第3章

软件测试方法

按测试过程是否在实际应用环境中运行分类，可以将传统的测试技术分为静态测试和动态测试。静态测试技术是单元测试中的重要手段之一，测试对象可以是需求文件、设计文件或源程序等，适用于新开发的和重用的代码，通常在代码完成并无错误地通过编译或汇编后进行。动态测试是指通过运行程序发现错误，通过观察代码运行过程获取系统的各方面信息，并从中发现缺陷。所有的黑盒测试和绝大多数的白盒测试都可以算作动态测试。

本章要点

- 静态测试的内容及方法
- 动态测试的内容及方法
- 主动测试与被动测试
- 白盒测试的内容及方法
- 黑盒测试的内容及方法
- 不同黑盒测试方法的优缺点和应用场合
- 白盒测试与黑盒测试的对比

3.1 静态测试

静态测试不执行被分析的程序，而是通过对模块源代码进行研读，找出其中的错误或可疑之处，收集一些度量数据。静态测试包括对软件产品的需求和设计规格说明书的评审、对程序代码的复审等。静态测试的查错和分析功能是其他方法所不能替代的，可以采用人工或计算机辅助静态测试手段进行检测。

人工检测指的是完全靠人工审查或评审软件，偏重于编码风格、质量的检验。除了审查编码，还要对各阶段的软件产品进行检验。这种方法可以有效地发现逻辑设计和编码错误，发现计算机辅助静态测试不易发现的问题。

计算机辅助静态测试是利用静态测试工具对被测程序进行特性分析，从程序中提取一

些信息，以便检查程序逻辑的各种缺陷和可疑的程序构造，如用错的局部变量和全局变量、不匹配的参数、潜在的死循环等。静态测试中还可以用符号代替数值求得程序结果，以便对程序进行运算规律的检验。

3.1.1　代码检查

代码检查包括桌面检查、代码审查和走查等。它主要检查代码和设计的一致性、代码对标准的遵循、可读性、代码逻辑表达正确性、代码结构合理性等方面；发现程序中不安全、不明确和模糊部分，找出程序中不可移植部分；发现违背程序编写风格问题。代码检查包括变量检查、命名和类型审查、程序逻辑审查、程序语法检查和程序结构检查等内容。

代码检查应该在编译和动态测试之前进行。在检查前，应准备好需求描述文档、程序设计文档、程序的源代码清单、代码编写标准和代码错误检查表等。

在实际使用中，代码检查法能快速找到缺陷，发现30%～70%的逻辑设计和编码缺陷，而且代码检查法看到的是问题本身而非征兆。但是，代码检查法非常耗费时间，并且需要经验和知识的积累。

代码检查法可以使用测试软件进行自动化测试，以提高测试效率，降低劳动强度；或者使用人工进行测试，以充分发挥人力的逻辑思维能力。

1. 桌面检查

桌面检查是一种传统的检查方法，由程序员自己检查编写的程序。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，目的是发现程序中的错误。

由于程序员熟悉自己的程序和程序设计风格，程序员自己进行桌面检查可以节省很多检查时间，但应避免主观片面性，因为人们一般不能有效地测试自己编写的程序。桌面检查需要首先运行拼写检查器、语法检查器、句法检查器等进行字面检查，现在大多数集成开发环境集成了这些相应的工具，帮助程序员在编写代码的同时就注意这些可能存在的缺陷。然后，程序员就可以慢慢地复审文档，寻找文档中的不一致、不完全和漏掉信息的地方。在这个过程中所检测到的问题，应该由程序员自己直接修改，这当中可能会需要项目管理者或项目中其他专家提供建议。一旦所有的改正工作完成，就应该重新运行前面所说的桌面检查，发现并修改所有由于修改内容而引起的新的拼写、语法、标点错误。

图3-1所示为一段未经桌面检查的源代码，由集成开发环境进行了初步的检查，并指出了基本的拼写、语法、标点错误。

```
28    public Int getCount() {
29        return count;
30    }
31
32    public void setCount(int count) {
33        this.count = count
34    }
35
36    public String getNumber() {
37        return number;
38    }
39
40    public void setNumber(String number) {
41        that.number = number;
42    }
```

图3-1　桌面检查案例

（1）第28行：返回数据类型应该为int，写成了Int。

（2）第33行：缺少标点符号“;”。

（3）第37行：返回的关键字“return”拼写错误，写成了“returm”。

（4）第41行：关键字“this”写成了“that”。

在利用集成开发环境进行编码测试时，大多数工具都会提供桌面检查的工具，若使用文

本编辑器进行代码编辑，那么就需要开发人员仔细检查自己编写的代码。

2. 代码审查

代码审查是由若干程序员和测试人员组成一个审查小组，通过阅读、讨论和争议，对程序进行静态分析的过程。代码审查分为以下两步。

(1) 小组负责人提前把设计规格说明书、控制流程图、程序文本以及有关要求、规范等分发给小组成员，作为审查的依据。

(2) 小组成员在充分阅读这些材料后，召开程序审查会，在会上首先由程序员逐行讲解程序逻辑，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。

实践表明，程序员在讲解过程中能够发现许多自己以前没有发现的错误，而讨论和争议则促进问题暴露。

在会前应该给每个小组成员准备一份常见错误清单，把以往发现的常见错误罗列出来，供与会者对照检查，以提高审查效率。这份常见错误清单也称为检查表，它把程序中可能出现的各种错误进行分类，对每类列举出尽可能多的典型错误，然后把它们列成表格，供再审查时使用。

表 3-1 所示为一张常规的 Java 代码审查检查表，测试小组人员根据该表格中的激活项逐一审查被测程序，并可根据级别对被测代码进行评审。

表 3-1 Java 代码审查检查表

重要性	激活	结果	检 查 项
总计			
命名			
重要			命名规则是否与所采用的规范保持一致
			是否遵循了最小长度最多信息原则
重要			has/can/is 前缀的函数是否返回布尔型
注释			
重要			注释是否较清晰且必要
重要	Y		复杂的分支流程是否已经被注释
			距离较远的}是否已经被注释
			非通用变量是否全部被注释
重要	Y		函数是否已经有文档注释(功能、输入、返回及其他可选)
			特殊用法是否被注释
声明、空白、缩进			
			每行是否只声明了一个变量(特别是那些可能出错的类型)
重要			变量是否已经在定义的同时初始化
重要			类属性是否都执行了初始化
			代码段落是否被合适地以空行分隔
	Y		是否合理地使用了空格使程序更清晰
			代码行长度是否在要求之内
			折行是否恰当
语句/功能的分布/规模			
			包含复合语句的{}是否成对出现并符合规范
			是否给单个的循环、条件语句也加了{}

续表

重要性	激活	结果	检　查　项
			if/if-else/if-else if-else/do-while/switch-case 语句的格式是否符合规范
			单个变量是否只作单个用途
重要			单行是否只有单个功能(不要使用；进行多行合并)
重要			单个函数是否执行了单个功能并与其命名相符
	Y		＋＋和－－操作符的应用是否复合规范
规模			
重要			单个函数不超过规定行数
重要			缩进层数是否不超过规定
重要			是否已经消除了所有警告
重要	Y		常数变量是否声明为 final
重要			对象使用前是否进行了检查
重要			局部对象变量使用后是否被复位为 NULL
重要			对数组的访问是不是安全的(合法的 index 取值为[0,MAX_SIZE-1])
重要			是否确认没有同名变量局部重复定义问题
			程序中是否只使用了简单的表达式
重要	Y		是否已经用()使操作符优先级明确化
重要	Y		所有判断是否都使用了(常量＝＝变量)的形式
			是否消除了流程悬挂
重要			是否每个 if-else if-else 语句都有最后一个 else 以确保处理了全集
重要			是否每个 switch-case 语句都有最后一个 default 以确保处理了全集
			for 循环是否都使用了包含下限不包含上限的形式(k＝0；k＜MAX)
重要			XML 标记书写是否完整,字符串的拼写是否正确
			对于流操作代码的异常捕获是否有 finally 操作以关闭流对象
			退出代码段时是否对临时对象做了释放处理
重要			对浮点数值的相等判断是否是恰当的(严禁使用＝＝直接判断)
可靠性(函数)			
重要	Y		入口对象是否都被判断不为空
重要	Y		入口数据的合法范围是否都被判断(尤其是数组)
重要	Y		是否对有异常抛出的方法都执行了 try…catch 保护
重要	Y		是否函数的所有分支都有返回值
重要			int 的返回值是否合理(负值为失败,非负值成功)
			对于反复进行的 int 返回值判断,是否定义了函数来处理
			关键代码是否做了捕获异常处理
重要			是否确保函数返回 CORBA 对象的任何一个属性都不能为 NULL
重要			是否对方法返回值对象做了 NULL 检查,该返回值定义时是否被初始化
重要			是否对同步对象的遍历访问做了代码同步
重要			是否确认在对 Map 对象使用迭代遍历过程中没有做增减元素操作
重要			线程处理函数循环内部是否有异常捕获处理,防止线程抛出异常而退出
			原子操作代码异常中断,使用的相关外部变量是否恢复先前状态
重要			函数对错误的处理是恰当的

续表

重要性	激活	结果	检 查 项
可维护性			
重要			实现代码中是否消除了直接常量(用于计数起点的简单常数例外)
			是否消除了导致结构模糊的连续赋值(如 a=(b=d+c))
			是否每个 return 前都要有日志记录
			是否有冗余判断语句(如 if(b)return true; else return false;)
			是否把方法中的重复代码抽象成私有函数

3. 走查

走查与代码审查基本相同,其过程分为以下两步。

(1) 把材料先发给走查小组的每个成员,认真研究程序,然后开会。

(2) 开会的程序与代码审查不同,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所测程序准备一批有代表性的测试用例,提交给走查小组,走查小组开会扮演计算机角色,让测试用例沿程序逻辑运行一遍,随时记录程序的跟踪,供分析和讨论用。

与会者借助测试用例媒介的作用,对程序逻辑和功能提出各种疑问,结合问题展开讨论和争议,能够发现更多问题。

以 Java 代码的走查与审查为例,以下常见错误需要在走查和审查中多加注意。

1) 多次复制字符串

测试所不能发现的一个错误是生成不可变(Immutable)对象的多份副本。不可变对象是不可改变的,因此不需要复制它。最常用的不可变对象是 String。如果必须改变一个 String 对象的内容,应该使用 StringBuffer。下面的代码可以正常工作。

```
String s = new String ("Text here");
```

但是,这段代码性能差,而且没有必要这么复杂。可以用以下方式重写代码。

```
String temp = "Text here";
String s = new String (temp);
```

但是,这段代码包含额外的 String。更好的代码为

```
String s = "Text here";
```

2) 没有复制返回的对象

封装(Encapsulation)是面向对象编程的重要概念。但是 Java 为打破这种封装提供了方便——Java 允许返回私有数据的引用(Reference)。下面的代码揭示了这一点。

```
import java.awt.Dimension;
/***Example class.The x and y values should never*be negative.*/
public class Example{
       private Dimension d = new Dimension (0, 0);
```

```
        public Example (){ }
        /*** Set height and width. Both height and width must be
        nonnegative * or an exception is thrown.*/
        public synchronized void setValues (int height,int width)
        throws IllegalArgumentException{
                if (height < 0 || width < 0)
                        throw new IllegalArgumentException();
                d.height = height;
                d.width = width;
        }
        public synchronized Dimension getValues(){
                // Ooops! Breaks encapsulation
                return d;
        }
}
```

