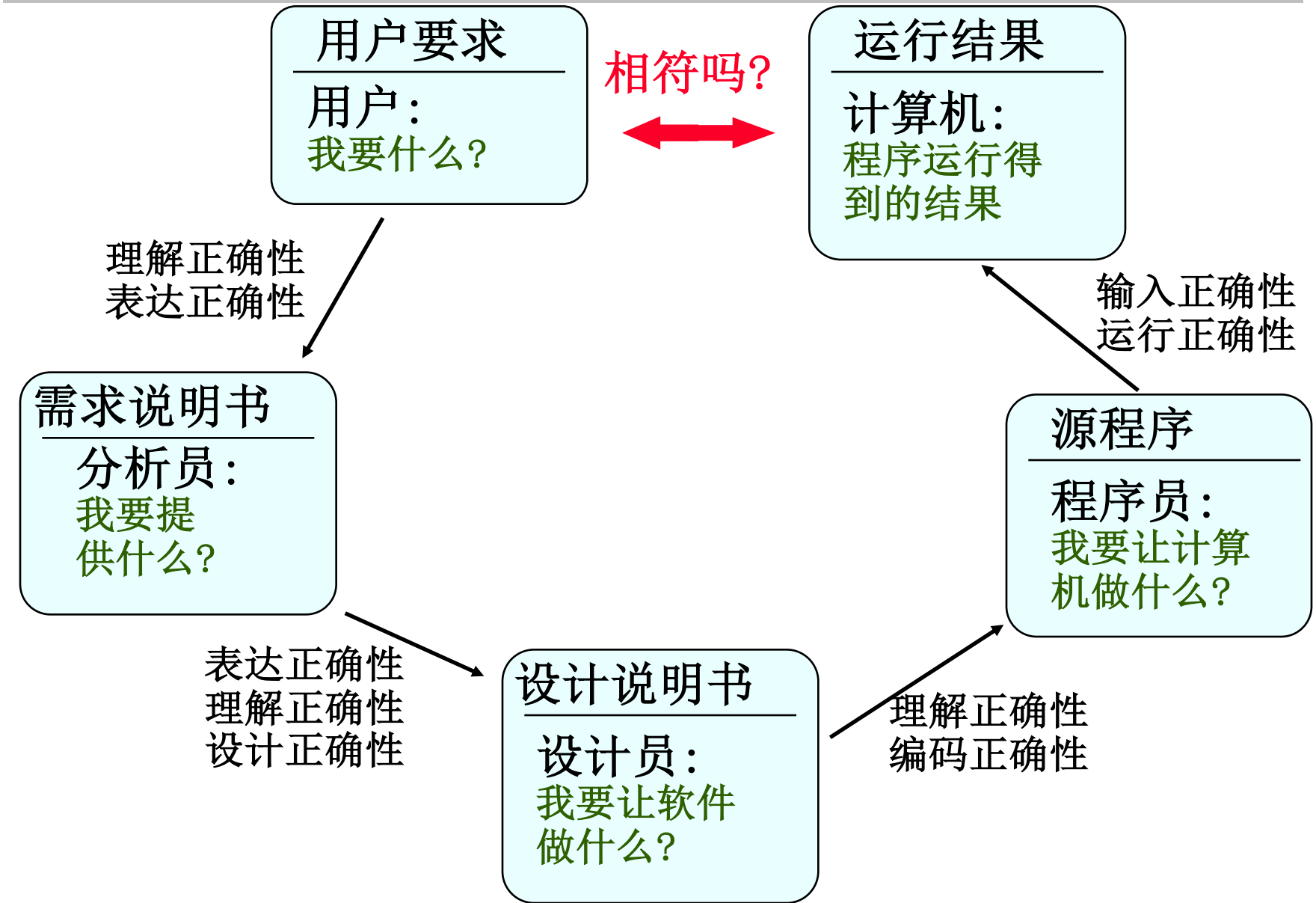
到程序的测试为止，软件开发工作已经经历了许多环节，每个环节都可能发生问题。为了把握各个环节的正确性，人们需要进行各种确认和验证工作。

10.1 软件测试的基本概念

确认（validation），是一系列的活动和过程，其目的是想证实在一个给定的外部环境中软件的逻辑正确性。它包括需求规格说明的确认和程序的确认，而程序的确认又分为静态确认与动态确认。

验证（verification），则试图证明在软件生存期各个阶段，以及阶段间的逻辑协调性、完备性和正确性。下图为软件生存期各个重要阶段之间所要保持的正确性。

10.1 软件测试的基本概念



测试方法与技术

- **机器测试与人工测试**
  - **机器测试**
    在设定的测试数据上执行被测程序的过程。又称动态测试。
  - **人工测试**
    采用人工方法进行，目的在于检查程序的静态结构，找出编译不能发现的错误。

测试方法与技术

- **人工测试的分类**
  - **代码审查**

    以小组会的形式，发现程序在结构、功能、编码风格等方面存在的问题。可查出30%~70%的错误
  - **走查**

    以小组会的形式进行，把测试数据“输入”到被测程序，并在纸上跟踪监视程序的执行情况，让人代替机器沿着程序的逻辑走一遍。
  - **桌前检查**

    设计模块时，程序员自己检查。

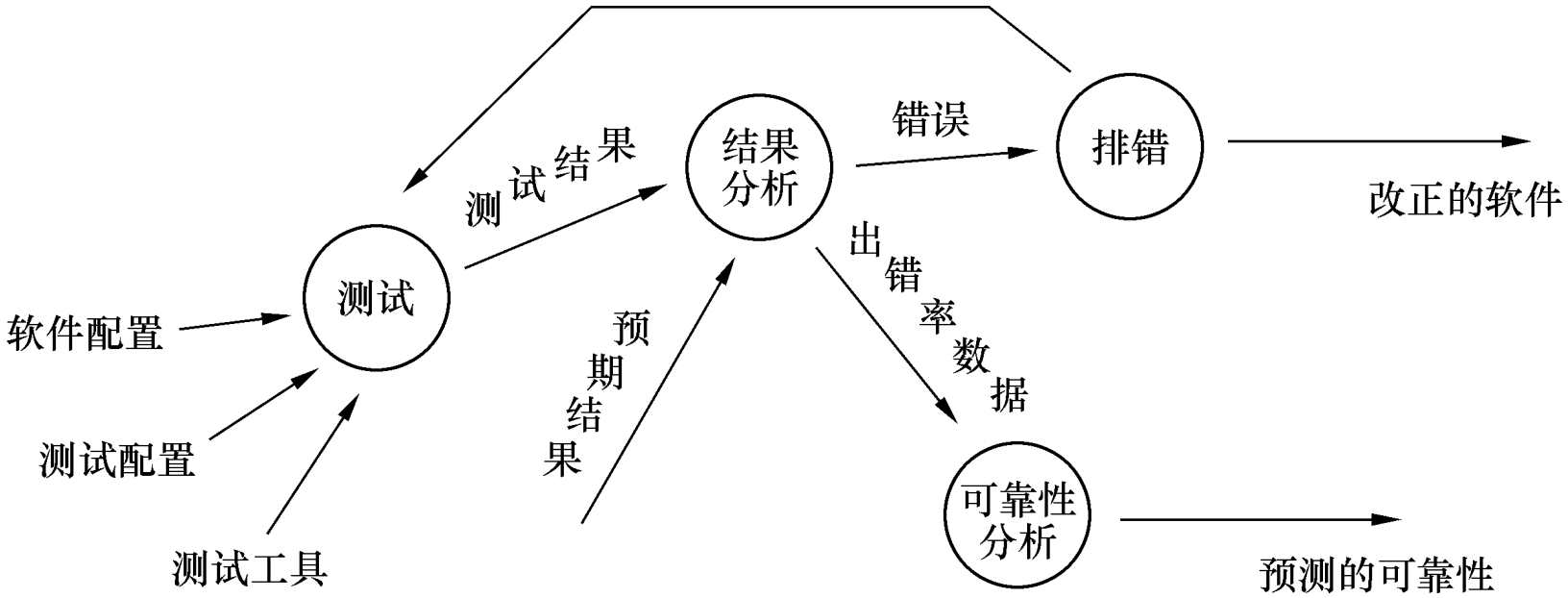
测试方法与技术

- **机器测试的分类**
  - **黑盒测试**
  - **白盒测试**

10.1 软件测试的基本概念

- 测试信息流

测试信息流如下图所示。



10.1 软件测试的基本概念

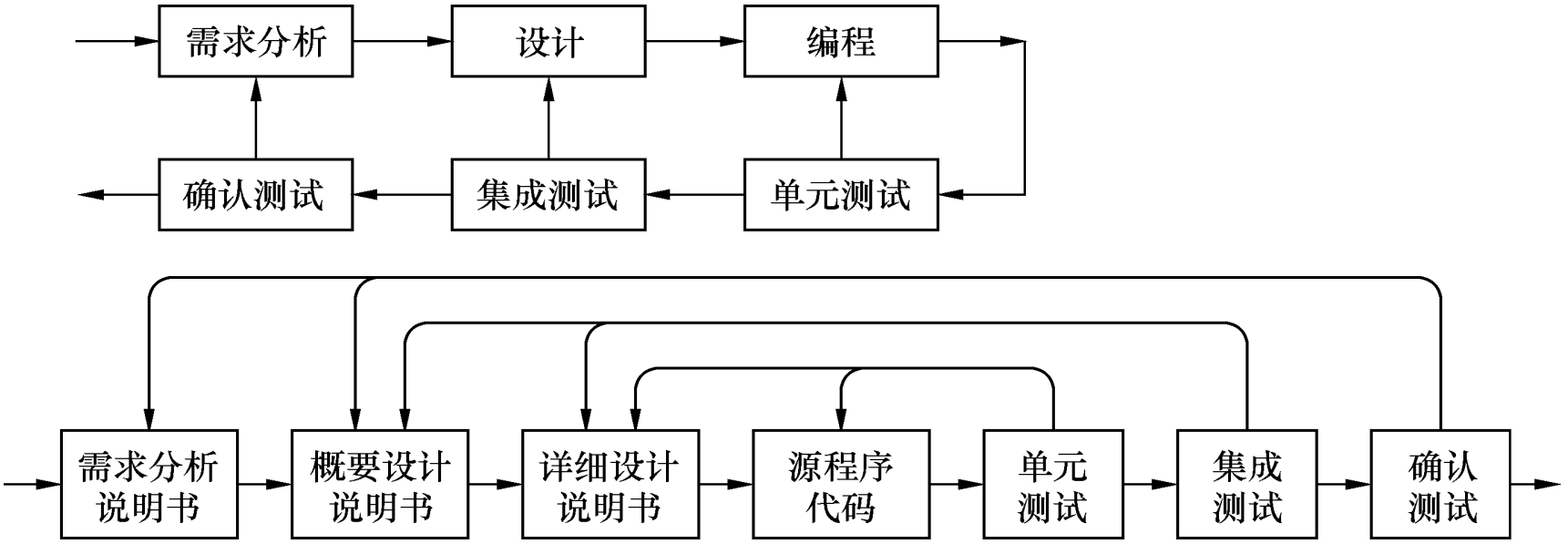
- 测试与软件开发各阶段的关系

软件开发过程是一个自顶向下、逐步细化的过程，而测试过程则是依相反的顺序安排的自底向上、逐步集成的过程。低一级测试为上一级测试准备条件。

当然不排除两者平行地进行测试。

10.1 软件测试的基本概念

- 测试与软件开发各阶段的关系



10.1 软件测试的基本概念

- 黑盒测试

  黑盒测试是把测试对象看做一个黑盒子，测试人员完全不考虑程序内部的逻辑结构和内部特性，只依据程序的需求规格说明书，检查程序的功能是否符合它的功能说明。

10.1 软件测试的基本概念

- 黑盒测试

黑盒测试方法主要是为了发现以下错误：

- **是否有不正确或遗漏了的功能？**
- **输入能否正确地接收？**
- **能否输出正确的结果？**
- **是否有数据结构错误或外部信息（例如数据文件）访问错误？**
- **性能上是否能够满足要求？**
- **是否有初始化或终止性错误？**

10.1 软件测试的基本概念

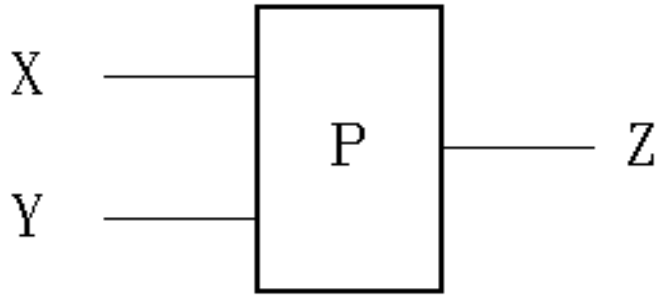
- **黑盒穷举测试**

对所有输入数据的各种可能值的排列组合都进行测试，来检查程序是否都能产生正确的输出。

实际上这是不可能的。

10.1 软件测试的基本概念

- 假设一个程序*P*有输入量*X*和*Y*及输出量*Z*。在字长为32位的计算机上运行。若*X*、*Y*取整数，按黑盒方法进行穷举测试：



- 可能采用的测试数据组：

$$2^{32} \times 2^{32} = 2^{64}$$

- 如果测试一组数据需要1毫秒，一年工作365×24小时，完成所有测试需5亿年。

10.1 软件测试的基本概念

2. 白盒测试

- **白盒测试是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做一个打开的盒子或透明的盒子，它允许测试人员利用程序内部的逻辑结构及有关信息，设计或选择测试用例，对程序所有逻辑路径进行测试。**
- **通过在不同点检查程序的状态，确定实际的状态是否与预期的状态一致。**
- **因此，白盒测试又称为结构测试或逻辑驱动测试。**

10.1 软件测试的基本概念

2. 白盒测试

白盒测试主要是对程序模块进行检查：

- **对程序模块的所有独立的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界限内执行循环体；测试内部数据结构的有效性等。**

