第5章 白盒测试

学习目的与要求

本章介绍白盒测试的基本概念和类型，白盒测试主要分为控制流测试和数据流测试。其中较常用到的是控制流测试中的相关覆盖准则。通过本章的学习，能够对白盒测试有深入的理解和体会。本章要求重点掌握相关覆盖准则的具体应用。

本章主要内容

- 白盒测试的概念。
- 控制流测试、数据流测试。
- 测试覆盖率。
- 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。
- 白盒测试工具。

5.1 白盒测试的概念

白盒测试作为测试人员常用的一种测试方法，越来越受到测试工程师的重视。白盒测试并不是简单地按照代码设计用例进行测试，而是需要根据不同的测试需求，结合不同的测试对象，使用适合的方法进行测试。因为对于不同复杂度的代码逻辑，可以衍生出许多种执行路径，只有适当的测试方法，才能帮助我们从代码的迷雾森林中找到正确的方向。

白盒测试也称为结构化测试、基于代码的测试，是一种测试用例设计方法。白盒测试从程序的控制结构导出测试用

例，是针对被测单元内部如何进行工作的测试。它根据程序的控制结构设计测试用例，主要用于软件或程序验证。

白盒测试与程序内部结构相关，需要利用程序结构的实现细节等知识，才能有效进行测试用例的设计工作。白盒测试方法有程序控制流测试、数据流测试、逻辑驱动测试、域测试、符号测试、路径测试、程序插桩及程序变异等，本章重点介绍前两种白盒测试方法。

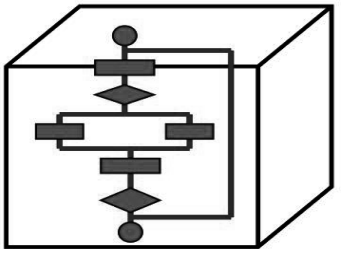


图 5-1 白盒测试示意图

白盒测试把测试对象看作一个透明的盒子，如图 5-1 所示，所以又称玻璃盒测试。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。

白盒测试检查程序内部逻辑结构，对所有逻辑路径进行测试，是一种穷举路径的测试方法。但即使每条路径都测试过了，仍然可能存在错误。这是因为：①穷举路径测试无法检查出程序本身是否违反了设计规范，即程序是否是一个错误的程序；②穷举路径测试不可能查出程序因为遗漏路径而出现的错误；③穷举路径测试发现不了一些与数据相关的错误。

采用白盒测试方法必须遵循以下几条原则，才能达到测试的目的。

(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。

5.1.1 控制流测试

由于非结构化程序会给测试带来许多不必要的困难，所以业界要求写出的程序具有良好的结构。自 20 世纪 70 年代以来，结构化程序的概念逐渐被人们接受。体现这一要求对某些语言并不困难，例如 Pascal、C 语言，因为它们都具有反映基本控制结构的控制语句。但对于有些开发语言要做到这一点并不容易，程序人员需要注意程序结构化的要求，例如汇编语言，若使用汇编语言编写程序，开发人员尤其要注意程序的结构化要求。

1. 控制流的基本概念

在进一步介绍控制流测试方法之前，先来回顾图论的相关概念术语。

定义 5.1(图) 图(又称线性图)是一种由两个集合定义的抽象数学结构，即一个节点集合和一个构成节点之间连接的边集合。图 $G=<V,E>$ 由节点的有限(并且非空)集合 V 和节点无序对偶集合 E 组成，即由 $V=\{n_1,n_2,\cdots,n_m\}$ 和 $E=\{e_1,e_2,\cdots,e_p\}$ 组成。其中，每条边 $e_k=\{n_i,n_j\}$，$n_i,n_j\in V$。$\{n_i,n_j\}$ 是一个无序对偶，有时记作 (n_i,n_j)。

如图 5-2 所示图例，节点集合为 $V=\{n_1,n_2,n_3,n_4,n_5,n_6,n_7\}$，边集合为 $E=\{e_1,e_2,e_3,e_4,e_5\}=\{(n_1,n_2),(n_1,n_4),(n_3,n_4),(n_2,n_5),(n_4,n_6)\}$。