Example 类保证了它所存储的 height 和 width 值永远为非负数，试图使用 setValues() 方法设置负值会触发异常。然而，getValues()方法返回 d 的引用，而不是 d 的副本，可以编写以下破坏性代码。

```
Example ex = new Example();
Dimension d = ex.getValues();
d.height = -5;
d.width = -10;
```

现在，Example 对象拥有负值。如果 getValues()方法的调用者永远也不设置返回的 Dimension 对象的 width 和 height 值，那么仅凭测试是不可能检测到这类错误的。随着时间的推移，客户代码可能会改变返回的 Dimension 对象的值，这时，追寻错误的根源是一件枯燥且费时的事情，尤其是在多线程环境中。更好的方式是让 getValues()方法返回副本，代码如下。

```
public synchronized Dimension getValues(){
        return new Dimension (d.x, d.y);
}
```

现在，Example 对象的内部状态就安全了。调用者可以根据需要改变它所得到的副本的状态，但是要修改 Example 对象的内部状态，必须通过 setValues()方法才可以。

3）不必要的复制

我们知道，get()方法应该返回内部数据对象的副本，而不是引用。但是也有例外的情况。

```
/*** Example class.The value should never * be negative.*/
public class Example{
     private Integer i = new Integer (0);
     public Example (){ }

     /*** Set x. x must be nonnegative* or an exception will be thrown*/
     public synchronized void setValues (int x) throws
     IllegalArgumentException{
          if (x < 0)
               throw new IllegalArgumentException();
          i = new Integer (x);
     }
```

```
    public synchronized Integer getValue(){
        // We can't clone Integers so we makea copy this way.
        return new Integer (i.intValue());
    }
}
```

上面这段代码是安全的，但是做了多余的工作。Integer 对象就像 String 对象，一旦被创建就是不可变的。因此，返回内部 Integer 对象，而不是它的副本，也是安全的。getValue()方法应该写为

```
public synchronized Integer getValue(){
      return i;
}
```

4）自编代码复制数组

Java 允许复制数组，但是开发者通常会自己编写以下代码。

```
public class Example{
    private int[] copy;
    /*** Save a copy of 'data'. 'data' cannot be null.*/
    public void saveCopy (int[] data){
        copy = new int[data.length];
        for (int i = 0; i < copy.length; ++i)
          copy[i] = data[i];
    }
}
```

这段代码是正确的，却引入了不必要的复杂性。saveCopy()方法的一个更好的实现如下。

```
void saveCopy (int[] data){
  try{
      copy = (int[])data.clone();
  }catch (CloneNotSupportedException e){
      // Can't get here.
  }
}
```

5）复制错误的数据

有时候程序员知道必须返回一个副本，却不小心复制了错误的数据。由于仅做了部分的数据复制工作，下面的代码与程序员的意图有偏差。

```
import java.awt.Dimension;
/*** Example class. The height and width values should never * be negative. */
public class Example{
      static final public int TOTAL_VALUES = 10;
      private Dimension[] d = new Dimension[TOTAL_VALUES];
      public Example (){ }

      /*** Set height and width. Both height and width must be
      nonnegative * or an exception will be thrown. */
      public synchronized void setValues (int index, int height, int
      width) throws IllegalArgumentException{
            if (height < 0 || width < 0)
                  throw new IllegalArgumentException();
            if (d[index] == null)
            d[index] = new Dimension();
```

```
            d[index].height = height;
            d[index].width = width;
        }
        public synchronized Dimension[] getValues()
        throws CloneNotSupportedException{
            return (Dimension[])d.clone();
        }
}
```

这里的问题在于 getValues()方法仅复制了数组，而没有复制数组中包含的 Dimension 对象，因此，虽然调用者无法改变内部的数组使其元素指向不同的 Dimension 对象，但是调用者却可以改变内部的数组元素（也就是 Dimension 对象）的内容。getValues()方法的更好版本为

```
public synchronized Dimension[] getValues() throws CloneNotSupportedException{
        Dimension[] copy = (Dimension[])d.clone();
        for (int i = 0; i < copy.length; ++i){
            // NOTE: Dimension isn't cloneable.
            if (d != null) copy[i] = new Dimension (d[i].height, d[i].width);
        }
        return copy;
}
```

6）检查 new 操作的结果是否为 NULL

Java 编程新手有时候会检查 new 操作的结果是否为 NULL。可能的检查代码为

```
Integer i = new Integer (400);
if (i == null) throw new NullPointerException();
```

检查虽然没有错误，却没有必要。C/C++程序员在开始写 Java 程序的时候常常会这么做，这是由于检查 C/C++中 malloc()函数的返回结果是必要的，不这样做就可能产生错误。但在 Java 中，new 操作不允许返回 NULL，如果真的返回 NULL，很可能是虚拟机崩溃了，这时即便检查返回结果也无济于事。

7）用＝＝替代 equals()方法

在 Java 中，有两种方式检查两个数据是否相等：使用＝＝操作符或使用所有对象都实现的 equals()方法。原子类型（int、float、char 等）不是对象，因此它们只能使用＝＝操作符，如下所示。

```
int x = 4;
int y = 5;
if (x == y) System.out.println ("Hi");
// This 'if' test won't compile.
if (x.equals (y)) System.out.println ("Hi");
```

对象更复杂些，＝＝操作符检查两个引用是否指向同一个对象，而 equals()方法则实现更专门的相等性检查。而 java.lang.Object 所提供的默认的 equals()方法的实现使用＝＝简单地判断被比较的两个对象是否为同一个。许多类覆盖了默认的 equals()方法，如 String 类，它的 equals()方法检查两个 String 对象是否包含同样的字符串，而 Integer 类的 equals()方法检查所包含的 int 值是否相等。大部分时候，在检查两个对象是否相等时应该使用 equals()方法，而对于原子类型的数据，则应该使用＝＝操作符。

8）混淆原子操作和非原子操作

Java 保证读和写 32 位数或更小的值是原子操作，也就是说可以一步完成，因而不可能被打断，因此这样的读和写不需要同步。以下代码是线程安全的。

```
public class Example{
       private int value; // More code here...
       public void set (int x){
              // NOTE: No synchronized keyword
              this.value = x;
       }
}
```

不过，这个保证仅限于读和写。以下代码不是线程安全的。

```
public void increment (){
       // This is effectively two or three instructions:
       // 1) Read current setting of 'value'.
       // 2) Increment that setting.
       // 3) Write the new setting back.
       ++this.value;
}
```

在测试的时候，可能不会捕获到这个错误。首先，测试与线程有关的错误是很难的，而且很耗时间。其次，在有些机器上，这些代码可能会被翻译成一条指令，因此工作正常，只有在其他虚拟机上测试时这个错误才可能显现。因此，最好在开始时就正确地同步代码。

```
public synchronized void increment (){
       ++this.value;
}
```

9）在 catch 块中做清除工作

一段在 catch 块中做清除工作的代码如下所示。

```
OutputStream os = null;
try{
       os = new OutputStream ();
       // Do something with os here.
       os.close();
}catch (Exception e){
       if (os != null)
       os.close();
}
```

尽管这段代码在几个方面都是有问题的，但是在测试中很容易漏掉这个错误。下面列出了这段代码存在的 3 个问题：(1)os.close()语句在两处出现，多此一举，而且会带来维护方面的麻烦；(2)上述代码仅处理了 Exception，而没有涉及 Error，但是当 try 块运行出现了 Error，流也应该被关闭；(3)close()方法可能会抛出异常。上述代码的一个更优版本为

```
OutputStream os = null;
try{
       os = new OutputStream ();
       // Do something with os here.
}finally{
       if (os != null)
```

```
        os.close();
}
```

这个版本消除了上面所提到的两个问题：代码不再重复，Error 也可以被正确处理了，但是没有很好地处理第 3 个问题。最好的方法是把 close()语句单独放在一个 try/catch 块中。

10）增加不必要的 catch 块

一些开发者听到 try/catch 块这个名字后，就会想当然地以为所有的 try 块必须要有与之匹配的 catch 块。C/C++程序员尤其会这样想，因为在 C/C++中不存在 finally 块的概念，而且 try 块存在的唯一理由只不过是为了与 catch 块相配对。增加不必要的 catch 块的代码就会出现以下情况，捕获到的异常又立即被抛出。

```
try{
        // Nifty code here
}catch(Exception e){
        throw e;
}finally{
        // Cleanup code here
}
```

不必要的 catch 块被删除后，代码就缩短为

```
try{
        // Nifty code here
}finally{
        // Cleanup code here
}
```

11）没有正确实现 equals()、hashCode()或 clone()等方法

equals()、hashCode()和 clone()方法是由 java.lang.Object 提供的默认实现。但是，这些默认实现在大部分时候毫无用处，因此许多类覆盖其中的若干个方法以提供更有用的功能。而当继承一个覆盖了若干个这些方法的父类时，子类通常也需要覆盖这些方法。在进行代码审查时，应该确保如果父类实现了 equals()、hashCode()或 clone()等方法，那么子类也必须正确地实现这些方法。

4. 代码检查常用检查项

通过代码检查法可以获得软件组成的重要基本因素，如变量标识符、过程标识符、常量等，组合这些基本因素就可以得到软件的基本信息。

（1）标号交叉引用表：列出在各模块出现的全部标号，在表中标出标号的属性，包括已说明、未说明、已使用、未使用，表中还包括在模块以外的全局标号、计算标号等。

（2）变量交叉引用表：在表中应标明各变量的属性，包括已说明、未说明、隐式说明、类型及使用情况，进一步还可以区分是否出现在赋值语句的右边，以及是否属于普通变量、全局变量或特权变量等。

（3）子程序、宏和函数表：在表中列出各个子程序、宏和函数的属性，包括已定义、未定义和定义类型。

（4）参数表：输入参数个数、顺序、类型，输出参数个数、顺序、类型，已引用、未引用、引用次数等。

(5) 等价表：列出在等价语句或等值语句中出现的全部变量和符号。

(6) 常数表：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。

通过这些软件的基本信息可以实现以下功能。

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

(2) 为用户提供辅助信息，如子程序、宏和函数表、等价表和常数表。

(3) 用来做错误预测和程序复杂度的计算，如操作符合操作数表等。

代码检查项目包括检查变量的交叉引用表，检查标号的交叉引用表，检查子程序、宏和函数表，等价性检查，标准检查，风格检查，比较控制流，选择、激活路径，对照程序说明，充分文档等。

(1) 检查变量的交叉引用表重点检查未说明变量和违反了类型规定的变量，还要对照源程序，逐个检查变量的引用、变量的使用序列、临时变量在某条路径上的重写情况，局部变量、全局变量与特权变量的使用。

(2) 检查标号的交叉引用表验证所有标号的正确性，检查所有标号的命名是否正确，转向指定位置的标号是否正确。

(3) 检查子程序、宏和函数表，调用每次调用和所调用位置是否正确，确定每次调用的子程序、宏和函数是否存在，检验调用序列中调用方式与参数顺序、个数、类型上的一致性。

(4) 等价性检查，检查所有等价变量类型的一致性，解释所包含的类型差异。

(5) 标准检查，用标准检查工具软件或手工检查程序中违反标准的问题。

(6) 风格检查，检查发现程序在设计风格方面的问题。

(7) 比较控制流，比较由程序员设计的控制流图和由程序生成的实际控制流图，寻找和解释每个差异，修改文档并修正错误。

(8) 选择、激活路径，在程序员设计的控制流图上选择路径，再到实际控制流图上激活这条路径，如果选择的路径在实际控制流图上不能被激活，则源程序可能存在错误。

(9) 对照程序说明，阅读程序源代码，逐行进行分析思考，比较实际的代码和期望的代码，从它们的差异中发现程序的错误和问题。

(10) 充分文档，代码检查的文档是一种过渡性文档，不是公开的正式文档，通过编写文档，也是对程序的一种下意识的检查和测试，可以帮助程序员发现更多的错误，管理部门也可以通过检查文档，了解模块质量、完全性、测试方法和程序员能力。