10.1 软件测试的基本概念

- **白盒穷举测试**

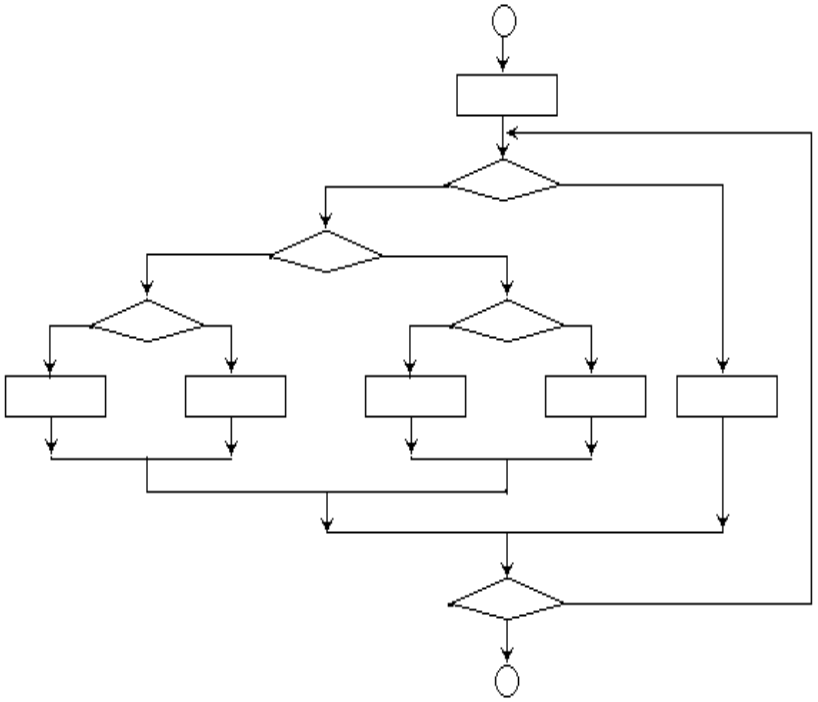
对每条通路都应在每种可能的输入数据下执行一次。

实际上这是不可能的。

10.1 软件测试的基本概念

- 对一个具有多重选择和循环嵌套的程序，不同的路径数目可能是天文数字。给出一个小程序的流程图，它包括了一个执行20次的循环。

- 包含的不同执行路径数达5^{20}条，对每一条路径进行测试需要1毫秒，假定一年工作365 × 24小时，要想把所有路径测试完，需3170年。



10.2 白盒测试的测试用例设计

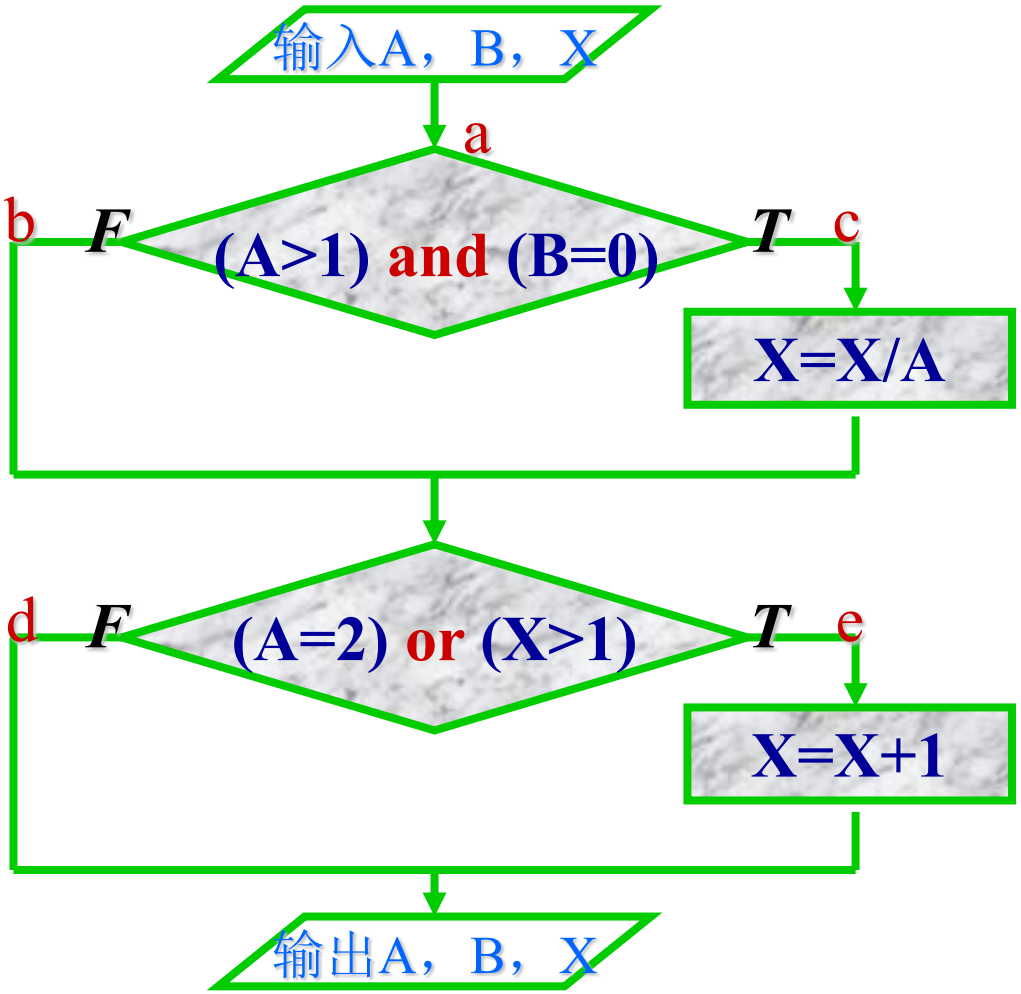
- 逻辑覆盖

  逻辑覆盖是以程序内部的逻辑结构为基础的设计测试用例的技术，它属于白盒测试。

  由于覆盖测试的目标不同，逻辑覆盖又可分为：

  - **语句覆盖**
  - **判定覆盖**
  - **判定—条件覆盖**
  - **条件组合覆盖**
  - **路径覆盖**

10.2 白盒测试的测试用例设计



- 图中所示的程序段有4条不同的路径：

  L1（a→c→e）

  L2（a→b→d）

  L3（a→b→e）

  L4（a→c→d）

  或简写为ace、abd、abe和acd。

10.2 白盒测试的测试用例设计

L1 (a → c → e)

= {(A>1) and (B=0)} and

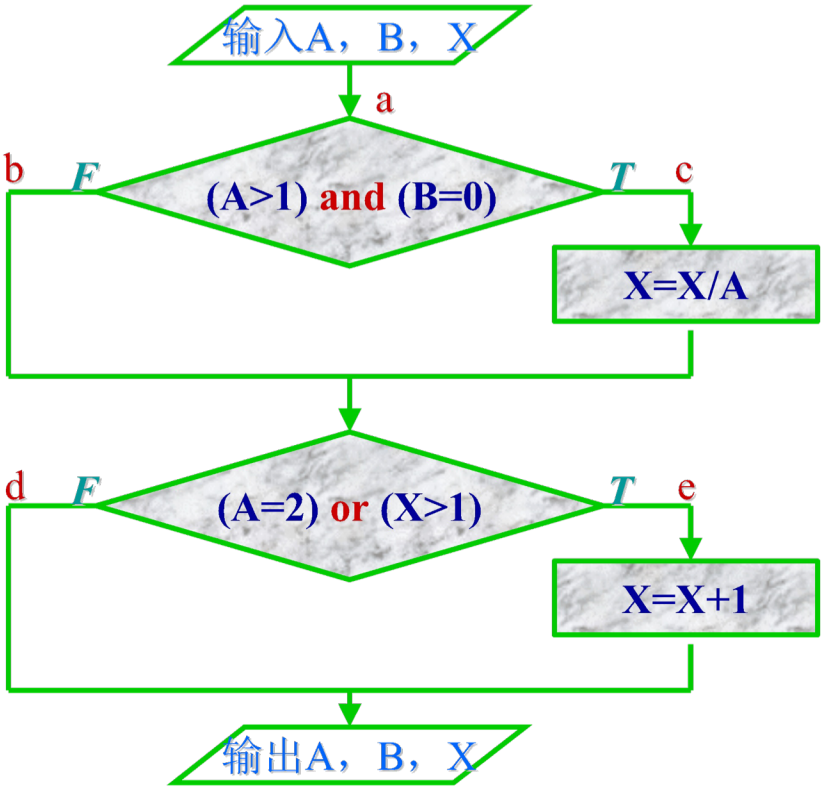
{(A=2) or (X/A>1)}

= (A>1) and (B=0) and (A=2) or

(A>1) and (B=0) and (X/A>1)

= (A=2) and (B=0) or

(A>1) and (B=0) and (X/A>1)



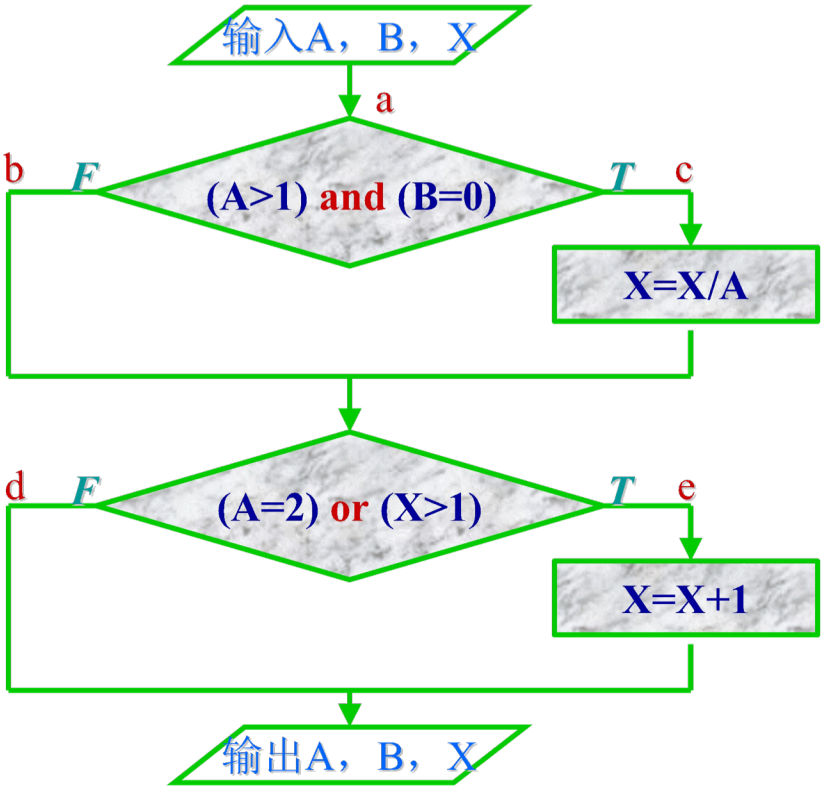
10.2 白盒测试的测试用例设计

L2 (a→ b → d)

= not{(A>1) and (B=0)} and
not{(A=2) or (X>1)}

= { not (A>1) or not (B=0) } and
{ not (A=2) and not (X>1) }

= (A≤1) and (X≤1) or (B≠0)
and (A ≠ 2) and (X ≤ 1)



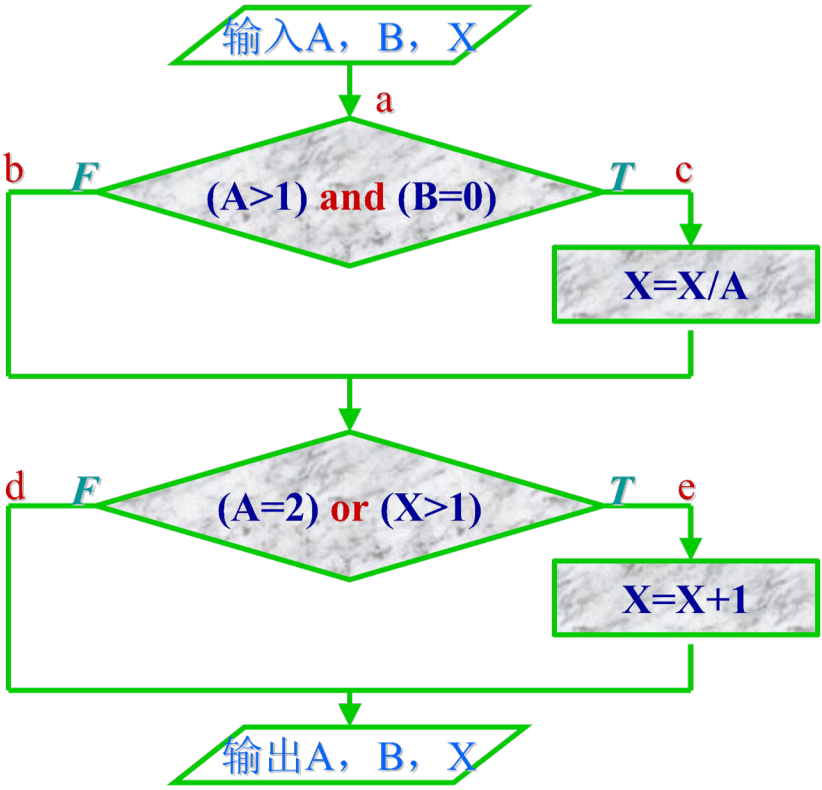
10.2 白盒测试的测试用例设计

L3 (a→ b→ e)

= not {(A>1) and (B=0)} and
{(A=2) or (X>1)}

= { not (A>1) or not (B=0)} and
{(A=2) or (X>1)}

= (A≤1) and (X>1) or (B≠0) and
(A=2) or (B≠0) and (X>1)



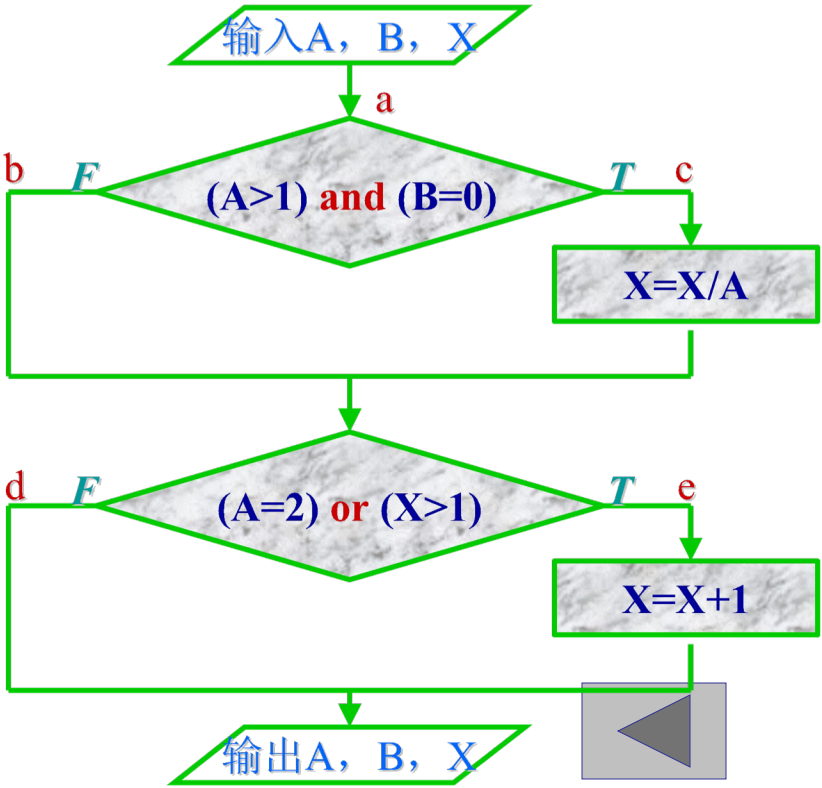
10.2 白盒测试的测试用例设计

L4 (a→ c → d)

= {(A>1) and (B=0)} and
not {(A=2) or (X/A>1)}

= (A>1) and (B=0) and not (A=2) and
not (X/A>1)

