白盒测试

1、**白盒测试**又称为结构测试或逻辑驱动测试，是针对被测试程序单元内部如何工作的测试，特点是基于被测试程序的源代码，而不是软件的需求规格说明。

使用白盒测试方法时，测试者必须全面了解程序内部逻辑结果，检查程序的内部结构，从检查程序从逻辑着手，对相关的逻辑路径进行测试，最后的得出测试结果。

2、**采用白盒测试方法必须遵循以下几条原则，才能达到测试目的：**

（1）保证一个模块中的所有独立路径至少被测试一次

（2）所有逻辑值均需测试真(true)和假(false)两种情况

（3）检查程序的内部数据结构，保证其结构的有效性

（4）在上下边界及可操作范围内运行所有循环

白盒测试需要完全了解程序结构和处理过程，他按照程序内部逻辑测试程序，检验程序中每条通路是否按预定要求正确工作，也被称做程序员测试。

需要注意：全面的白盒测试将产生百分之百正确的程序，实际上是不可能的

3、**白盒测试分静态和动态两种：**

（1）静态白盒测试是在不执行的条件下有条理地仔细审查软件设计、体系结构和代码，从而找出软件缺陷的过程，有时也称为结构分析。

（2）动态白盒测试也称结构化测试，通过查看并使用代码的内部结构，设计和执行测试。

4、**静态白盒测试**主要通过审查、走查、检验等方法，来查找代码中的问题和缺陷。主要原因是为了尽早发现软件缺陷，以找出黑盒测试难以发现或隔离的软件缺陷。其次，为黑盒测试员在接受软件进行测试设计时，设计和应用测试用例提供思路。通过审查评论，可以确定有问题或者容易产生软件缺陷的特性范围。

正式审查的四要素：确定问题、遵守规则、准备、编写报告

正式审查的效果：正式审查的主要的目的是找出软件中存在的缺陷，除此之外，还可以形成一些间接的效果。如：程序员与程序、测试人员之间的交流，增强相互了解；程序员会更仔细的编程，提高正确率等。正式审查是把大家聚在一起讨论同一个项目问题的良机。

正式审查的几种类型：同事审查、走查、检验

在编程和审查程序代码时，建立相关的规范和标准，并坚持标准或规范。三个重要的原因：

（1）可靠性：坚持按照某种标准和规范编写的代码更加可靠和安全。

（2）可读性/维护性：符合设备标准和规范的代码易于阅读、理解和维护。

（3）移植性：代码符合设备标准，迁移到另一个平台就会轻而易举，甚至完全没有障碍。

编程标准的 4 个组成部分

①标题：描述标准包含的主题。

②标准（或规范）：描述标准或规范的内容。

③解释说明：给出标准背后的原因，以使程序员理解为什么这样是好的编程习惯。

④示例：给出如何使用标准的简单程序示例。

通用代码审查清单

（1）数据引用错误

数据引用错误是指使用未经正确声明和初始化的变量、常量、数组、字符串或记录而导致的软件缺陷。数据引用错误是缓冲区溢出的主要原因。

（2）数据声明错误

数据声明缺陷产生的原因是不正确地声明或使用变量和常量。

（3）计算错误

计算或运算错误就是计算无法得到预期的结果。

（4）比较错误

在使用比较和判断运算时产生的比较和判断错误，这种错误很可能是因为边界条件问题。

（5）控制流程错误

控制流程错误产生的原因是编程语言中循环等控制结构未按预期的方式工作。通常由计算或者比较错误直接或间接造成。

（6）子程序参数错误

子程序参数错误的来源是软件子程序不正确地传递数据。

（7）输入/输出错误

输入输出错误包括文件读取、接受键盘或鼠标输入，以及向打印机或屏幕等输出设备写入错误。

（8）其他检查

5、动态白盒测试方法主要是按一定步骤和方法生成测试用例，并驱动相关模块去执行程序并发现软件中的错误和缺陷。测试人员要求对被测系统内的程序结构有深入的认识，清楚程序的结构、各个组成部分及其之间的关联，以及其内部的运行原理、逻辑等。

内容包括的 4 部分：

（1）直接测试底层函数、过程、子程序和库。

（2）以完整程序的方式从顶层测试软件，有时根据对软件运行的了解调整测试用例。

（3）从软件获得读取变量和状态信息的访问权，以便确定测试结果与预期结果是否相符，同时强制软件以正常测试难以实现的方式运行。

（4）估算执行测试时“命中”的代码量和具体代码，然后调整测试，去掉多余的测试用例，补充遗漏的测试用例。

动态白盒测试方法主要包括：逻辑覆盖法、基本路径测试法、循环测试、程序插桩技术

6、逻辑覆盖法

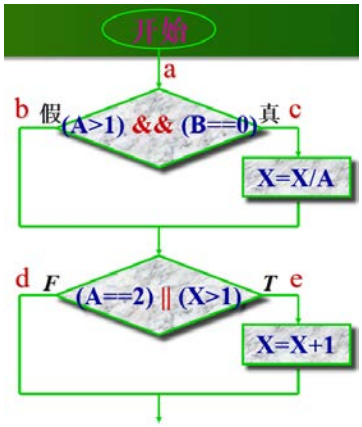
逻辑覆盖法是动态白盒测试中常用的测试技术，是一系列测试过程的总称。有选择地执行程序中的某些最具有代表性的通路来对尽穷测试的唯一可行的替代方法。