3.1.2 静态结构分析

静态结构分析主要是以图的形式表现程序的内部结构，供测试人员对程序结构进行分析。程序结构形式是白盒测试的主要依据。研究表明，程序员 38％的时间花费在理解软件系统上，因为代码以文本格式被写入多重文件中，这是很难阅读理解的，需要其他一些东西帮助人们阅读理解，如各种图表等，而静态结构分析满足了这样的需求。

静态结构分析是一种对代码机械性的、程式化的特性进行分析的方法。在静态结构分析中，测试者通过使用测试工具分析程序源代码的系统结构、数据接口、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、内部文件调用关系图、子程序表、宏和函数参

数表等各类图形图表，可以清晰地标识整个软件系统的组成结构，使其便于阅读和理解，然后可以通过分析这些图表，检查软件有没有存在缺陷或错误。静态结构分析包括控制流分析、数据流分析、信息流分析、接口分析、表达式分析等。

常用的关系图主要有函数调用关系图和模块控制流图。

函数调用关系图列出所有函数，用连线表示调用关系，通过应用程序各函数之间的调用关系展示系统的结构。利用函数调用关系图可以检查函数的调用关系是否正确，是否存在孤立的函数而没有被调用，明确函数被调用的频繁度，对调用频繁的函数可以重点检查。通过查看函数调用关系图，可以发现系统是否存在结构缺陷，发现哪些函数是重要的，哪些是次要的，需要使用什么级别的覆盖要求等。

模块控制流图是由许多节点和连接节点的边组成的图形，其中每个节点代表一条或多条语句，边表示控制流向，模块控制流图可以直观地反映出一个函数的内部结构，通过检查这些模块控制流图可以很快地发现软件错误与缺陷。

模块控制流图图元符号如图 3-2 所示。

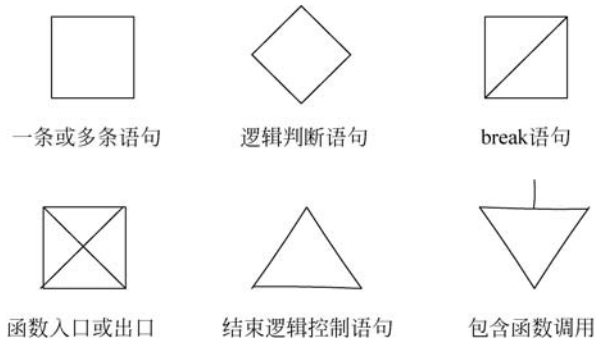

图 3-2　模块控制流图图元符号

用模块控制流图表示的基本程序结构如图 3-3 所示。

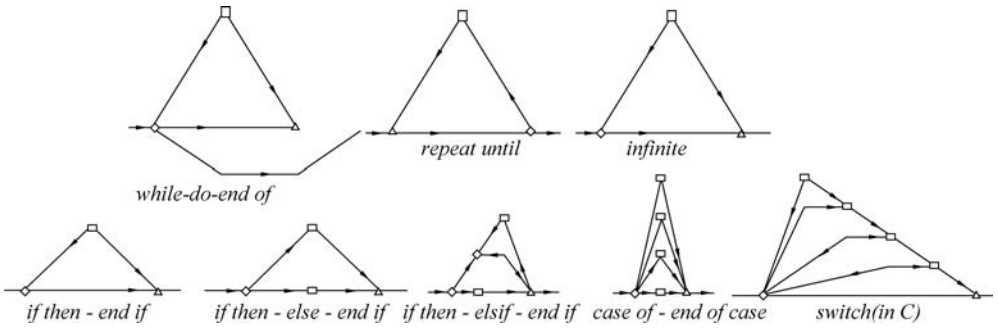

图 3-3　模块控制流图基本结构

静态错误分析用于确认源程序中是否存在某类错误的危险结构，有以下几种形式。

1. 类型和单元分析

为了强化在源程序中数据类型的检查，在程序设计语言中扩展了一些新的数据类型，如仅能在数组中使用的下标类型和在循环语句中当作控制变量使用的计数器类型，这样就可以静态预处理程序，分析程序中的类型错误。

2. 引用分析

在静态错误分析中，最广泛使用的技术就是发现引用异常。如果沿着程序的控制路径，变量在赋值以前被引用，或变量在赋值以后未被引用，这时就发生引用异常。

为了检测引用异常，需要检查通过程序的每条路径。通常采用类似深度优先的方法遍历程序流程图的每条路径，也可以建立引用异常的探测工具，这类工具包含两个表：定义表和未引用表。每张表都包含一组变量表。未引用表包含已被赋值但未必引用的一些变量。

当扫描达到一个出度大于 1 的节点 V 时，深度优先探测算法要求先检查最左分支的那部分程序流程图，然后再检查其他分支。在最左分支检查完之前，应把定义表和未引用表的当前内容用一个栈暂时存储起来，当最左分支检查完之后，算法控制返回到节点 V，从栈中恢复该节点的定义表和未引用表的旧的副表，然后再去遍历该节点的下一个分支，这个过程要持续到全部分支检查完为止。

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：

(1) 在表达式中不正确地使用括号造成的错误；

(2) 数组下标越界造成的错误；

(3) 除数为零造成的错误；

(4) 对负数开平方或对 π 求正切造成的错误。

最复杂的一类表达式分析是对浮点数计算造成的误差的检查。由于使用二进制数不能精确地表示十进制浮点数，常常使计算结果出乎意料。

4. 接口分析

接口分析是程序的静态错误分析和设计分析共同研究的问题。接口一致性的设计可以分析检查模块之间接口的一致性和外部数据库之间接口的一致性。

程序关于接口的静态错误分析检查过程与实参在类型、函数过程接口之间具有一致性，因此要检查形参与实参在类型、数量、维数、顺序、使用上的一致性；以及检查全局变量和公共数据区在使用上的一致性。

在“猜数字游戏”案例项目中，导出其函数调用关系图，如图 3-4 所示。

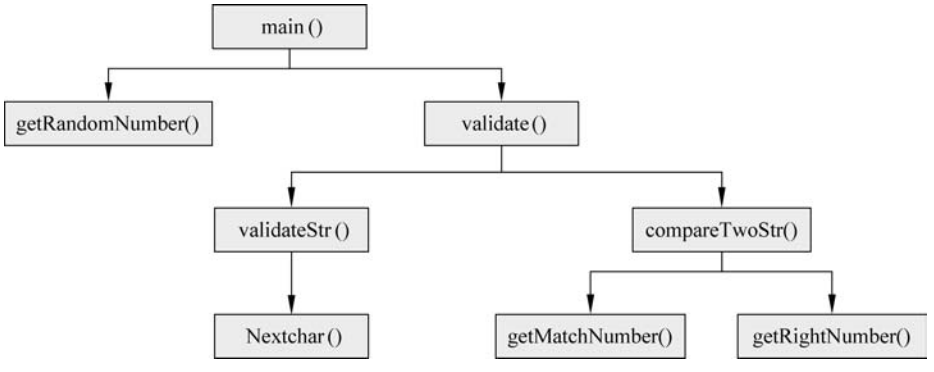

图 3-4　函数调用关系图

从该函数调用关系图中可以得到以下信息。

(1) 函数之间的调用关系符合设计规格说明书的要求；

(2) 不存在递归调用；

(3) 调用层次最深为 4 层；

(4) 不存在独立的和没有被调用的函数；

(5) 比较重要的函数有 validate()、compareTwoStr()、validateStr()等。

也可以对每个函数使用模块控制流图，考查每个函数的结构是否合理、是否存在错误等。

3.2　动态测试

动态测试是软件测试中使用最普遍的方法，通过运行程序发现错误，通过观察代码运行过程获取系统行为、变量实时结果、内存、堆栈、线程以及测试覆盖率等各方面的信息，从而判断系统是否存在问题，或者通过有效的测试用例、对应的输入输出关系分析被测程序的运行情况，从中发现缺陷。

3.2.1　主动测试

在软件测试中，比较常见的是主动测试方法，测试人员主动向被测试对象发送请求，或借助数据、事件驱动被测试对象的行为，验证被测试对象的反应或输出结果。如图 3-5(a)所示，在主动测试中，测试人员和被测试对象之间发生直接相互作用，而且被测试对象完全受测试人员的控制，被测试对象处于测试状态，而不是实际工作状态。

3.2.2　被动测试

由于主动测试中被测试对象受人为因素影响较大，而且一般是在测试环境中进行，而非软件产品的实际运行环境，所以主动测试不适用于产品的在线测试。为了实现产品在线测试，这就需要用到被动测试。在被动测试中，软件产品在实际环境中运行，测试人员被动地监控产品的运行，通过一定的机制获取系统运行的数据，包括输入、输出数据，如图 3-5(b)所示。被动测试适合性能测试和在线监控，在嵌入式系统测试中常常采用被动测试方法。另外，大规模复杂系统的性能测试，为了节省成本，可以采用这种方法。

在主动测试中，测试人员需要设计测试用例、设法输入各种数据；而在被动测试中，系统运行过程中的各种数据自然生成，测试人员不需要设计测试用例，只要设法获取系统运行的各种数据，但是数据的完整性得不到保证。被动测试的关键是建立监控程序，并通过数据分析掌握系统的状态。

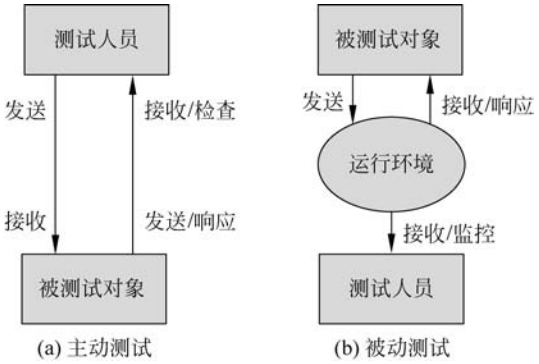

图 3-5　主动测试与被动测试

3.3 白盒测试

白盒测试，有时也称为玻璃盒测试、结构化测试、逻辑驱动测试等，它关注软件产品的内部细节和逻辑结构，即把被测的程序看作一个透明的盒子。白盒测试利用构件层设计的一部分描述的控制结构生成测试用例，需要对系统内部结构和工作原理有一个清楚的了解。白盒测试的准备时间较长，如果要完成覆盖全部程序语句、分支的测试，一般要花费比编程更长的时间。白盒测试对技术的要求较高，测试成本也比较高。

白盒测试的测试方法有程序插桩法、逻辑覆盖法、基本路径法等。

3.3.1 程序插桩法

在调试程序时，常常需要插入一些打印语句，在执行程序时能够打印有关信息，进一步通过这些信息了解程序执行时的一些动态特性，如程序的执行路径或特定变量在特定时刻的取值。这一思想发展出来的程序插桩技术在软件动态测试中作为一种基本的测试手段，有着广泛的应用。

简单来说，程序插桩技术是借助向被测程序中插入操作实现测试目的的方法，即向源程序中添加一些语句，实现对程序语句的执行、变量的变化等情况进行检查。例如，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，或是每个语句的实际执行次数，就可以利用程序插桩技术。

通过插入的语句获取程序执行时的动态信息，这一做法如同在刚研制成机器的特定部位安装记录仪表一样。安装好以后启动机器试运行，除了可以对机器加工的成品进行检验得到机器的运行特性外，还可以通过记录了解其动态特性。这就相当于在运行程序以后，一方面可检测测试的结果数据，另一方面还可以借助插入的语句给出的信息了解程序的执行特性。由于这个原因，有时把插入的语句称为“探测器”，借以实现探测和监控的功能。