= (A>1) and (B=0) and (A ≠ 2)
and (X/A ≤1)



语句覆盖

- 语句覆盖就是设计若干个测试用例，运行被测程序，使得每一可执行语句至少执行一次。
- 在图例中，正好所有的可执行语句都在路径L1上，所以选择路径 L1设计测试用例，就可以覆盖所有的可执行语句。

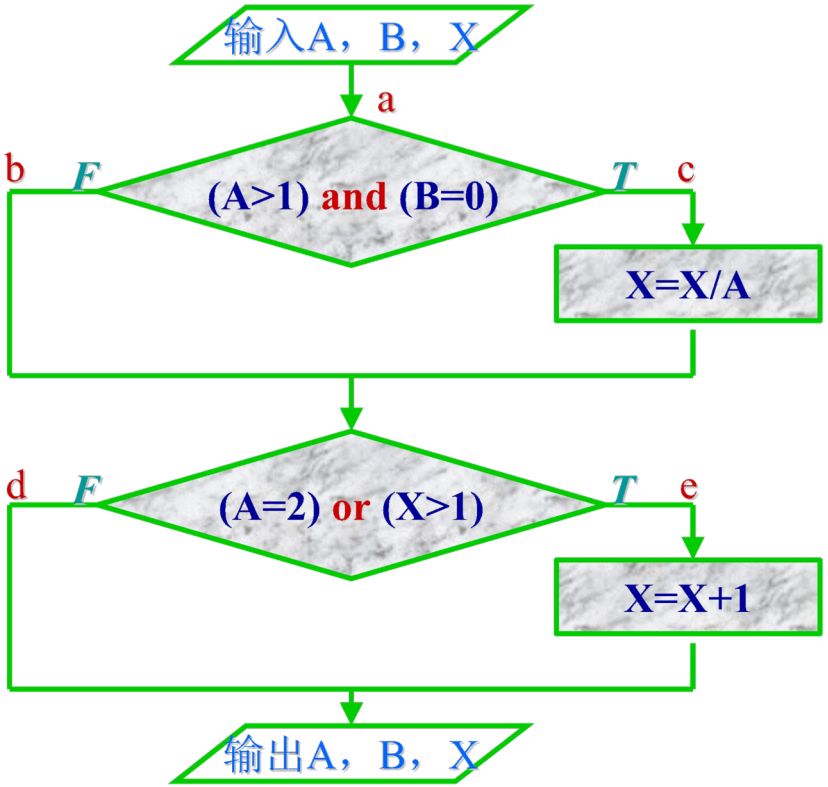
L1 (a → c → e)

= (A=2) and (B=0) or

(A>1) and (B=0) and (X/A>1)

满足语句覆盖的测试用例是:

【(2, 0, 4)，(2, 0, 3)】



语句覆盖

虽然语句覆盖检验了每一个可执行语句，但可能发现不了判断中逻辑运算的错误。语句覆盖是最弱的逻辑覆盖标准。如将A>1&&B= =0 错写成 A>1||B= =0，测试用例【(2, 0, 4)，(2, 0, 3)】依然成立。

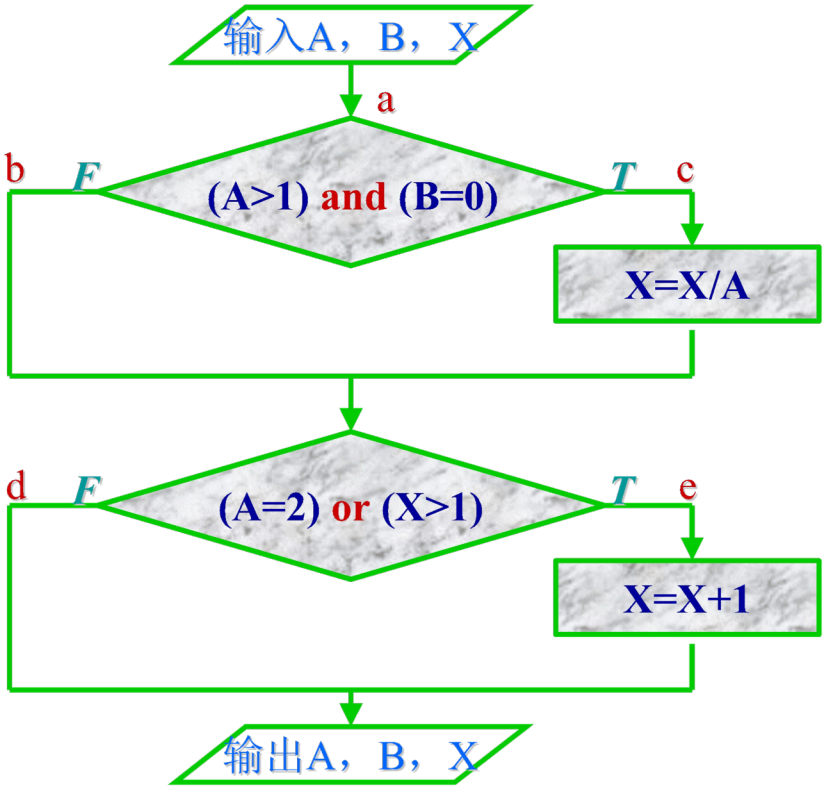
```
main()
{float A,B,X;
 scanf(“%f%f%f”,&A,&B,&X);
 if  (A>1 || B= =0)    X=X/A;
 if  (A==2||X>1)       X=X+1;
 printf(“A=%f,B=%f,X=%f”,A,B,X);
}
```

判定覆盖

- **判定覆盖就是设计若干个测试用例，运行被测程序，使得程序中每个判断的取真分支和取假分支至少经历一次。判定覆盖又称为分支覆盖。**

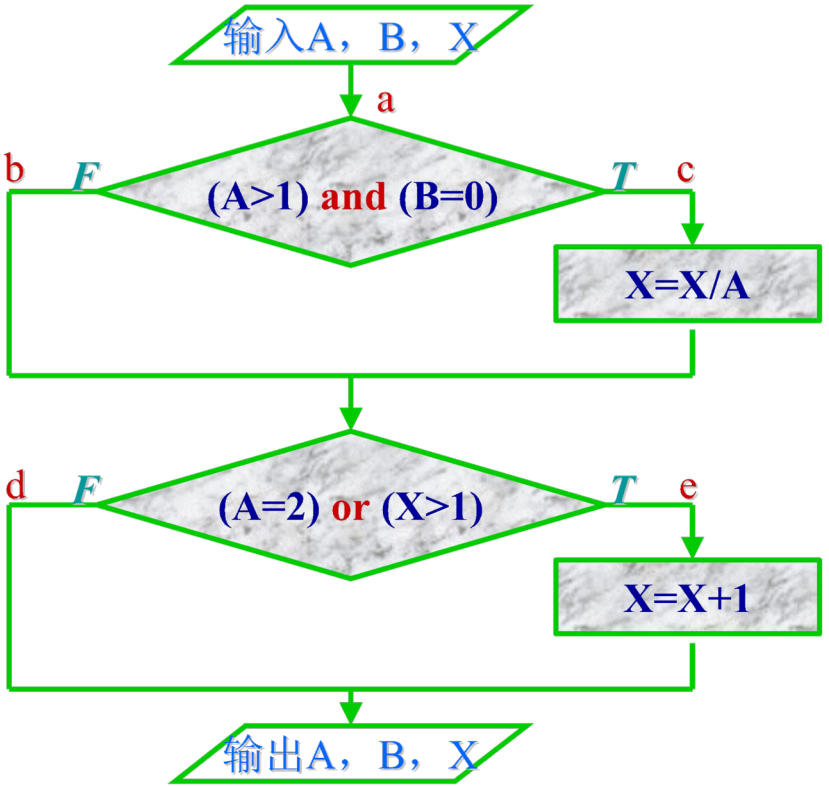
【(2, 0, 4)，(2, 0, 3)】
覆盖 ace【L1】

【(1, 1, 1)，(1, 1, 1)】
覆盖 abd【L2】



判定覆盖

- **如果选择路径L3和L4，还可得另一组可用的测试用例:**
  【(2, 1, 1)，(2, 1, 2)】覆盖 abe【L3】
  【(3, 0, 3)，(3, 1, 1)】覆盖 acd【L4】



判定覆盖

只是判定覆盖，还不能保证一定能查出在判断的条件中存在的错误。

若将第二个判断中的条件x>1错写成x<1，上面两组测试用例，仍能得到同样结果。

```
main()
{float A,B,X;
 scanf("%f%f%f",&A,&B,&X);
 if  (A>1 && B= =0)    X=X/A;
 if  (A==2||X<1)       X=X+1;
 printf("A=%f,B=%f,X=%f",A,B,X);
}
```

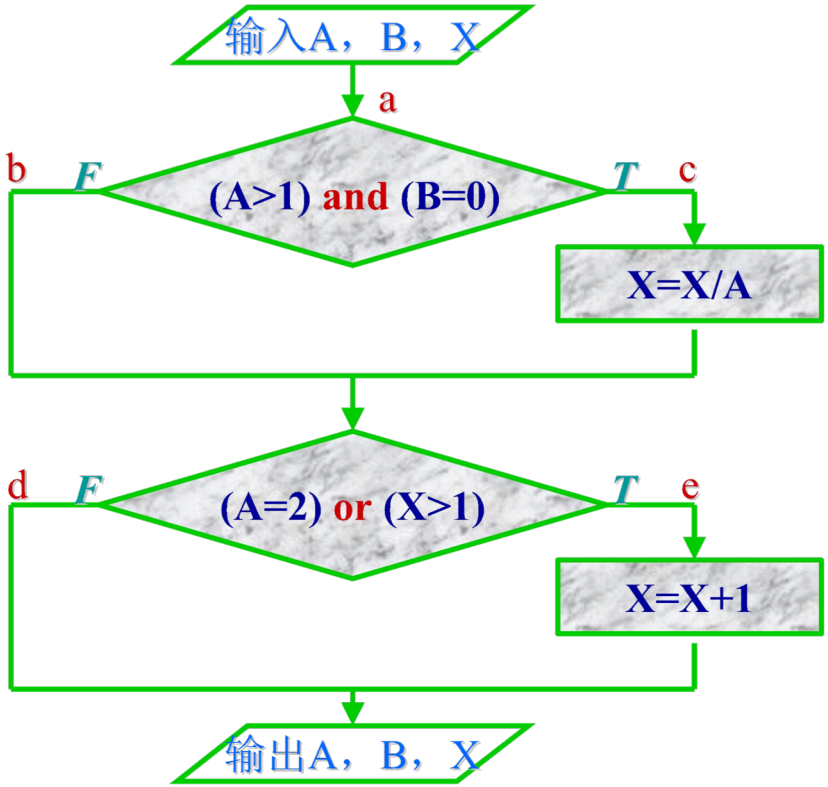
ace【L1】:
【(2, 0, 4)，(2, 0, 3)】
abd【L2】:
【(1, 1, 1)，(1, 1, 1)】
abe【L3】:
【(2, 1, 1)，(2, 1, 2)】
acd【L4】:
【(3, 0, 3)，(3, 1, 1)】

条件覆盖

- 条件覆盖就是设计若干个测试用例，运行被测程序，使得程序中每个判断的每个条件的可能取值至少执行一次。
- 在图例中，我们事先可对所有条件的取值加以标记。例如，
- 条件 A > 1 取真为**T1** ，取假为 $\overline{T_1}$
  条件 B = 0 取真为**T2** ，取假为 $\overline{T_2}$
  条件 A = 2 取真为**T3** ，取假为 $\overline{T_3}$
  条件 X > 1 取真为**T4** ，取假为 $\overline{T_4}$

条件覆盖

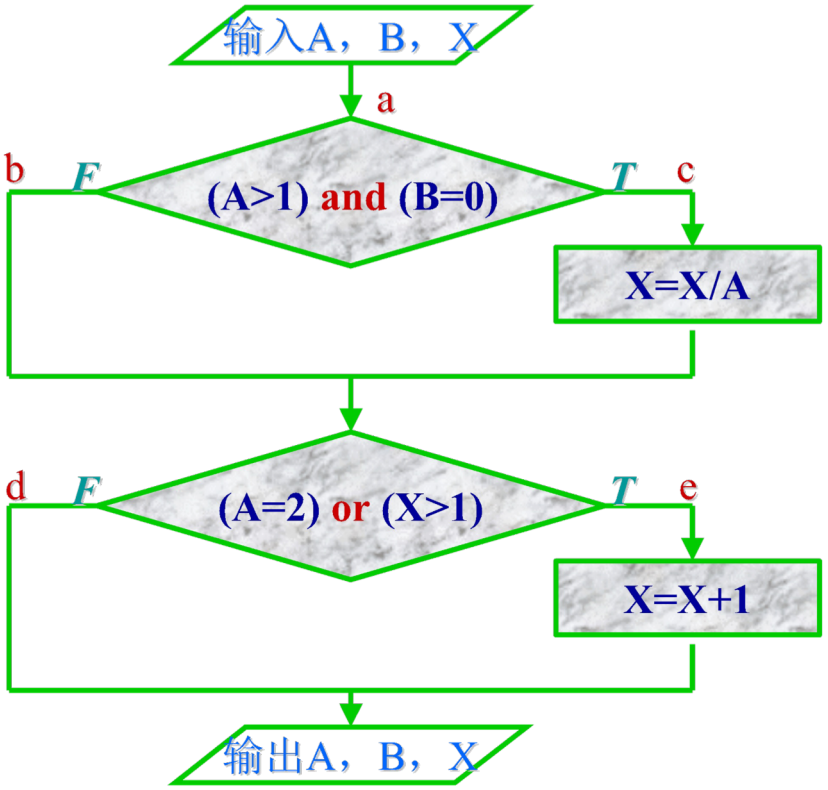
测试用例	覆盖分支	条件取值
【(2, 0, 4)，(2, 0, 3)】	L1(c, e)	$\overline{T_1}T_2\overline{T_3}\overline{T_4}$
【(1, 0, 1)，(1, 0, 1)】	L2(b, d)	$\overline{T_1}T_2\overline{T_3}T_4$
【(2, 1, 1)，(2, 1, 2)】	L3(b, e)	$T_1\overline{T_2}T_3\overline{T_4}$



这组测试用例不但覆盖了所有判断的取真分支和取假分支，而且覆盖了判断中所有条件的可能取值；

条件覆盖

测试用例	覆盖分支	条件取值
【(1, 0, 3)，(1, 0, 4)】	L3(b, e)	$\overline{T_1}T_2\overline{T_3}T_4$
【(2, 1, 1)，(2, 1, 2)】	L3(b, e)	$T_1\overline{T_2}T_3\overline{T_4}$



