第3章 白盒测试方法

代码检查法

代码检查法主要检查代码和程序设计的一致性，代码结构的合理性，代码编写的标准性、可读性，代码逻辑表达的正确性等方面。主要参考文档为：程序设计文档、程序的源代码清单、编码规范、代码缺陷检查表等。代码检查看到的是问题本身而非问题的征兆。代码检查非常耗费时间，而且代码检查需要知识和经验的积累。

代码检查法包括代码审查、代码走查和桌面检查三种方式。

代码审查和走查

- 代码审查（Code Review）

是一种正式的质量保证活动，由团队中其他成员（通常是资深开发者）对代码进行系统性检查，重点关注代码的正确性、安全性、性能、可读性、可维护性以及是否符合团队规范。其核心目的是发现代码中的缺陷、优化代码质量，并促进团队知识共享。

- 代码走查（Code Walkthrough）

是一种相对非正式的协作过程，由代码的作者主动引导团队成员逐行或逐模块浏览代码，重点关注代码的逻辑流程、设计意图和实现思路。其核心目的是让团队理解代码的工作原理，确认实现是否符合需求，并通过集体讨论发现潜在问题。

简单来说，代码审查更像 “专家质检”，通过严格检查确保代码质量；代码走查更像 “集体研讨”，通过共同理解促进团队对齐和问题发现。两者相辅相成，都是提升代码质量和团队协作效率的重要实践。

桌面检查

桌面检查是一种传统的检查方法，由程序员检查自己编写的程序。程序员在程序通过编译之后，对源程序代码进行分析、检验，并补充相关文档，由于程序员熟悉自己的程序及其程序设计风格，桌面检查由程序员自己进行可以节省时间，但应避免主观片面性。桌面检查的效果逊色于代码检查和走查，但桌面检查胜过没有检查。

静态结构分析法

静态结构分析法主要是以图形的方式表现程序的内部结构。测试者通过使用测试工具分析程序源代码的系统结构、数据结构、内部控制逻辑等内部结构，生成函数调用关系图、模块控制流图、函数内部控制流图等各种图形图表，清晰地标识整个软件的组成结构，便于理解。通过分析这些图表（包括控制流分析、数据流分析、接口分析、表达式分析），检查软件是否存在缺陷或错误。

第3章 白盒测试方法

章节概述/ Summary

白盒测试又称为透明盒测试、结构测试，它基于程序的内部逻辑结构进行测试，而不是程序的功能（黑盒测试）。因此，进行白盒测试时，测试人员需要了解程序的内部逻辑结构，根据使用的编程语言设计测试用例。白盒测试的方法包括基本路径法、逻辑覆盖法、程序插桩法，本章将对白盒测试的方法进行详细讲解。

基本块（Basic Block，BB）

基本块是程序中连续的、无分支进入（除入口外）、无分支跳出（除出口外）的指令序列。简单来说，一旦程序执行进入一个基本块的入口，就会“从头到尾”执行完所有指令，中途不会被其他分支打断，也不会有其他分支跳转到块内的中间位置。

```
int process(int x, int y) {
    int sum = x + y;          // 指令1
    int product = x * y;      // 指令2
BB  if (sum > product) {      // 指令3（条件分支）
        sum += 10;            // 指令4
        product = 0;          // 指令5
    } else {
        product -= 5;         // 指令6
    }
    int result = sum + product; // 指令7
    return result;            // 指令8
}
```

基本块的特点

- 单入口：只有一个 “开始执行点” ——即块的第一条指令。程序只能从这条指令进入该块，不能从块外的其他指令跳转到块内的中间指令。
- 单出口：只有一个 “结束执行点” ——即块的最后一条指令。执行完这条指令后，程序只会跳转到一个或两个其他基本块（若为条件分支，则跳转到两个块；若为无条件分支 / 顺序执行，则跳转到一个块）。
- 顺序执行：块内所有指令严格按顺序执行，没有分支、循环等控制流变化。

```
int process(int x, int y) {
    int sum = x + y;          // 指令1
    int product = x * y;      // 指令2
BB  if (sum > product) {      // 指令3（条件分支）
        sum += 10;            // 指令4（入口）
        product = 0;          // 指令5（出口）
    } else {
        product -= 5;         // 指令6
    }
    int result = sum + product; // 指令7
    return result;            // 指令8
}
```

基本块的划分

划分基本块的核心是先找到“基本块的边界”（即“入口点”和“出口点”），再将入口点到下一个入口点（或程序结束）的指令划分为一个基本块。具体步骤如下：

步骤 1：确定“入口点”（基本块的起始指令）

步骤 2：确定“出口点”（基本块的结束指令）

步骤 3：划分基本块

基本块的划分

步骤1：确定“入口点”（基本块的起始指令）

满足以下任一条件的指令，即为入口点：

- 程序的第一条指令（整个程序的入口）。
- 被分支指令（如 if 的 else、for 的循环体、goto、return）跳转指向的指令（例如，goto L1中的L1对应的指令）。
- 分支指令的下一条指令（例如，if (a>0) { ... }中，if块结束后的第一条指令 —— 因为if执行完后会跳转到这里）。

```
int process(int x, int y) {
    int sum = x + y;          // 指令1 （入口）
    int product = x * y;      // 指令2
    if (sum > product) {      // 指令3（条件分支）
        sum += 10;            // 指令4 （入口）
        product = 0;          // 指令5
    } else {
        product -= 5;         // 指令6 （入口）
    }
    int result = sum + product; // 指令7 （入口）
    return result;            // 指令8
}
```

基本块的划分

步骤 2：确定 “出口点” （基本块的结束指令）

满足以下任一条件的指令，即为出口点（出口点的下一条指令必为入口点）：

- 分支指令：包括条件分支（如if、switch）、无条件分支（如goto）、返回指令（如return）—— 这些指令会**改变程序的执行流**，因此是当前块的最后一条指令。
- 程序的最后一条指令（无后续执行流）。

- 出口点：会改变程序执行流的指令
- 隐含的分支结束： 指的是没有显式分支指令（如if、goto），但程序执行到此处后会自动跳转到下一个基本块

```
int process(int x, int y) {
    int sum = x + y;          // 指令1
    int product = x * y;      // 指令2
    if (sum > product) {      // 指令3（条件分支）  (出口)
        sum += 10;            // 指令4
        product = 0;          // 指令5          出口?
    } else {
        product -= 5;         // 指令6          出口?
    }
    int result = sum + product; // 指令7
    return result;            // 指令8  (出口)
}
```

基本块的划分

步骤 3：划分基本块

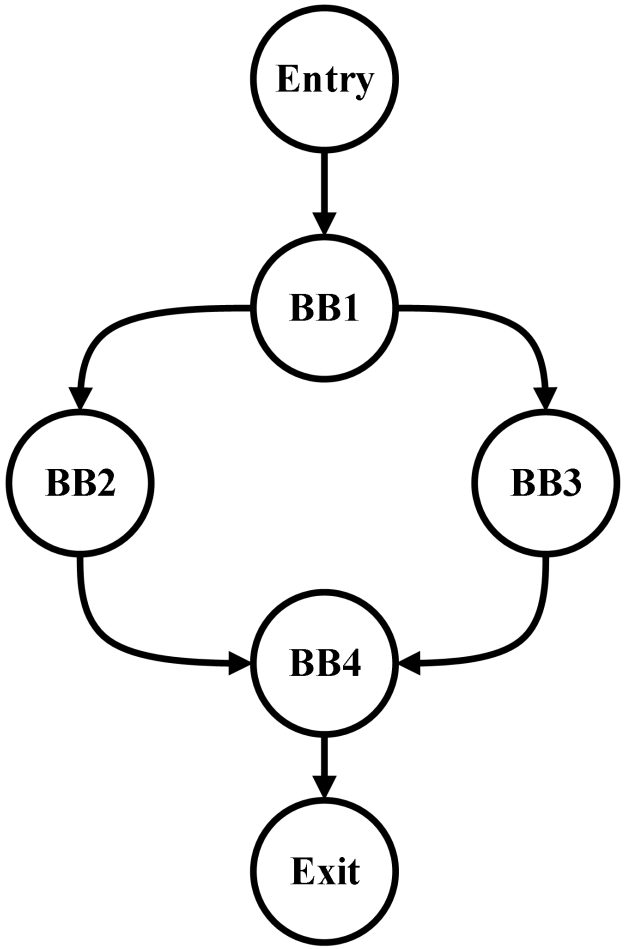
- 从每个入口点开始，依次收集指令，直到遇到下一个入口点或出口点，形成一个完整的基本块。

```
int process(int x, int y) {
BB1   int sum = x + y;          // 指令1  (入口)
      int product = x * y;      // 指令2
      if (sum > product) {      // 指令3（条件分支）  (出口)
BB2       sum += 10;            // 指令4  (入口)
          product = 0;          // 指令5  (分支结束)
      } else {
BB3       product -= 5;         // 指令6  (入口)  (分支结束)
      }
BB4   int result = sum + product; // 指令7  (入口)
      return result;            // 指令8  (出口)
}
```

控制流图（Control Flow Graph，CFG）：程序的“执行路径地图”