在程序的特定部位插入记录动态特性的语句，最终是为了把程序执行过程中发生的一些重要历史事件记录下来。设计插桩程序时需要考虑的问题如下。

(1) 需要探测哪些信息。

(2) 在程序的什么部分设置探测点。

(3) 需要设置多少个探测点。

其中第 1 个问题需要结合具体情况解决，并不能给出笼统的回答。

关于第 2 个问题，在实际测试中通常在以下一些部位设置探测点。

(1) 程序块的第 1 个可执行语句之前。

(2) for、do、do while、do until 等循环语句处。

(3) if、else if、else、end if 等条件语句各分支处。

(4) 输入或输出语句之后。

(5) 函数、过程、子程序调用语句之后。

(6) return 语句之后。

(7) goto 语句之后。

关于第 3 个问题，原则是需要考虑如何设置最少探测点的方案。一般情况下，在没有分支的程序段中只需要一个计数语句，如果出现了多种控制结构，使整个结构十分复杂，则需要针对程序的控制结构进行具体的分析。

在应用程序插桩技术时，可以在程序的某些部分插入某些用以判断变量特性的语句，使程序执行时这些语句得以证实，从而使程序运行特性得以证实，把这些插入的语句称为断言语句。这一方法是程序正确性证明的基本步骤，尽管算不上严格证明，但在实践中十分实用。有些编译系统支持表达式形式的断言语句识别。

在如下的简单代码中，从 main()函数开始执行，main()函数创建对象 A 并通过调用对象 A 的 Go()函数开始进入执行流程。main()函数和 Go()函数代码如下所示。

```
void A::Go()
{
       cout << "input the file name:" << endl;
       cin >> inf;
       if(( psin = fopen( inf, "r")) == NULL)
       {
              cout << "can't open file." << endl;
              return;
       }
       cout << "input a file name:" << endl;
       cin >> outf;
       cout << "running" << endl;
       int error;
       error=judge();
       if(error)
              cout << "errors found when judging." << endl;
       else
       {
              psout=fopen(outf,"w");
              Create ();
              cout << "succeed. executing." << endl;
              Run();
              fclose(psout);
       }
       fclose(psin);
}

int main()
{
       A a;                                          //声明A类实例
       a.Go();                                       //运行实例

       //将程序暂停，查看结果
       cout<<"press any character to exit"<<endl;
       char mh;
       cin >> mh;
       return 0;
}
```

为了使程序在运行时更清晰地表现其运行状态，可以对上面两个函数插入一些打印语句。

在 main()函数的入口插入如下语句，标识整个程序开始执行。

```
cout <<"主程序开始运行"<< endl;
```

在 Go()函数的入口插入如下语句,标识创建对象 A 成功并已开始运行 Go()函数。

```
cout <<"开始运行 A 程序"<< endl;
```

在 Go()函数的 judge()语句前插入如下语句。

```
cout <<"开始判断"<< endl;
```

并在 Go()函数的 judge ()语句后插入如下语句,标识判断部分的运行和返回状态。

```
cout <<"判断结束"<< endl;
```

在 Go()函数的 Create ()语句前插入如下语句。

```
cout <<"开始创建"<< endl;
```

并在 Go()函数的 Create()语句后插入如下语句,标识 Create ()函数的开始与结束执行。

```
cout <<"结束创建"<< endl;
```

在 Go()函数的 Run()语句前插入如下语句。

```
cout <<"开始运行"<< endl;
```

并在 Go()函数的 Run()语句后插入如下语句,标识 Run()函数的开始与结束执行。

```
cout <<"结束运行"<< endl;
```

在 main()函数的 a.Go()语句后插入如下语句,标识对象 A 的 Go()函数已经得到运行并返回。

```
cout <<"程序已运行结束"<< endl;
```

同样地,也可以为每个子程序和函数在入口处添加打印代码,在某些地方插入断言语句,在有循环语句的部分插入计数器统计循环语句的执行次数等。通过这些输出打印信息、计数器取值信息等表示程序的运行状态、函数的调用与返回、变量取值等,进行程序的追踪、调试与测试。

3.3.2 逻辑覆盖法

逻辑覆盖法也是常用的一类白盒动态测试方法,以程序内部逻辑结构为基础,通过对程序逻辑结构遍历实现程序测试的覆盖。逻辑覆盖法要求测试人员对程序的逻辑结构有清晰的了解。

逻辑覆盖法是一系列测试过程的总称,这组测试过程逐渐进行越来越完整的通路测试。

根据覆盖源程序语句的详尽程度，可以分为语句覆盖、判定覆盖、条件覆盖、条件判定覆盖、条件组合覆盖和路径覆盖。下面将一一介绍这些覆盖准则。

1. 语句覆盖

语句覆盖指的是代码中所有的语句都至少执行一遍，用于检查测试用例是否有遗漏，如果检查出存在没有执行到的语句，就需要增加测试用例将所有语句覆盖到。语句覆盖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。该测试用例虽然覆盖了可执行语句，但是不能检查判断逻辑是否有问题，因此一般认为语句覆盖是很不充分的一种测试，是最弱的逻辑覆盖准则。语句覆盖可以通过测试覆盖率工具（如 TrueCoverage、PureCoverage 等）来检查。

语句覆盖率是指被执行的语句数占总语句数的百分比。原则上，单元测试中的语句覆盖率必须达到 100%，尤其是对于一些对质量要求较高的软件，如航天航空软件、电信设备软件、医用设备软件等，就要求软件中的所有语句都必须执行到；对于质量要求不高的软件，根据实际项目情况，语句覆盖率一般也应该达到 90%以上，否则单元测试的效果会大打折扣。

考虑下面的程序。

```
if (A>1 and B=0)
X=X/A;
if(A=2 or X>1)
X=X+1;
```

上述代码的程序流程图如图 3-6 所示。

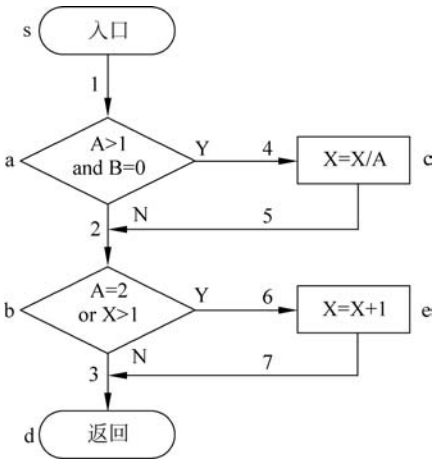

图 3-6　被测试模块的流程图

为了使每个语句都执行一次，程序的执行路径应该是 s-a-c-b-e-d。为此，只需要输入下面的测试数据（实际上 X 可以是任意实数）。

A＝2，B＝0，X＝4

语句覆盖对程序的逻辑覆盖很少。在上面例子中两个判定条件都只测试了条件为真的情况，条件为假时处理有错误，显然不能发现。此外，语句覆盖只关心判定表达式的值，而没有分别测试判定表达式中每个条件取不同值时的情况。在上面的例子中，为了执行 s-a-c-b-

e-d 路径，只需两个判定表达式(A>1)and(B=0)和(A=2)or(X>1)都取真值，因此使用上述一组测试数据就够了。但是，如果程序中把第 1 个判定表达式中的逻辑运算符 and 错写成 or，或把第 2 个判定表达式中的条件 X>1 错写成 X<1，使用上面的测试数据并不能查出这些错误。

综上所述，可以看出语句覆盖是很弱的逻辑覆盖标准。为了更充分地测试程序，可以采用下面的覆盖标准。

2. 判定覆盖

判定覆盖又称为判断覆盖、分支覆盖，是比语句覆盖稍强的覆盖标准。判定覆盖指的是设计足够的测试用例，使每个判断获得每种可能的结果至少一次，即对被测试模块中的每个判断要分别取真和假各一次进行测试。判定覆盖是单元测试中很常用的一类覆盖，利用判定覆盖可以检查测试用例的设计是否完整，但是判定覆盖准则依然不够严格。

对于上面的例子，能够分别覆盖路径 s-a-c-b-e-d 和 s-a-b-d 的两组测试数据，或者可以分别覆盖路径 s-a-c-b-d 和 s-a-b-e-d 的两组测试数据，都满足判定覆盖标准。例如，用下面两组测试数据就可做到完全的判定覆盖。

- A=3，B=0，X=3(覆盖 s-a-c-b-d)
- A=2，B=1，X=1(覆盖 s-a-b-e-d)

判定覆盖比语句覆盖强，但是对程序逻辑的覆盖程度仍然不高，如上述测试数据只覆盖了程序全部路径的一半。

3. 条件覆盖

条件覆盖是指程序中每个判断中的每个条件的所有可能取值至少要都执行一次，条件覆盖独立度量每个子表达式，并对控制流更敏感。但是完全的条件覆盖并不能满足完全地判定覆盖。

考虑下面的程序。

```
if (a && b) {
       return true;
}else {
       return false;
}
```

通过以下两个测试用例可以得到 100%的条件覆盖率。

- a=true，b=false
- a=false，b=true

但上述测试用例条件都不会使 if 的逻辑运算式成立，因此不符合判定覆盖的条件。

4. 条件判定覆盖

既然判定覆盖不一定包含条件覆盖，条件覆盖也不一定包含判定覆盖，自然会提出一种能同时满足这两种覆盖准则的逻辑覆盖，这就是条件判定覆盖。条件判定覆盖是判定覆盖和条件覆盖的组合，指的是设计足够的测试用例，使判定中每个条件的所有可能取值至少出现一次，并且每个判定取到的各种可能的结果也至少出现一次。条件判定覆盖具有两者的简单性并且没有两者的缺点，但是其没有考虑单个判定对整体结果的影响。

下面以一段代码为例，说明条件判定覆盖的测试用例的设计过程。

```
int x, y;
double z;
if(x>0 && y>0)
       z = z / x;
if(x>1 || z>1)
       z = z + 1;
z = y + z;
```

对其设计测试用例的第一步就是绘制出它的程序流程图，如图 3-7 所示。

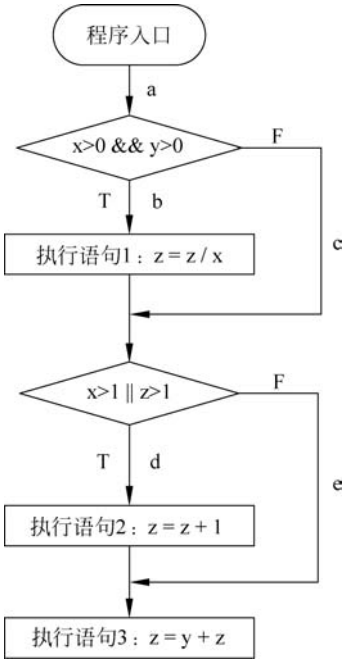

图 3-7　程序流程图

由于条件判定覆盖是条件覆盖与判定覆盖的组合，所以其测试用例取条件覆盖的用例和判定覆盖的用例的并集即可。

条件覆盖的思想是使每个判断的所有逻辑条件的每种可能取值至少执行一次。

对于判断语句 x＞0 && y＞0：条件 x＞0 取真为 T1，取假为-T1；条件 y＞0 取真为 T2，取假为-T2。

对于判断语句 x＞1 || z＞1：条件 x＞1 取真为 T3，取假为-T3；条件 z＞1 取真为 T4，取假为-T4。

设计测试用例如表 3-2 所示。

表 3-2　条件覆盖的测试用例

输　　入	通 过 路 径	条 件 取 值
x＝7，y＝1，z＝3	a-b-d	T1，T2，T3，T4
x＝－1，y＝－3，z＝0	a-c-e	-T1，-T2，-T3，-T4