可以把节点看作程序语句，边表示控制流或定义/使用关系。

定义 5.2(节点的度) 节点的度是以该节点作为端点的边的条数。节点 n 的度记作 deg(n)。

可以说，节点的度表示它在图中的“流行程度”。如果图中的节点表示对象，边表示消息，

则节点(对象)的度表示适合该对象的集成测试范围。

图 5-2 中节点的度：$deg(n_1)=2$，$deg(n_2)=2$，$deg(n_3)=1$，$deg(n_4)=3$，$deg(n_5)=1$，$deg(n_6)=1$，$deg(n_7)=0$。

定义 5.3(关联矩阵)　拥有 m 个节点和 n 条边的图 $G=<V,E>$ 的关联矩阵是一种 $m\times n$ 矩阵，其中第 i 行第 j 列的元素是 1，当且仅当节点 i 是边 j 的一个端点，否则元素是 0。

关联矩阵是对称的，行的和即该节点的度。

定义 5.4(相邻矩阵)　拥有 m 个节点和 n 条边的图 $G=<V,E>$ 的相邻矩阵是一种 $m\times m$ 矩阵，其中第 i 行第 j 列的元素是 1，当且仅当节点 i 和节点 j 之间存在一条边，否则元素是 0。

相邻矩阵是对称的，行的和即该节点的度。

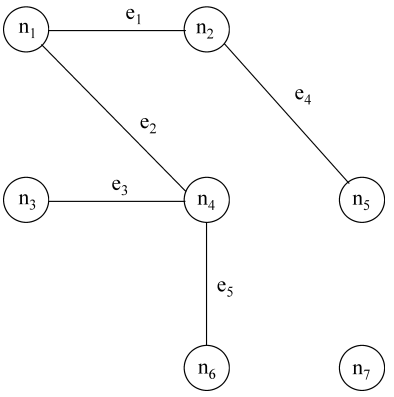


图 5-2　图例示意图

定义 5.5(路径)　路径是一系列的边，对于序列中的任何相邻边对偶 e_i，e_j，边都拥有相同的(节点)端点。图 5-2 中的一些路径如下。

路　径	节点序列	边　序　列
n_1 和 n_5	n_1，n_2，n_5	e_1，e_4
n_6 和 n_5	n_6，n_4，n_1，n_2，n_5	e_5，e_2，e_1，e_4

定义 5.6(连接性)　节点 n_i 和 n_j 是被连接的，当且仅当它们都在同一条路径上。

“连接性”是一种图的节点集合上的等价关系。

(1) 连接性是自反的。

(2) 连接性是对称的。

(3) 连接性是传递的。

图的组件是相连节点的最大集合。例如，图 5-2 中有两个组件：$\{n_1,n_2,n_3,n_4,n_5,n_6\}$ 和 $\{n_7\}$。

定义 5.7(圈数)　图 G 的圈数由 $V(G)=e-n+p$ 给出。其中，e 是 G 中的边数，n 是 G 中的节点数，p 是 G 中的组件数，V(G)是图中不同区域的个数。

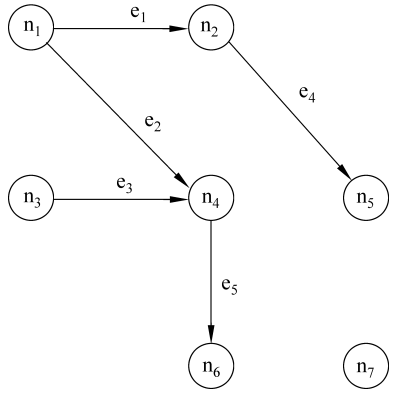


图 5-3　有向图示意图

定义 5.8(有向图)　有向图(又称框图)$D=(V,E)$ 包含：一个节点的有限集合 $V=\{n_1,n_2,\cdots,n_m\}$，一个边的集合 $E=\{e_1,e_2,\cdots,e_p\}$，其中每条 $e_k=<n_i,n_j>$ 是节点 $<n_i,n_j>\in V$ 的一个有序对偶。

对于有向图，边有了方向含义，在符号上，无序对偶 (n_i,n_j) 变成有序对偶 $<n_i,n_j>$，我们说有向边从节点 n_i 到 n_j，而不是在节点之间。

有向图如图 5-3 所示，节点集合为 $V=\{n_1,n_2,n_3,n_4,n_5,n_6,n_7\}$，边集合为 $E=\{e_1,e_2,e_3,e_4,e_5\}=\{<n_1,n_2>,<n_1,n_4>,<n_3,n_4>,<n_2,n_5>,<n_4,n_6>\}$。