逻辑覆盖法的覆盖率是程序中一组被测试用例执行到的百分比。

覆盖率=（至少被执行一次的被测试项数）/被测试项总数

根据测试覆盖的目标不同，以及覆盖的程度不同，可由弱到强分为：语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖、条件组合覆盖、路径覆盖。

①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。



L1：（a → c → e）
L2：（a → b → d）
L3：（a → b → e）
L4：（a → c → d）

在例图中，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。

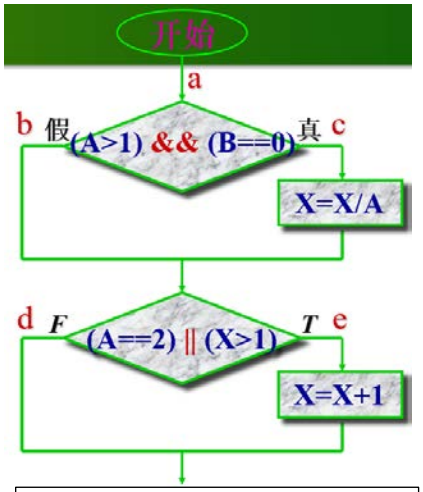
语句覆盖率：已执行的可执行语句占程序中可执行语句总数的百分比
注意：复杂的程序不可能达到语句的完全覆盖，语句覆盖率越高越好
语句覆盖的优点：（可以很直观地从源代码得到测试用例，无须细分每条判定表达式）
（1）检查所有语句
（2）结构简单的代码的测试效果较好
（3）容易实现自动测试
（4）代码覆盖率高
（5）如果是程序块覆盖，则不涉及程序块中的源代码
语句覆盖的缺点：由于这种测试方法仅仅针对程序逻辑中显式存在的语句，但对于隐藏的条件是无法测试的。如在多分支的逻辑运算中无法全面的考虑。语句覆盖是最弱的逻辑覆盖。
语句覆盖不能检查出的错误：
（1）逻辑运算（&&、||）错误，判定的第一个运算符“&&”错写成“||”，或第二个运算符“||”错写成“&&”，这时使用上述的测试用例仍然可以达到 100%的语句覆盖.
（2）循环语句错误，循环次数错误；跳出循环条件错误
语句覆盖存在的问题：语句覆盖看似语句覆盖率很高，但却有严重缺陷，比如：

```
if(x!=1)
{
      statements;
      ……;
}
else
{
      statement;
}
```

99句
1句

测试用例:
x = 2
语句覆盖率99%
50%的分支没有达到

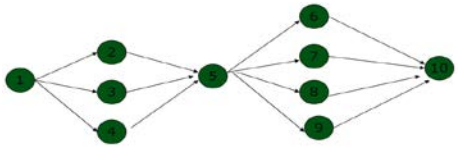
②判定覆盖就是设计若干个测试用例，运行被测程序，使得程序中每个判断的取真分支和取假分支至少经历一次。
判定覆盖又称为分支覆盖



L1：（a → c → e）
L2：（a→ b → d）
L3：（a→ b→ e）
L4：（a→ c → d）

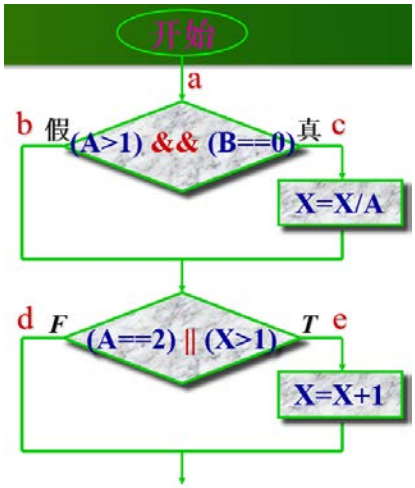
对于图例，如果选择路径 L1 和 L2，就可得满足要求的测试用例
【(2，0，4)，(2，0，3)】覆盖 ace【L1】
【(1，1，1)，(1，1，1)】覆盖 abd【L2】;
如果选择路径 L3 和 L4，还可得另一组可用的测试用例：
【(2，1，1)，(2，1，2)】覆盖 abe【L3】
【(3，0，3)，(3，1，1)】覆盖 acd【L4】

说明：以上仅考虑了两出口的判断，我们还应把判定覆盖准则扩充到多出口判断（如 Case 语句）的情况。因此，判定覆盖更为广泛的含义应该是使得每一个判定获得每一种可能的结果至少一次。



判定覆盖的**优点**：判定覆盖具有比语句覆盖更强的测试能力。同样判定覆盖也具有和语句覆盖一样的简单性，无须细分每个判定就可以得到测试用例。
判定覆盖的**缺点**：往往大部分的判定语句是由多个逻辑条件组合而成，若仅仅判断其整个最终结果，而忽略每个条件的取值情况，必然会遗漏部分测试路径。判定覆盖仍是弱的逻辑覆盖。