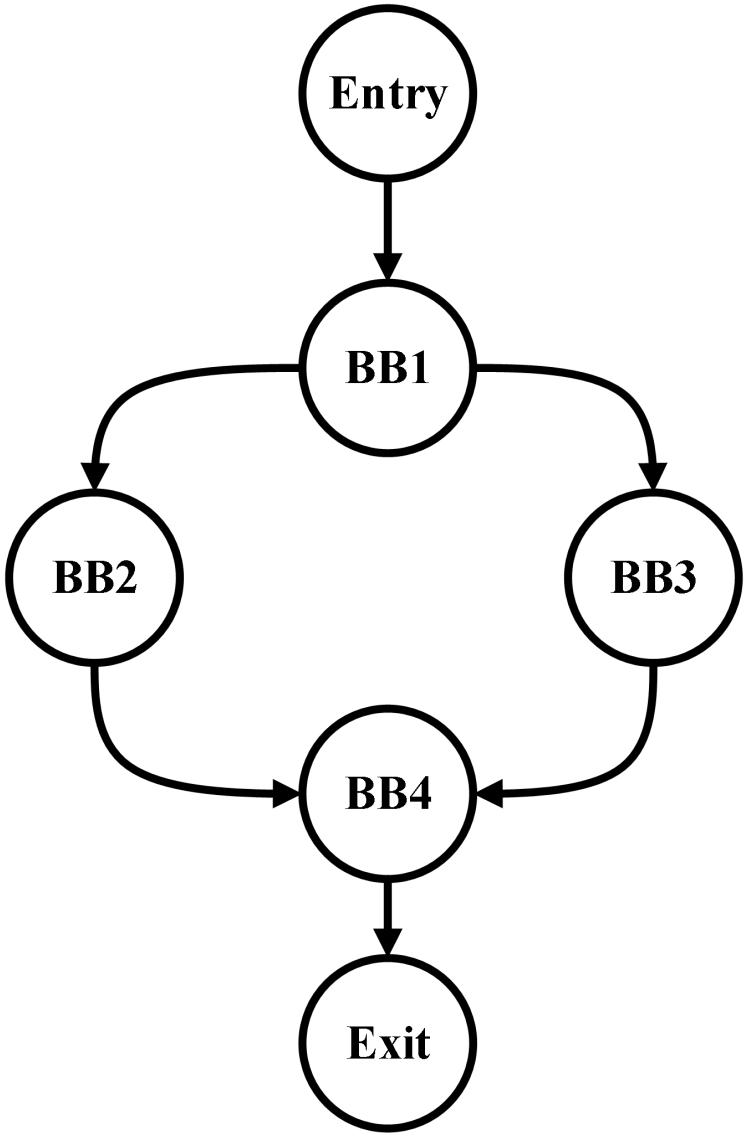
控制流图是用**有向图**表示程序执行流的工具，它以 “基本块” 为节点，以 “基本块之间的执行跳转关系” 为有向边，直观地展示程序所有可能的执行路径。

```
int process(int x, int y) {
    int sum = x + y;          // 指令1 (入口)
BB1 int product = x * y;      // 指令2
    if (sum > product) {      // 指令3 (条件分支) (出口)
        sum += 10;            // 指令4 (入口)
BB2     product = 0;          // 指令5 (分支结束)
    } else {
BB3     product -= 5;         // 指令6 (入口) (分支结束)
    }
    int result = sum + product; // 指令7 (入口)
BB4 return result;            // 指令8 (出口)
}
```



控制流图的组成

- 节点（Node）：每个节点对应一个基本块（BB），若基本块为空（如空循环体），也可表示为“空节点”；此外，通常会增加两个特殊节点：
  - ✓ 入口节点（Entry）：唯一的起始节点，指向程序的第一个基本块（表示程序开始执行）。
  - ✓ 出口节点（Exit）：唯一的结束节点，由所有return或程序结束的基本块指向（表示程序执行终止）。
- 有向边（Edge）：若基本块 A 执行完后，程序可能跳转到基本块 B，则从 A 到 B 画一条有向边（A→B），边的含义是“执行流从 A 转移到 B”。



控制流图的构建

在完成基本块划分后，按以下步骤构建 CFG：

1. 为每个基本块分配一个节点（记为 BB1、BB2、...、BBn），并添加 Entry 和 Exit 节点。
2. 对每个基本块 BBi，分析其出口指令的跳转逻辑：
   - 若 BBi 的出口是无条件分支（如 goto L）：找到 L 对应的基本块 BBj，添加边BBi → BBj。
   - 若 BBi 的出口是条件分支（如 if (cond) {X} else {Y}）：
     - ✓ 条件为 “真” 时，跳转到 X 对应的基本块 BBx，添加边BBi → BBx；
     - ✓ 条件为 “假” 时，跳转到 Y 对应的基本块 BBy（若 else 为空，则跳转到 BBi 的下一个基本块），添加边BBi → BBy。
   - 若 BBi 的出口是return 指令：添加边BBi → Exit（执行流终止）。
3. 连接 Entry 节点：添加边Entry → 程序的第一个基本块（如BB1）。

目录/Contents

西南石油大学
Southwest Petroleum University

01 基本路径法

02 逻辑覆盖法

03 程序插桩法

3.1 基本路径法

3.1.1 基本路径法概述

- 掌握基本路径法的概述，能够使用基本路径法设计测试用例

3.1.1 基本路径法概述

基本路径法是一种将程序的流程图转化为程序控制流图，并在程序控制流图的基础上，分析被测程序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。使用基本路径法设计的测试用例需要确保被测程序中的每个可执行语句至少被执行一次。

使用基本路径法设计测试用例的4个步骤

步骤1

步骤2

步骤3

步骤4

画出流程图

首先需要分析被测程序的源代码，并画出程序的流程图。

步骤1

步骤2

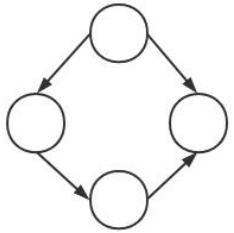
步骤3

步骤4

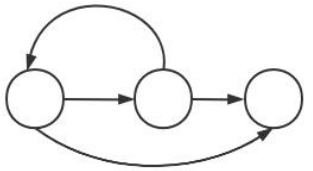
画出控制流图

控制流图是描述程序控制流的一种图示方法。控制流图可以由程序流程图转化而来。如果测试的源程序代码简洁，也可以直接通过分析源程序代码画出控制流图。在画程序的控制流图时，使用圆圈表示一条或多条无分支的语句（程序块）；使用箭头表示控制流方向。程序中常见的控制流图如下图所示。

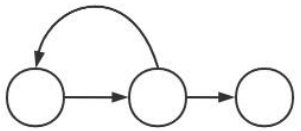
顺序结构



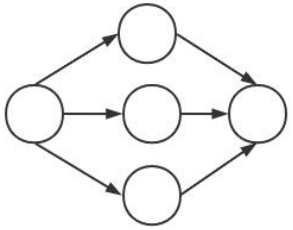
if条件语句结构



While循环语句结构



Until循环语句结构



多分支选择语句结构

步骤1

步骤2

步骤3

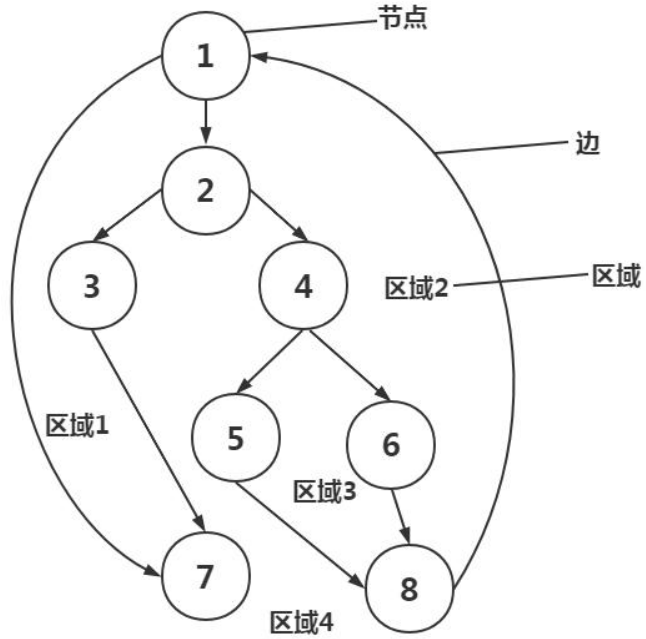
步骤4

计算程序的圈复杂度

计算程序圈复杂度的方法有3种，具体如下。

- 使用公式计算：V(G)=E-N+2，其中V(G)表示程序的圈复杂度，E表示控制流图中边的数量，N表示控制流图中节点的数量。
- 使用公式计算：V(G)=P+1，P表示控制流图中判定节点的数量。
- 程序的圈复杂度数量等于控制流图中的区域数量。

为了演示上述介绍的3种方法，假设某程序的控制流图如右图所示。



步骤1

步骤2

步骤3

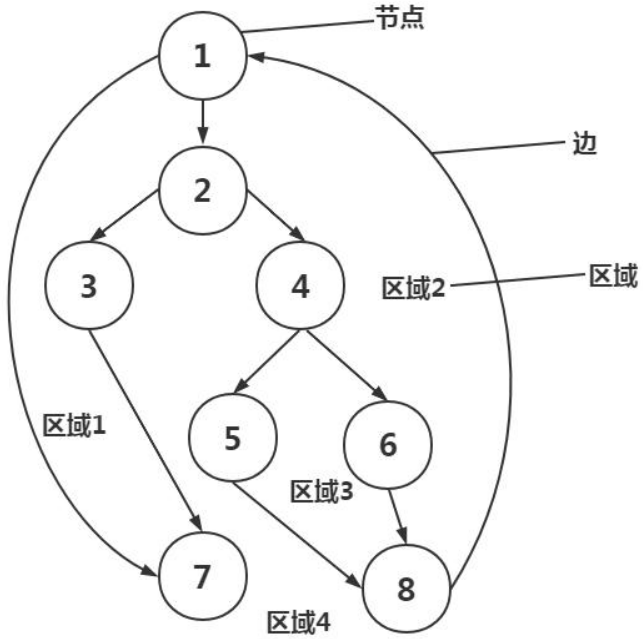
步骤4

设计测试用例

根据计算出的程序圈复杂度导出基本可执行路径集合，从而设计测试用例的输入数据和预期结果。以上一页中某程序的控制流图为例，由于圈复杂度为4，所以可以得到4条独立的路径，具体如下。

- 路径1：1→7。
- 路径2：1→2→3→7。
- 路径3：1→2→4→5→8→1→7。
- 路径4：1→2→4→6→8→1→7。

根据以上4条独立的路径即可设计测试用例，从而确保每一条路径都能被执行。



多学一招

如何将程序流程图转化为控制流图

特征	程序流程图	控制流图
最小单元	单个步骤（如输入、运算、判断）	基本块（连续的顺序执行步骤）
节点含义	具体操作或判断	抽象的基本块
侧重点	展示程序的执行步骤细节	展示基本块之间的跳转关系
用途	便于人理解程序逻辑	便于编译器分析和优化