判定覆盖的思想是使每个判断的取真分支和取假分支至少执行一次。

对于判断语句 x>0 && y>0：取真为 M；取假为-M。

对于判断语句 x>1 || z>1：取真为 N；取假为-N。

设计测试用例，如表 3-3 所示。

综合表 3-2 和表 3-3，条件判定覆盖的测试用例如表 3-4 所示。

表 3-3 判定覆盖的测试用例

输　入	通过路径	判定取值
x=7,y=1,z=3	a-b-d	M,N
x=-1,y=-3,z=0	a-c-e	-M,-N

表 3-4 条件判定覆盖的测试用例

输　入	通过路径
x=7,y=1,z=3	a-b-d
x=-1,y=-3,z=0	a-c-e

5. 条件组合覆盖

条件组合覆盖又称为多条件覆盖，指的是设计足够的测试用例，使判定条件中每个条件的可能组合至少出现一次。多条件覆盖需要的测试用例是用一个条件的逻辑操作符的真值表确定的。显然，满足多条件覆盖的测试用例一定满足判定覆盖、条件覆盖和条件判定覆盖。多条件覆盖是一个彻底的测试，但是它依然存在以下一些缺点。

(1) 它可能是非常冗长乏味的决定一个需要的测试用例的最小设置，特别是对于一些非常复杂的布尔表达式。

(2) 对于相似的复杂性的条件却需要非常大的变化。

(3) 可能会存在路径遗漏。

以上一段代码为例，给出其测试用例的设计过程。

对各判断语句的逻辑条件的取值组合标记如下。

(1) x>0,y>0，记作 T1、T2，条件组合取值 M。

(2) x>0,y<=0，记作 T1、-T2，条件组合取值-M。

(3) x<=0,y>0，记作-T1、T2，条件组合取值-M。

(4) x<=0,y<=0，记作-T1、-T2，条件组合取值-M。

(5) x>1,z>1，记作 T3、T4，条件组合取值 N。

(6) x>1,z<=1，记作 T3、-T4，条件组合取值 N。

(7) x<=1,z>1，记作-T3、T4，条件组合取值 N。

(8) x<=1,z<=1，记作-T3、-T4，条件组合取值-N。

设计测试用例如表 3-5 所示。

表 3-5 多条件覆盖的测试用例

输　入	通过路径	条件取值	覆盖组合号
x=1,y=3,z=2	a-b-d	T1,T2,-T3,T4	1,7
x=2,y=0,z=8	a-c-d	T1,-T2,T3,T4	2,5
x=-1,y=1,z=1	a-c-e	-T1,T2,-T3,-T4	3,8
x=-2,y=-3,z=0	a-c-e	-T1,-T2,-T3,-T4	4,8
x=5,y=9,z=0	a-b-d	T1,T2,T3,-T4	1,6

6. 路径覆盖

路径覆盖是指测试用例中执行到的路径数量占被测试模块所有可能的执行路径的比例。在路径覆盖中，我们只需要考虑所有可能的执行路径，对于不可能执行的路径，是不需要考虑的。而且对于一些大型程序，其包含的路径总量是非常庞大的，如果要把所有路径都找出来去覆盖也是不现实的。因此，我们可以借助以下方法寻找程序中的路径。

1）单个判断语句的路径计算

单个判断语句中，路径只有两条，一条是判断条件为真，另一条是判断条件为假。在不考虑判断分支中的路径分支时，路径数与判断分支数相等。

2）单个循环语句中的路径计算

在循环语句中，通常循环的每次迭代都可以看作一条路径，但这样计算出的路径的测试工作量太大，所以通过以下方式简化循环中的路径计算。

（1）当循环中条件一定会满足，循环内语句一定会执行时，循环语句中的可能执行的路径可看作一条。

例如下面的循环语句：

```
int i , sum;
for(i=1;i<=100;i++)
{
       sum+=i;
}
```

循环中的条件一定满足，循环语句 sum＋＝i 一定会被执行，直到 i＝101 时循环结束。在这种情况下，整个循环语句可能的执行路径可看作只有一条。

（2）当循环条件不一定满足，循环内的语句有可能被执行到，也可能不被执行时，循环语句中可能的执行路径可看作两条：一条路径循环被执行，另一条路径循环不被执行。

例如下面的循环语句：

```
int sum=0;
while  (x<100) {
       sum+=x;
       x++;
}
```

上面的循环中，当 x≥100 时，循环不执行，因此可能的执行路径有两条。

（3）当循环过程中有可能出现中断的情况。

例如下面的循环语句：

```
int GetSum(int maxnum){
       int sum=0;
       for(int i=0;i<50;i++){
              sum+=i;
              if(sum>=maxnum){
                     break;
              }
       }
       return sum;
}
```

上面这段程序，由于参数 maxnum 的不同，在循环的每次迭代都有两种可能，一种是满足判断条件，一种是不满足判断条件，所以可以认为路径的数量是 51 条，其中 50 条是 for 循环中 if 判断条件都满足的情况，另外一条是 for 循环中 if 判断条件没有被满足，break 语句没有被执行的情况。实际上，程序中可能执行的路径总数不多于输入参数的等价类的所有组合数量，上述代码中 GetSum()函数的输入参数实际上有 51 个等价类，可能的执行路径数量也是 51 条。

3）有嵌套判断或循环时的路径计算

下面是有嵌套判断语句的例子。

```
int GetMaxnum(int a,int b,int c){
      if(a>=b){
            if(a>=c)
                  return a;
            else
                  return c;
      }else{
            if(b>=c)
                  return b;
            else
                  return c;
      }
}
```

对上述程序的路径统计，先计算出第 1 个 if 中的路径为两条，再计算出 else 中的路径为两条，因此相乘得到总路径为 4 条，对于有嵌套判断或循环的程序，一般先从内层开始计算路径，然后通过相乘得到总的路径数。

3.3.3 基本路径法

基本路径法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，导出基本可执行的路径集合，从而设计测试用例的方法。在基本路径测试中，设计出的测试用例要保证在测试中程序的每条可执行语句至少执行一次。在基本路径法中，需要使用程序的控制流图进行可视化表达。

程序的控制流图是描述程序控制流的一种图示方法。其中，圆圈称为控制流图的一个节点，表示一个或多个无分支的语句或源程序语句；箭头称为边或连接，代表控制流。在将程序流程图简化成控制流图时，应注意：

（1）在选择或多分支结构中，分支的汇聚处应有一个汇聚节点；

（2）边和节点圈定的区域叫作区域，当对区域计数时，图形外的区域也应记为一个区域。

控制流图如图 3-8 所示。

环路复杂度是一种为程序逻辑复杂性提供定量测度的软件度量，将该度量用于计算程序的基本的独立路径数目，为确保所有语句至少执行一次的测试数量的上界。独立路径必须包含一条在定义之前不曾用到的边。以下 3 种方法可用于计算环路复杂度。

（1）流图中区域的数量对应于环路的复杂度。

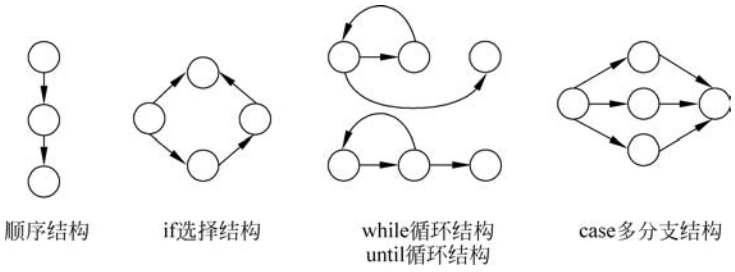

图 3-8　控制流图表示

(2) 给定流图 G 的环路复杂度 $V(G)$,定义为 $V(G)=E-N+2$,其中,E 为流图中边的数量,N 为流图中节点的数量。

(3) 给定流图 G 的环路复杂度 $V(G)$,定义为 $V(G)=P+1$,其中,P 为流图 G 中判定节点的数量。

基本路径测试法适用于模块的详细设计及源程序,其步骤如下。

(1) 以详细设计或源代码为基础,导出程序的控制流图。

(2) 计算控制流图 G 的环路复杂度 $V(G)$。

(3) 确定线性无关的路径的基本集。

(4) 生成测试用例,确保基本路径集中每条路径的执行。

每个测试用例执行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可执行语句至少被执行了一次。但是必须注意,一些独立路径往往不是完全孤立的,有时它是程序正常控制流的一部分,这时对这些路径的测试可以是另一条测试路径的一部分。

下面将以一个具体实例为出发点,讲解使用基本路径法测试的细节。

对于下面的程序,假设输入的取值范围为 1000＜year＜2001,使用基本路径法为变量 year 设计测试用例,使其满足基本路径覆盖的要求。

```
int IsLeap(int 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 leap;
}
```

根据源代码绘制程序的控制流图,如图 3-9 所示。

通过控制流图,计算环路复杂度 $V(G)$=区域数=4。

线性无关的路径集为：

(1) 1-3-8；

(2) 1-2-5-8；

(3) 1-2-4-7-8；

(4) 1-2-4-6-8。

设计测试用例如下。

路径 1：输入数据：year＝1999，预期结果：leap＝0；

路径 2：输入数据：year＝1996，预期结果：leap＝1；

路径 3：输入数据：year＝1800，预期结果：leap＝0；

路径 4：输入数据：year＝1600，预期结果：leap＝1。

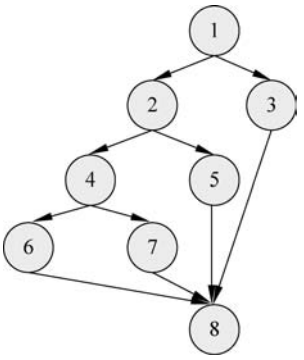

图 3-9　控制流图

3.3.4　白盒测试方法选择

白盒测试还有静态质量度量、域测试、Z 路径覆盖等方法，本书不再展开叙述。白盒测试的每种测试方法都有各自的优点和不足，需要测试人员根据实际软件特点、实际测试目标和测试阶段选择合适的方法设计测试用例，这样能有效地发现软件错误，提高测试效率和测试覆盖率。以下是选择方法的几条经验。

(1) 在测试中，可采取先静态再动态的组合方式，先进行代码检查和静态结构分析，再进行覆盖测试。

(2) 利用静态分析的结果作为引导，通过代码检查和动态测试的方式对静态分析的结果做进一步确认。

(3) 覆盖测试是白盒测试的重点，一般可使用基本路径测试法达到语句覆盖标准，对于软件的重点模块，应使用多种覆盖标准衡量测试的覆盖率。

(4) 不同的测试阶段，测试重点不同。在单元测试阶段，以代码检查、覆盖测试为主；在集成测试阶段，需要增加静态结构分析等；在系统测试阶段，应根据黑盒测试的结果，采用相应的白盒测试方法。

3.4　黑盒测试

黑盒测试又称为功能测试，它主要关注被测软件功能的实现，而不是其内部逻辑。在黑盒测试中，被测对象的内部结构、运作情况对测试人员是不可见的，测试人员把被测试的软件系统看作一个黑盒子，并不需要关心盒子的内部结构和内部特性，而只关注于软件产品的输入数据和输出结果，从而检查软件产品是否符合它的功能说明。黑盒测试有多种方法，如场景法、等价类划分、边界值分析、因果图法、决策表法等。