③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。**



L1：（a → c → e）
L2：（a→ b → d）
L3：（a→ b→ e）
L4：（a→ c → d）

在图例中，我们事先可对所有条件的取值加以标记。
例如：对于第一个判断，条件 A > 1 取真为 T1，A≤1 取假为$\overline{T1}$
条件 B = 0 取真为 T2，B≠0 取假为$\overline{T2}$
对于第二个判断：
条件 A = 2 取真为 T3，A ≠2 取假为$\overline{T3}$
条件 X > 1 取真为 T4，X 1 取假为$\overline{T4}$

测试用例	覆盖分支	条件取值
【(2, 0, 4)，(2, 0, 3)】	L1(c, e)	$T_1T_2T_3T_4$
【(1, 1, 1)，(1, 1, 1)】	L2(b, e)	$\overline{T_1}\overline{T_2}\overline{T_3}\overline{T_4}$

或

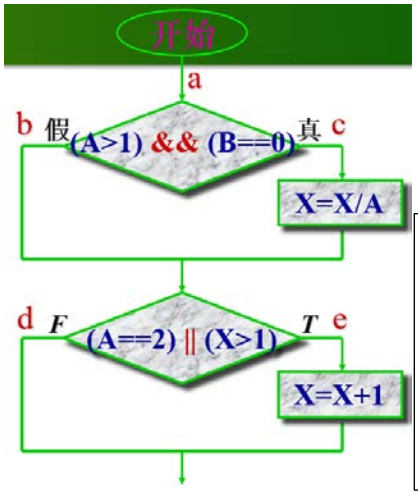
测试用例	覆盖分支	条件取值
【(2, 0, 4)，(2, 0, 3)	L1(c, e)	$T_1T_2T_3T_4$
【(1, 0, 1)，(1, 0, 1)】	L2(b, d)	$\overline{T_1}T_2\overline{T_3}\overline{T_4}$
【(2, 1, 1)，(2, 1, 2)】	L3(b, e)	$T_1\overline{T_2}T_3\overline{T_4}$

或

或 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)	$T_1\overline{T_2}T_3\overline{T_4}$

条件覆盖的**优点**：增加了对条件判定情况的测试，增加了测试路径
条件覆盖的**缺点**：条件覆盖下不一定包含判定覆盖，条件覆盖只能保证每个条件至少有一次为真，而不考虑所有的判定结果。

④判定-条件覆盖实际上是将判定覆盖和条件覆盖结合起来的一种方法。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，同时每个判定的可能结果也至少出现一次。



L1：（a → c → e）
L2：（a → b → d）
L3：（a → b → e）
L4：（a → c → d）

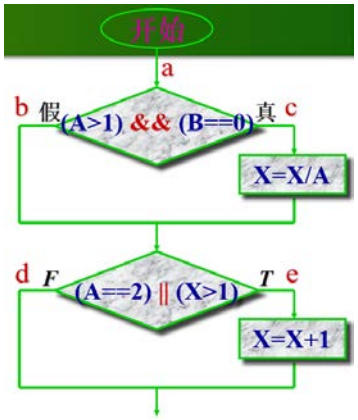
设计测试用例覆盖 4 个条件的 8 种取值以及 4 个判定分支。

设计测试用例——首先要想符合判定覆盖，则需要两个判定分支分别都能取到真和假，可以为**“真真”与“假假”**，然后要符合条件覆盖，每个条件的可能取值至少被执行一次
真真：(A=2) and (B=0) or (A>1) and (B=0) and (X=X/A>1)
假假：not (A>1) and not (A=2) and not (X>1) or not (B=0) and not (A=2) and not (X>1)

注：判定-条件覆盖也不一定能够完全检查出逻辑表达式中的错误。
比如：对于第一个判定(A>1)&&(B==0)来说，必须 A>1 和 B==0 这两个条件同时满足才能确定该判定为真。如果 A>1 为假，则编译器将不再检查 B==0 这个条件，那么即使这个条件有错也无法被发现。对于第二个判定(A==2)||(X>1)来说，若条件 A==2 满足，就认为该判定为真，这时将不会再检查 X>1，那么同样也无法发现这个条件中的错误。
判定-条件覆盖的优点：能同时满足判定、条件两种覆盖标准
判定-条件覆盖的缺点：判定/条件覆盖准则的缺点是未考虑条件的组合情况。

⑤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。



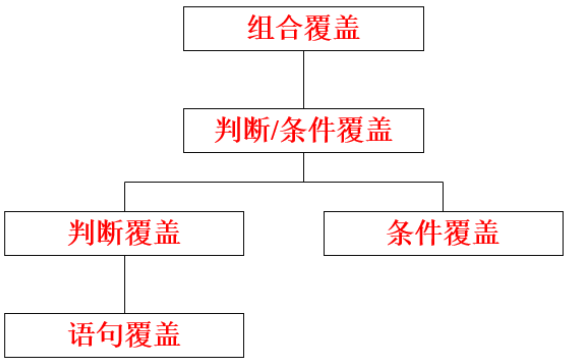
L1：（a → c → e）
L2：（a → b → d）
L3：（a → b → e）
L4：（a → c → d）