3.1.1 程序流程图转化为控制流图

转化的核心思路是：提取基本块并建立块间跳转关系，本质是将流程图中 “过程化的步骤” 抽象为 “结构化的基本块节点”。以下为具体步骤：

核心转化规则

1. 基本块提取：将流程图中 “连续顺序执行的步骤” 合并为一个基本块（BB），基本块需满足 “单入口、单出口”。
2. 节点替换：用 “基本块” 替换流程图中的 “处理步骤”，用 “有向边” 替换流程图中的 “控制流箭头”。
3. 保留分支结构：流程图中的分支（如if-else）、循环（如while）会转化为基本块之间的分支边。

步骤 1：识别 “入口点” （基本块的起始位置）

在流程图中，以下位置为基本块入口点：

- 流程图的起始节点（程序开始处）。
- 所有分支的目标节点（如if的 “真” 分支目标、 “假” 分支目标）。
- 分支结构（如if-else、循环）的结束后第一个节点。

步骤 2：识别 “出口点” （基本块的结束位置）

- 所有分支判断节点（如if条件判断、while循环条件）。
- 程序的终止节点（如return、结束框）。

步骤 3：划分基本块

从每个入口点开始，按顺序收集流程图中的处理步骤，直到遇到下一个入口点或出口点，形成一个基本块。

规则：基本块内只能有 “处理步骤” （如赋值、运算），不能包含分支判断或跳转。

步骤 4：为基本块命名

用 BB1、BB2、... 等标识每个基本块，便于后续连接。

步骤 5：绘制节点与边

- 节点：每个基本块对应一个或多个矩形节点，内部可标注块内的核心操作（无需列出所有步骤）
- 有向边：根据流程图的控制流箭头，连接相关基本块：
  - 分支判断节点（如if）所在的基本块，需引出两条边（对应 “真” “假” 分支）。
  - 循环结构中，循环条件块需引出两条边（“进入循环体” 和 “退出循环”）。
  - 顺序执行的基本块之间用单条边连接。

步骤 6：添加入口和出口节点

- 入口节点（Entry）：唯一的起始节点，指向程序的第一个基本块。
- 出口节点（Exit）：唯一的终止节点，由所有结束程序的基本块（如含return的块）指向。

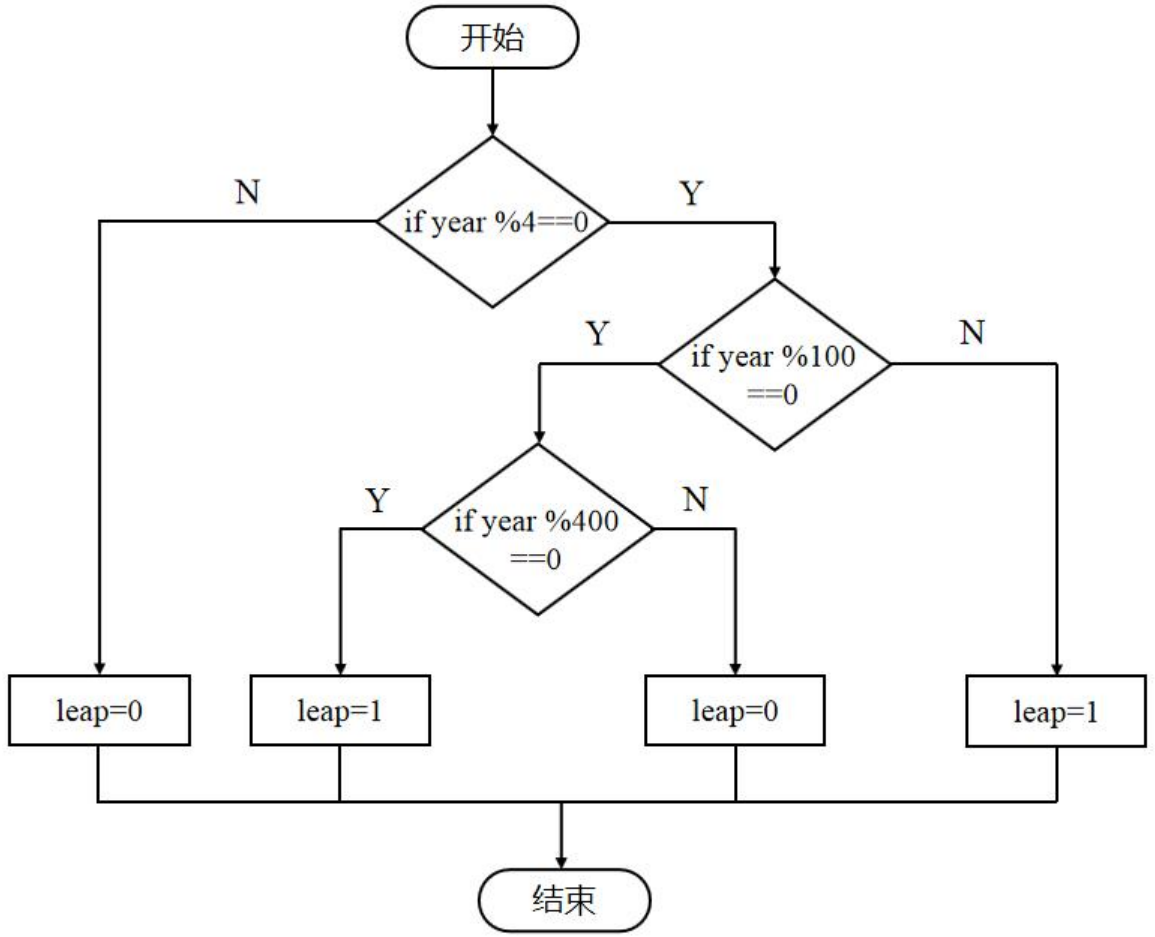
3.1.2 实例：判断年份是否为闰年

在白盒测试中，经常使用基本路径法测试程序的代码。为了让读者更好地掌握基本路径法的使用，下面以判断闰年问题的C语言程序代码为例，讲解如何通过基本路径法设计测试用例。

3.1.2 实例：判断年份是否为闰年

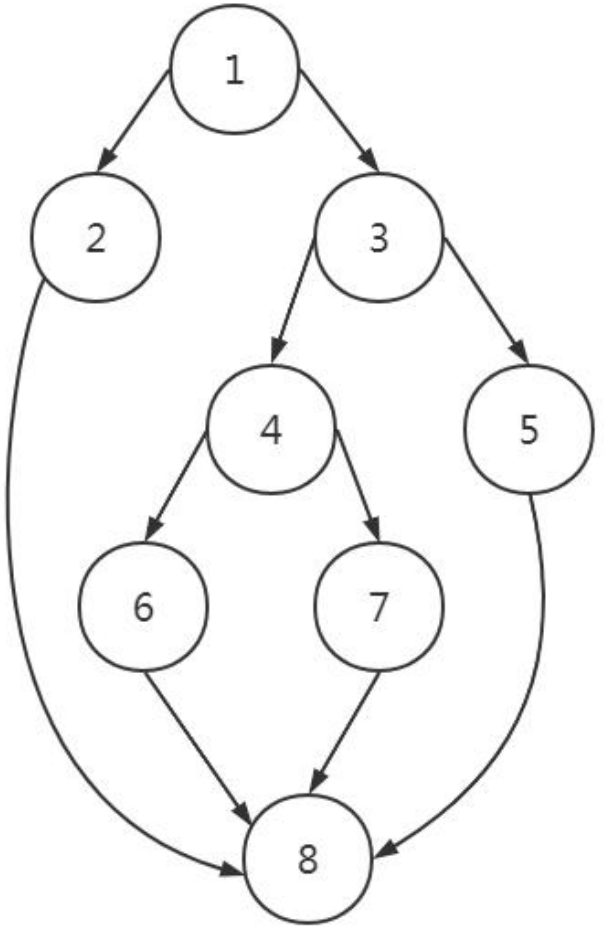
当年份能够被4但不能被100整除时为闰年，或者年份能够被400整除时为闰年，据此可以设计判断输入的年份是否为闰年的C语言程序代码，具体代码和流程图如下。