定义 5.9(内度与外度)　有向图中节点的内度，是将

该节点作为终止节点的不同边的条数，记为 indeg(n)；有向图中节点的外度，是将该节点作为开始节点的不同边的条数，记为 outdeg(n)。

如在图 5-3 中节点具有以下内度和外度：indeg(n_1)＝0，outdeg(n_1)＝2；indeg(n_2)＝1，outdeg(n_2)＝1。

一般图和有向图存在某种联系，例如有 deg(n)＝indeg(n)＋outdeg(n)。

定义 5.10 节点的类型：①内度为 0 的节点是源节点；②外度为 0 的节点是吸收节点；③内度不为 0 且外度不为 0 的节点是传递节点。

源节点和吸收节点构成图的外部边界。既是源节点又是吸收节点的节点是孤立节点。

在图 5-3 中，n_1、n_3 和 n_7 是源节点，n_5、n_6 和 n_7 是吸收节点，n_2 和 n_4 是传递节点，n_7 是孤立节点。

定义 5.11(有向图的相邻矩阵) 有 m 个节点和 n 条边的图 D＝＜V，E＞的相邻矩阵是一种 m×m 矩阵，A＝(a_{ij})，其中 a_{ij}＝1，当且仅当从节点 i 到节点 j 之间有一条边，否则该元素是 0。

图 5-3 所示的有向图的相邻矩阵如下所示。

	n_1	n_2	n_3	n_4	n_5	n_6	n_7
n_1	0	1	0	1	0	0	0
n_2	0	0	0	0	1	0	0
n_3	0	0	0	1	0	0	0
n_4	0	0	0	0	0	1	0
n_5	0	0	0	0	0	0	0
n_6	0	0	0	0	0	0	0
n_7	0	0	0	0	0	0	0

定义 5.12(路径与半路径)(有向) 路径是一系列边，使得第一条边的终止节点是第二条边的初始节点。

环路是一个在同一个节点上开始和结束的有向路径。

半路径是一系列边，使得对于该序列中至少有一对相邻边对偶＜e_i，e_j＞来说，第一条边的初始节点是第二条边的初始节点，或第一条边的终止节点是第二条边的终止节点。

在图 5-3 中，从 n_1 到 n_6 存在一条路径，n_1 和 n_3 之间有一条半路径。

定义 5.13(n-连接性) 有向图中的两个节点 n_i 和 n_j 是：

0 连接，当且仅当 n_i 和 n_j 之间没有路径。

1 连接，当且仅当 n_i 和 n_j 之间有一条半路径，但是没有路径。

2 连接，当且仅当 n_i 和 n_j 之间有一条路径。

3 连接，当且仅当 n_i 和 n_j 之间有一条路径，并且 n_j 和 n_i 之间有一条路径。

定义 5.14(可到达性矩阵) 有 m 个节点和 n 条边的图 D＝＜V，E＞的相邻矩阵是一种 m×m 矩阵，**R**＝(r_{ij})，其中 r_{ij}＝1，当且仅当从节点 i 到节点 j 之间有一条路径，否则该元素是 0。