① A > 1, B = 0 作 T_1T_2
② A > 1, B ≠ 0 作 $T_1\overline{T_2}$
③ A ≯ 1, B = 0 作 $\overline{T_1}T_2$
④ A ≯ 1, B ≠ 0 作 $\overline{T_1}\overline{T_2}$
⑤ A = 2, X > 1 作 T_3T_4
⑥ A = 2, X ≯ 1 作 $T_3\overline{T_4}$
⑦ A ≠ 2, X > 1 作 $\overline{T_3}T_4$
⑧ A ≠ 2, X ≯ 1 作 $\overline{T_3}\overline{T_4}$

测试用例	覆盖条件	覆盖组合
【(2, 0, 4), (2, 0, 3)】(L1)	$T_1T_2T_3T_4$	①, ⑤
【(2, 1, 1), (2, 1, 2)】(L3)	$T_1\overline{T_2}T_3\overline{T_4}$	②, ⑥
【(1, 0, 3), (1, 0, 4)】(L3)	$\overline{T_1}T_2\overline{T_3}T_4$	③, ⑦
【(1, 1, 1), (1, 1, 1)】(L2)	$\overline{T_1}\overline{T_2}\overline{T_3}\overline{T_4}$	④, ⑧

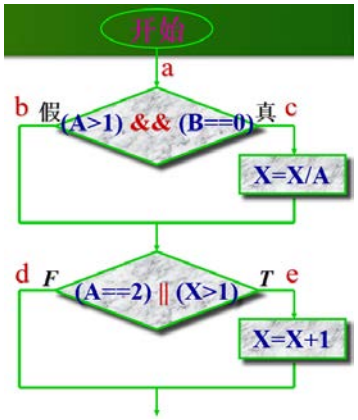
左边这组测试用例覆盖了所有 8 种条件取值的组合，覆盖了所有判定的真假分支，但是却丢失了一条路径 L4

条件组合覆盖的优点：条件组合覆盖准则满足判定覆盖、条件覆盖和判定-条件覆盖准则
条件组合覆盖的缺点：线性地增加了测试用例数量



逻辑覆盖法中各种方法的逻辑覆盖程度

7、**路径覆盖**就是设计足够的测试用例，覆盖程序中所有可能的路径（或程序图中有环，则每个环至少经过一次）



L1：（a → c → e）
L2：（a → b → d）
L3：（a → b → e）
L4：（a → c → d）

测试用例	通过路径	覆盖条件
【(2, 0, 4), (2, 0, 3)】	ace (L1)	$T_1T_2T_3T_4$
【(1, 1, 1), (1, 1, 1)】	abd (L2)	$\overline{T_1}\overline{T_2}\overline{T_3}\overline{T_4}$
【(1, 1, 2), (1, 1, 3)】	abe (L3)	$\overline{T_1}\overline{T_2}\overline{T_3}T_4$
【(3, 0, 3), (3, 0, 1)】	acd (L4)	$T_1T_2\overline{T_3}\overline{T_4}$

虽然前面一组测试用例满足了路径覆盖，但并没有覆盖程序中所有的条件组合，即**满足路径覆盖的测试用例并不一定满足组合覆盖。**

对于比较简单的小程序，实现路径覆盖是可能做到的。但如果程序中出现较多判断和较多循环，可能的路径数目将会急剧增长，要在测试中覆盖所有的路径是无法实现的。为了解决这个难题，只有把覆盖路径数量压缩到一定的限度内，如程序中的循环体只执行一次。

在实际测试中，即使对于路径数很有限的程序已经做到路径覆盖，仍然不能保证被测试程序的正确性，还需要采用其他测试方法进行补充。

8、基本路径法

基本路径是指程序中至少引进一条新的语句或一个新的条件的任一路径。

基本路径测试法又称独立路径测试，是在程序控制流图的基础上，通过分析控制结构的环路复杂性，导出基本可执行路径集合，从而设计出相应的测试用例的方法。

路径测试就是从一个程序的入口开始，执行所经历的各个语句的完整过程。从广义的角度讲，任何有关路径分析的测试都可以被称为路径测试。

完成路径测试的理想情况是做到路径覆盖，但对于复杂性大的程序要做到所有路径覆盖是不可能的。

在不能做到所有路径覆盖的前提下，如果某一程序的每一个**独立路径**都被测试过，那么可以认为程序中的每个语句都已经检验过了，即达到了语句覆盖。这种测试方法就是通常所说的基本路径测试法。

基本路径测试方法是在**控制流图**的基础上，通过分析控制结构的**环形复杂度**，**导出执行路径的基本集**，再从该基本集**设计测试用例**。基本路径测试方法包括 4 个步骤：