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
	int year,leap;
	printf("Enter year:");
	scanf("%d",&year);
	if(year%4==0)
	{
		if(year%100==0)
		{
			if(year%400==0)
				leap=1;
			else
				leap=0;
		}
		else
			leap=1;
	}
	else
		leap=0;
	return 0;
}
```



3.1.2 实例：判断年份是否为闰年

画出程序的控制流图，如右图所示。



右图中，一共有10条边，8个节点，4个区域，其中判定节点有3个，分别是1、3、4，程序的圈复杂度为4。

根据圈复杂度可以得到4条独立的路径，具体如下。

- 路径1：1→2→8。
- 路径2：1→3→4→6→8。
- 路径3：1→3→4→7→8。
- 路径4：1→3→5→8。

3.1.2 实例：判断年份是否为闰年

根据上一页中的4条独立路径即可设计测试用例。判断闰年问题的测试用例如下表所示。

测试用例	执行路径	输入数据	预期结果
test1	路径1	year=1999	leap=0
test2	路径2	year=2000	leap=1
test3	路径3	year=1900	leap=0
test4	路径4	year=2020	leap=1

当预期结果leap=0时，表示平年；当预期结果leap=1时，表示闰年。

3.2 逻辑覆盖法

测试覆盖率

测试覆盖率：用于确定测试所执行到的覆盖项的百分比，其中覆盖项是指作为测试基础的一个入口或属性，如语句、分支、条件等。

测试覆盖率是对测试充分性的表示，它可以作为在测试分析报告中的一个可量化的指标依据，一般认为测试覆盖率越高，测试效果越好。但是测试覆盖率并非测试的绝对目标，而只是一种手段。

常用的测试覆盖评测

1. 基于需求的测试覆盖

基于需求的测试覆盖在测试生命周期中要评测多次，并在测试生命周期的里程碑处提供测试覆盖的标识（如已计划的、已实施的、已执行的和成功的测试覆盖）。

2. 基于代码的测试覆盖

基于代码的测试覆盖评测测试过程中已经执行的代码的多少，与之相对的是要执行的剩余代码的多少。基于代码的测试覆盖可以建立在控制流（语句、分支或路径）或数据流的基础上。

3.2.1 语句覆盖

- 掌握语句覆盖法的使用，能够应用语句覆盖法设计测试用例

3.2.1 语句覆盖

语句覆盖(Statement Coverage)又称行覆盖、段覆盖、基本块覆盖，它是最常见的覆盖方式之一。语句覆盖的目的是测试程序中的代码是否被执行，它只测试代码中的执行语句，这里的执行语句不包括头文件、注释、空行等。语句覆盖在多分支的程序中只能覆盖某一条路径，使得该路径中的每一个语句至少被执行一次，不会考虑各种分支组合的情况。

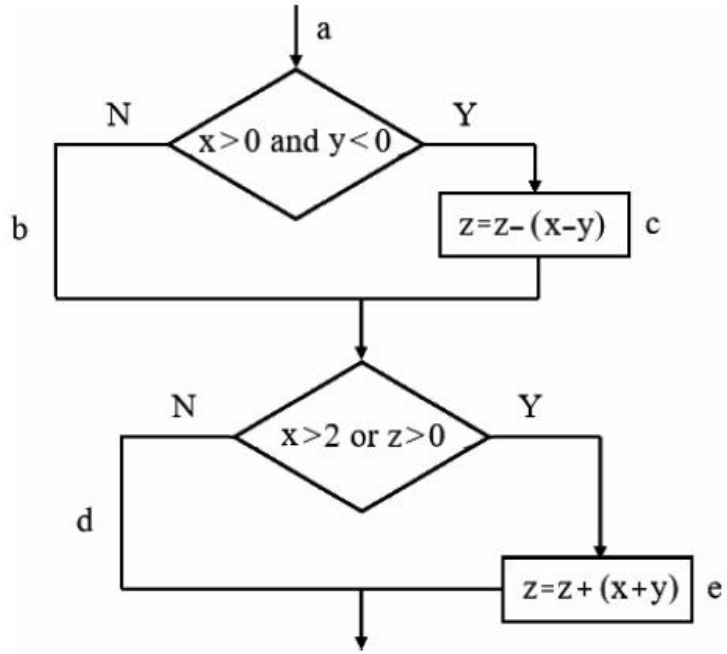
下面结合一段小程序介绍语句覆盖中方法的执行，程序伪代码如下。

```
if x>0 and y<0    //条件1
   z=z-(x-y)
if x>2 or z>0     //条件2
   z=z+(x+y)
```

上述代码中，and表示逻辑运算&&，or表示逻辑运算||。第1~2行代码表示如果x>0成立并且y<0成立，则执行z=z-(x-y)语句；第3~4行代码表示如果x>2成立或者z>0成立，则执行z=z+(x+y)语句。

3.2.1 语句覆盖

根据程序伪代码可以画出流程图，程序执行的流程图如右图所示。



根据程序执行流程图中标示的语句执行路径设计测试用例，具体如下。