这组测试用例虽满足了条件覆盖，但只覆盖了第一个判断的取假分支和第二个判断的取真分支。

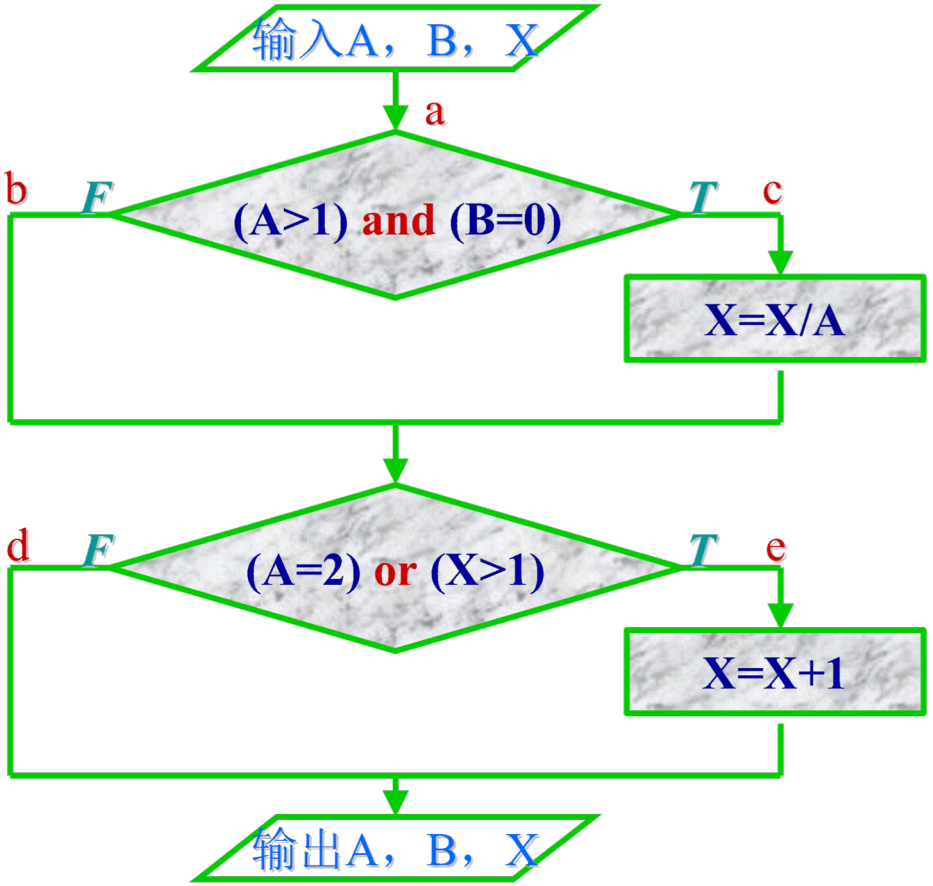
因此，满足条件覆盖，并不一定能满足分支覆盖。

判定-条件覆盖

所谓判定-条件覆盖就是设计足够的测试用例，使得判断中每个条件的所有可能取值至少执行一次，同时每个判断本身的所有可能判断结果至少执行一次。

判定-条件覆盖

测试用例	通过路径	条件取值	覆盖分支
【(2, 0, 4), (2, 0, 3)】	ace (L1)	T1 T2 T3 T4	c, e
【(1, 1, 1), (1, 1, 1)】	abd (L2)	$\overline{T1}$ $\overline{T2}$ $\overline{T3}$ $\overline{T4}$	b, d



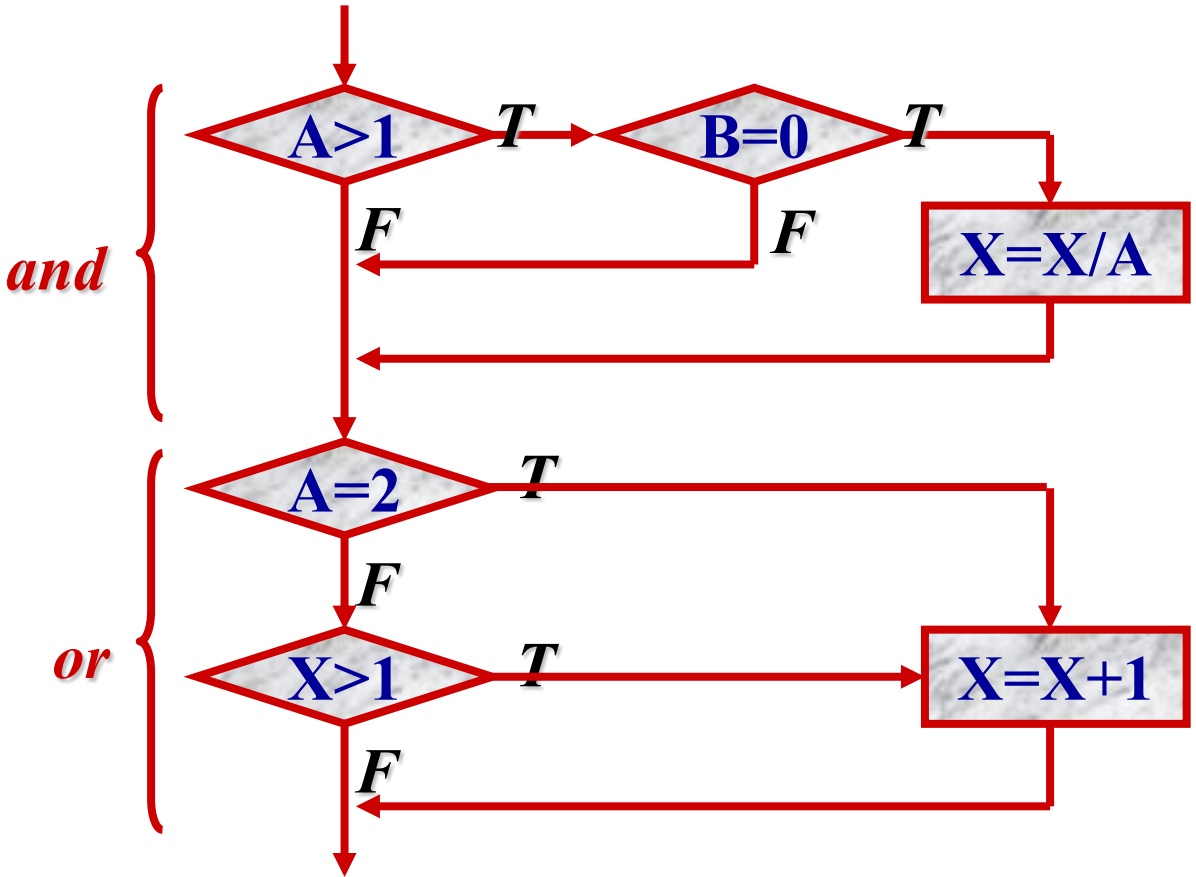
判定-条件覆盖

判定-条件覆盖也有缺陷。从表面上看，它测试了所有条件的取值，但事实并非如此。因为往往某些条件掩盖了另一些条件。

对于表达式**(A>1) and (B=0)**来说，若(A>1)的测试结果为假，往往就不再测试(B=0)的取值了。

判定-条件覆盖

为彻底检查所有条件的取值，可将多重条件判定分解，形成由多个基本判断组成的流程图。这样就可以有效检查所有的条件是否正确了。



条件组合覆盖

- **条件组合覆盖就是设计足够的测试用例，运行被测程序，使得每个判断的所有可能的条件取值组合至少执行一次。**

① $A>1, B=0$ 记作 $T_1\overline{T_2}$

② $A>1, B\neq 0$ 记作 T_1T_2

③ $A\not>1, B=0$ 记作 $\overline{T_1}T_2$

④ $A\not>1, B\neq 0$ 记作 $\overline{T_1}\,\overline{T_2}$

⑤ $A=2, X>1$ 记作 T_3T_4

⑥ $A=2, X\not>1$ 记作 $T_3\overline{T_4}$

⑦ $A\neq 2, X>1$ 记作 $\overline{T_3}T_4$

⑧ $A\neq 2, X\not>1$ 记作 $\overline{T_3}\,\overline{T_4}$

条件组合覆盖

测 试 用 例	通 过 路 径	覆 盖 条 件	覆盖组合号
【（2，0，4），（2，0，3）】	ace（L1）	T1 T2 T3 T4	①，⑤
【（2，1，1），（2，1，2）】	abe（L3）	T1 $\overline{T2}$ T3 $\overline{T4}$	②，⑥
【（1，0，3），（1，0，4）】	abe（L3）	$\overline{T1}$ T2 $\overline{T3}$ T4	③，⑦
【（1，1，1），（1，1，1）】	abd（L2）	$\overline{T1}$ $\overline{T2}$ $\overline{T3}$ $\overline{T4}$	④，⑧

路径漏掉了L4

条件组合覆盖

- 条件组合覆盖是一种相当强的覆盖准则，可以有效地检查各种可能的条件取值的组合是否正确。
- 它不但可覆盖所有条件的可能取值的组合，还可覆盖所有判断的可取分支，但可能有的路径会遗漏掉。
- 因此，满足条件组合覆盖的测试还不完全。

路径测试

路径测试是设计足够的测试用例，覆盖程序中所有可能的路径。若仍以最初的图为例，则可以选择如下的一组测试用例，覆盖该程序段的全部路径。

测试用例	通过路径	覆盖条件
【(2, 0, 4), (2, 0, 3)】	ace (L1)	T1 T2 T3 T4
【(1, 1, 1), (1, 1, 1)】	abd (L2)	$\overline{T1}$ $\overline{T2}$ $\overline{T3}$ $\overline{T4}$
【(1, 1, 2), (1, 1, 3)】	abe (L3)	$\overline{T1}$ $\overline{T2}$ $\overline{T3}$ T4
【(3, 0, 3), (3, 0, 1)】	acd (L4)	T1 T2 $\overline{T3}$ $\overline{T4}$

10.3 基本路径覆盖

- **基本路径测试**是在程序控制流图的基础上，通过分析控制构造的环路复杂性，导出基本可执行路径集合，从而设计测试用例的方法。
- 设计出的测试用例要保证在测试中程序的每一个可执行语句至少执行一次。

10.3 基本路径覆盖

- 实施基本路径测试需要利用程序环路复杂性计算的McCabe方法。基本路径测试法适用于模块的详细设计及源程序，其主要步骤如下：

（1）以详细设计或源代码作为基础，导出程序的控制流图；

（2）计算得到的控制流图G的环路复杂性V（G）；

（3）确定线性无关的基本路径集；

（4）生成测试用例，确保基本路径集中每条路径的执行。

下面以一个求平均值的过程averagy为例，说明测试用例的设计过程。用PDL描述的averagy过程如下所示。

10.3 基本路径覆盖

```
PROCEDURE averagy;
    *This procedure computes the averagy of 100 of fewer numbers that lie bounding values; it also computes
        the total input and the total vatid.
    INTERFACE RETURNS averagy, total. input, total, valid,
    INTERFACE ACCEPTS vaiue, minimum, maximum;
    TYPE value [1:100] IS SCALAR ARRAY;
    TYPE averagy, total. input, total. valid, minimum, maximum, sum IS SCALAR;
    TYPE i IS INTEGER;
    i=1;
    total. input=total. valid=0;
    sum=0;
    DO WHILE valuel[i] <>-999 AND total. imput<100
        increment total. input by 1;
        IF value[i]>=minimum AND value[i]<= maximum
            THEN increment total.valid by 1;
                 sum=sum+value [i];
            ELSE skip
        ENDIF;
        increment i by 1;
    ENDDO
    IF total. valid>0
        THEN averagy=sum/total. valid;
        ELSE averagy=-999;
    ENDIF
END averagy
```

10.3 基本路径覆盖

1. 由过程描述导出控制流图

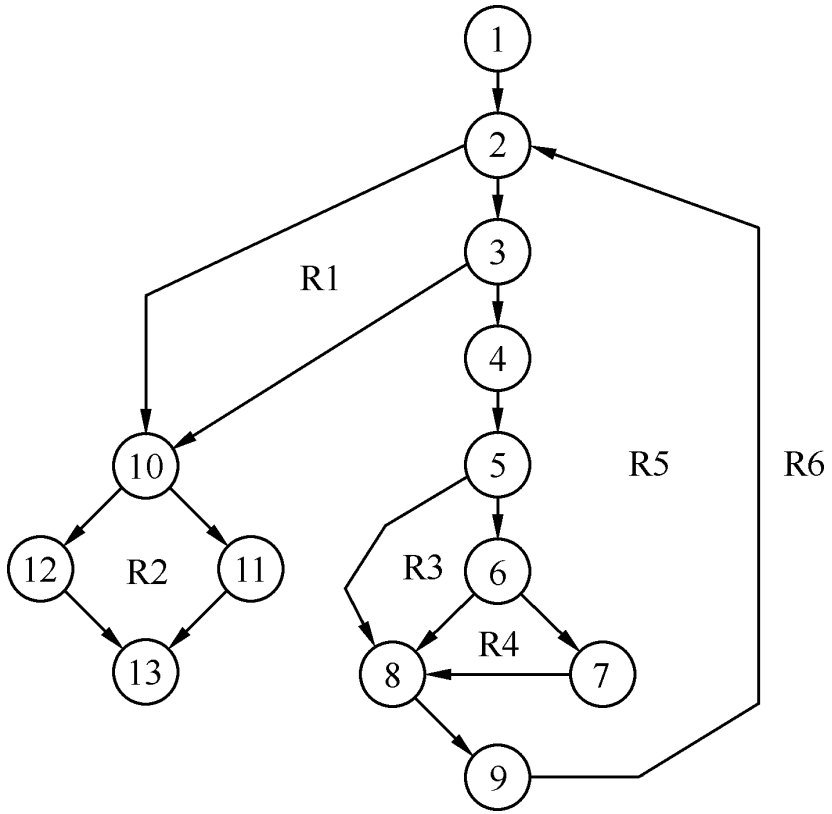
利用第五章介绍的符号和构造规则生成控制流图。对于以上用PDL描述的averagy过程 ，对将要映射为对应控制流图中一个结点的PDL语句或语句组，加上用数字表示的标号。加了标号的PDL程序及对应的控制流图如下所示。

10.3 基本路径覆盖

```
①  i = 1;
   total.input = total.valid = 0;
   sum = 0;
   DO WHILE value [i] <> -999 AND total.input<100
④  increment total.input by 1;
      IF value [i] >= minimum AND value [i] <= maximum
⑤        THEN increment total.valid by 1;
⑦             sum =sum + value [i];
         ELSE skip
⑧    ENDIF;
     increment i by 1;
⑨  ENDD0
      IF total.valid > 0
⑪       THEN averagy = sum/total.valid;
⑫       ELSE averagy = -999;
⑬  ENDIF
   END averagy
```

②、③、⑥、⑩ 为判定结点

对averagy过程定义结点



averagy过程的控制流图

10.3 基本路径覆盖

2. 由过程描述导出控制流图

利用在前面给出的计算控制流图环路复杂性的方法，算出控制流图G的环路复杂性。如果一开始就知道判断结点的个数，甚至不必画出整个控制流图，就可以计算出该图的环路复杂性的值。对于以上控制流图，可以算出：