（1） 以详细或源代码作为基础，画出程序的控制流图

（2） 计算程序的环形复杂度，导出程序基本路径集中的独立路径条数，这是确定程序中每个可执行语句至少执行一次所必需的测试用例数目的上界

（3） 导出基本路径集，确定程序的独立路径

（4） 根据(3)中的独立路径，设计测试用例的输入数据和预期输出。确保基本路径集中每条路径的执行

【注：控制流图是对程序流程图做出了一些简化的结果，更加能突出控制流的结构】

控制流图的基本概念

在控制流图中只有两种图形符号，他们是：

（1）节点：以标有编号的圆圈表示

程序流程图中矩形框所表示的处理

菱形表示的两个甚至多个出口判断

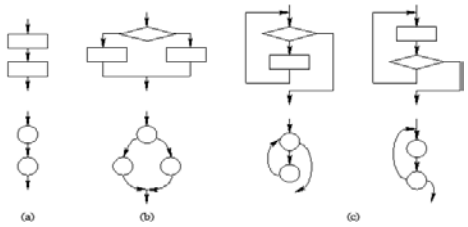
多条流线相交的汇合点

（2）控制流线或弧：以箭头表示

与程序流程图中的流线一致，表明了控制的顺序

控制流线通常标有名字

几种基本结构的控制流图

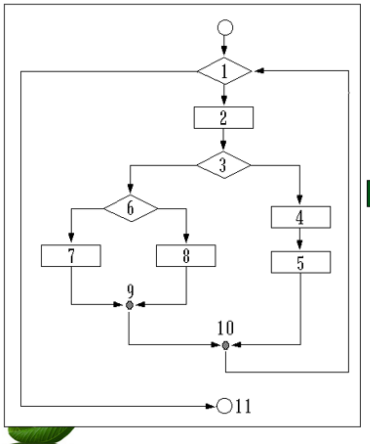


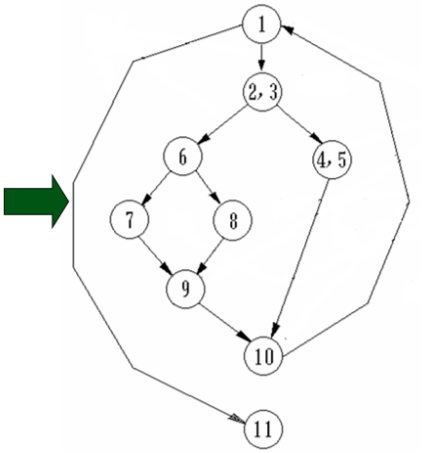
程序的控制流图中需要注意：

（1）在选择或多分支结构中，分支的汇聚出应有一个汇聚节点

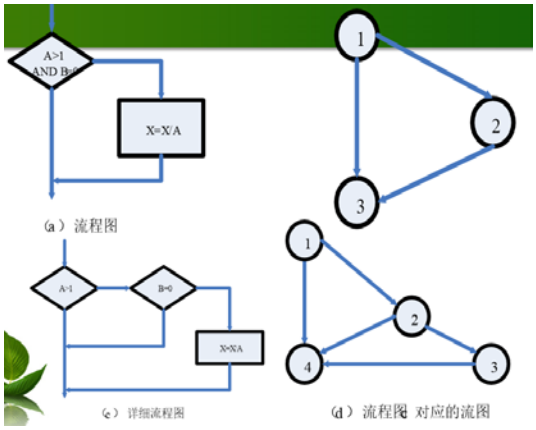
（2）边和节点圈定的区域叫做区域，当对区域计数时，图形外的区域也是应记为一个区域

（3）复合条件的控制流图，如果判断中的条件表达式是由一个或多个逻辑运算符(OR,AND,…)连接的符合条件表达式，则需改为一系列只有单个条件的嵌套的判断





可以合并，也可以不合并



环路复杂性（=独立路径条数）

程序的环路复杂性给出了**程序基本路径中的独立路径条数**，这是确保程序中每个可执行语句至少执行一次所必需的测试用例数目的**上界**。

从控制流图看，一条**独立路径**是至少包含有一条在其他独立路径中从未有过的边的路径。

程序环路复杂性计算方法（三种）：

（1） 流图中区域的数量对应于环形复杂度

（2） 给定流图 G 的环形复杂度 V(G)，定义为 V(G)=E-N+2，E 是流图中边的数量，

N 是流图中节点的数量

（3） V(G)=P+1，P 是流图 G 中的判定节点数

导出测试用例

（1）导出测试用例，确保基本路径集中的每一条路径的执行。

（2）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 —— 用逻辑覆盖方法。

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。

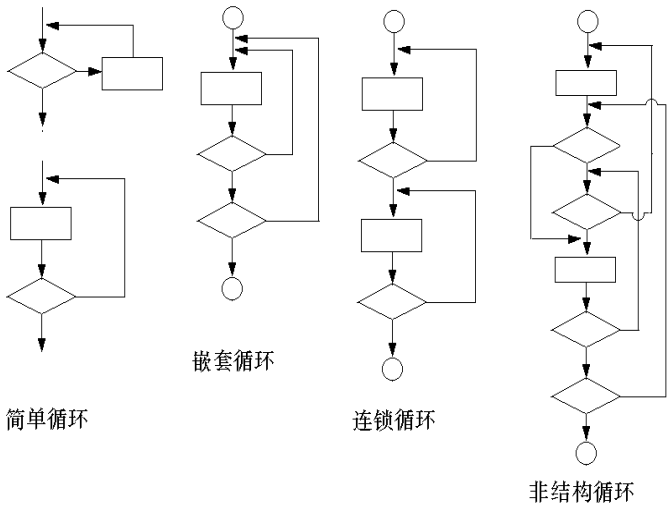
必须注意，一些独立的路径，往往不是完全孤立的，有时它是程序正常的控制流的一部分，这时，这些路径的测试可以是另一条路径测试的一部分。