```
test1:x=3   y=-1   z=2
```

执行上述测试用例，程序的运行路径为a→c→e。可以看出程序中a→c→e路径上的每个语句都能被执行，但是语句覆盖无法全面反映多分支的逻辑，仅仅执行一次不能进行全面覆盖。因此，语句覆盖是弱覆盖方法。

3.2.2 判定覆盖

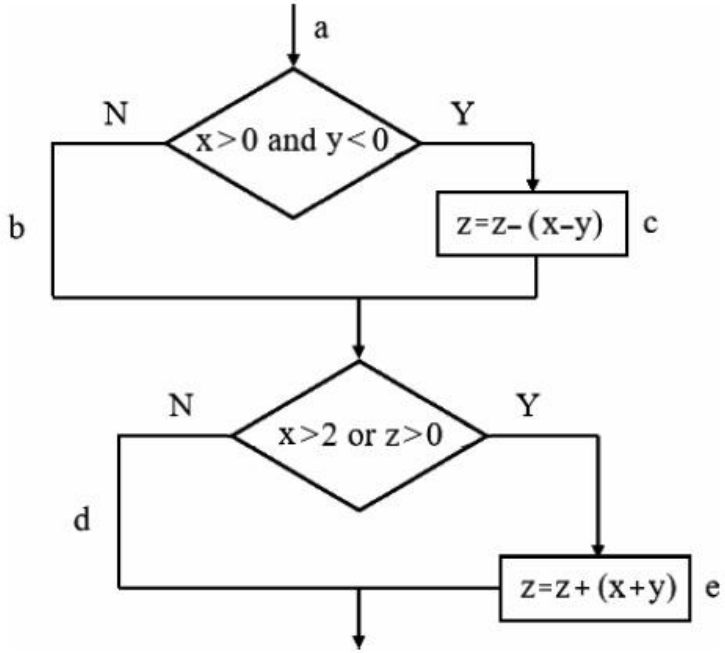
- 掌握判定覆盖法的使用，能够应用判定覆盖法设计测试用例

3.2.2 判定覆盖

判定覆盖(Decision Coverage)又称为分支覆盖，其原则是设计足够多的测试用例，在测试过程中保证每个判定条件至少有一次为真值，有一次为假值。判定覆盖的作用是使真假分支均被执行，虽然判定覆盖比语句覆盖测试能力强，但仍然具有和语句覆盖一样的单一性。

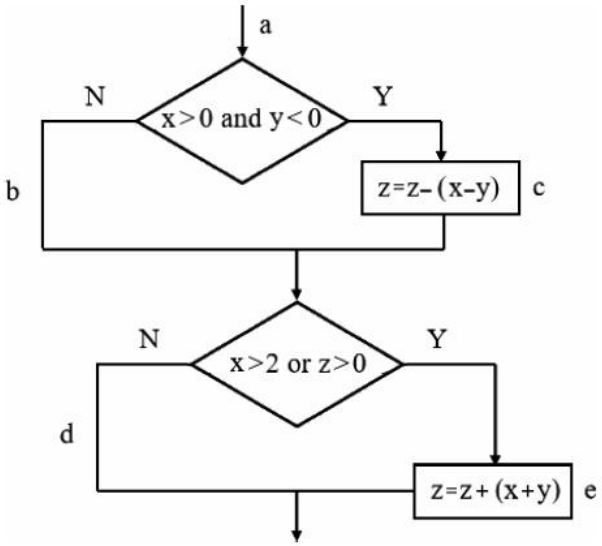
3.2.2 判定覆盖

在该程序中，有2个判定语句，每个判定语句有2个逻辑条件，共有4个逻辑条件，使用标识符标记各个逻辑条件取真值与取假值的情况，判定条件如下表所示。



判定1	条件标记	判定2	条件标记
x>0	S1	x>2	S3
x<=0	-S1	x<=2	-S3
y<0	S2	z>0	S4
y>=0	-S2	z<=0	-S4

判定覆盖测试用例如下表所示。



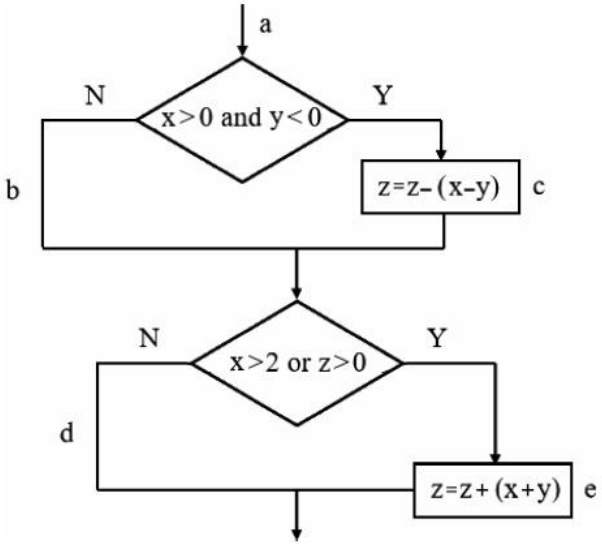
测试用例	x	y	z	条件标记	判定1	判定2	执行路径
test1	-3	1	-5	-S1、-S2、-S3、-S4	0	0	a→b→d
test2	3	-1	-5	S1、S2、S3、-S4	1	1	a→c→e

3.2.3 条件覆盖

- 掌握条件覆盖法的使用，能够应用条件覆盖法设计测试用例

3.2.3 条件覆盖

条件覆盖(Condition Coverage)是指设计足够多的测试用例，使判定语句中的每个逻辑条件取真值与取假值至少出现一次。



测试用例	x	y	z	条件标记	判定1	判定2	执行路径
test1	-3	-1	1	-S1、S2、-S3、S4	0	1	a→b→e
test2	3	1	-1	S1、-S2、S3、-S4	0	1	a→b→e

3.2.4 判定-条件覆盖

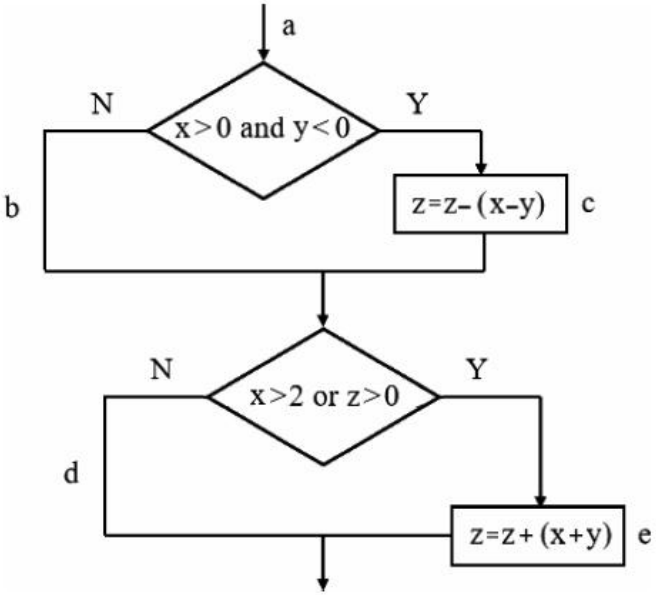
- 掌握判定-条件覆盖法的使用，能够应用判定-条件覆盖法设计测试用例

3.2.4 判定-条件覆盖

判定-条件覆盖(Condition/Decision Coverage)要求设计较多的测试用例，使得判定语句中所有条件的可能取值至少出现一次，同时，所有判定语句的可能结果也至少出现一次。例如，对于判定语句if(x>0 and y<0)，该判定语句有x>1、y<1这2个条件，则在设计测试用例时，要保证x>0和y<0这2个条件取真值、假值至少一次，同时，判定语句if(x>0 and y<0)取真值、假值也至少出现一次。判定-条件覆盖弥补了判定覆盖和条件覆盖的不足之处。

3.2.4 判定-条件覆盖

根据判定-条件覆盖原则，设计判定-条件覆盖测试用例。判定-条件覆盖测试用例如下表所示。



测试用例	x	y	z	条件标记	条件1	条件2	执行路径
test1	3	1	1	S1、-S2、S3、S4	0	1	a→b→e
test2	-3	1	-1	-S1、-S2、-S3、-S4	0	0	a→b→d
test3	3	-1	1	S1、S2、S3、S4	1	1	a→c→e

相比条件覆盖、判定覆盖，判定-条件覆盖弥补了前两者的不足，但是由于判定-条件覆盖没有考虑判定语句与条件判断的组合情况，其覆盖范围并没有比条件覆盖更全面，判定-条件覆盖也没有覆盖a→c→d路径，因此判定-条件覆盖也存在遗漏测试的情况。

3.2.5 条件组合覆盖

- 掌握条件组合覆盖法的使用，能够应用条件组合覆盖法设计测试用例

3.2.5 条件组合覆盖

条件组合覆盖(Multiple Condition Coverage)是指设计足够多的测试用例，使判定语句中每个条件的所有可能情况至少出现一次，并且每个判定语句本身的判定结果也至少出现一次。它与判定-条件覆盖的区别是，它不是简单地要求每个条件都出现真与假2种结果，而是要求让这些结果的所有可能组合都至少出现一次。

3.2.5 条件组合覆盖

程序中共有4个条件：x>0、y<0、x>2、z>0。下面继续使用S1、S2、S3、S4标记这4个条件成立，用-S1、-S2、-S3、-S4标记这4个条件不成立。S1与S2属于一个判定语句，两两组合有4种情况，如下所示。

- S1，S2
- S1，-S2
- -S1，S2
- -S1，-S2。

同样，S3与S4属于一个判定语句，两两组合也有4种情况。2个判定语句的组合情况各有4种。在执行程序时，只要能分别覆盖2个判定语句的组合情况即可，因此，条件组合覆盖至少要设计4个测试用例。

条件组合覆盖的4种情况如下表所示。

序号	组合	含义
1	S1、S2、S3、S4	x>0成立，y<0成立，x>2成立，z>0成立
2	S1、-S2、S3、-S4	x>0成立，y<0不成立，x>2成立，z>0不成立
3	-S1、S2、-S3、S4	x>0不成立，y<0成立，x>2不成立，z>0成立
4	-S1、-S2、-S3、-S4	x>0不成立，y<0不成立，x>2不成立，z>0不成立

条件组合覆盖测试用例

序号	组合	测试用例			条件1	条件2	覆盖路径
		x	y	z			
test1	S1、S2、S3、S4	3	-1	5	1	1	a→c→e
Test2	S1、-S2、S3、-S4	6	1	-2	0	1	a→b→e
test3	-S1、S2、-S3、S4	-5	-2	1	0	1	a→b→e
test4	-S1、-S2、-S3、-S4	-3	1	-1	0	0	a→b→d

a→c→d?

3.2.6 实例：三角形的逻辑覆盖

- 掌握三角形的逻辑覆盖，能够设计三角形程序判定覆盖测试用例

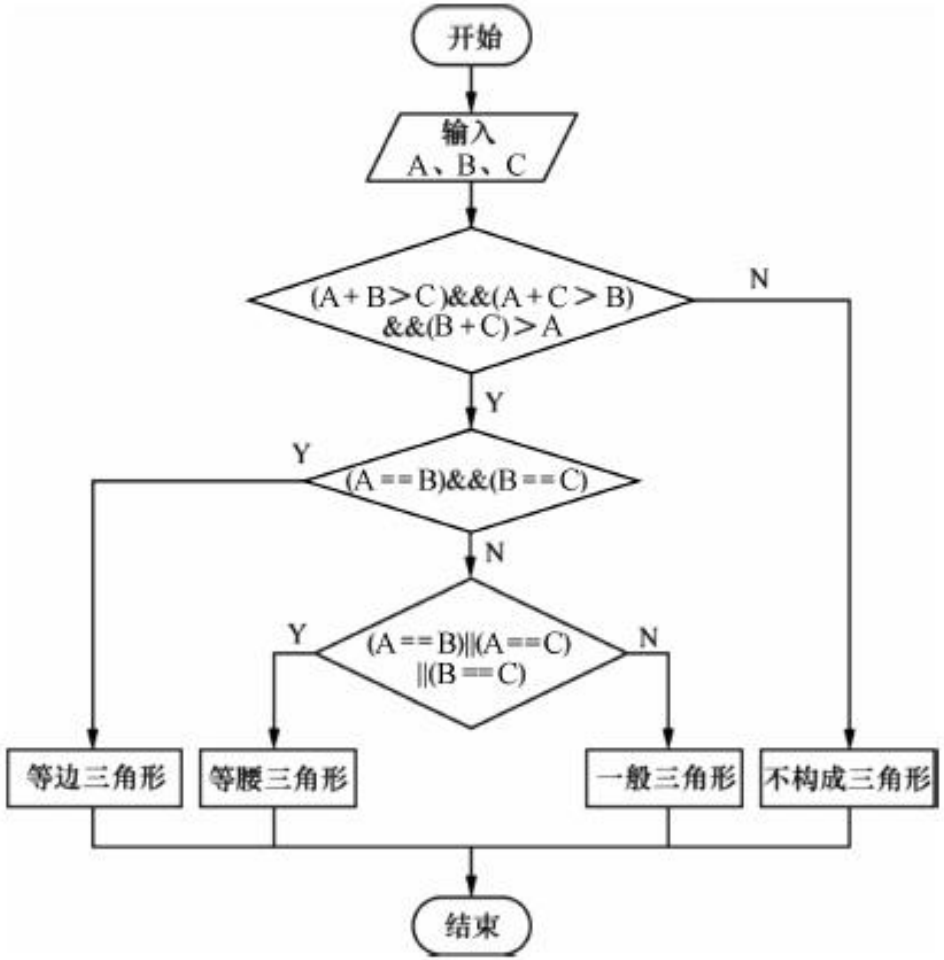
3.2.6 实例：三角形的逻辑覆盖

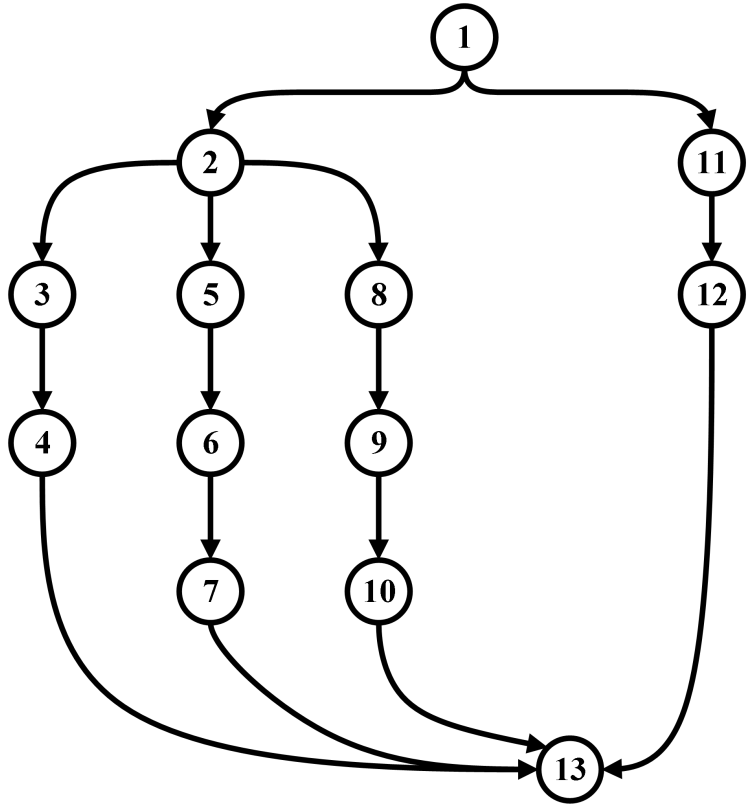
在第2章的黑盒测试中使用了决策表法判断三角形的类型，根据三角形三边关系可知可能出现4种情况：不构成三角形、一般三角形、等腰三角形、等边三角形。据此实现一个判断三角形的程序，伪代码如下。

