第5章

软件测试过程

5.1 单元测试

5.1.1 简介

1. 什么是单元测试

单元测试是指对软件中的最小可测试单元进行检查和验证，其目的在于发现每个程序模块内部可能存在的问题和缺陷。单元测试一般是将最小可测试单元与程序的其他部分隔离的情况下对其进行测试。

单元的粒度具体划分可以不同，在传统的结构化编程语言如 C 语言中，单元一般是模块，也就是函数或子过程；在面向对象语言中（如 C++ 、Java 等），单元是类或类的方法；在 Ada 语言中，单元可为独立的过程、函数或 Ada 包；在第四代语言中，单元对应为一个菜单或显示界面。

单元测试中的一个可测“单元”应符合以下要求。

（1）是可测试的、最小的、不可再分的程序模块。

（2）有明确的功能、规格定义。

（3）有明确的接口定义，清晰地与同一程序的其他单元划分开来。

2. 单元测试的执行人

单元测试通常是在代码完成后由程序员自己来完成，有时测试人员也会参加单元测试，但程序员在单元测试中仍会起到主要作用，单元测试是程序员的一项基本职责。程序员必须对自己所编写的代码保持认真负责的态度，这是程序员的基本职业素质之一。同时，直接影响单元测试能力也是程序员的一项基本能力，这种能力的高低直接影响程序员的工作效率与软件的质量。在编码的过程中进行单元测试，由于程序员对软件的设计和代码都很熟悉，不需要额外花时间去阅读、理解、分析程序的设计书和源代码，所以测试成本是最小的，测试效率也是最高的，得到的将是更优质的代码。程序员通过单元测试，发现代码中的各种问题，也能增加经验、提高编程能力和水平。

3. 单元测试的必要性

在软件开发过程中，越早发现问题、解决问题，花费的成本越小。经验表明，一个尽责

的单元测试在软件开发的早期会发现很多的缺陷和问题，当时修改它们的成本会很低，而如果拖到后期阶段，缺陷的发现和修改将会变得更加困难，并要消耗更多的时间和费用。进行充分的单元测试，是提高软件质量，降低开发成本的必由之路。

有统计数据表明，以软件的一个功能点为基准，单元测试的成本效率大约是集成测试的两倍，是系统测试的三倍。

4. 单元测试方法

单元测试的方法一般以白盒测试方法为主，黑盒测试方法为辅。

白盒测试主要是检查程序的内部结构、逻辑、循环和路径。通常是理解程序的单元内部结构，分析单元的输入/输出，构造合适的单元测试用例，达到单元内程序路径的最大覆盖，保证单元内部程序运行路径处理正确，它侧重于单元内部结构测试，依赖于对单元实施细节的了解。常用的单元测试用例设计方法有逻辑覆盖测试和基本路径测试。

黑盒测试方法通过对程序单元的输入/输出的用例构造验证单元的特性和行为，侧重于核实单元的可观测行为和功能，它只用到程序的规格说明，不需要了解程序的内部实现细节，重点判断程序单元是否能完成需求规格说明的功能要求。常用测试用例设计方法有等价类划分、边界值分析、错误推测和因果图分析等方法。

5. 单元测试目标

单元测试是软件测试的基础。单元测试的目标是要确保各单元模块按照设计被正确地进行了编码实现，还需要保证代码在结构上可靠健全，能够在所有情况下正确响应。通过单元测试，应能确保每个单元模块都能正常工作，程序代码符合各种要求和规范。

单元模块要确定被正确编码实现，重点从以下几方面考虑。

(1) 信息能否正确地流入和流出单元模块。

(2) 在单元模块工作过程中，其内部数据能否保持其完整性，包括内部数据的形式、内容及相互关系不发生错误，也包括全局变量在单元模块中的处理和影响。

(3) 在未限制数据加工而设置的边界处，单元模块能否正确工作。

(4) 单元模块的运行能否做到满足特定的逻辑覆盖。

(5) 单元模块中发生了错误，其中的出错处理措施是否有效等。

同时合格的单元模块代码应该具备正确性、清晰性、规范性、一致性、高效性等，其中优先级最高的正确性，只有先满足正确性，其他特性才具有实际意义。而对有些会反复执行的代码，还需要具有高效性，否则会影响整个系统的性能。