逻辑覆盖与判定覆盖的比较

方法	判定覆盖	条件覆盖	条件组合覆盖	基本路径测试
优点	简单、无须细分每个判定	增加了对符号判定情况的测试	对程序进行较彻底的测试，覆盖面广	清晰、测试用例有效
缺点	往往大部分的判定语句是由多个逻辑条件组合而成（如包含AN’D、OR等的组合），若仅仅判断其组合条件的结果，而忽略每个条件的取值情况，必然会遗漏部分测试场景	达到条件覆盖，需要足够多的测试用例，但条件覆盖还是不能保证判定覆盖，这是由于AND和OR不同的组合效果造成的	对所有可能条件进行测试，需要设计大量、复杂测试用例，工作量比较大	基本路径法，类似于分支的方法，不能覆盖一些特定的条件，这些条件往往是容易出错的地方

9、循环测试

循环分为 4 种类型：简单循环、嵌套循环、连锁循环(串接循环)、非结构循环(不规则循环)



（1）简单循环测试

①零次循环：从循环入口到出口

②一次循环：检查循环初始值

③二次循环：两次通过循环

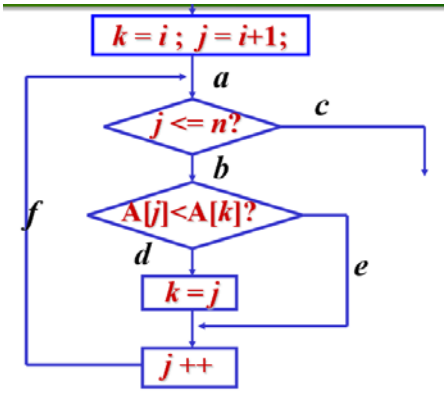
④m 次循环：检查多次循环

⑤最大次数循环 n、比最大次数多一次 n+1、少一次的循环 n-1

例：下列求最小值的程序代码

```
k = i;
for ( j = i+1;   j <= n;   j++ )
      if ( A[j] < A[k] )   k = j;
```

测试用例选择



循环	i	n	A[i]	A[i+1]	A[i+2]	k	路 径
0	1	1				i	a c
1	1	2	1	2		i	a b e f c
			2	1		i+1	a b d f c
2	1	3	1	2	3	i	a b e f e f c
			2	3	1	i+2	a b e f d f c
			3	2	1	i+2	a b d f d f c
			3	1	2	i+1	a b d f e f c

（2）嵌套循环测试

① 对最内层循环做简单循环的全部测试。所有其它层的循环变量置为最小值；

② 逐步外推，对其外面一层循环进行测试。测试时保持所有外层循环的循环变量取最小值，所有其它嵌套内层循环的循环变量取“典型”值。

③ 反复进行，直到所有各层循环测试完毕。

④ 对全部各层循环同时取最小循环次数，或者同时取最大循环次数

（3）连锁循环

如果各个循环互相独立，则可以用与简单循环相同的方法进行测试。但如果几个循环不是互相独立的，则需要使用测试嵌套循环的办法来处理。

（4）非结构循环

这一类循环应该使用结构化程序设计方法重新设计测试用例。

10、程序插桩技术

在软件动态测试中，程序插桩(Program Instrumentation)是一种基本的测试手段。

方法简介：借助往被测程序中插入操作，来实现测试目的的方法。如果我们想要了解一个程序在某次运行中所有可执行语句被覆盖的情况，或是每个语句实际执行次数，最好的办法就是利用程序插桩技术.

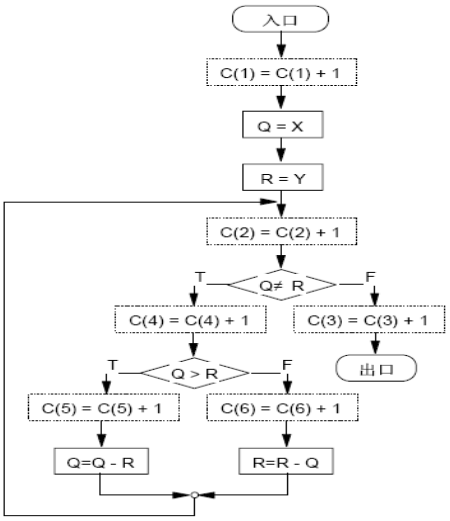
最简单的插桩：在程序中插入打印语句 printf(“…”)语句

例:求取两个整数 X 和 Y 的最大公约数程序如下:

```
int gsd (int X,int Y)
{ int Q=X;
   int R=Y;
```

```
    while(Q!=R)
     { if(Q>R)
          Q=Q-R;
       else    R=R-Q; }
     return Q;
}
```

可以根据程序绘制出其流程图
为了记录该程序中语句的执行次数，我们使用插桩技术插如如下语句：
C(i)=C(i)+1,　　i=1,2,…,6
插桩之后的流程图如下：



程序从入口开始执行，到出口结束，凡经历的计数语句都能记录下该程序点的执行次数。
如果我们在程序的入口处还插入了对计数器 C(i)初始化的语句，在出口处插入了打印这些计数器的语句，就构成了**完整的插桩程序**。它就能记录并输出在各程序点上语句的实际执行次数.