```
int  A B C      //三角形的三个边
if((A+B>C)&&(A+C>B)&&(B+C)>A)  //是否满足三角形构成条件
    if((A==B)&&(B==C))  //等边三角形
        等边三角形
    else if((A==B)||(B==C)||(A==C))     //等腰三角形
        等腰三角形
    else                                    //一般三角形
        一般三角形
else
    不是三角形
end
```

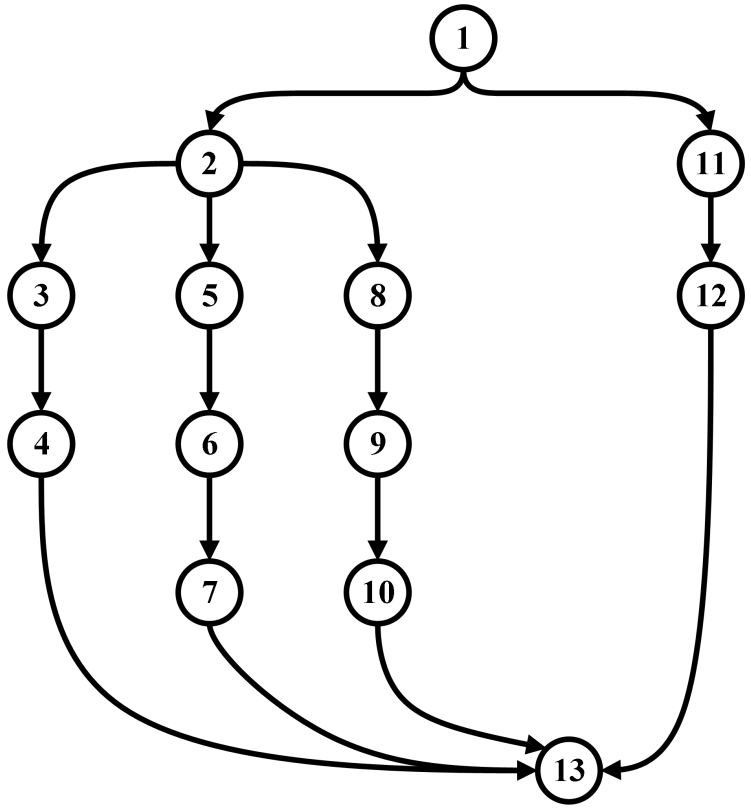
3.2.6 实例：三角形的逻辑覆盖

根据上述代码可以画出流程图和控制流图





3.2.6 实例：三角形的逻辑覆盖



三角形程序测试用例

编号	测试用例			路径	预期结果
	A	B	C		
test1	6	6	6	1→2→3→4→13	等边三角形
test2	6	6	8	1→2→5→6→7→13	等腰三角形
test3	3	4	5	1→2→8→9→10→13	一般三角形
test4	3	3	6	1→11→12→13	不构成三角形

总结对比

覆盖准则	覆盖目标（核心）	优点	缺点
语句覆盖	所有语句至少执行 1 次	简单易实现	不关心判定 / 条件，漏洞多
判定覆盖	每个判定的 T/F 至少执行 1 次	覆盖判定结果	不关心条件的真假状态
条件覆盖	每个条件的 T/F 至少执行 1 次	覆盖条件状态	可能遗漏判定结果（如本例未覆盖 T）
判定 - 条件覆盖	同时覆盖判定 T/F 和条件 T/F	兼顾判定和条件	未覆盖条件的所有组合
条件组合覆盖	所有条件的真假组合至少执行 1 次	覆盖最全面，漏洞最少	用例数多（2^n），测试成本高

- 覆盖程度从低到高：**语句覆盖 < 判定覆盖 < 判定 - 条件覆盖 < 条件组合覆盖**（条件覆盖与判定覆盖无绝对高低，需看场景）；
- 测试成本从低到高：与覆盖程度正相关（条件组合覆盖成本最高）；
- 实际测试中需平衡 “覆盖程度” 和 “成本”，并非越全面越好（如简单逻辑用判定 - 条件覆盖即可，复杂逻辑需考虑条件组合覆盖）。

3.3 程序插桩法

3.3.1 目标代码插桩

- 了解目标代码插桩法的原理，能够描述目标代码插桩法的3种执行模式

3.3.1 目标代码插桩

目标代码插桩是指向目标代码（即二进制代码）插入测试代码，以获取程序运行信息的测试方法，也称为动态程序分析方法。在进行目标代码插桩之前，测试人员要对目标代码的逻辑结构进行分析，从而确认需要插桩的位置。

目标代码插桩对程序运行时的内存监控、指令跟踪、错误检测等有着重要意义。相比于逻辑覆盖法，目标代码插桩在测试过程中不需要重新编译代码或链接程序，并且目标代码的格式与具体的编程语言无关，主要与操作系统相关，因此目标代码插桩被广泛使用。

1. 目标代码插桩的原理

目标代码插桩法的原理是在不改变目标程序原有逻辑功能的基础上，通过向目标代码（二进制代码）中插入特定的测试代码（桩代码），获取程序运行时的各种信息，从而实现对程序的分析与测试。具体如下：

（1）插桩位置确定

在插桩之前，测试人员或分析工具会对目标代码进行静态分析。包括解析目标代码的结构，比如函数调用关系、控制流（如循环、分支语句）和数据流（变量的赋值、传递等）。通过这些分析，确定哪些位置插入桩代码能够获取到有价值的信息。根据不同的测试目的，选择不同的插桩位置。如果要进行性能测试，可能会在关键函数、循环体等性能瓶颈可能出现的地方插桩；若要进行逻辑正确性测试，则会在条件判断语句、分支跳转语句处插桩，以检查程序是否按照预期的逻辑执行。

（2）桩代码插入

对于二进制形式的目标代码，需要借助专门的二进制编辑工具或插桩工具，在确定的插桩位置直接修改目标代码，将桩代码插入进去。如果是高级语言编写的源程序，一些插桩工具可以在编译阶段介入，在源程序被编译成目标代码的过程中，按照插桩规则将桩代码插入到合适的位置。

（3）桩代码执行与信息获取

当插入桩代码的目标程序运行时，一旦执行到插桩位置，桩代码就会被触发执行。桩代码可以实现各种功能，比如记录当前程序的执行状态（如变量的值、程序计数器的值等）。桩代码执行后获取到的信息，会按照预先设定的方式进行处理。有些桩代码会将信息直接打印输出，方便开发人员或测试人员查看；有些则会将信息存储到特定的数据结构（如日志文件、内存缓冲区等）中，供后续进一步分析。

2. 目标代码插桩的两种方式

由于目标代码是可执行的二进制代码，所以目标代码的插桩可分为两种方式。

静态插桩

在目标代码尚未运行之前，借助特定的工具或技术，对目标代码（可执行的二进制代码 ）进行分析，确定需要插入测试代码（桩代码）的位置，然后将桩代码按照从头到尾的顺序插入到目标代码中，完成插桩后再去执行程序。这种方式适用于需要实现完整系统或仿真（模拟真实系统）进行的代码覆盖测试，更容易系统和全面的规划。

动态插桩

在程序已经处于运行状态时，利用专门的动态插桩工具，在不中断程序正常运行或者尽可能减少对程序运行影响的前提下，将测试代码插入到正在运行的程序中。通过这种方式，可以实时获取程序在特定时间点或时间段内的运行状态信息。

3. 目标代码插桩的执行模式

目标代码插桩具有以下3种执行模式。

3种执行模式

Just-In-Time Mode

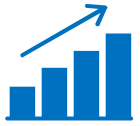
即时模式

将修改部分的二进制代码以副本的形式在新的内存区域中，在测试时仅执行修改部分的目标代码。

Interpretation Mode

解释模式

测试人员插入的测试代码作为目标代码指令的解释语言。每当执行一条目标代码指令时，程序就会在测试代码中查找并执行相应的替代指令。



Probe Mode

探测模式

探测模式使用新指令覆盖旧指令进行测试，这种模式在某些体系结构（如x86体系结构，指令丰富且兼容性强）中比较适用。

4. 目标代码插桩工具

常见的目标代码插桩工具主要有以下2种。

（1）Pin-A Dynamic Binary Instrumentation Tool（Pin）

Pin是由Intel公司开发的免费框架，它可以用于二进制代码检测与源代码检测。Pin支持IA-32、x86-64、MIC（Many Integrated Core，众核架构）体系，可以运行在Linux平台、Windows平台和Android平台上。Pin具有基本块分析器、缓存模拟器、指令跟踪生成器等模块，使用该工具可以创建程序分析工具、监视程序运行的状态信息等。Pin非常稳定可靠，常用于大型程序测试，（如Office办公软件、虚拟现实引擎等）测试。

（2）DynamoRIO

DynamoRIO是一个受许可的动态二进制代码检测框架，作为应用程序和操作系统的中间平台，它可以在程序执行时实现程序任何部分的代码转换。DynamoRIO支持IA-32、x86-64、AArch64体系，可以运行在Linux平台、Windows平台和Android平台上。DynamoRIO包含内存调试工具、内存跟踪工具、指令跟踪工具等。

3.3.2 源代码插桩

- 掌握源代码插桩法的使用，能够应用探针代码测试程序

3.3.2 源代码插桩

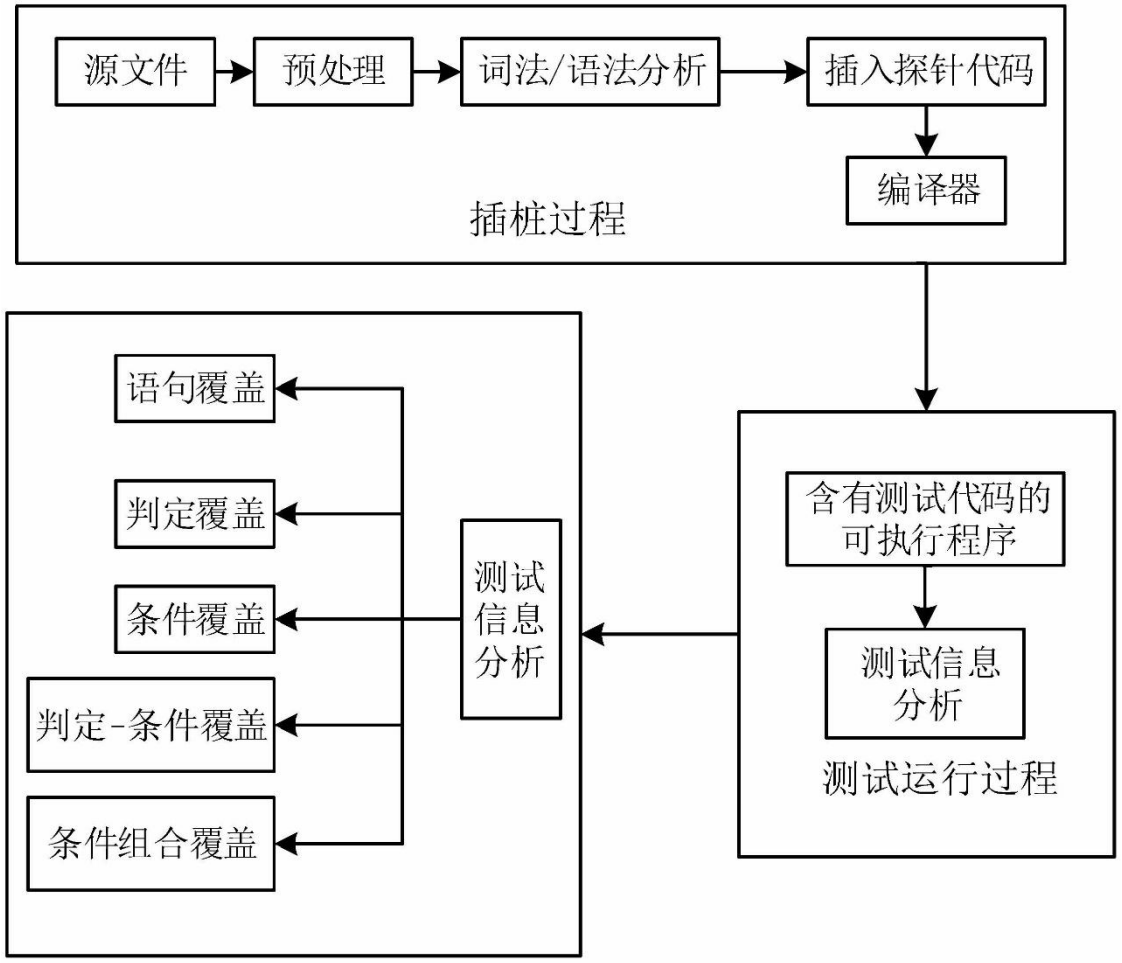
源代码插桩是在软件开发过程中，对高级语言编写的源代码进行插桩处理，以此获取程序运行信息、辅助程序调试与测试的一种技术手段。

在程序的源代码层面，依据测试、调试或分析的需求，在合适的位置插入特定的代码片段（桩代码）。当对插入桩代码后的源程序进行编译、链接并运行时，这些桩代码会随着程序的执行而被触发，进而实现对程序运行状态和行为的监控。 比如，在一个用 C 语言编写的计算函数中，为了获取函数中某个变量在每次循环时的值，可在循环体中插入打印该变量值的桩代码。

3.3.2 源代码插桩

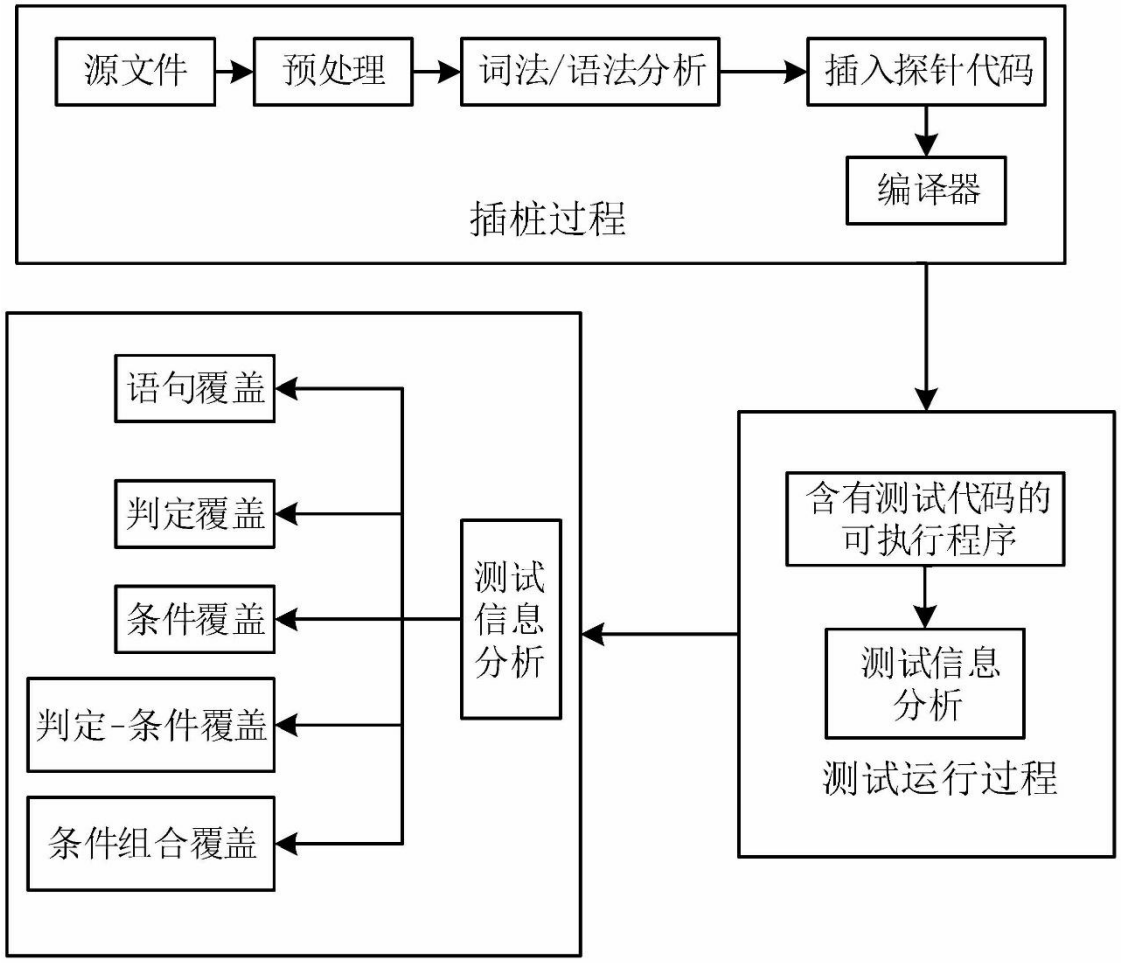
一、插桩过程

- 源文件：待测试的程序源代码。
- 预处理：对源文件进行预处理操作。
- 词法 / 语法分析：对预处理后的代码进行 “拆解” —— 词法分析将代码拆分为一个个标记（如关键字、变量名、运算符）；语法分析则基于标记构建语法树，明确代码的逻辑结构（如函数、分支、循环的层次关系）。
- 插入探针代码：根据测试需求，在代码的关键位置（如函数入口 / 出口、分支判断处等）插入 “探针代码”（即用于收集测试信息的代码片段，比如记录代码执行次数、变量值变化等）。
- 编译器：将插入探针代码后的源代码编译成含有测试代码的可执行程序。



二、测试运行过程

- 含有测试代码的可执行程序：运行插桩后的程序，此时程序不仅执行原有功能，还会通过探针代码收集测试信息。
- 测试信息分析：程序运行过程中，探针代码会持续收集数据；运行结束后，对这些数据进行分析。
- 测试覆盖分析：通过分析收集到的测试信息，评估程序的测试覆盖程度，常见的覆盖类型包括：
  - 语句覆盖。
  - 判定覆盖。
  - 条件覆盖。
  - 判定 - 条件覆盖。
  - 条件组合覆盖。



3.3.2 源代码插桩

通过一个小案例来讲解源代码插桩，该案例是一个除法运算，具体代码如图所示。

程序运行后，提示输入被除数和除数，在输入除数后，程序宏函数ASSERT(y)判断除数是否为0，若除数为0则输出错误信息，程序运行结束；若除数不为0，则进行除法运算并输出计算结果。