定义 5.15 环形复杂度，也称为圈复杂度，是一种为程序逻辑复杂度提供定量尺度的软件度量。

可以将环形复杂度用于基本路径方法，它可以提供程序基本集的独立路径数量，确保所有语句至少执行一次测试数量的上界。

独立路径是指程序中至少引入了一个新的处理语句集合或一个新条件的程序通路。采用流图的术语，独立路径必须至少包含一条在本次定义路径之前不曾用过的边。

测试可以被设计成基本路径集的执行过程，但基本路径集通常并不唯一。

环形复杂度以图论为基础，为我们提供了非常有用的软件度量。可用如下方法来计算环形复杂度。

(1) 控制流图中区域的数量对应于环形复杂度。

(2) 给定控制流图 G 的环形复杂度 V(G)定义为 V(G)＝E－N＋2。其中，E 是控制流图中边的数量，N 是控制流图中的节点数量。

(3) 给定控制流图 G 的环形复杂度 V(G)也可定义为 V(G)＝P＋1。其中，P 是控制流图 G 中判定节点的数量。

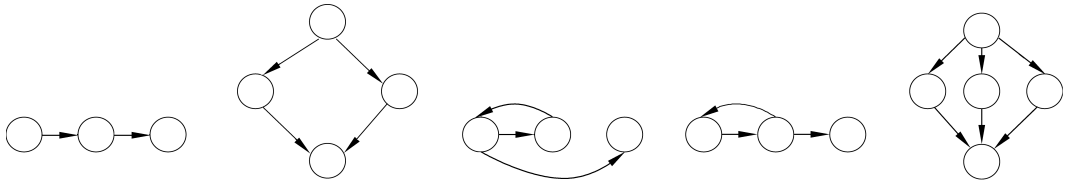
定义 5.16(图矩阵)　图矩阵是控制流图的矩阵表示形式。

图矩阵是一个方形矩阵，其维数等于控制流图的节点数。矩阵中的每列和每行都对应于标识的节点，矩阵元素对应于节点间的边。

通常，控制流图中的节点用数字标识，边则用字母标识。如果在控制流图中从第 i 个节点到第 j 个节点有一个标识为 x 的边相连接，则在对应图矩阵的第 i 行第 j 列有一个非空的元素 x。

2. 程序的控制流图

程序流程图又称框图，是人们最熟悉也最容易理解的一种程序控制结构的图形表示。常见结构的控制流图如图 5-4 所示。在这种图上的框里面常常标明了处理要求或条件，但是，这些标注在进行路径分析时是不重要的。为了更加突出控制流的结构，需要对程序流程图做一些简化。图 5-5 给出了简化的例子。其中图 5-5(a)是一个含有两个出口判断和循环的程序流程图，我们把它简化成图 5-5(b)的形式，这种简化了的程序流程图称为控制流图。其中，包含条件的节点称为判定节点(又称谓词节点)，由判定节点发出的边必须终止于某一个节点，由边和节点所限定的范围称为区域。



(a) 顺序结构　(b) if选择结构　(c) while循环结构　(d) until循环结构　(e) case多分支结构

图 5-4　常见结构的控制流图

可以观察到，在控制流图中只有节点和控制线(弧)两种图形符号。

(1) 节点：以标有编号的圆圈表示。它代表了程序流程图中矩形框表示的处理、菱形表示的两个到多个出口判断，以及两条到多条流线相交的汇合点。

(2) 控制流线(弧)：以箭头表示。它与程序流程图中的流线是一致的，表明了控制的顺序。为了方便讨论，线通常标有名字，如 a、b、c 等。

为了使控制流图在机器上表示，可以把它表示成矩阵的形式，称为控制流图矩阵。图 5-6

表示了图 5-5 的控制流图矩阵，这个矩阵有 5 行 5 列，是由该控制图中 5 个节点决定的。矩阵中 6 个元素 a、b、c、d、e 和 f 的位置决定了它们所连接节点的号码。例如，弧 d 在矩阵中处于第 3 行第 4 列，那是因为它在控制流图中连接了节点 3 至节点 4。这里必须注意方向，图中节点 4 到节点 3 没有弧，所以矩阵中第 4 行第 3 列也就没有元素。

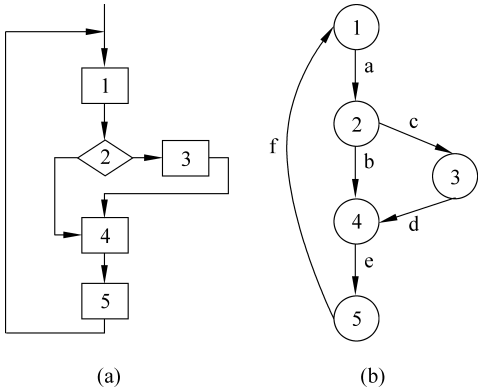


图 5-5 程序流程图和控制流图

	a			
		c	b	
			d	
				e
f				

图 5-6 控制流图矩阵

除了程序流程图可以转换成控制流图以外，还可以把伪代码表示的处理过程转换成控制流图。根据程序建构控制流图很容易，如图 5-7 所示，把三角形伪代码实现过程转换成了控制流图。对于不可执行语句我们不把它映射成节点，比如变量和类型说明语句。

```
(1) //Program triangle2 version of simple
(2) int a,b,c;
(3) boolean IsATriangle;
(4) cout<<"Enter 3 integers which are sides of a triangle";
(5) cin>>a>>b>>c;
(6) cout<<"Side A is"<<a;
(7) cout<<"Side A is"<<b;
(8) cout<<"Side A is"<<c;
(9) if((a<b+c) &&(b<a+c)&&(c<a+b))
(10) IsATriangle=True;
(11) else IsATriangle=False;
(12) //EndIf
(13) if(IsATrangle)
(14) if((a=b)&& (b=c))
(15) cout<<"Equilateral";
(16) else if((a!=b)&&(a!=c)&&(b!=c))
(17) cout<<"Scalene";
(18) cout<<"Isosceles";
(19) //EndIf
(20) //EndIf
(21) else cout<<"NOT a Triangle";
(22) //EndIf
(23) //End triangle2
```