V（G）=6（区域数）=5（判定结点数）+1=6

3. 确定线性无关的基本路径集

计算出的环路复杂性的值，就是该图已有的线性无关基本路径集中路径的数目。该图所有的6条路径是：

10.3 基本路径覆盖

- **path1：1-2-10-11-13**
- **path2：1-2-10-12-13**
- **path3：1-2-3-10-11-13**
- **path4：1-2-3-4-5-8-9-2……**
- **path5：1-2-3-4-5-6-8-9-2……**
- **path6：1-2-3-4-5-6-7-8-9-2……**

路径4、5、6后面的省略号（……）表示在控制结构中以后剩下的路径是可选择的。在很多情况下，标识判断结点，常常能够有效地帮助导出测试用例。在上例中，结点2、3、5、6和10都是判断结点。

10.3 基本路径覆盖

4. 准备测试用例，确保基本路径集中的每一条路径的执行

根据判断结点给出的条件，选择适当的数据以保证某一条路径可以被测试到。满足上例基本路径集的测试用例如下。

path1：输入数据：value [k] =有效输入，限于k<i（i定义如下）
value [i] =−999，　当2≤i≤100

预期结果：*n*个值的正确的平均值、正确的总计数。

注意：不能孤立地进行测试，应当作为路径4、5、6测试的一部分来测试。

path2：输入数据：value [1] =−999

预期结果：平均值=−999，总计数取初始值。

path3：输入数据：试图处理101个或更多的值，而前100个应当是有效的值。

预期结果：与测试用例1相同。

10.3 基本路径覆盖

path4：输入数据：value [i] =有效输入，且i<100
value [k] <最小值，当k<i时
预期结果：*n*个值的正确的平均值，正确的总计数
path5：输入数据：value [i] =有效输入，且i<100
value [k] >最大值，当k≤i时
预期结果：*n*个值的正确的平均值，正确的总计数
path6：输入数据：value [i] =有效输入，且i<100
预期结果：*n*个值的正确的平均值，正确的总计数

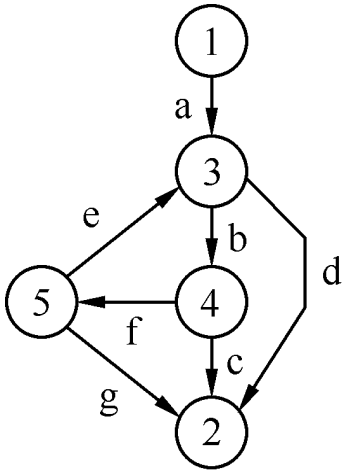
每个测试用例执行之后，与预期结果进行比较。如果所有测试用例都执行完毕，则可以确信程序中所有的可执行语句至少被执行了一次。但是必须注意的是，一些独立的路径（如此例中的路径1），往往不是完全孤立的，有时它是程序正常的控制流的一部分，这时，这些路径的测试可以是另一条路径测试的一部分。

10.3 基本路径覆盖

5. 图形矩阵

- **图形矩阵是在基本路径测试中起辅助作用的软件工具，利用它可以实现自动地确定一个基本路径集。**
- **一个图形矩阵是一个方阵，其行/列数等于控制流图中的结点数。**
- **每行和每列依次对应到一个被标识的结点，矩阵元素对应到结点间的连接（即边）。**

10.3 基本路径覆盖



控制流图

结点 \ 相连结点	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

图形矩阵

在控制流图中对每一条边加上一个连接权，图形矩阵就成为测试过程中评价程序控制结构的工具。连接权提供了关于控制流的附加信息。最简单的情形，连接权为“1”，表示存在一个连接，或者为“0”，表示不存在一个连接。但在其他情况，连接权可以表示如下特性：连接（边）执行的可能性（概率）、通过一个连接需花费的时间、在通过一个连接时所需的存储、在通过一个连接时所需的资源。

10.3 基本路径覆盖

为了举例说明，用最简单的权限（0或者1）来表明连接。下图为上图图形矩阵改画后的结果。每个字母用“1”取代，表明存在一个连接。在图中，“0”未画出。采用这种表示时，图形矩阵称为连接矩阵。

图中，若一行有2个或更多的元素，则这行所代表的结点一定是判定结点。因而通过计算排列在连接矩阵右边的算式，可以得到确定该图环路复杂性的另一种方法。

结点 \ 相连结点	1	2	3	4	5
1			1		
2					
3		1		1	
4		1			1
5		1	1		

连接矩阵

连接			
1	−1	=	0
2	−1	=	1
2	−1	=	1
2	−1	=	1
环路复杂性	3 + 1	=	4

10.4 黑盒测试的测试用例设计

- 等价类划分
  - **等价类划分是一种典型的黑盒测试方法，也是一种非常实用的重要测试方法,它是用来解决如何选择适当的子集，使其尽可能多地发现错误。**
  - **使用这一方法设计测试用例要经历划分等价类（列出等价类表）和选取测试用例两步。**

10.4 黑盒测试的测试用例设计

1. 划分等价类

- 所谓等价类是指某个输入域的子集合。在该子集合中，各个输入数据对于揭露程序中的错误都是等效的，并合理地假定：测试某等价类的代表值等价于对这一类其他值的测试。
- 或者说，如果某个等价类中的一个数据作为测试数据进行测试查出了错误，那么使用这一等价类中的其他数据进行测试也会查出同样的错误；
- 反之，若使用某个等价类中的一个数据作为测试数据进行测试没有查出错误，则使用这个等价类中的其他数据也同样查不出错误。

10.4 黑盒测试的测试用例设计

等价类的划分有两种不同的情况：

（1）**有效等价类**：是指对于程序的规格说明来说，是合理的、有意义的输入数据构成的集合。利用它，可以检验程序是否实现了规格说明预先规定的功能和性能。

（2）**无效等价类**：是指对于程序的规格说明来说，是不合理的、无意义的输入数据构成的集合。程序员主要利用这一类测试用例检查程序中功能和性能的实现是否有不符合规格说明要求的地方。

在设计测试用例时，要同时考虑有效等价类和无效等价类的设计。

10.4 黑盒测试的测试用例设计

以下结合具体实例给出几条划分等价类的原则。

（1）如果输入数据规定了取值范围或值的个数，则可以确定一个有效等价类和两个无效等价类。例如，在程序的规格说明中，对输入数据有一句话：

“……项数可以从1到999……”

则有效等价类是“1≤项数≤999”，两个无效等价类是“项数<1”及“项数>999”。在数轴上表示为

10.4 黑盒测试的测试用例设计

（2）如果规格说明规定了数据值的集合，或者是规定了“必须如何”的条件，这时可确定一个有效等价类和一个无效等价类。例如，在PASCAL语言中对变量标识符规定为“以字母打头的……串”，那么所有以字母打头的串构成有效等价类，而不在此集合内（不以字母打头）的串归于无效等价类。

（3）如果规格说明中规定的是一个条件数据，则可确定一个有效等价类和一个无效等价类。例如：“……成人（年满18岁）须……”，则考虑成人为一有效等价类；未满18岁者为无效等价类。

（4）如果我们确知，已划分的等价类中各元素在程序中的处理方式不同，则应将此等价类进一步划分成更小的等价类。

10.4 黑盒测试的测试用例设计

2. 确定测试用例

在确定了等价类之后，建立等价类表，列出所有划分出的等价类如下：

输 入 数 据	有效等价类	无效等价类
……	……	……
……	……	……

再从划分出的等价类中按以下原则选择测试用例。

（1）为每一个等价类规定一个唯一的编号。

（2）设计一个新的测试用例，使其尽可能多地覆盖尚未被覆盖的有效等价类，重复这一步，直到所有的有效等价类都被覆盖为止。

（3）设计一个新的测试用例，使其仅覆盖一个尚未被覆盖的无效等价类，重复这一步，直到所有的无效等价类都被覆盖为止。

10.4 黑盒测试的测试用例设计

3. 用等价类划分法设计测试用例的实例

在某一PASCAL语言版本中规定：“标识符是由字母开头、后跟字母或数字的任意组合构成。有效字符数为8个，最大字符数为80个。”并且规定：“标识符必须先说明，再使用。”“在同一说明语句中，标识符至少必须有一个。”

为用等价类划分的方法得到上述规格说明所规定的要求，本着前述的划分原则，建立输入等价类表，如下表所示（表中括号中的数字为等价类编号）。

输 入 数 据	有效等价类	无效等价类
标识符个数	1个（1），多个（2）	0个（3）
标识符字符数	1～80个（4）	0个（5），>80个（6）
标识符组成	字母（7），数字（8）	非字母数字字符（9），保留字（10）
第一个字符	字母（11）	非字母（12）
标识符使用	先说明后使用（13）	未说明已使用（14）

10.4 黑盒测试的测试用例设计

下面选取了8个测试用例，它们覆盖了所有的等价类。

① VAR x，T1234567：REAL；} (1) (2) (4) (7) (8) (11) (13)

BEGIN x：=3.414；T1234567：=2.732；……

② VAR ：REAL； } (3)

③ VAR x，：REAL； } (5)

④ VAR T12345……：REAL； } (6) 多于80个字符

⑤ VAR T $：CHAR； } (9)

⑥ VAR GOTO：INTEGER； } (10)

⑦ VAR 2T：REAL； } (12)

⑧ VAR PAR：REAL； } (14)

BEGIN……

PAP：=SIN (3.14*0.8) /6；

10.4 黑盒测试的测试用例设计

- 边界值分析

1. 边界值分析方法的考虑

➢**边界值分析**也是一种黑盒测试方法，是对等价类划分方法的补充。人们从长期的测试工作经验中得知，大量的错误是发生在输入或输出范围的边界上，而不是在输入范围的内部。

➢这里所说的边界是指，相当于输入等价类和输出等价类而言，稍高于其边界值及稍低于其边界值的一些特定情况。

10.4 黑盒测试的测试用例设计

2. 选择测试用例的原则

(1)如果输入数据规定了值的范围，则应取刚达到这个范围的边界的值，以及刚刚超越这个范围边界的值作为测试输入数据。

例如，若输入值的范围是“–1.0～1.0”，则可选取“–1.0”，“1.0”，“–1.001”，“1.001”作为测试输入数据。

(2) 如果输入数据规定了值的个数，则用最大个数、最小个数、比最大个数多1、比最小个数少1的数作为测试数据。例如，一个输入文件有1～255个记录，设计测试用例时则可以分别设计有1个记录、255个记录以及0个记录和256个记录的输入文件。

10.4 黑盒测试的测试用例设计

（3）根据规格说明的每个输出数据，使用前面的原则(1)。

例如，某程序的功能是计算折扣量，最低折扣量是0元，最高折扣量是1 050元，则设计一些测试用例，使它们恰好产生0元和1 050元 的结果。

此外，还可考虑设计结果为负值或大于1 050元的测试用例。

由于输入值的边界不与输出值的边界相对应，所以要检查输出值的边界不一定可能，要产生超出输出值值域之外的结果也不一定办得到。尽管如此，必要时还需一试。

10.4 黑盒测试的测试用例设计

（4）根据规格说明的每个输出数据，使用前面的原则(2)。

例如，一个信息检索系统根据用户打入的命令，显示有关文献的摘要，但最多只显示4篇摘要。

这时可设计一些测试用例，使得程序分别显示1篇、4篇、0篇摘要，并设计一个有可能使程序错误地显示5篇摘要的测试用例。

10.4 黑盒测试的测试用例设计

（5）如果程序的规格说明给出的输入域或输出域是有序集合（如有序表，顺序文件等），则应选取集合的第一个元素和最后一个元素作为测试用例。

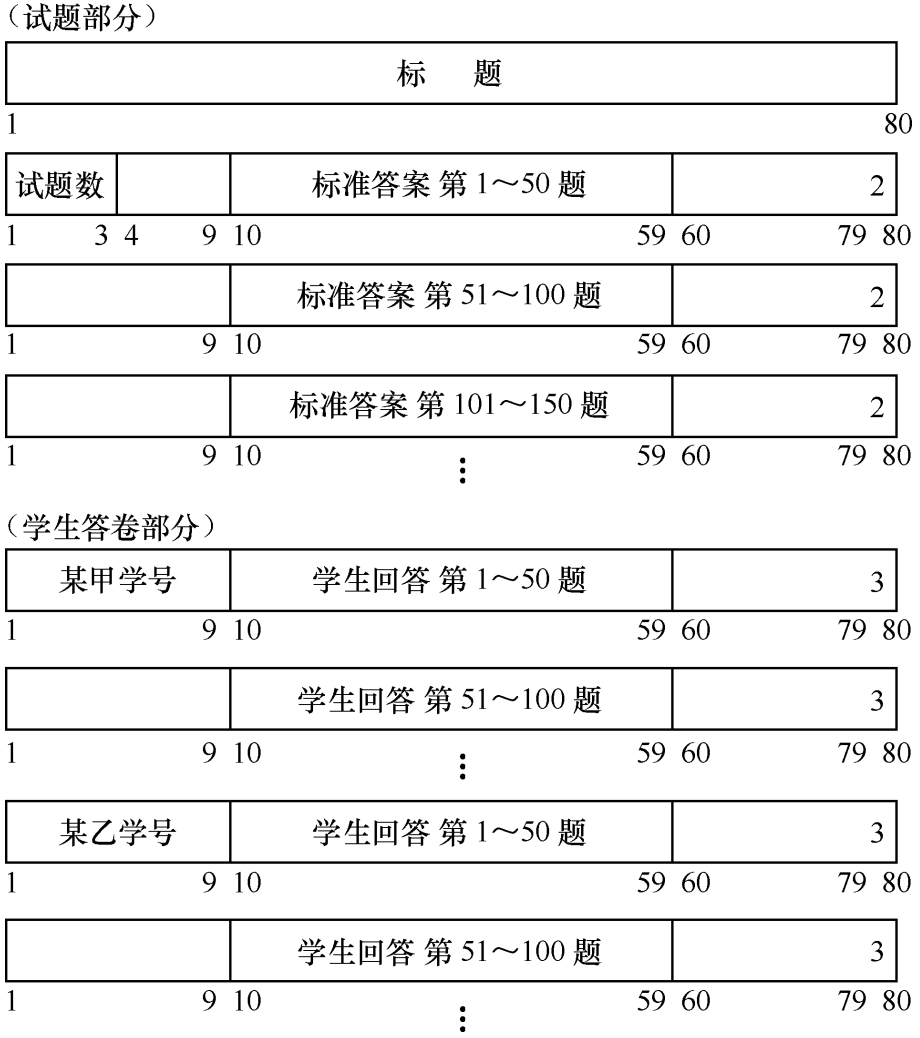
（6）如果程序中使用了一个内部数据结构，则应当选择这个内部数据结构的边界上的值作为测试用例。例如，如果程序中定义了一个数组，其元素下标的下界是0，上界是100，那么应选择达到这个数组下标边界的值，如0与100，作为测试用例。

（7）分析规格说明，找出其他可能的边界条件。

10.4 黑盒测试的测试用例设计

3. 应用边界值分析方法设计测试用例的实例

假如一个为学生标准化考试批阅试卷、产生成绩报告的程序，程序的输入文件由一些有80个字符的记录（卡片）组成。输入数据记录格式如右图所示。



10.4 黑盒测试的测试用例设计

把以上所有记录分为3组:

（1）标题。这一组只有一个记录，其内容是成绩报告的名字。

（2）各题的标准答案。每个记录均在第80个字符处标以数字“2”。该组的第1个记录的第1～3个字符为试题数（取值为1～999）。第10～59个字符给出第1～50题的标准答案（每个合法字符表示一个答案）。该组的第2、第3……个记录相应为第51～100题、第101～150题……题的标准答案。

（3）学生的答卷。每个记录均在第80个字符处标以数字“3”，每个学生的答卷在若干个记录中给出。例如，某甲的首记录第1～9个字符给出学生的学号，第10～59个字符列出的是某甲所作的第1～50题的解答。若试题数超过50，则其第2、第3……个记录分别给出的第51～100题、第101～150……题的解答。然后是某乙的答卷记录。学生人数不超过200人，试题个数不超过999。程序的输出有4个报告。

10.4 黑盒测试的测试用例设计

① 按学号排列的成绩单，列出每个学生的成绩（百分制）、名次。

② 按学生成绩排序的成绩单。

③ 平均分数及标准偏差的报告。

④ 试题分析报告。按试题号排列，列出各题学生答对的百分比。

下面分别考虑输入数据和输出数据，以及边界条件，选择测试用例。

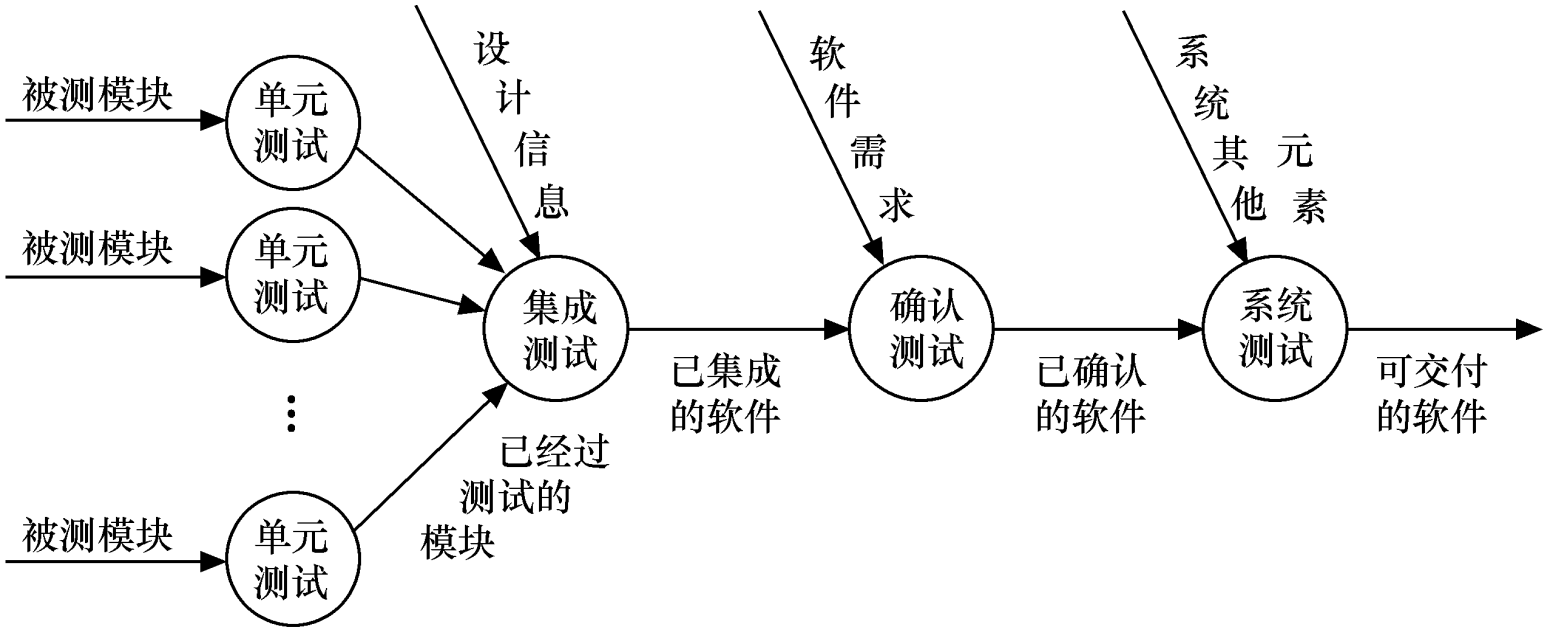
输 入 数 据	测 试 用 例
输入文件	[空输入文件]
标题	[没有标题记录] [标题只有一个字符] [标题有 80 个字符]
试题数	[试题数为 1] [试题数为 50] [试题数为 51] [试题数为 100] [试题数为 999] [试题数为 0] [试题数含有非数字字符]
标准答案记录	[没有标准答案记录，有标题] [标准答案记录多一个] [标准答案记录少一个]
学生人数	[0 个学生] [1 个学生] [200 个学生] [201 个学生]
学生答题	[某学生只有一个回答记录，但有两个标准答案记录] [该学生是文件中的第一个学生] [该学生是文件中最后一个学生（记录数出错的学生）]
学生答题	[某学生有两个回答记录，但只有一个标准答案记录] [该学生是文件中第一个学生（指记录数出错的学生）] [该学生是文件中最后一个学生]

10.4 黑盒测试的测试用例设计

输 出 数 据	测 试 用 例
学生成绩	[所有学生的成绩都相等] [每个学生的成绩都互不相同] [部分（不是全体）学生的成绩相同（检查是否能按成绩正确排名次）] [有个学生得 0 分] [有个学生得 100 分]
输出报告 ① ②	[有个学生的学号最小（检查按学号排序是否正确）] [有个学生的学号最大（检查按学号排序是否正确）] [适当的学生人数，使产生的报告刚好印满一页（检查打印页数）] [学生人数使报告印满一页尚多出 1 人（检查打印换页）]
输出报告 ③	[平均成绩为 100 分（所有学生都得满分）] [平均成绩为 0 分（所有学生都得 0 分）] [标准偏差为最大值（有一半学生得 0 分，其他 100 分）] [标准偏差为 0（所有学生的成绩都相等）]
输出报告 ④	[所有学生都答对了第一题] [所有学生都答错了第一题] [所有学生都答对了最后一题] [所有学生都答错了最后一题] [选择适当的试题数，使第四个报告刚好打满一页] [试题数使报告打满一页后，刚好剩下一题未打]

10.5 软件测试的策略

通常**软件测试过程**按4个步骤进行，即**单元测试**、**组装测试**、**确认测试**和**系统测试**。如下图所示。



10.5 软件测试的策略

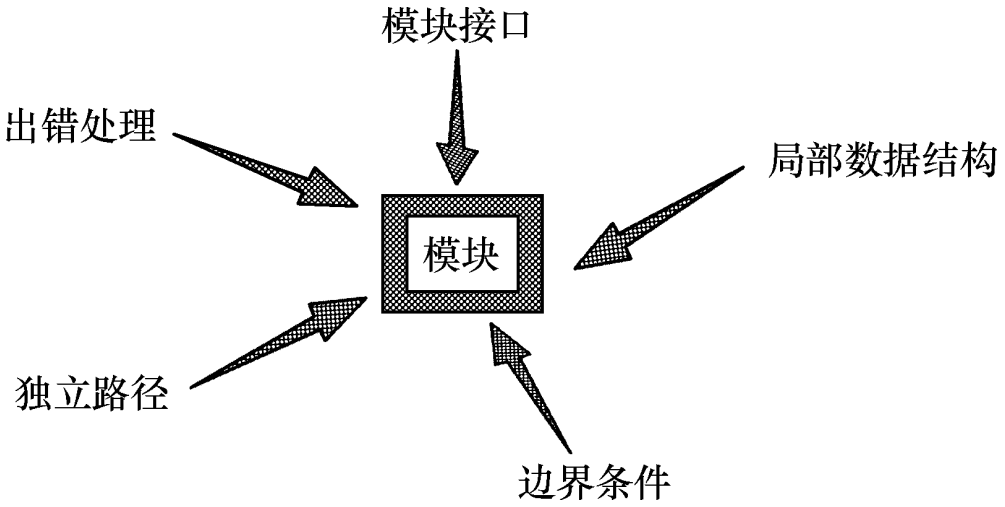
- 单元测试

单元测试（unit testing）**又称模块测试，是针对软件设计的最小单位—程序模块，进行正确性检验的测试工作。其目的在于发现各模块内部可能存在的各种差错。单元测试需要从程序的内部结构出发设计测试用例。多个模块可以平行地独立进行单元测试。**

1．单元测试的内容

单元测试主要采用白盒测试方法设计测试用例，辅之以黑盒测试的测试用例，使之对任何合理的输入和不合理的输入，都能鉴别和响应。在单元测试中进行的测试工作如下图所示，需要在5个方面对被测模块进行检查。

10.5 软件测试的策略



（1）模块接口测试。在单元测试的开始，应对通过被测模块的数据流进行测试。对模块接口可能需要如下的测试项目：调用本模块时的输入参数与模块的形式参数的匹配情况；本模块调用子模块时，它输入给子模块的参数与子模块中的形式参数的匹配情况；是否修改了只作输入用的形式参数；全局量的定义在各模块中是否一致；限制是否通过形式参数来传送。

10.5 软件测试的策略

（2）局部数据结构测试。模块的局部数据结构是最常见的错误来源，应设计测试用例以检查以下各种错误：不正确或不一致的数据类型说明；使用尚未赋值或尚未初始化的变量；错误的初始值或错误的默认值；变量名拼写错；不一致的数据类型。可能的话，除局部数据之外的全局数据对模块的影响也需要查清。

（3）路径测试。选择适当的测试用例，对模块中重要的执行路径进行测试。应当设计测试用例查找由于错误的计算、不正确的比较或不正常的控制流而导致的错误。对基本执行路径和循环进行测试可以发现大量的路径错误。

10.5 软件测试的策略

（4）**错误处理测试**。比较完善的模块设计要求能预见出错的条件，并设置适当的出错处理，以便在一旦程序出错时，能对出错程序重作安排，保证其逻辑上的正确性。若出现下列情况之一，则表明模块的错误处理功能包含有错误或缺陷：出错的描述难以理解；出错的描述不足以对错误定义，不足以确定出错的原因；显示的错误与实际的错误不符；对错误条件的处理不正确；在对错误进行处理之前，错误条件已经引起系统的干预等。

（5）**边界测试**。在边界上出现错误是常见的，要特别注意数据流、控制流中刚好等于、大于或小于确定的比较值时出错的可能性，对这些地方要仔细地选择测试用例，认真加以测试。

10.5 软件测试的策略

2. 单元测试的步骤

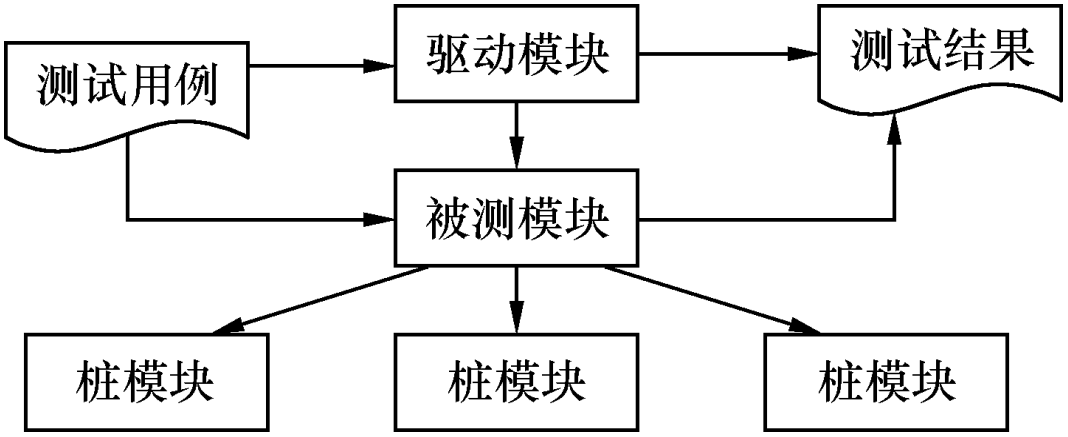
通常单元测试是在编码阶段进行的。在源程序代码编制完成，经过评审和验证，肯定没有语法错误之后，就开始进行单元测试的测试用例设计。

模块并不是一个独立的程序，在考虑测试模块时，同时要考虑它和外界的联系，用一些辅助模块去模拟与被测模块相联系的其他模块。这些辅助模块分为如下两种。

（1）驱动模块（driver）——相当于被测模块的主程序，它接收测试数据，并把这些数据传送给被测模块，最后再输出实测结果。

10.5 软件测试的策略

（2）桩模块（stub）——也叫做存根模块，用以代替被测模块调用的子模块。桩模块可以做少量的数据操作，不需要把子模块所有功能都带进来，但不允许什么事情也不做。被测模块、与它相关的驱动模块及桩模块共同构成了一个“测试环境”，如下图所示。



10.5 软件测试的策略

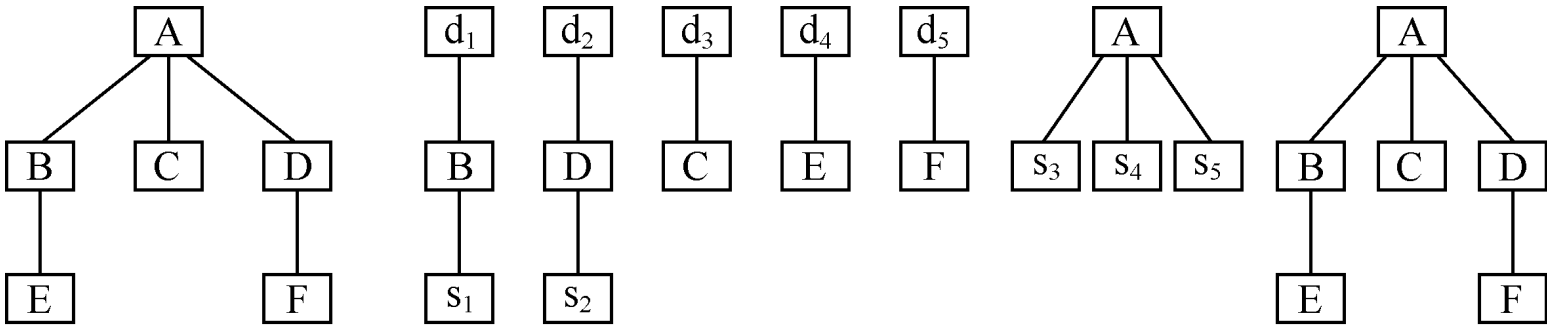
- 组装测试

组装测试（integrated testing）也叫做**集成测试**或**联合测试**。通常，在单元测试的基础上，需要将所有模块按照设计要求组装成为系统，把模块组装为系统的方式有两种：**一次性组装方式**（big bang）和**增值式组装方式**。