设计插桩程序时需要考虑的问题包括：
（1）需要探测哪些信息
（2）在程序的什么部位设置探测点
（3）需要设置多少个探测点
前两个问题需要结合具体的问题解决，并不能给出笼统的回答。至于第三个问题，需要考虑如何设置最少的探测点！

11、白盒测试问题类别分为以下几大类:
各层公用问题、JAVA 语言规范、数据类型、SQL 语句规范、界面 UI 、VO 数值对象、BO 业务对象、DMO 数据管理对象、业务逻辑重点、事务处理与隔离级别测试、效率测试。

问题属性分为四类：错误、缺陷、失效、故障。

错误是指计算值、观测值、测量值之间，或条件与真值之间，不符合规定的或理论上的正确值或条件。

缺陷是指与期望值或特征值的偏差。

故障是指功能部件不能执行所要求的功能。故障可能由错误、缺陷或失效引起。

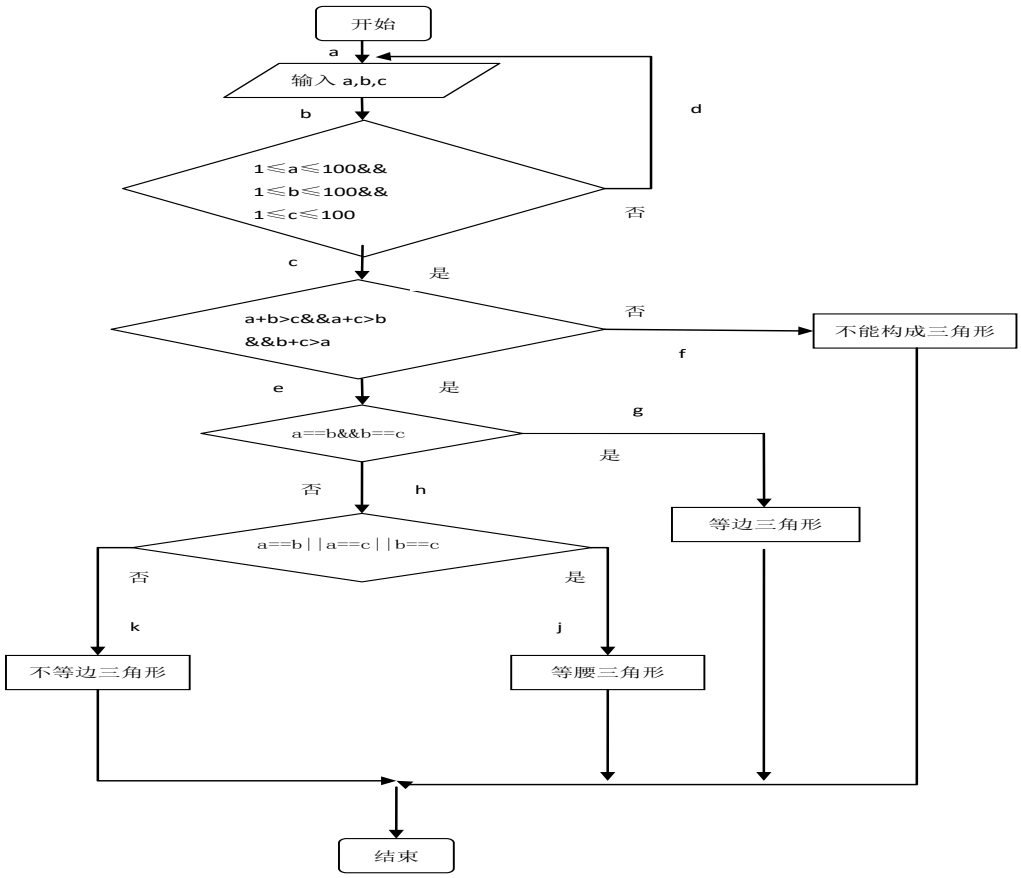
失效是指功能部件执行其功能的能力丧失，系统或系统部件丧失了在规定限度内执行所要求功能的能力。

白盒测试运用实例

三角形问题：

输入三个整数 a,b,c(1<=a,b,c<=100)， 判断是否构成三角形？若能构成三角形，指出构成的是等边三角形？ 等腰三角形？还是不等边三角形？

首先，画出流程图



逻辑测试方法

（1）语句覆盖

ID	输入			预期输出结果	实际输出结果	通过路径
	a	b	c			
TE-01	5	5	5	构成等边三角形	构成等边三角形	abcegi
TE-02	5	5	6	构成等腰三角形	构成等腰三角形	abcehjl
TE-03	1	1	3	不能构成三角形	不能构成三角形	abcfl
TE-04	5	6	7	构成不等三角形	构成不等边三角形	abcehkl

（2）判定覆盖

ID	输入			预期输出结果	实际输出结果	通过路径
	a	b	c			
TE-05	105	5	8	输入a错误，重新输入	输入a错误，重新输入	abd
TE-06	5	1	2	不能构成三角形	不能构成三角形	adcfl
TE-07	6	6	6	构成等边三角形	构成等边三角形	abcegil
TE-08	6	6	5	构成等腰三角形	构成等腰三角形	abcehjl
TE-09	6	8	10	构成不等三角形	构成不等边三角形	abcehkl