(1) 正确性是指代码逻辑必须正确，能够实现预期的功能。

(2) 清晰性是指代码必须简明、易懂，注释准确没有歧义。

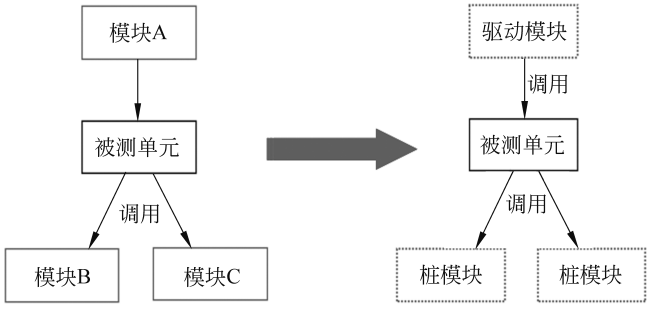
(3) 规范性是指代码必须符合企业或部门所定义的共同规范包括命名规则，代码风格等。

(4) 一致性指代码必须在命名、风格上保持统一。

(5) 高效性是指需要尽可能降低代码的执行时间。

6. 单元测试模型

一个单元模块或一个方法等并不是一个独立的程序，在测试它时要同时考虑它和外界的联系，需要用一些辅助模块去模拟与被测模块相联系的其他模块。这些辅助模块分为驱动模块和桩模块两种，如图 5-1 所示。



(a) 驱动模块　　(b) 桩模块

图 5-1　驱动模块和桩模块

（1）驱动模块

驱动模块是用来模拟被测单元的上层模块的程序模块。驱动模块能够接收或者设置测试数据、参数、环境变量等，调用被测单元，将数据传递给被测单元，如果需要还可以显示或者打印测试的执行结果。可将驱动模块理解为被测单元的主程序。

（2）桩模块

桩模块用来模拟被测单元的子模块。设计桩模块的目的是模拟被测单元所调用的子模块，接受被测单元的调用，并返回调用结果给被测单元。桩模块不一定需要包括子模块的全部功能，但至少应能模拟满足被测单元的调用需求，而不至于让被测单元在调用它时出现错误。

驱动模块和桩模块的编写会产生一定的工作量，会带来额外的开销。因为它们在软件交付时并不作为产品的一部分一同交付。特别是桩模块，为了能够正确、充分地测试软件，桩模块可能需要模拟实际子模块的功能，这样桩模块的建立就不是很轻松了。有时编写桩模块是非常困难和费时的，但也可以采取一定的策略，来避免编写桩模块，只需在项目进度管理时将实际桩模块的代码编写工作安排在被测模块前编写即可。而且这样可以提高测试工作的效果，因为不断调用实际的桩模块可以更好地对其进行测试，保证产品的质量。