3.4.1　黑盒测试方法

1. 等价类划分法

根据软件测试原则可以知道，要做到穷举测试是不可能的，事实上也是不必要的。为了

减少测试工作量，需要对测试用例进行适当的选取。等价类划分法便提供了一种选取测试用例的方法。

等价类划分法是一种典型的黑盒测试方法，用这种方法设计测试用例完全不用考虑程序的内部结构，只根据对程序的要求和说明，即需求规格说明书。必须仔细分析和推敲说明书的各项内容，特别是功能需求，把说明中对输入的要求和输出的要求区别开来并加以分解。

等价类划分法把程序的输入域划分为若干部分，然后从每个部分中选取少数代表性数据当作测试用例。每类的代表性数据在测试中的作用等价于这一类中的其他值。也就是说，如果某一类中的一个用例发现了错误，这一类中的其他用例也能发现同样的错误；反之，如果某一类中的一个用例没有发现错误，则这一类中的其他用例也不会查出错误。

使用这一方法设计测试用例，首先必须在分析需求规格说明书的基础上划分等价类，列出等价类表。等价类划分有两种不同的情况：有效等价类和无效等价类。有效等价类是指对程序的规格说明是有意义的、合理的输入数据所构成的集合。无效等价类是指对程序的规格说明是无意义的、不合理的输入数据构成的集合。

在划分等价类时，有一些规则应该遵循。

(1) 如果输入条件规定了取值范围或个数，则可以确定一个有效等价类和两个无效等价类。例如，输入值是选课人数，为 0～100，那么有效等价类是 0≤学生人数≤100；无效等价类是学生人数<0 和学生人数>100。

(2) 如果输入条件规定了输入值的集合或是规定了“必须如何”的条件，则可以确定一个有效等价类和一个无效等价类。例如，输入值是日期类型的数据，那么有效等价类是日期类型的数据；无效等价类是非日期类型的数据。

(3) 如果输入是布尔表达式，可以分为一个有效等价类和一个无效等价类。例如，要求密码非空，则有效等价类为非空密码；无效等价类为空密码。

(4) 如果输入条件是一组值，且程序对不同的值有不同的处理方式，则每个允许的输入值对应一个有效等价类，所有不允许的输入值的集合为一个无效等价类。例如，输入条件“职称”的值是初级、中级或高级，那么有效等价类应该有 3 个：初级、中级和高级；无效等价类有一个：其他任何职称。

(5) 如果规定了输入数据必须遵循的规则，可以划分出一个有效等价类(符合规则)和若干个无效等价类(从不同的角度违反规则)。

划分好等价类后，就可以设计测试用例了。设计测试用例的步骤可以归结为 3 步。

(1) 对每个输入和外部条件进行等价类划分，画出等价类表，并为每个等价类进行编号。

(2) 设计一个测试用例，使其尽可能多地覆盖有效等价类，重复这一步，直到所有的有效等价类被覆盖。

(3) 为每个无效等价类设计一个测试用例。

下面将以一个测试 NextDate 函数的具体实例为出发点，讲解使用等价类划分法设计测试用例的过程。输入 3 个变量(年、月、日)，函数返回输入日期后面一天的日期：1≤月≤12，1≤日≤31，1812≤年≤2012。给出等价类划分表并设计测试用例。

划分等价类，得到等价类划分表，如表 3-6 所示。

表 3-6 等价类划分表

输入及外部条件	有效等价类	等价类编号	无效等价类	等价类编号
日期的类型	数字字符	1	非数字字符	8
年	1812～2012	2	＜1812	9
			＞2012	10
月	1～12	3	＜1	11
			＞12	12
非闰年的 2 月	日为 1～28	4	＜1	13
			＞28	14
闰年的 2 月	日为 1～29	5	＜1	15
			＞29	16
月份为 1 月、3 月、5 月、7 月、8 月、10 月、12 月	日为 1～31	6	＜1	17
			＞31	18
月份为 4 月、6 月、9 月、11 月	日为 1～30	7	＜1	19
			＞30	20

为有效等价类设计测试用例，如表 3-7 所示。

表 3-7 有效等价类的测试用例

序号	输入数据			预期输出			覆盖范围（等价类编号）
	年	月	日	年	月	日	
1	2003	3	15	2003	3	16	1,2,3,6
2	2004	2	13	2004	2	14	1,2,3,5
3	1999	2	3	1999	2	4	1,2,3,4
4	1970	9	29	1970	9	30	1,2,3,7

为无效等价类设计测试用例，如表 3-8 所示。

表 3-8 无效等价类的测试用例

序号	输入数据			预期结果	覆盖范围（等价类编号）
	年	月	日		
1	xy	5	9	输入无效	8
2	1700	4	8	输入无效	9
3	2300	11	1	输入无效	10
4	2005	0	11	输入无效	11
5	2009	14	25	输入无效	12
6	1989	2	－1	输入无效	13
7	1977	2	30	输入无效	14
8	2000	2	－2	输入无效	15
9	2008	2	34	输入无效	16
10	1956	10	0	输入无效	17
11	1974	8	78	输入无效	18
12	2007	9	－3	输入无效	19
13	1866	12	35	输入无效	20

通过案例可以了解，等价类划分法可以作为一种有效的黑盒测试方法，设计测试用例能够覆盖程序功能，而又不存在冗余的测试用例。但是需要对程序规格说明书进行深入了解并合理地划分等价类。有些时候，规格说明书中可能没有定义对无效输入的预期输出应该是什么样子，因此测试人员需要花费大量时间来定义这些测试用例的预期输出。这也是等价类划分法存在的一个缺陷。

2. 边界值分析法

人们从长期的测试工作经验中得知，大量的错误往往发生在输入和输出范围的边界上，而不是范围的内部。因此，针对边界情况设计测试用例，能够更有效地发现错误。

边界值分析法是一种补充等价类划分法的黑盒测试方法，它不是选择等价类中的任意元素，而是选择等价类边界的测试用例。实践证明，这些测试用例往往能取得很好的测试效果。边界值分析法不仅重视输入范围边界，也从输出范围中导出测试用例。

通常情况下，软件测试所包含的边界条件有以下几种类型：数字、字符、位置、质量、大小、速度、方位、尺寸、空间等；边界值应该为最大/最小、首位/末位、上/下、最快/最慢、最高/最低、最短/最长、空/满等情况。

用边界值分析法设计测试用例时应当遵守以下几条原则。

(1) 如果输入条件规定了取值范围，应以该范围的边界内及刚刚超范围的边界外的值作为测试用例。如以 a 和 b 作为输入条件，测试用例应当包括 a 和 b，以及略大于 a 和略小于 b 的值。

(2) 若规定了值的个数，应分别以最大、最小个数和稍小于最小个数和稍大于最大个数作为测试用例。

(3) 针对每个输出条件，也使用上面两条原则。

(4) 如果程序规格说明书中提到的输入或输出范围是有序的集合，如顺序文件、表格等，应注意选取有序集的第 1 个和最后一个元素作为测试用例。

(5) 分析规格说明，找出其他的可能边界条件。

边界值分析法利用输入变量的最小值、略大于最小值、输入范围内任意值、略小于最大值、最大值设计测试用例。如图 3-10 所示，对于 n 个变量，使除一个以外的所有变量都取正常值，使剩余的变量取上述 5 个值，对每个变量都重复进行。一个 n 变量函数的边界值有 $4n+1$ 个测试用例。

健壮性测试是边界值分析的一种简单扩展，除了使用 5 个边界值分析取值，还要采用一个略小于最小值和一个略大于最大值的取值。健壮性测试更关注例外情况如何处理。如图 3-11 所示，一个 n 变量函数的健壮性边界值有 $6n+1$ 个测试用例。

如果对于每个变量，首先取边界值的 5 个取值作为集合，然后对这些集合进行笛卡尔积运算生成测试用例，则称为最坏情况测试。一个 n 变量函数的最坏情况测试有 $5n$ 个测试用例。同理，可以进行健壮最坏情况测试。一个 n 变量函数的最坏情况测试有 $7n$ 个测试用例。可以看到，测试的完全度与测试复杂度是成正比的，在实际测试中选择哪种边界值分析法，要根据项目具体需要确定。

普通边界条件是很容易找到的，它们在规格说明书中有定义，或者在使用软件过程中确定。有些边界在软件内部，最终用户几乎看不到，但是软件测试仍有必要检查，这样的边界条件成为次边界条件。寻找这样的边界条件就需要测试人员了解软件大概的工作方式。边

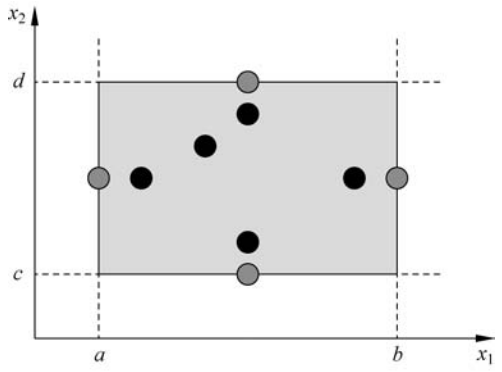

图 3-10　边界值分析测试用例示意图

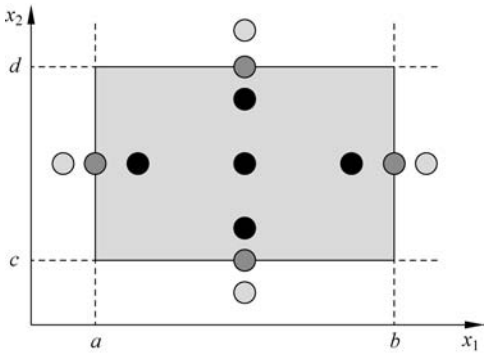

图 3-11　健壮性边界值测试用例示意图

界条件的确定有时也需要一定的领域知识。

以上面提到的 NextDate 函数为例，除了已经用等价类划分法设计出的测试用例外，还应该用边界值分析法再补充如表 3-9 所示的测试用例。

表 3-9　边界值分析法设计的测试用例

序号	边　界　值	输入数据			预期输出		
		年	月	日	年	月	日
1	使年刚好等于最小值	1812	3	15	1812	3	16
2	使年刚好等于最大值	2012	3	15	2012	3	16
3	使年刚刚小于最小值	1811	3	15	输入无效		
4	使年刚刚大于最大值	2013	3	15	输入无效		
5	使月刚好等于最小值	2000	1	15	2000	1	16
6	使月刚好等于最大值	2000	12	15	2000	12	16
7	使月刚刚小于最小值	2000	0	15	输入无效		
8	使月刚刚大于最大值	2000	13	15	输入无效		
9	使闰年的 2 月的日刚好等于最小值	2000	2	1	2000	2	2
10	使闰年的 2 月的日刚好等于最大值	2000	2	29	2000	3	1
11	使闰年的 2 月的日刚刚小于最小值	2000	2	0	输入无效		
12	使闰年的 2 月的日刚刚大于最大值	2000	2	30	输入无效		
13	使非闰年的 2 月的日刚好等于最小值	2001	2	1	2001	2	2
14	使非闰年的 2 月的日刚好等于最大值	2001	2	28	2001	3	1
15	使非闰年的 2 月的日刚刚小于最小值	2001	2	0	输入无效		
16	使非闰年的 2 月的日刚刚大于最大值	2001	2	29	输入无效		
17	使 1 月、3 月、5 月、7 月、8 月、10 月、12 月的日刚好等于最小值	2001	10	1	2001	10	2
18	使 1 月、3 月、5 月、7 月、8 月、10 月、12 月的日刚好等于最大值	2001	10	31	2001	11	1