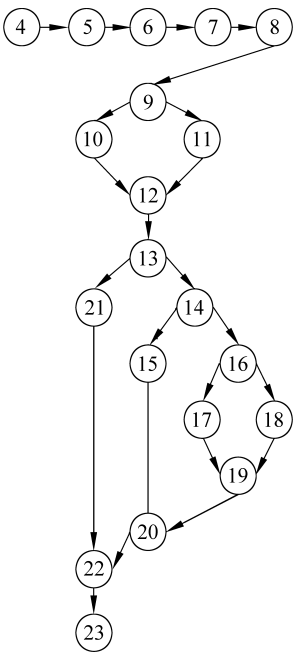


图 5-7 三角形伪代码映射成的控制流图

有时为了方便会把一条伪代码语句作为一个节点，但有时也可以把几个节点合并成一个节点，合并的原则是：序列中没有分支，则可以把这个序列的节点都合并成一个节点。比如图 5-7，可以合并成如图 5-8 所示的形式。对于不可执行语句，则不把它映射成节点，比如变量和类型说明语句。

当过程设计中包含复合条件时，生成控制流图的方法要复杂一些。在这种情况下，需要把复合条件拆开成一个个简单条件，让每一个简单条件对应流图中一个节点，这样的节点称为判定节点，它会引出两条或者多条边，如图 5-9 所示。

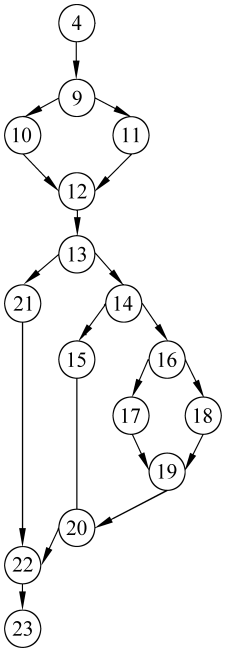


图 5-8　简化后的三角形控制流图

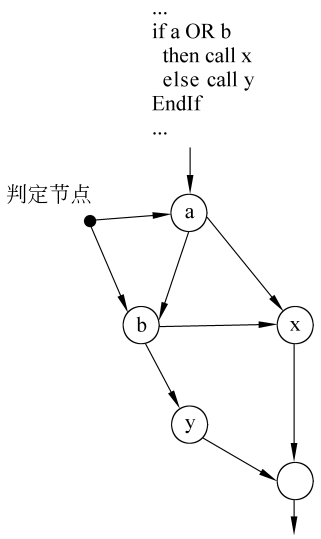


图 5-9　包含复合条件的伪代码转换成的控制流图

3. 控制流测试覆盖准则

控制流测试覆盖准则具体是指覆盖测试的标准。具体来说，可分为逻辑覆盖和路径覆盖，而逻辑覆盖又包括语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖和条件组合覆盖，如图 5-10 所示。

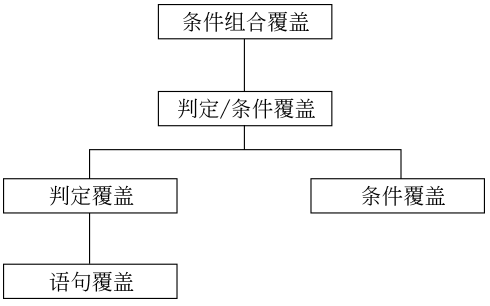


图 5-10　逻辑覆盖

5.1.2 数据流测试

1. 数据流分析

在单元测试中，数据仅仅在一个模块或一个函数中流通。但是，数据流的通路往往涉及多个集成模块甚至整个软件，尽管它非常耗时，但是有必要进行数据流的测试。

数据流分析最初是随着编译系统要生成有效的目标代码而出现的，这类方法主要用于优化代码。数据流测试是指一个基于通过程序的控制流，从建立的数据目标状态的序列中发现异常的结构测试方法。数据流测试用作路径测试的“真实性检查”。