（3）条件覆盖

ID	输入			预期输出结果	实际输出结果	通过路径
	a	b	c			
TE-10	101	105	105	输入a、b、c错误，重新输入	输入a、b、c错误，重新输入	abd
TE-11	-1	-1	-1	输入a、b、c错误，重新输入	输入a、b、c错误，重新输入	abd
TE-12	2	8	12	不能构成三角形	不能构成三角形	adcfl
TE-13	2	12	8	不能构成三角形	不能构成三角形	adcfl
TE-14	12	2	8	不能构成三角形	不能构成三角形	adcfl
TE-15	6	6	6	构成等边三角形	构成等边三角形	abcegil
TE-16	6	8	10	构成不等边三角形	构成不等边三角形	abcehkl

（4）判定-条件覆盖

ID	输入			预期输出结果	实际输出结果	通过路径
	a	b	c			
TE-17	101	105	105	输入a、b、c错误，重新输入	输入a、b、c错误，重新输入	abd
TE-18	-1	-1	-1	输入a、b、c错误，重新输入	输入a、b、c错误，重新输入	abd
TE-19	2	8	12	不能构成三角形	不能构成三角形	adcfl
TE-20	2	12	8	不能构成三角形	不能构成三角形	adcfl
TE-21	12	2	8	不能构成三角形	不能构成三角形	adcfl
TE-22	6	6	8	构成等腰三角形	构成等腰三角形	abcehjl
TE-23	8	6	6	构成等腰三角形	构成等腰三角形	abcehjl
TE-24	6	8	6	构成等腰三角形	构成等腰三角形	abcehjl
TE-25	6	6	6	构成等边三角形	构成等边三角形	abcegil
TE-26	6	8	10	构成不等边三角形	构成不等边三角形	abcehkl

（5）条件组合覆盖

ID	输入			预期输出结果	实际输出结果	通过路径
	a	b	c			
TE-26	101	105	105	输入a、b、c错误，重新输入	输入a、b、c错误，重新输入	abd
TE-27	-1	-1	-1	输入a、b、c错误，重新输入	输入a、b、c错误，重新输入	abd
TE-28	2	8	12	不能构成三角形	不能构成三角形	adcfl
TE-29	2	12	8	不能构成三角形	不能构成三角形	adcfl
TE-30	12	2	8	不能构成三角形	不能构成三角形	adcfl
TE-31	6	6	10	构成等腰三角形	构成等腰三角形	abcehjl
TE-32	8	8	8	构成等边三角形	构成等边三角形	abcegil
TE-33	3	4	5	构成不等边三角形	构成不等边三角形	abcehkl

基本路径测试

ID	输入			预期输出结果	实际输出结果	通过路径
	a	b	c			
TE-037	103	5	8	输入a错误，重新输入	输入a错误，重新输入	abd
TE-038	2	2	5	不能构成三角形	不能构成三角形	abcfl
TE-039	10	10	10	构成等边三角形	构成等边三角形	abcegi
TE-040	5	5	7	构成等腰三角形	构成等腰三角形	abcehjl
TE-041	10	12	15	构成不等边三角形	构成不等边三角形	abcehkl

12、NC（Network Computer）系统中的对象主要分为如下几种：

界面对象 UI(UI Object)

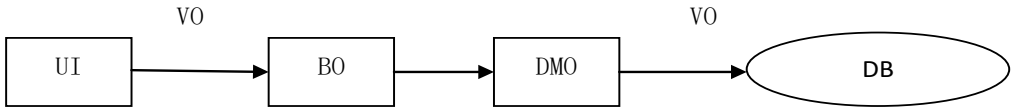
数值对象 VO(Value Object)

业务对象 BO(Business Object)

数据管理对象 DMO(Data Manage Object)

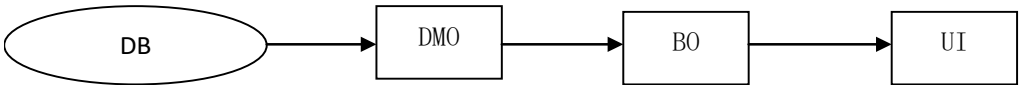
1、界面对象测试

界面对象测试的流程图如图 3-16 所示。



2、业务对象测试

业务对象测试的流程图如图 3-17 所示。



其优缺点比较如下：

界面对象测试流程的优点是便于测试者从界面层直观地录入数据，缺点是做回归测试时，录入数据需重复。

业务对象测试原则是从底层测试，底层测试通过了，再依次往上一层测试；否则不需往上层测试，优点是做回归测试时，不用再构造输入数据，只要再执行一遍小测试程序。缺点是需给中间层做一测试小程序，根据程序中类的对象构造输入数据及将结果输出到控制台上。

写在最后：采用任何一种覆盖方法都不能满足我们的要求，所以，在实际的测试用例设计过程中，可以根据需要将不同的覆盖方法组合起来使用，以实现最佳的测试用例设计 。