续表

序号	边　界　值	输入数据			预期输出		
		年	月	日	年	月	日
19	使1月、3月、5月、7月、8月、10月、12月的日刚刚小于最小值	2001	10	0	输入无效		
20	使1月、3月、5月、7月、8月、10月、12月的日刚刚大于最大值	2001	10	32	输入无效		
21	使4月、6月、9月、11月的日刚好等于最小值	2001	6	1	2001	6	2
22	使4月、6月、9月、11月的日刚好等于最大值	2001	6	30	2001	7	1
23	使4月、6月、9月、11月的日刚刚小于最小值	2001	6	0	输入无效		
24	使4月、6月、9月、11月的日刚刚大于最大值	2001	6	31	输入无效		

3. 因果图法

等价类划分法和边界值分析法都主要考虑的是输入条件，而没有考虑输入条件的各种组合以及各个输入条件之间的相互制约关系。然而，如果在测试时考虑到输入条件的所有组合方式，可能其本身非常大甚至是个天文数字。因此，必须考虑描述多种条件的组合，相应地产生多个动作的形式，设计测试用例。这就需要利用因果图法。

因果图法是一种黑盒测试方法，它从自然语言书写的程序规格说明书中寻找因果关系，即输入条件与输出和程序状态的改变，通过因果图产生判定表。它能够帮助人们按照一定的步骤高效地选择测试用例，同时还能指出程序规格说明书中存在的问题。

在因果图中，用 C 表示原因，E 表示结果，各节点表示状态，取值为 0 表示某状态不出现，取值为 1 表示某状态出现。因果图有 4 种关系符号，如图 3-12 所示。

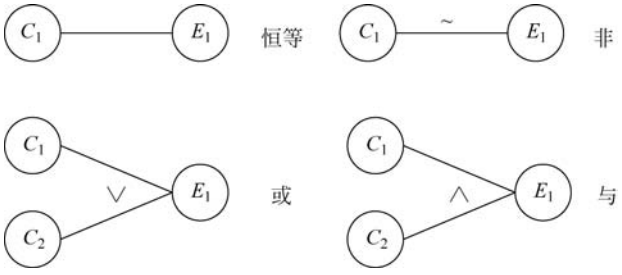

图 3-12　因果图基本符号

(1) 恒等：若原因出现，则结果出现；若原因不出现，则结果不出现。

(2) 非(～)：若原因出现，则结果不出现；若原因不出现，则结果反而出现。

(3) 或(∨)：若几个原因中有一个出现，则结果出现；若几个原因都不出现，则结果

不出现。

(4) 与(∧)：若几个原因都出现，结果才出现；若其中一个原因不出现，则结果不出现。

为了表示原因与原因之间、结果与结果之间可能存在的约束关系，在因果图中可以附加一些表示约束条件的符号，如图 3-13 所示。从输入考虑，有以下 4 种约束。

(1) E 约束(互斥)：表示 a 和 b 两个原因不会同时成立，最多有一个可以成立。

(2) I 约束(包含)：表示 a 和 b 两个原因至少有一个必须成立。

(3) O 约束(唯一)：表示 a 和 b 两个原因必须有且仅有一个成立。

(4) R 约束(要求)：表示 a 出现时，b 也必须出现。

从输出考虑，有一种约束。

M 约束(强制)：表示 a 为 1 时，b 必须为 0。

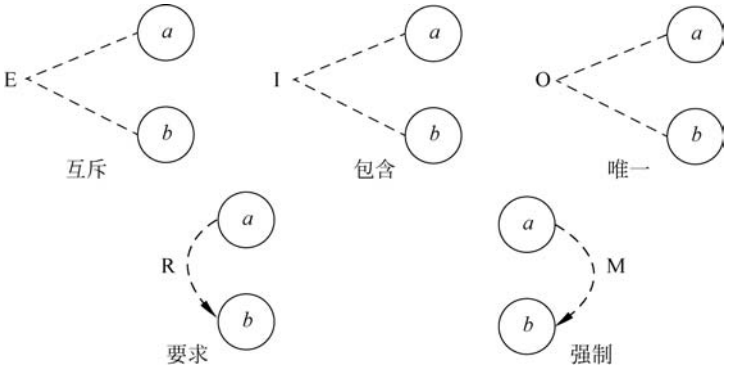

图 3-13　因果图约束符号

因果图法设计测试用例的步骤如下。

(1) 分析程序规格说明书的描述中，哪些是原因，哪些是结果。原因常常是输入条件或输入条件的等价类，而结果常常是输出条件。

(2) 分析程序规格说明书中描述的语义内容，并将其表示成连接各个原因与各个结果的因果图。

(3) 由于语法或环境的限制，有些原因和结果的组合情况是不可能出现的，为表明这些特定的情况，在因果图上使用若干特殊的符号标明约束条件。

(4) 把因果图转化为决策表。

(5) 为决策表中每列表示的情况设计测试用例。

后面两个步骤中提到的决策表，将在后续进行详细介绍。如果项目在设计阶段已存在决策表，则可以直接使用而不必再画因果图。

下面以一个自动饮料售货机软件为例，展示因果图分析方法。该自动饮料售货机软件的规格说明如下。

有一个处理单价为 1 元 5 角的盒装饮料的自动售货机软件。若投入 1 元 5 角硬币，按下“可乐”“雪碧”或“红茶”按钮，相应的饮料就送出来；若投入 2 元硬币，在送出饮料的同时退还 5 角硬币。

首先从软件规格说明中分析原因、结果以及中间状态。分析结果如表 3-10 所示。

表 3-10　自动饮料售货机软件分析结果

原因	C_1：投入 1 元 5 角硬币 C_2：投入 2 元硬币 C_3：按“可乐”按钮 C_4：按“雪碧”按钮 C_5：按“红茶”按钮	中间状态	11：已投币 12：已按钮	结果	E_1：退还 5 角硬币 E_2：送出可乐 E_3：送出雪碧 E_4：送出红茶

根据表 3-10 中的原因与结果，结合软件规格说明，连接成如图 3-14 所示的因果图。

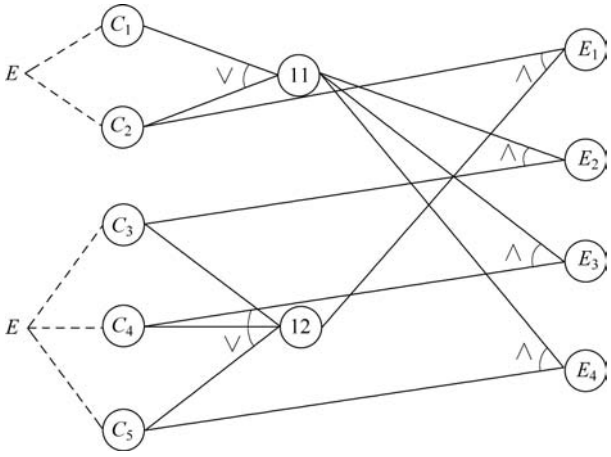

图 3-14　自动饮料售货机软件因果图

4. 决策表法

在一些数据处理问题中，某些操作是否实施依赖于多个逻辑条件的取值。在这些逻辑条件取值的组合所构成的多种情况下，分别执行不同的操作。处理这类问题的一个非常有力的工具就是决策表。

决策表是分析和表达多逻辑条件下执行不同操作的情况的工具，可以把复杂逻辑关系和多种条件组合的情况表达得比较明确。决策表通常由 4 部分组成，如图 3-15 所示。

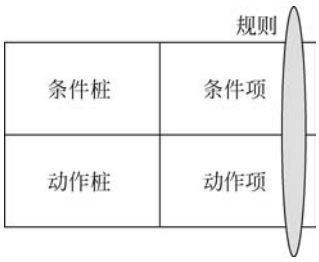

图 3-15　决策表组成

（1）条件桩：列出问题的所有条件。

（2）条件项：列出所列条件下的取值在所有可能情况下的真假值。

（3）动作桩：列出问题规定可能采取的动作。

（4）动作项：列出在条件项的各种取值情况下应采取的动作。

规则规定了任何一个条件组合的特定取值及其相应要执行的操作。在决策表中贯穿条件项和动作项的一列就是一条规则。有两条或多条规则具有相同的动作，并且其条件项之间存在着极为相似的关系的规则可以进行规则合并。

决策表的建立应当根据软件规格说明书，分为以下几个步骤。

（1）确定规则个数。

（2）列出所有条件桩和动作桩。

（3）填入条件项。

（4）填入动作项，制订初始决策表。

（5）简化，合并相似规则或者相同动作。

在简化并得到最终决策表后，只要选择适当的输入，使决策表每列的输入条件得到满足即可生成测试用例。

将上面得到的自动饮料售货机软件因果图转换为决策表，如表 3-11 所示。

表 3-11　自动饮料售货机软件决策表

		1	2	3	4	5	6	7	8	9	10	11
条件	C_1：投入 1 元 5 角硬币	1	1	1	1	0	0	0	0	0	0	0
	C_2：投入 2 元硬币	0	0	0	0	1	1	1	1	0	0	0
	C_3：按“可乐”按钮	1	0	0	0	1	0	0	0	1	0	0
	C_4：按“雪碧”按钮	0	1	0	0	0	1	0	0	0	1	0
	C_5：按“红茶”按钮	0	0	1	0	0	0	1	0	0	0	1
中间状态	11：已投币	1	1	1	1	1	1	1	1	0	0	0
	12：已按钮	1	1	1	0	1	1	1	0	1	1	1
动作	E_1：退还 5 角硬币	0	0	0	0	1	1	1	0	0	0	0
	E_2：送出可乐	1	0	0	0	1	0	0	0	0	0	0
	E_3：送出雪碧	0	1	0	0	0	1	0	0	0	0	0
	E_4：送出红茶	0	0	1	0	0	0	1	0	0	0	0

可以根据上述决策表设计测试用例，从而验证适当的输入组合能否得到正确的输出。特别是在本案例中，利用因果图和决策表法能够很清晰地验证自动饮料售货机软件的功能完备性。

5. 正交试验法

在将因果图转换成决策表生成测试用例时，若要进行全面测试，其得到的测试用例数目大得惊人。例如，对于有 n 个原因导致一个结果的因果图，如果每个原因的取值有两种：存在或不存在，则进行全面测试需要为此设计 2^n 种测试用例，再考虑到其他因果图，最后得出的测试用例数量无法想象，这给软件测试带来了沉重的负担。为了有效、合理地减少测试的工时与费用，可利用正交试验法进行测试用例的设计。

正交试验法是从大量的实验数据中挑选适量的、有代表性的点，合理安排测试的设计方法。

日本著名统计学家田口玄一将正交试验选择的水平组合列成表格，称为正交表。例如，做一个 3 因素 3 水平的试验，按全面试验要求，须进行 $3^3=27$ 种组合的试验，且尚未考虑每种组合的重复数。若按 $L_9(3^3)$ 正交表安排试验，只需 9 次试验，按 $L_{18}(3^7)$ 正交表也只需 18 次试验，显然大大减少了工作量。因而正交试验设计在很多领域的研究中已经得到广泛应用。

正交表的形式为 $L_{行数}(水平数^{因素数})$。其中，行数表示正交表中的行的个数，即试验的次数，也是我们通过正交试验法设计的测试用例的个数；因素数是正交表中列的个数，即要测试的功能点；水平数是任何单个因素能够取得的值的最大个数。正交表中包含的值为从

0～(水平数－1)或从 1～水平数,即要测试功能点的输入条件。

正交表具有以下两项性质。

(1) 每列中,不同的数字出现的次数相等。例如,在 2 水平正交表中,任何一列都有数码 1 和 2,且任何一列中它们出现的次数是相等的;在 3 水平正交表中,任何一列都有数码 1、2 和 3,且在任何一列的出现数均相等。