早期的数据流分析常常集中于定义/引用异常的缺陷。

(1) 变量被定义，但是从来没有使用。

(2) 所使用的变量没有被定义。

(3) 变量在使用之前被定义了两次。

若一个变量在程序中的某处出现使数据与该变量相绑定，则称该出现是变量的定义性出现。

若一个变量在程序中的某处出现使与该变量相绑定的数据被引用，则称该出现是变量的引用性出现。

因为程序内的语句因变量的定义和使用而彼此相关，所以用数据流测试方法更能有效地发现软件缺陷。但是，在度量测试覆盖率和选择测试路径的时候，数据流测试很困难。

2. 数据流测试覆盖准则

数据流测试按照程序中的变量定义和使用的位置来选择程序的测试路径。数据流测试关注变量接收值的点和使用这些值的点。

定义 5.17(定义节点) 节点 $n \in G(P)$是变量 $v \in V$ 的定义节点，记作 DEF(v,n)，当且仅当变量 v 的值由对应节点 n 的语句片断处定义。

输入语句、赋值语句、循环语句和过程调用，都是定义节点语句的例子。当执行这种语句的节点时，与该变量关联的存储单元的内容就会改变。

定义 5.18(使用节点) 节点 $n \in G(P)$是变量 $v \in V$ 的使用节点，记作 USE(v,n)，当且仅当变量 v 的值在对应节点 n 的语句片断处使用。

使用节点语句的语句包括：①输出语句；②赋值语句；③条件语句；④循环控制语句；⑤过程调用。

定义 5.19(谓词使用、计算使用) 使用节点 USE(v,n)是一个谓词使用(记作 P-use)，当且仅当语句 n 是谓词语句；否则，USE(v,n)是计算使用(记作 C-use)。

(1) 对应于谓词使用的节点，其外度$\geqslant$2。

(2) 对应于计算使用的节点，其外度$\leqslant$1。

定义 5.20(定义-使用路径) 定义-使用路径(记作 du-path)是 PATHS(P)中的路径，使得对某个 $v \in V$，存在定义和使用节点 DEF(v,m)和 USE(v,n)，使得 m 和 n 是该路径的最初节点和最终节点。

定义 5.21(定义清除路径) 定义清除路径(记作 dc-path)是具有最初节点 DEF(v,m)和最终节点 USE(v,n)的 PATHS(P)中的路径，使得该路径中没有其他节点是 v 的定义节点。

数据流测试有如下几条具体准则。

(1) 全定义准则：集合 T 满足程序 p 的全定义准则，当且仅当所有变量 $v \in V$，T 包含从 v

的每个定义节点到 v 的一个使用的定义清除路径。T 是拥有变量集合 V 的程序 p 的程序图 G(p)中的一个路径集合。

(2) 全使用准则：集合 T 满足程序 p 的全使用准则，当且仅当所有变量 v∈V，T 包含从 v 的每个定义节点到 v 的所有使用，以及到所有 USE(v,n)后续节点的定义清除路径。

(3) 全谓词使用/部分计算使用准则：集合 T 满足程序 p 的全谓词使用/部分计算使用准则，当且仅当所有变量 v ∈V，T 包含从 v 的每个定义节点到 v 的所有谓词使用的定义清除路径，并且如果 v 的一个定义没有谓词使用，则定义清除路径导致至少一个计算使用。

(4) 全计算使用/部分谓词使用准则：集合 T 满足程序 p 的全计算使用/部分谓词使用准则，当且仅当所有变量 v∈v，T 包含从 v 的每个定义节点到 v 的所有计算使用的定义清除路径，并且如果 v 的一个定义没有计算使用，则定义清除路径导致至少一个谓词使用。

(5) 全定义-使用路径准则：集合 T 满足程序 p 的全定义-使用路径准则，当且仅当所有变量 v ∈V，T 包含从 v 的每个定义节点到 v 的所有使用，以及到所有 USE(v,n)后续节点的定义清除路径，并且这些路径要么有一次的环经过，要么没有环路。

5.2 测试覆盖率

测试覆盖率是指用于确定测试所执行到的覆盖项的百分比。其中，覆盖项指的是作为测试基础的一个入口或属性，例如语句、分支、条件等。

测试覆盖率可以表示出测试的充分性，在测试分析报告中可以作为量化指标的依据，测试覆盖率越高效果越好。但覆盖率不是目标，只是一种手段。

测试覆盖率包括功能点覆盖率和结构覆盖率。功能点覆盖率用于表示软件已经实现的功能与软件需要实现的功能之间的比例关系。结构覆盖率包括语句覆盖率、分支覆盖率、循环覆盖率、路径覆盖率等。