```
#include <stdio.h>
//定义ASSERT(y)
#define ASSERT(y) if(y){   printf("出错文件：%s\n",__FILE__);\
                           printf("在%d行：\n",__LINE__\);
                           printf("提示：除数不能为0！\n");\
}
int main()
{
        int x,y;
        printf("请输入被除数：");
        scanf("%d",&x);
        printf("请输入除数：");
        scanf("%d",&y);
        ASSERT(y==0); //插入的桩（即探针代码）
        printf("%d",x/y);
        return 0;
}
```

根据除法运算程序设计测试用例，除法运算测试用例如下表所示。

测试用例	测试数据	预期结果
test1	1,1	1
test2	1,-1	-1
test3	-1,-1	1
test4	-1,1	-1
test5	1,0	错误
test6	-1,0	错误
test7	0,0	错误
test8	0,1	0
test9	0,-1	0

3.3.2 源代码插桩

程序的目标代码插桩与源代码插桩测试方法有效提高了代码测试覆盖率，但是使用插桩测试方法会出现代码膨胀、执行效率低下、HeisenBugs等问题。在一般情况下，插桩后的代码膨胀率在20%~40%，甚至能达到100%，导致插桩测试失败。

小提示：HeisenBugs

HeisenBugs即海森堡Bug，它是一种软件缺陷，这种缺陷的重现率很低，当开发人员试图研究时，它会消失或改变。实际软件测试中，这种缺陷也比较常见，例如，测试人员测试到一个缺陷并提交给开发人员后，开发人员按照测试人员提交的缺陷报告中的步骤执行时，却无法重现缺陷，其原因是缺陷已经消失或者出现了其他缺陷。

3.3.2 源代码插桩与目标代码插桩对比

对比维度	源代码插桩	目标代码插桩
操作对象	高级语言源代码（如 C/C++、Java、Python 代码）	二进制目标代码（可执行文件、库文件等）
是否需要源码	必须需要，无源码则无法操作	无需源码，直接处理二进制文件
编程语言依赖	强依赖：需适配不同语言的语法（如 Java 用 AspectJ，C 用宏定义）	弱依赖：与源码语言无关，仅依赖操作系统 / 指令集（如 x86、ARM）
插桩时机	程序编译前（源码阶段）	程序编译后（二进制阶段），可分静态（运行前）、动态（运行中）
编译依赖	插桩后需重新编译、链接生成新程序	无需重新编译，直接修改二进制或运行时插入
适用场景	开发阶段调试、单元测试、源码级性能分析	闭源软件测试、运行时监控、漏洞检测、跨语言程序分析
对程序影响	可能增加源码体积，编译后才能验证插桩效果	不改变原始源码，动态插桩可实时启停，影响较小

3.3.3 实例：求3个数的中间值

- 掌握求3个数的中间值的案例，能够使用源代码插桩求3个数的中间值

3.3.3 实例：求3个数的中间值

下面通过一个案例对源代码插桩进行讲解，以加深读者对源代码插桩的理解。该案例要求用键盘输入3个数并求中间值，源程序代码如下。测试人员通过写入的文件可以查看源程序执行的过程，插桩后的代码如下。