被测模块及与它相关的驱动模块和桩模块共同构成了一个“测试环境”。建立单元测试的环境时，需完成以下一些工作。

（1）构造最小运行调度系统，即构造被测单元的驱动模块。

（2）模拟被测单元的接口，即构造被测单元调用的桩模块。

（3）模拟生成测试数据及状态，为被测单元运行准备动态环境。

单元测试模型如图 5-2 所示。

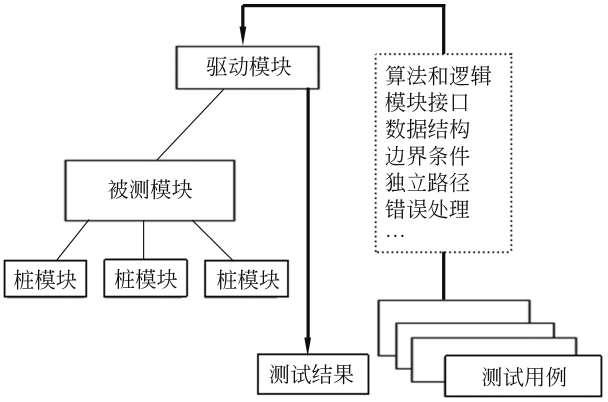


图 5-2 单元测试模型

5.1.2 单元测试的任务

对编码完成的软件进行单元测试，所测试的内容包括程序的内部结构以及程序单元的功能和可观测的行为。它主要分为两个步骤，即静态检查和动态测试。

静态检查是单元测试的第一步，其工作主要是保证代码中算法或流程的逻辑正确性，应通过人工检查发现代码的逻辑错误，其次还要检查代码的清晰性、规范性、一致性，考虑算法的执行效率等。第二步是通过设计测试用例，执行被测程序，比较实际结果与预期结果的异同，以发现程序中的错误。但是代码中仍会有大量的隐性错误无法通过静态检查发现，必须通过动态测试才能够捕捉和发现。动态测试是单元测试的重点与难点。一般而言，应当对程序模块进行以下动态单元测试。

(1) 对模块内所有独立的执行路径至少测试一次。

(2) 对所有的逻辑判定，取“真”与“假”的两种情况都至少执行一次。

(3) 在循环的边界和运行界限内执行循环体。

(4) 测试内部数据的有效性等。

单元测试的依据主要是软件的详细设计说明书、编码规范等，检查和测试的对象主要就是源程序。单元测试的主要任务如下。

(1) 验证代码能否达到详细设计的预期要求。

(2) 发现代码中不符合编码规范的地方。

(3) 准确定位错误，以便排除错误。

具体而言，单元测试应检查和测试的内容包括如下方面。

1. 算法和逻辑

算法和逻辑，即检查算法的和内部各个处理逻辑的正确性。例如，某程序员编写的打印下降三角形的九九乘法表的程序如下。

```
public static void printTable() {
for (int i =1; i <=9; i++) {
```

```
    for (int j =1; j<=9; j++) {
         System.out.print(String.format("%d * %d =%-2d ", i, j, i * j));}
         System.out.println();
        }  }
```

通过检查和测试应能发现程序逻辑是错误的，打印出来的不是下降三角形的九九乘法表，而是9×9的方阵。改正的办法是把第二个循环 **for**（**int** j = 1；j<= 9；j++），修改为 **for**（**int** j = 1；j<= i；j++）。

2. 模块接口

对模块自身的接口做正确性检查，确定形式参数个数、数据类型、顺序是否正确，确定返回值类型，检查返回值的正确性。检查调用其他模块的代码的正确性，调用其他模块时给定的参数类型正确与否、参数个数正确与否、参数顺序正确与否，特别是具有多态的方法尤其需要注意。检查返回值正确与否，有没有误解返回值所表示的意思。必要时可以对每个被调用的方法的返回值用显式代码如程序插桩，做正确性检查，如果被调用方法出现异常或错误程序应该给予反馈，并添加适当的出错处理代码。

例如，某程序员编写的求平均成绩的代码段如下。

```
public class getScoreAverage
{  public float getAverage(String[] scores)
       {  if (scores==null || scores.length==0)
          {  throw new NullPointerException(); }
          float sum =0.0F;
          int j=scores.length;
          for (int i=0; i<j; i++)
          {  sum +=scores[i]; }
        return sum/j;
   }
public static void main(String[] args) {
    getScoreAverage cj =new getScoreAverage();
    int[] scores={60,80,70};
    System.out.println(cj.getAverage(scores)); } }
```

程序中的问题是：函数内部把成绩当成数值型数据处理，直接进行累加，而形式参数中存放成绩的是字符型数组，所以接口和内部实现是不一致的。要改正的话，既可以修改程序内部实现，也可以修改接口，但如果事先没有规定程序接口，显然修改接口要比修改内部实现简单一些，只把 **public float** getAverage（String[] scores）改为 **public float** getAverage(int[] scores)即可。