根据覆盖目标的不同，测试覆盖又可分为语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖、条件组合覆盖和路径覆盖。

(1) 语句覆盖：选择足够多的测试用例，使得程序中的每个可执行语句至少执行一次。

(2) 判定覆盖：通过执行足够的测试用例，使得程序中的每个判定至少都获得一次“真”值和“假”值，也就是使程序中的每个取“真”分支和取“假”分支至少均经历一次，也称为“分支覆盖”。

(3) 条件覆盖：设计足够多的测试用例，使得程序中每个判定包含的每个条件的可能取值(真/假)都至少满足一次。

(4) 判定/条件覆盖：设计足够多的测试用例，使得程序中每个判定包含的每个条件的所有情况(真/假)至少出现一次，并且每个判定本身的判定结果(真/假)也至少出现一次。

(5) 条件组合覆盖：通过执行足够的测试用例，使得程序中每个判定所有可能的条件取值组合都至少出现一次。

(6) 路径覆盖：设计足够多的测试用例，要求覆盖程序中所有可能的路径。

经过观察分析，不难发现：

(1) 满足判定/条件覆盖的测试用例一定同时满足判定覆盖和条件覆盖。

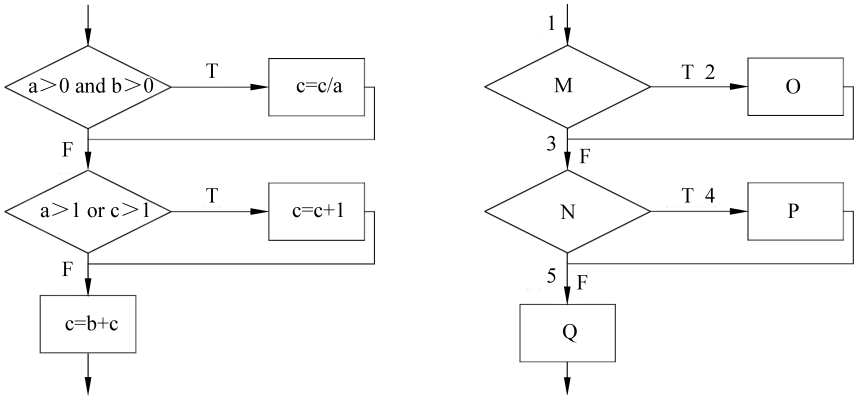
(2) 满足组合覆盖的测试用例一定满足判定覆盖、条件覆盖和判定/条件覆盖。

5.2.1 语句覆盖

例 5-1 实现一个简单的数学运算。

```
int a, b;
double c;
if(a>0 and b>0)   c=c/a;
if(a>1 or c>1)    c=c+1;
c=b+c
```

上面程序的流程如图 5-11 所示。



(a) 流程图　　(b) 模板

图 5-11　例 5-1 的程序流程图和模板

由图 5-11(b)可以看出，该程序模块有 4 条不同的路径：①P1：(1-2-4)；②P2：(1-2-5)；③P3：(1-3-4)；④P4：(1-3-5)。

将里面的判定条件和过程记录为：条件 M＝{a＞0 and b＞0}；条件 N＝{a＞1 or c＞1}。

这样，程序的 4 条不同路径可以表示为：①P1：(1-2-4)＝M and N；②P2：(1-2-5)＝～M and N；③P3：(1-3-4)＝M and ～N；④P4：(1-3-5)＝～M and ～N。

语句覆盖的基本思想：设计若干测试用例，运行被测程序，使程序中每个可执行语句至少执行一次。

看到上面的例题，P1 包含了所有可执行语句，按照语句覆盖的测试用例设计原则，可以使用 P1 来设计测试用例，例如：

令 a＝2，b＝1，c＝6，此时满足条件 M{a＞0 and b＞0}和条件 N＝{a＞1 or c＞1}(注：此时 c＝c/a＝3)，这样，测试用例的输入{a＝2，b＝1，c＝6}和对应的输出{a＝2，b＝1，c＝5}覆盖路径 P1。

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。

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，但对于隐藏的条件和可能到达的隐式逻辑分支是无法测试的。例如，在 Do-While 结构中，语句覆盖执行其中某一个条件分支。显然，语句覆盖对于多分支的逻辑运算也是无法全面反映的，它只运行一次，而不考虑其他情况。