```c
#include <stdio.h>
int main()
{
    int i,mid,a[3];
    for(i=0;i<3;i++)
        scanf("%d",&a[i]);
    mid=a[2];
    if(a[1]<a[2])
    {
        if(a[0]<a[1])
            mid=a[1];
        else if(a[0]<a[2])
            mid=a[1];
    }
    else
    {
        if(a[0]>a[1])
            mid=a[1];
        else if(a[0]>a[2])
            mid=a[0];
    }
    printf("中间值是:%d\n",mid);
    return 0;
}
```

```c
#include <stdio.h>
#define LINE() fprintf(__POINT__,"%3d",__LINE__)
FILE *__POINT__;
int main()
{
    if((__POINT__=fopen("test.txt","w"))==NULL)
        fprintf(stderr,"不能打开test.txt文件");
    int i,mid,a[3];
    for(LINE(),i=0;i<3;LINE(),i++)
        LINE(),scanf("%d",&a[i]);
    LINE(),mid=a[2];
    if(LINE(),a[1]<a[2])
    {
        if(LINE(),a[0]<a[1])
            LINE(),mid=a[1];
        else if(LINE(),a[0]<a[2])
            LINE(),mid=a[1];
    }
    else
    {
        if(LINE(),a[0]>a[1])
            LINE(),mid=a[1];
        else if(LINE(),a[0]>a[2])
            LINE(),mid=a[0];
    }
    LINE(),printf("中间值是：%d\n",mid);
    LINE(),fclose(__POINT__);
    return 0;
}
```

3.3.3 实例：求3个数的中间值

源代码插桩完成之后，根据3个数的不同组合方式设计测试用例，具体测试用例如下表所示。

测试用例	测试数据	预期结果
test1	1,1,2	1
test2	1,2,3	2
test3	3,2,1	2
test4	3,3,3	3
test5	6,4,5	5
test6	6,8,4	6
test7	8,4,9	8

3.3.3 实例：求3个数的中间值

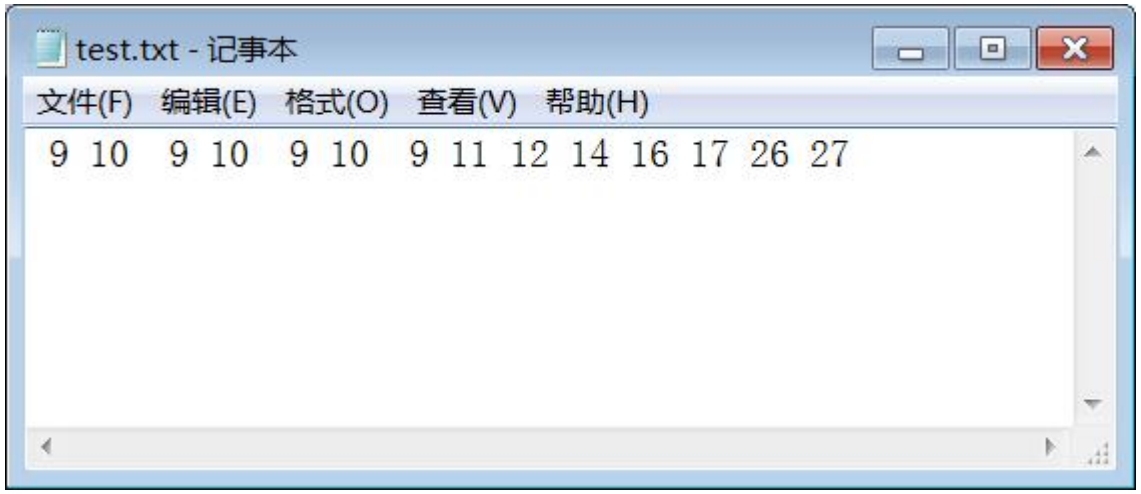
程序运行后得到的输出结果与程序执行路径如下表所示。

测试用例	输出结果	源程序执行路径
test1	1	9→10→9→10→9→10→9→11→12→14→16→17→26→27
test2	2	9→10→9→10→9→10→9→11→12→14→15→26→27
test3	2	9→10→9→10→9→10→9→11→12→21→22→26→27
test4	3	9→10→9→10→9→10→9→11→12→21→23→26→27
test5	5	9→10→9→10→9→10→9→11→12→14→16→26→27
test6	6	9→10→9→10→9→10→9→11→12→21→23→24→26→27
test7	4	9→10→9→10→9→10→9→11→12→14→16→17→26→27

3.3.3 实例：求3个数的中间值

分析输出结果与程序执行路径表中测试用例输出结果会发现，输出结果与测试用例表中的测试用例test7的期望结果不相符。对test7数据及执行过程进行分析，test7的数据为8、4、9，其执行路径为9→10→9→10→9→10→9→11→12→14→16→17→26→27。

```c
#include <stdio.h>
#define LINE() fprintf(__POINT__,"%3d",__LINE__)
FILE *__POINT__;
int main()
{
    if((__POINT__=fopen("test.txt","w"))==NULL)
        fprintf(stderr,"不能打开test.txt文件");
    int i,mid,a[3];
    for(LINE(),i=0;i<3;LINE(),i++)
        LINE(),scanf("%d",&a[i]);
    LINE(),mid=a[2];
    if(LINE(),a[1]<a[2])
    {
        if(LINE(),a[0]<a[1])
            LINE(),mid=a[1];
        else if(LINE(),a[0]<a[2])
            LINE(),mid=a[1];
    }
    else
    {
        if(LINE(),a[0]>a[1])
            LINE(),mid=a[1];
        else if(LINE(),a[0]>a[2])
            LINE(),mid=a[0];
    }
    LINE(),printf("中间值是：%d\n",mid);
    LINE(),fclose(__POINT__);
    return 0;
}
```



3.3.3 实例：求3个数的中间值

除了逻辑错误，源程序还会将程序执行的路径写入test.txt文件中，此时会覆盖test.txt文件中原有的数据，这样在查看test.txt文件时只能看到最近一次的执行过程，这违背了测试可溯源的原则。在修改代码逻辑错误时，同时修改test.txt的写入方式为追加写入，修改后的代码如下。

```
#include <stdio.h>
#define LINE() fprintf(__POINT__,"%3d",__LINE__)
FILE *__POINT__;
int i,mid,a[3];
int main()
{
    if((__PROBE__=fopen("test.txt","a+"))==NULL)
        fprintf(stderr,"不能打开test.txt文件");
    for(LINE(),i=0;i<3;LINE(),i++)
        LINE(),scanf("%d",&a[i]);
    LINE(),mid=a[2];
    if(LINE(),a[1]<a[2])
    {
        if(LINE(),a[0]<a[1])
            LINE(),mid=a[1];
        else if(LINE(),a[0]<a[2])
            if(a[0]<a[1])
                LINE(),mid=a[1] ;
            else
                mid=a[0] ;
    }
    else
    {
        if(LINE(),a[0]>a[1])
            LINE(),mid=a[1];
        else if(LINE(),a[0]>a[2])
            LINE(),mid=a[0];
    }
    LINE(),printf("中间值是：%d\n",mid);
    fprintf(__POINT__,"\n");
    fclose(__POINT__);
    return 0;
}
```

本章小结

本章讲解了白盒测试方法中的基本路径法、逻辑覆盖法和程序插桩法。对于基本路径法，需要掌握绘制控制流图和计算圈复杂度的方法。逻辑覆盖法包含语句覆盖、判定覆盖、条件覆盖、判定-条件覆盖、条件组合覆盖，读者需要掌握这些方法以及它们之间的差别，在实际测试中选择合适的方法进行测试。程序插桩法包含目标代码插桩和源代码插桩，使用源代码插桩并设置合理的探针有助于在程序开发中查找逻辑错误。通过本章的学习，读者应能够掌握白盒测试的方法。

黑盒测试和白盒测试的对比

对比维度	黑盒测试（Black-Box Testing）	白盒测试（White-Box Testing）
测试核心依据	仅依赖软件的**需求规格说明书 / 用户手册**，不关注内部逻辑	依赖软件的**源代码、数据结构、算法、控制流程**等内部信息
测试视角	站在**用户角度**，验证软件 “输入→输出” 是否符合预期	站在**开发 / 测试角度**，验证代码逻辑执行是否正确
常用测试方法	等价类划分法、边界值分析法、因果图法、错误推测法	逻辑覆盖法（语句 / 判定 / 条件覆盖等）、基本路径测试法
人员能力要求	无需懂编程，需熟悉需求和用户场景	需具备编程能力，能读懂代码、分析逻辑结构
可发现的问题	功能缺失、输入输出异常、UI 交互问题、兼容性问题	代码逻辑错误（如死循环）、变量未初始化、内存泄漏、安全漏洞
适用测试阶段	集成测试、系统测试、验收测试（后期阶段）	单元测试、代码审查（前期阶段）
优点	1. 不依赖源码，适合闭源软件； 2. 贴近用户实际使用场景； 3. 测试用例设计简单	1. 能深入定位代码级问题； 2. 测试覆盖更全面（如分支、路径）； 3. 提前发现底层风险
缺点	1. 无法检测内部逻辑错误； 2. 需求不完整时测试不全面； 3. 可能遗漏复杂逻辑组合问题	1. 依赖源码，闭源软件无法使用； 2. 测试成本高（需懂代码）； 3. 代码变更后测试用例需频繁更新

谢谢