例如，有一个 Web 软件，其中管理员登录的几个类分别如下。

```
cn.appsys.service.backend包下的 BackendUserService.java
//定义一个管理员登录的接口
public interface BackendUserService {
```

```
    /**
     * 用户登录
     * @param userCode
     * @param userPassword
     * @return
     */
     public BackendUser login(String userCode, String userPassword) throws
     Exception;
    }
```

cn.appsys.service.backend包下的BackendUserServiceImpl.Java

```
//定义一个类实现管理员登录的接口
    @Service
public class BackendUserServiceImpl implements BackendUserService {
    @Resource
    private BackendUserMapper mapper;

    @Override
        public BackendUser login(String userCode, String userPassword) throws
Exception {
        BackendUser user =null;
        user =mapper.getLoginUser(userCode);
            if(null !=user){                    //匹配密码
            if(!user.getUserPassword().equals(userPassword))
                user =null;}
        return user;
    }}
```

cn.appsys.controller.backend包下的UserLoginController.java

```
//管理员登录控制类
Public class UserLoginController {
    private Logger logger =Logger.getLogger(UserLoginController.class);

    @Resource
    private BackendUserService backendUserService;

    @RequestMapping(value="/login")
    public String login(){
        logger.debug("LoginController welcome AppInfoSystem");
        return "backendlogin";}

    @RequestMapping(value="/dologin",method=RequestMethod.POST)
    public String doLogin(@RequestParam String userCode,@RequestParam String
    userPassword,HttpServletRequest request,HttpSession session){
        logger.debug("doLogin==========================");
        BackendUser user =null;                 //调用 service()方法.进行用户匹配
```

```
        try {
            user =backendUserService.login(userCode,userPassword);
        } catch (Exception e) {
            e.printStackTrace();
        }
        if(null !=user){                                    //登录成功
            //放入 session
            session.setAttribute(Constants.USER_SESSION, user);
            //页面跳转(main.jsp)
            return "redirect:/manager/backend/main";
        }else{
            //页面跳转(login.jsp)带出提示信息--转发
            request.setAttribute("error", "用户名或密码不正确");
            return "backendlogin";
        }
    }

    @RequestMapping(value="/backend/main")
    public String main(HttpSession session){
        if(session.getAttribute(Constants.USER_SESSION) ==null){
            return "redirect:/manager/login";
        }
        return "backend/main";
    }

    @RequestMapping(value="/logout")
    public String logout(HttpSession session){       //清除 session
        session.removeAttribute(Constants.USER_SESSION);
        return "backendlogin";
    }
}
```

在单元测试中要仔细分析这些接口、类之间的关系，判别接口、函数参数的正确性，并进行验证。

3. 数据结构

检查全局和局部数据结构的定义（如队列、堆栈等）是否能实现模块或方法所要求的功能。例如，某程序中需要实现先来先服务的任务调度，但为此定义的数据结构为栈，这显然是错误的，因为栈用于实现后进者先出。改正的办法是定义一个队列，而不是栈。

在模块功能实现中，局部数据结构是正确实现模块功能的基础，数据结构多，容易出错，因此，在对局部数据结构的测试时，应仔细设计测试用例，重点发现以下问题。

（1）不合适或不相容的类型说明。

（2）变量未初始化。

(3) 变量初始化有错或默认值不正确。

(4) 变量命名不确切。

(5) 数据处理过程中出现越界或地址访问错误。

4. 边界条件

检查各种边界条件发生时程序执行是否仍然正确,包括检查判断条件的边界等。主要检查普通合法数据的处理、普通非法数据的处理、边界值内合法边界数据的处理、边界值外非法边界数据的处理等等。

例如,某程序用于实现将百分制成绩转换为五级计分制成绩,代码如下。

```
public class ScoreException extends Exception
{
    public ScoreException(String msg)
        {  super(msg); } }
public class ScoreToGradeUtil
{  public enum GradeEnum
       {  EXCELLENT,                                    //优秀
          GOOD,                                         //良好
          FAIR,                                         //中等
          PASS,                                         //及格
          FAIL                                          //不及格 }
public static GradeEnum convert(Double score) throws ScoreException
    {  if (score >100 || score <0)
       {  throw new ScoreException("分数输入错误");}
       if (score >90)
       {  return GradeEnum.EXCELLENT;
       }else if (score <90 && score >80)
       {  return GradeEnum.GOOD;
       }else if (score <80 && score >70)
       {  return GradeEnum.FAIR;
       }else if (score <70 && score >60)
       {  return GradeEnum.PASS;
       }else
           return GradeEnum.FAIL;  }  }
```