1．一次性组装方式

它是一种**非增值式**组装方式，也叫做**整体拼装**。使用这种方式，首先对每个模块分别进行模块测试，然后再把所有模块组装在一起进行测试，最终得到要求的软件系统。例如，有一个模块系统结构，如下图（a）所示，其单元测试和组装顺序（b）所示。

10.5 软件测试的策略



（a）　　（b）

上图中，模块d1，d2，d3，d4，d5是对各个模块作单元测试时建立的驱动模块，s1，s2，s3，s4，s5是为单元测试而建立的桩模块。这种一次性组装方式试图在辅助模块的协助下，在分别完成模块单元测试的基础上，将被测模块连接起来进行测试。但是，由于程序中不可避免地存在涉及模块间接口、全局数据结构等方面的问题，所以一次试运行成功的可能性不很大。

10.5 软件测试的策略

2. 增值式组装方式

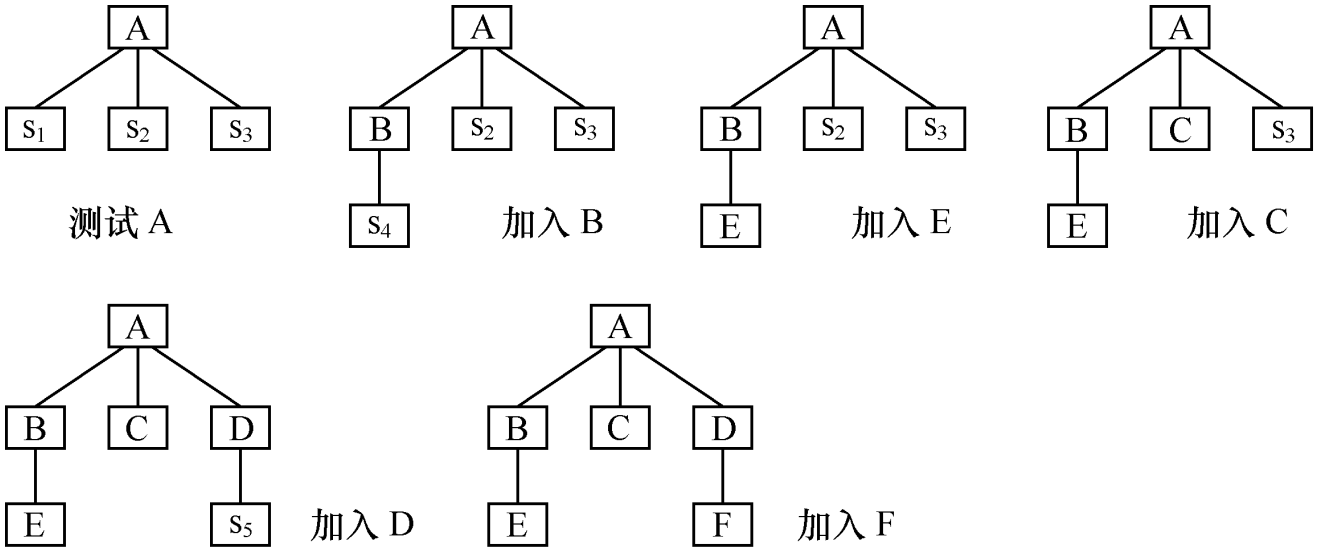
这种组装方式又称**渐增式组装**，首先是对一个个模块进行模块测试，然后将这些模块逐步组装成较大的系统，在组装的过程中边连接边测试，以发现连接过程中产生的问题。最后通过增值逐步组装成为要求的软件系统。增值组装有以下3种做法。

（1）**自顶向下的增值方式**。这种组装方式是将模块按系统程序结构，沿控制层次自顶向下进行组装，其步骤如下：

① 以主模块为被测模块兼驱动模块，所有直属于主模块的下属模块全部用桩模块代替，对主模块进行测试。

10.5 软件测试的策略

② 采用深度优先（如下图）或宽度优先的策略，逐步用实际模块替换已用过的桩模块，再用新的桩模块代替它们的直接下属模块，与已测试的模块或子系统组装成新的子系统。



按深度方向组装的例子

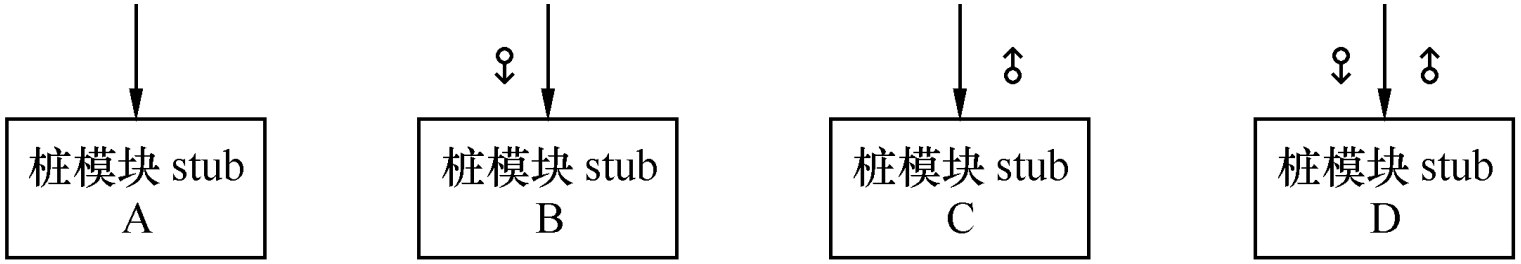
10.5 软件测试的策略

③ 进行回归测试（即重新执行以前做过的全部测试或部分测试），排除组装过程中引入新的错误的可能。

④ 判断是否所有的模块都已组装到系统中，若是则结束测试，否则转到②去执行。

自顶向下的组装和测试存在一个逻辑次序问题。在为了充分测试较高层的处理而需要较低层处理的信息时，就会出现这类问题。在自顶向下组装阶段，还需要用桩模块代替较低层的模块，根据不同情况，桩模块的编写，可能如下所示的几种选择。

10.5 软件测试的策略



显示跟踪信息

显示传递的信息

从一个表（或外部文件）返回一个值

进行一项表查询以根据输入参数返回输出参数

⇳ ⇕ 表示传递的数据信息

为了能够准确地实施测试，应当让桩模块正确而有效地模拟子模块的功能和合理的接口，不能是只包含返回语句或只显示该模块已调用信息，不执行任何功能的哑模块。

10.5 软件测试的策略

（2）**自底向上的增值方式**。这种组装方式是从程序模块结构的最底层的模块开始组装和测试。因为模块是自底向上进行组装，对于一个给定层次的模块，它的子模块（包括子模块的所有下属模块）已经组装并测试完成，所以不再需要桩模块。在模块的测试过程中需要从子模块得到的信息可以由直接运行子模块得到。

自底向上增值的步骤如下：

① 由驱动模块控制最底层模块的并行测试；也可以把最底层模块组合成实现某一特定软件功能的簇，由驱动模块控制它进行测试。

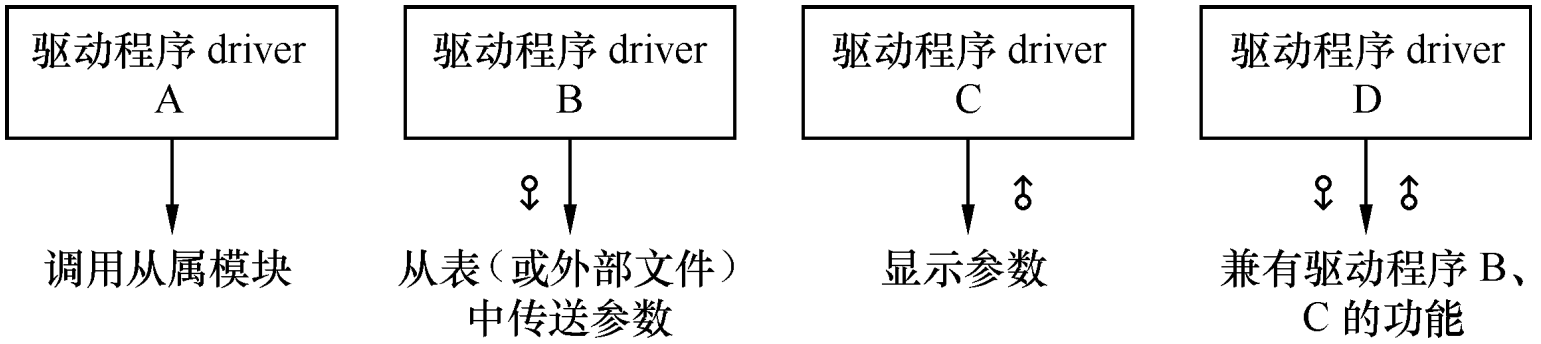
② 用实际模块代替驱动模块，与它已测试的直属子模块组装成为子系统。

10.5 软件测试的策略

③ 为子系统配备驱动模块，进行新的测试。

④ 判断是否已组装到达主模块。若是则结束测试，否则执行②。

自底向上进行组装和测试时，需要为被测模块或子系统编制相应的驱动模块。常见的几种类型的驱动模块如下图所示。



10.5 软件测试的策略

（3）混合增值式测试。自顶向下增值的方式和自底向上增值的方式各有优缺点。自顶向下增值方式的缺点是需要建立桩模块。自底向上增值方式的缺点是“程序一直未能作为一个实体存在，直到最后一个模块加上去后才形成一个实体”。也就是说，在自底向上组装和测试的过程中，对主要的控制直到最后才接触到。

鉴于此，通常是把以上两种方式结合起来进行组装和测试。下面简单介绍3种常见的综合增值方式测试。

① 衍变的自顶向下的增值测试：它的基本思想是强化对输入/输出模块和引入新算法模块的测试，并自底向上组装成为功能相当完整且相对独立的子系统，然后由主模块开始自顶向下进行增值测试。

10.5 软件测试的策略

② 自底向上—自顶向下的增值测试：它首先对含读操作的子系统自底向上直至根结点模块进行组装和测试，然后对含写操作的子系统作自顶向下的组装与测试。

③ 回归测试：这种方式采取自顶向下的方式测试被修改的模块及其子模块，然后将这一部分视为子系统，再自底向上测试，以检查该子系统与其上级模块的接口是否适配。

3．组装测试的组织和实施

组装测试是一种正规测试过程，必须精心计划，并与单元测试的完成时间协调起来。在制定测试计划时，应考虑如下因素：

（1）采用何种系统组装方法进行组装测试。

（2）组装测试过程中连接各个模块的顺序。

10.5 软件测试的策略

（3）模块代码编制和测试进度是否与组装测试的顺序一致。

（4）测试过程中是否需要专门的硬件设备。

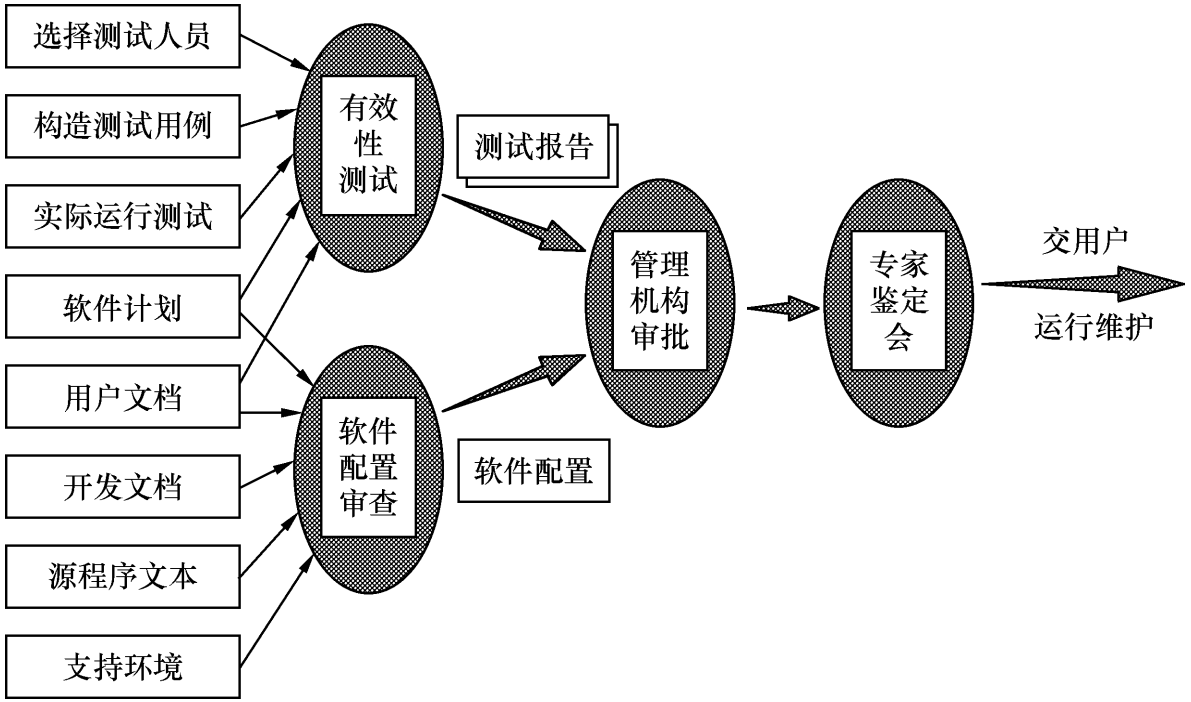
解决了上述问题之后，就可以列出各个模块的编制、测试计划表，标明每个模块单元测试完成的日期、首次组装测试的日期、组装测试全部完成的日期，以及需要的测试用例和所期望的测试结果。

在完成预定的组装测试工作之后，测试小组应负责对测试结果进行整理、分析，形成测试报告。测试报告中要记录实际的测试结果，在测试中发现的问题，解决这些问题的方法以及解决之后再次测试的结果。此外，还应提出目前不能解决、还需要管理人员和开发人员注意的一些问题，提供测试评审和最终决策，以提出处理意见。

10.5 软件测试的策略

- 确认测试

确认测试（validation testing）又称有效性测试。它的任务是验证软件的有效性，即验证软件的功能和性能及其他特性是否与用户的要求一致。在确认测试阶段需要做的工作如下图所示。



10.5 软件测试的策略

从上图中可看出，首先要进行有效性测试以及软件配置复审，然后进行验收测试和安装测试，在通过了专家鉴定之后，才能成为可交付的软件。

1．进行有效性测试（黑盒测试）

有效性测试是在模拟的环境（可能就是开发的环境）下，运用黑盒测试的方法，验证被测软件是否满足需求规格说明书列出的需求。为此，需要首先制订测试计划，规定要进行测试的种类。还需要制订一组测试步骤，描述具体的测试用例。通过实施预定的测试计划和测试步骤，确定软件的特性是否与需求相符，确保所有的软件功能需求都能得到满足，所有的软件性能需求都能达到，所有的文档都正确且便于使用。同时，对其他软件需求，如可移植性、兼容性、出错自动恢复、可维护性等，也都要进行测试，确认是否满足。