(2) 任意两列中数字的排列方式齐全而且均衡。例如,在 2 水平正交表中,任何两列(同一行内)有序对共有 4 种:(1,1)、(1,2)、(2,1)、(2,2),每对出现次数相等;在 3 水平情况下,任何两列(同一行内)有序对共有 9 种:(1,1)、(1,2)、(1,3)、(2,1)、(2,2)、(2,3)、(3,1)、(3,2)、(3,3),且每对出现次数也相等。

以上两点充分地体现了正交表的两大优越性,即均匀分散性和整齐可比性。通俗地说,每个因素的每个水平与另一个因素的每个水平各碰一次,这就是正交性。

下面以一个用户注册功能为例,展示正交试验法设计测试用例的方法。该用户注册页面有 7 个输入框,分别是用户名、密码、确认密码、真实姓名、地址、手机号、电子邮箱。假设每个输入框只有填与不填两种状态,则可以设计 $L_8(2^7)$正交表,如表 3-12 所示。其中因素 C_1～C_7 分别表示上述 7 个输入框。表 3-12 中 1 表示填该输入框,0 表示不填。

根据表 3-12 可以得到 8 个测试用例,读者可以根据各因素代表的输入框含义自己生成测试用例。

表 3-12　$L_8(2^7)$正交表

因素 / 行号	C_1	C_2	C_3	C_4	C_5	C_6	C_7
1	1	1	1	1	1	1	1
2	1	1	1	0	0	0	0
3	1	0	0	1	1	0	0
4	1	0	0	0	0	1	1
5	0	1	0	1	0	1	0
6	0	1	0	0	1	0	1
7	0	0	1	1	0	0	1
8	0	0	1	0	1	1	0

6. 场景法

现在软件很多都是用事件触发控制流程,事件触发时的情形便形成场景,而同一事件不同的触发顺序和处理结果就形成了事件流。这种在软件设计中的思想也可以应用到软件测试中,可生动地描绘出事件触发时的情形,有利于测试者执行测试用例,同时测试用例也更容易理解和执行。

用例场景是通过描述流经用例的路径确定的过程,这个流经过程要从用例开始到结束遍历其中所有的基本流和备选流。

(1) 基本流:采用黑直线表示,是经过用例的最简单路径,表示无任何差错,程序从开始执行到结束。

(2) 备选流:采用不同颜色表示,一个备选流可以从基本流开始,在某个特定条件下执

行，然后重新加入基本流，也可以起源于另一个备选流，或终止用例，不再加入基本流。

应用场景法进行黑盒测试的步骤如下。

（1）根据规格说明，描述出程序的基本流和各个备选流。

（2）根据基本流和各个备选流生成不同的场景。

（3）对每个场景生成相应的测试用例。

（4）对生成的所有测试用例进行复审，去掉多余的测试用例，对每个测试用例确定测试数据。

下面以一个经典的 ATM 机为例，介绍使用场景法设计测试用例的过程。ATM 机取款流程的场景分析如图 3-16 所示，其中灰色框构成的流程为基本流。

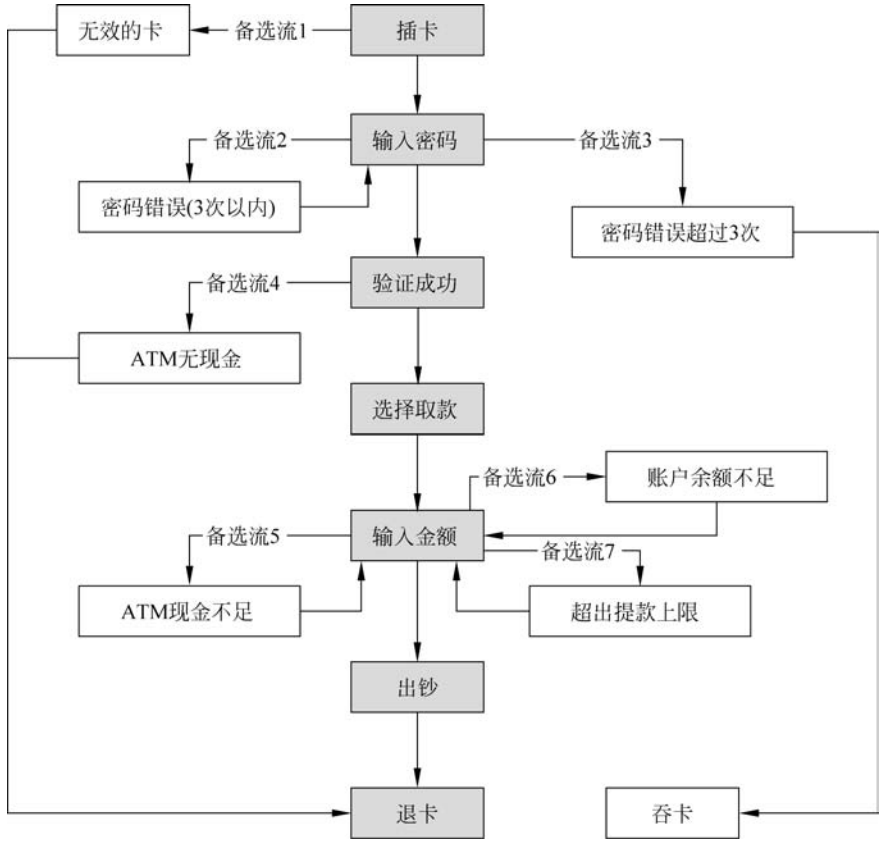

图 3-16　ATM 取款流程场景法分析图

该程序用例场景如表 3-13 所示。

表 3-13　用例场景

序　　号	场　　景	基　本　流	备　选　流
场景 1	成功提款	基本流	
场景 2	无效卡	基本流	备选流 1
场景 3	密码错误 3 次以内	基本流	备选流 2
场景 4	密码错误超过 3 次	基本流	备选流 3
场景 5	ATM 无现金	基本流	备选流 4

续表

序　号	场　景	基 本 流	备 选 流
场景 6	ATM 现金不足	基本流	备选流 5
场景 7	账户余额不足	基本流	备选流 6
场景 8	超出提款上限	基本流	备选流 7

接下来设计用例覆盖每个用例场景，如表 3-14 所示。

表 3-14　场景法测试用例

用例号	场景	账　户	密码	操　作	预 期 结 果
1	场景 1	621226XXXXXXXXX3481	123456	插卡，取 500 元	成功取款 500 元
2	场景 2	—	—	插入一张无效卡	系统退卡，显示该卡无效
3	场景 3	621226XXXXXXXXX3481	123456	插卡，输入密码 111111	系统提示密码错误，请求重新输入
4	场景 4	621226XXXXXXXXX3481	123456	插卡，输入密码 111111 超过 3 次	系统提示密码输入错误超过 3 次，卡被吞掉
5	场景 5	621226XXXXXXXXX3481	123456	插卡，选择取款	系统提示 ATM 无现金，退卡
6	场景 6	621226XXXXXXXXX3481	123456	插卡，取款 2000 元	系统提示现金不足，返回输入金额界面
7	场景 7	621226XXXXXXXXX3481	123456	插卡，取款 3000 元	系统提示账户余额不足，返回输入金额界面
8	场景 8	621226XXXXXXXXX3481	123456	插卡，取款 3500 元	系统提示超出取款上限（3000 元），返回输入金额界面

3.4.2　黑盒测试方法选择

此外，黑盒测试还有错误推测等方法，本书不再展开叙述。黑盒测试的每种测试方法都有各自的优缺点，需要测试人员根据实际项目特点和需要选择合适的方法设计测试用例。以下是选择黑盒测试方法的几条经验。

（1）在任何情况下都必须选择边界值分析方法。经验表明用这种方法设计出的测试用例发现程序错误的能力最强。

（2）必要时用等价类划分法补充一些测试用例。

（3）用错误推测法再追加一些测试用例。

（4）如果程序的功能说明中含有输入条件的组合情况，则可选用因果图法和决策表法。

选择合适的测试方法能够极大地提高黑盒测试的效率和效果。除了上述几条经验，还需要测试人员积累实际的测试经验，做出合适的选择。

3.4.3　白盒测试与黑盒测试的比较

白盒测试和黑盒测试是两类软件测试方法，传统的软件测试活动基本上都可以划分到

这两类测试方法中。表 3-15 给出了两种方法的基本比较。

表 3-15 黑盒测试和白盒测试比较

黑 盒 测 试	白 盒 测 试
不涉及程序结构	考查程序逻辑结构
用软件规格说明书生成测试用例	用程序结构信息生成测试用例
适用于从单元测试到系统验收测试	主要适用于单元测试和集成测试
某些代码段得不到测试	对所有逻辑路径进行测试

白盒测试和黑盒测试各有侧重点，不能相互取代，在实际测试活动中，这两种测试方法不是截然分开的。通常在白盒测试中交叉着黑盒测试，黑盒测试中交叉着白盒测试。相对来说，白盒测试比黑盒测试成本要高得多，它需要测试在可以被计划前产生源代码，并且在确定合适数据和决定软件是否正确方面需要花费更多的工作量。

在实际测试活动中，应当尽可能使用可获得的软件规格从黑盒测试方法开始测试计划，白盒测试计划应当在黑盒测试计划已成功通过之后再开始，使用已经产生的流程图和路径判定。路径应当根据黑盒测试计划进行检查并且决定和使用额外需要的测试。

灰盒测试结合了白盒测试和黑盒测试的要素，考虑了用户端、特定的系统知识和操作环境。它在系统组件的协同性环境中评价应用软件的设计。可以认为集成测试就是一类灰盒测试。关于灰盒测试，本书不再展开叙述。

3.5 本章小结

本章主要介绍了静态测试、动态测试的定义与内容，以及静态测试、动态测试的分类及方法。

白盒测试关注软件产品的内部细节和逻辑结构，可以分为静态测试和动态测试。静态测试不通过执行程序进行测试，其关键是检查软件的表示与描述是否一致，是否存在冲突或歧义；动态测试需要执行程序，当程序在模拟的或真实的环境中执行之前、之中和之后，对程序行为分析，主要验证一个程序在检查状态下是否正确。本章介绍了白盒测试常用的方法，并着重介绍了程序插桩技术、逻辑覆盖法以及基本路径法，并对各个技术方法附以相关实例进行详细说明。

黑盒测试主要关注被测软件功能的实现，而不是其内部逻辑。本章重点介绍了常用的几种黑盒测试方法，并给出了相应的案例说明。


拓展练习

习题 3

(1) 代码检查法主要包括哪些主要内容？可以发现哪些问题？

(2) 试比较代码审查与走查的异同。

(3) 静态结构分析有哪几种形式？

(4) 动态测试可分为哪些方法？分类依据分别是什么？

(5) 什么是白盒测试？包括哪些技术？

（6）利用基本路径测试技术为下面一段程序设计测试用例。

```
while(a > 0)
{
    a = a - 1;
    if(b < 0 || c > = 1)
    {
        c = c - b;
    }
    else
        c = c + b;
}
a = b + c;
```

（7）什么是黑盒测试？有哪些主要方法？

（8）给出白盒测试与黑盒测试的不同。

（9）某程序功能说明书指出，该程序的输入数据为每个学生的学号。其中，学号由以下3部分构成：

- 入学年份：4位数字(1900～2999)；
- 专业编码：0或1开头的4位数字；
- 序号：两位数字。

试用等价类划分法设计测试用例。

（10）对于一个需要输入姓名、身份证号码、手机号码的系统，按每个输入有两个状态(填与不填)设计一个最小行数的正交表。