显然,程序中的判断条件漏掉了相等的情况,例如,当 score =90 时,程序会执行最后一个 else 分支给出 **FAIL** 作为转换结果。改正的办法是在适当的位置加上“ = ”。

5. 独立路径

在模块中应该对每一条独立执行路径进行测试,保证模块中的每条语句至少能够被执行一次,因为程序编写时可能存在疏漏,应对照程序详细设计书的要求对程序进行检查和测试,看是否漏掉了某些原本需要的处理逻辑,也就是少了某些应当有的独立路径,或

者某些独立路径存在处理错误。重点检查内容包括算符优先级、混合类型运算、精度不够、表达式符号、循环条件和死循环。例如，某程序用于实现将百分制成绩转换为五级计分制成绩，代码如下。

```
public static GradeEnum convert(Double score) throws ScoreException{
        if (score>100 || score<0) {
            throw new ScoreException("分数输入错误");}
        if (score>=90) {
            return GradeEnum.EXCELLENT;
        }else if (score<90 && score>=80) {
            return GradeEnum.GOOD;
        }else if (score<70 && score>=60) {
            return GradeEnum.PASS;
        }else
            return GradeEnum.FAIL;} }
```

对照程序详细设计书可以发现，程序漏掉了 score＜80 and score＞＝70 这种情况，当 score＜80 and score＞＝70 时，程序给出转换结果 FAIL，明显不符合逻辑。

6. 异常处理

单元模块应能预见某些代码运行可能出现异常的条件和情况，并设置适当的异常处理代码，以便在相关代码行运行出现异常时，能妥善处理，保证整个单元模块处理逻辑的正确性，这种异常处理应当是模块功能的一部分。

例如，有代码段如下，在 Date 类中有一个成员方法如下。

```
public void set(int y, int m, int d)                    //成员方法.设置日期值
    {
        this.year =y;
        this.month =m;
        this.day =d;
    }
```

程序中，我们发现在使用 set()方法设置日期时，有一些常识没有考虑在内，比如日期不能大于 31 天，月份不能大于 12 月等；因此需要修改程序应对处理一些异常数据的情况。

```
public void set(int year,int month,int day) throws DateFormatException
                                                                    //设置日期
    {
        if (year<=-2000 || year>2500)
         throw new DateFormatException(year+".年份不合适.有效年份为-2000
         ～2500。");
        if (month<1 || month>12)
        throw new DateFormatException(month+"月.月份错误");
```

```
        if (day<1 || day>MyDate.daysOfMonth(year, month))
         throw new DateFormatException(year+"年"+month+"月"+day+"日.日期错
         误");
        this.year =year;
        this.month =month;
        this.day =day;
    }
```

系统应当能对输入数据进行完备性、正确性、规范性或者合理性检查，经验表明，没有对输入数据进行必要和有效的检查，是造成软件系统不稳定或者执行出问题的主要原因之一。下列程序中，设计了一个银行类和主程序类。

```
public class Bank {
    private long balance=10000L;
    public Bank(){
    }
    public void withDraw(long cash) throws InterruptedException{
        if (cash>balance){
            throw new InterruptedException("您的余额不足!");
        }
        this.balance-=cash;
        System.out.println("您的取款金额为: "+cash+";账户余额为"+balance);
    }
}

import java.util.Scanner;
public class MainClass {
    public static void main(String[] args) {
        long cash=0L;
        Scanner in=new Scanner(System.in);
        System.out.print("请输入取款金额: ");
        cash=Long.parseLong(in.nextLine());
        Bank bank=new Bank();
        try{
            bank.withDraw(cash);
        }catch (InterruptedException ie){
            System.out.print(ie.getMessage());
        }
        in.close();
    }
}
```

程序调试中，执行“请输入取款金额时”存在出错的可能，如数据输入不正确等。为此，应设置适当的出错处理代码，以便在相关代码运行出现异常时，能妥善处理。修改后的代码段如下。