10.5 软件测试的策略

2．软件配置复查

软件配置复查的目的是保证软件配置的所有成分都齐全，各方面的质量都符合要求，具有维护阶段所必须的细节，而且已经编排好分类的目录。

除了按合同规定的内容和要求，由人工审查软件配置之外，在确认测试的过程中，应当严格遵守用户手册和操作手册中规定的使用步骤，以便检查这些文档资料的完整性和正确性。必须仔细记录发现的遗漏和错误，并且适当地补充和改正。

10.5 软件测试的策略

3．α测试和β测试

在软件交付使用之后，用户将如何实际使用程序，对于开发者来说是无法预测的。如果软件是为多个用户开发的产品，让每个用户逐个执行正式的验收测试是不切实际的。很多软件产品生产者采用一种称之为α测试和β测试的测试方法，以发现可能只有最终用户才能发现的错误。

α测试是由一个用户在开发环境下进行的测试，也可以是公司内部的用户在模拟实际操作环境下进行的测试。软件在一个自然设置状态下使用，开发者坐在用户旁边，随时记下错误情况和使用中的问题。 α测试的目的是评价软件产品的FLURPS（即功能、局域化、可使用性、可靠性、性能和支持），尤其注重产品的界面和特色。

10.5 软件测试的策略

β测试是由软件的多个用户在一个或多个用户的实际使用环境下进行的测试。这些用户是与公司签定了支持产品预发行合同的外部客户。与α测试不同的是，开发者通常不在测试现场，由用户记下遇到的所有问题。开发者在综合用户的报告之后进行修改，最后将软件产品交付给全体用户使用。β测试主要衡量产品的FLURPS，着重于产品的支持性，包括文档、客户培训和支持产品生产能力。只有当α测试达到一定的可靠程度时，才能开始β测试。

由于β测试的主要目标是测试可支持性，所以β测试应尽可能由主持产品发行的人员管理。

10.5 软件测试的策略

4．验收测试

在通过了系统的有效性测试及软件配置审查之后，应开始系统的验收测试（acceptance testing）。**验收测试**是以用户为主的测试，软件开发人员和QA（质量保证）人员也应参加。由用户参加设计测试用例，使用用户界面输入测试数据，并分析测试的输出结果。一般使用生产中的实际数据进行测试，在测试过程中，除了考虑软件的功能和性能外，还应对软件的可移植性、兼容性、可维护性、错误的恢复功能等进行确认。

10.5 软件测试的策略

5．确认测试的结果

在全部确认测试的测试用例运行完后，所有的测试结果可以分为两类。

（1）测试结果与预期的结果相符，这说明软件的这部分功能或性能特征与需求规格说明书相符合，从而这部分程序可以接受。

（2）测试结果与预期的结果不符，这说明软件的这部分功能或性能特征与需求规格说明不一致，因此，需要开列一张软件各项缺陷表或软件问题报告，通过与用户的协商，解决所发现的缺陷和错误。

10.5 软件测试的策略

- 系统测试

系统测试（system testing）是将通过确认测试的软件，作为整个计算机系统的一个元素，与计算机硬件、外设、某些支持软件、数据、人员等其他系统元素结合在一起，在实际运行（使用）环境下，对计算机系统进行一系列的组装测试和确认测试。

系统测试的**目的**在于通过与系统的需求定义作比较，发现软件与系统定义不符合或与之矛盾的地方。系统测试的测试用例应根据系统的需求分析说明书设计，并在实际使用环境下运行。

10.5 软件测试的策略

- 测试的类型

软件测试实际上是由一系列不同的测试组成。几种常见的软件测试及它们与各个测试步骤中的关系如右图所示。

测试步骤 / 测试种类	开发阶段的测试					产品阶段的测试				
	设计	单元测试	模块测试	组装测试	部件测试	有效性测试	α测试	β测试	验收测试	系统测试
设计评审	M			S						
代码审查		M		H						S
功能测试(黑盒)		H	M	M	M	M	M	M	M	M
结构测试(白盒)		H	M		S					
回归测试			S	H	M	M				M
可靠性测试					H	M	M	M	M	M
强度测试					H	M				H
性能测试			S		H	M	M	M	M	H
恢复测试						M				
启动/停止测试						M				
配置测试					H	M				M
安全性测试						H				
可使用性测试					S	H	M	M		
可支撑性测试							H	M		
安装测试						M	M	M		
互连测试			S			M				M
兼容性测试					M	M				
容量测试					H	M				H
文档测试						M	S	H	M	

说明: M = 必要的（mandatory）　H = 积极推荐（highly recommended）
S = 建议使用（suggested）

10.5 软件测试的策略

上图中各类测试的定义如下：

（1）功能测试（function testing）：功能测试是在规定的一段时间内运行软件系统的所有功能，以验证这个软件系统有无严重错误。

（2）回归测试（regression testing）：这种测试用于验证对软件修改后有没有引出新的错误，或者说，验证修改后的软件是否仍然满足系统的需求规格说明。

（3）可靠性测试（reliability testing）：如果系统需求说明书中有对可靠性的要求，则需进行可靠性测试。通常使用平均失效间隔时间（MTBF）与因故障而停机的时间（MTTR）来度量系统的可靠性。

10.5 软件测试的策略

（4）**强度测试**（stress testing）：也称压力测试，是要检查在系统运行环境恶劣的情况下，系统可以运行到何种程度的测试。因此，进行强度测试，需要提供非正常数量、频率或总量资源来运行系统。实际上，这是对软件的“超负荷”环境或临界环境的运行检验。

（5）**性能测试**（performance testing）：是要检查系统是否满足在需求说明书中规定的性能。特别是对于实时系统或嵌入式系统，软件只满足要求的功能而达不到要求的性能是不可接受的，所以还需要进行性能测试。

（6）**恢复测试**（recovery testing）：恢复测试是要证实在克服硬件故障（包括掉电、硬件或网络出错等）后，系统能否正常地继续进行工作，并不对系统造成任何损害。

10.5 软件测试的策略

（7）**启动/停止测试**（startup/shutdown testing）：这类测试的目的是验证在机器启动及关机阶段，软件系统正确处理的能力。包括反复启动软件系统（例如，操作系统自举、网络的启动、应用程序的调用等），以及在尽可能多的情况下关机。

（8）**配置测试**（configuration testing）：这类测试是要检查计算机系统内各个设备或各种资源之间的相互连接和功能分配中的错误。配置测试主要包括以下3种。

① **配置命令测试**：验证全部配置命令的可操作性（有效性）；特别对最大配置和最小配置要进行测试。软件配置和硬件配置都要测试。

10.5 软件测试的策略

② 循环配置测试：证明对每个设备物理与逻辑的、逻辑与功能的每次循环置换配置都能正常工作。

③ 修复测试：检查每种配置状态及哪个设备是坏的，并用自动的或手工的方式进行配置状态间的转换。

（9）**安全性测试**（security testing）：检验在系统中已经存在的系统安全性和保密性措施是否发挥作用，有无漏洞。为此要了解破坏安全性的方法和工具，并设计一些模拟测试用例对系统进行测试，力图破坏系统的保护机构以进入系统。

（10）**可使用性测试**（usability testing）：可使用性测试主要从使用的合理性、方便性等角度对软件系统进行检查，以发现人为因素或使用上的问题。

10.5 软件测试的策略

（11）**可支持性测试**（supportability testing）：验证系统的支持策略对于公司与用户方面是否切实可行。它所采用的方法是试运行支持过程（如对有错部分打补丁的过程，热线界面等），对其结果进行质量分析，评审诊断工具、维护过程、内部维护文档；衡量修复一个明显错误所需的平均最少时间。还有一种常用的方法是，在发行前把产品交给用户，向用户提供支持服务的计划，从用户处得到对支持服务的反馈。

（12）**安装测试**（installation testing）：安装测试的目的不是查找软件错误，而是查找安装错误。在安装软件系统时，会有多种选择。要分配和装入文件与程序库，布置适用的硬件配置，进行程序的连接。而安装测试是要查找出在这些安装过程中出现的错误。

10.5 软件测试的策略

（13）互连测试（interoperability testing）：验证两个或多个不同的系统之间的互连性。这类测试对支持标准规格说明，或承诺支持与其他系统互连的软件系统有效。

（14）兼容性测试（compatibility testing）：验证软件产品在不同版本之间的兼容性。有两类基本的兼容性测试：向下兼容和交错兼容。向下兼容测试是测试软件新版本，保留它早期版本的功能的情况；交错兼容测试是要验证共同存在的两个相关但不同的产品之间的兼容性。

（15）容量测试（volume testing）：容量测试是要检验系统的能力最高能达到什么程度。

（16）文档测试（documentation testing）：检查用户文档（如用户手册）的清晰性和精确性。用户文档中所使用的例子必须在测试中一一试过，确保叙述正确无误。

10.6 人工测试

人工测试不要求在计算机上实际执行被测程序，而是以一些人工的模拟技术和一些类似动态分析所使用的方法对程序进行分析和测试。

- 静态分析

静态分析是要对源程序进行静态检验。通常采用以下方法进行。

1．生成各种引用表

在源程序编制完成后生成各种引用表，这是为了支持对源程序进行静态分析。这些表可用手工方式从源程序中提取所需的信息，也可借助于专用的软件工具自动生成。引用表按功能分类，有以下3种。

10.6 人工测试

（1）直接从表中查出说明/使用错误，如循环层次表、变量交叉引用表、标号交叉引用表等。

（2）为用户提供辅助信息，如子程序（宏、函数）引用表、等价（变量、标号）表、常数表等。

（3）用来作错误预测和程序复杂度计算，如操作符和操作数的统计表等。

常用的引用表有如下几种：

（1）**标号交叉引用表**：它列出在各模块中出现的全部标号。在表中标出标号的属性：已说明、未说明、已使用和未使用。表中还有在模块以外的全局标号、计算标号等。

10.6 人工测试

（2）**变量交叉引用表**：即变量定义与引用表。在表中标明各变量的属性：已说明、未说明、隐式说明，以及类型及使用情况。进一步还可区分是否出现在赋值语句的右边，是否属于COMMON变量、全局变量或特权变量等。

（3）**子程序、宏和函数表**：在表中，各个子程序、宏和函数的属性：已定义、未定义和定义类型；参数表：输入参数的个数、顺序和类型；输出参数的个数、顺序和类型；已引用、未引用、引用次数等。

（4）**等价表**：表中列出在等价语句或等值语句中出现的全部变量和标号。

（5）**常数表**：在表中列出全部数字常数和字符常数，并指出它们在哪些语句中首先被定义，即首先出现在哪些赋值语句的左部或哪些数据语句或参数语句中。

10.6 人工测试

2．静态错误分析

静态错误分析用于确定在源程序中是否有某类错误或“危险”结构，它有以下几种。

（1）**类型和单位分析**：为了发现源程序中数据类型、单位上的不一致性，建立一些程序语言的预处理程序，分析程序中在“下标”类型及循环控制变量方面的类型错误，以及通过使用一般的组合/消去规则，确定表达式的单位错误。

（2）**引用分析**：沿着程序的控制路径，检查程序变量的引用异常问题。

（3）**表达式分析**：对表达式进行分析，以发现和纠正在表达式中出现的错误，包括：

10.6 人工测试

① 在表达式中不正确地使用了括号造成错误；

② 数组下标越界造成错误；

③ 除式为零造成错误；

④ 对负数开平方，或对π求正切值造成错误；

⑤ 浮点数计算的误差。

（4）**接口分析**：分析接口的一致性错误，包括：

① 模块之间接口的一致性和模块与外部数据库之间接口的一致性；

② 过程和函数过程之间接口的一致性，全局变量和公共数据区在使用上的一致性。

10.6 人工测试

- 人工测试的几种形式

静态分析中进行人工测试的主要方法有桌前检查、代码评审和走查。经验表明，使用这种方法能够有效地发现30%～70%的逻辑设计和编码错误。

1．桌前检查

桌前检查（desk checking）是一种传统的检查方法，由程序员自己检查自己编写的程序。程序员在程序通过编译之后，进行单元测试设计之前，对源程序代码进行分析、检验并补充相关的文档，目的是发现程序中的错误。检查项目包括如下内容：

10.6 人工测试

（1）检查变量的交叉引用；

（2）检查标号的交叉引用；

（3）检查子程序、宏结构、函数；

（4）常量检查；

（5）标准检查；

（6）风格检查；

（7）比较控制流；

（8）选择、激活路径；

（9）对照程序的规格说明，详细阅读源代码；

（10）补充文档；

10.6 人工测试

2．代码评审

代码评审（code reading review）是由若干程序员和测试员组成一个评审小组，通过阅读、讨论和争议，对程序进行静态分析的过程。

代码评审分**两步**：

- 小组负责人提前把设计规格说明书、控制流程图、程序文本及有关要求、规范等分发给小组成员，作为评审的依据；
- 召开程序评审会。在会上，由程序员逐句讲解程序的逻辑。在此过程中，程序员或其他小组成员可以提出问题，展开讨论，审查错误是否存在。

10.6 人工测试

在会前，应当给评审小组每个成员准备一份常见错误的清单。这个常见错误清单也叫做检查表，它把程序中可能发生的各种错误进行分类，对每一类列举出尽可能多的典型错误，然后把它们制成表格，供会审时使用。这种检查表类似于本章单元测试中给出的检查表。在代码评审之后，需要做以下几件事。

（1）把发现的错误登记造表，并交给程序员。

（2）若发现错误较多，或发现重大错误，则在改正之后，再次组织代码评审。

（3）对错误登记表进行分析、归类、精练，以提高审议效果。

10.6 人工测试

3．走查

走查（walkthroughs）与代码评审基本相同，其过程分为两步。

（1）把材料先发给走查小组每个成员，让他们认真研究程序，然后再开会。开会的议程与代码评审不同，不是简单地读程序和对照错误检查表进行检查，而是让与会者“充当”计算机，即首先由测试组成员为被测程序准备一批有代表性的测试用例，提交给走查小组。走查小组开会，集体扮演计算机角色，让测试用例沿程序的逻辑运行一遍，随时记录程序的踪迹，供分析和讨论用。

（2）人们借助于测试用例的媒介作用，对程序的逻辑和功能提出各种疑问，结合问题开展热烈的讨论和争议，能够发现更多的问题。

10.7 调试

调试（debug）也称排错或纠错，它是紧跟在测试之后要做的工作，但与测试不同之处在于：测试着重于发现软件中有错，发现异常或软件运行的可疑之处；而**调试的任务**在于为错误确切地定位，找到出错的根源，并且通过修改程序将其排除。

一般地，**调试的步骤**如下：

（1）针对测试提供的信息，分析错误的外部表现形式，确定程序出错的位置。

（2）研究程序的相关部分，找出导致错误的内在原因。

（3）修改相关的程序段，如果是设计导致的错误，则需修改相关的设计，以排除错误。

10.7 调试

（4）重复执行以前发现错误的测试，以确认：

① 该错误确已通过修改而消除；

② 这次修改并未引进新的错误。

（5）如果重新测试表明修改无效，发生错误的现象仍然出现，则要撤销上述修改，再次进行信息分析，实施上述过程，直至修改有效为止。

That's All!