



单元测试 Unit Testing

Xu Sihan (徐思涵)
College of Computer Science
Nankai University

*Slides adapted from materials by Prof. Qiang Liu (Tsing Hua University) and
Ivan Marsic (Rugers University)*

数组中最大的子数组之和

- 输入，一个数组，和它的大小
- 输出，这个数组中最大子数组的和
- 例如

输入	输出
[-1, 2, 3, -4]	5
[-1, 2, -5, 3, -4]	3
[-1, 20, -5, 30, -4]	45
[-2, -3, -5, -1, -9]	-1

用类/函数来实现

如果设计了一个类 Class

MSA (Maximum Sub-array Sum)

里面有一个函数

Calc()

如何设计这个函数

输入参数是什么，输出是什么

单元测试—示例1

```
int[] test1 = { 1, -2, 3, 5, -3, 6, 1, -1 };  
msa.Calc(test1, 8);  
Debug.Assert(msa.MaxSumPosition.sum == 12);  
Debug.Assert(msa.MaxSumPosition.start == 2);  
Debug.Assert(msa.MaxSumPosition.end == 6);
```

单元测试—示例2

```
[TestMethod()]  
public void ConstructorTest()  
{  
    string userEmail = "someone@somewhere.com";  
  
    User target = new User(userEmail);  
  
    Assert.IsTrue(target != null);  
}
```

//我们还可以进一步测试E-mail是否的确是保存在User类型中。

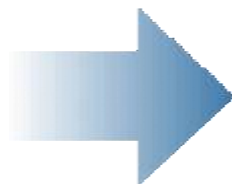
```
[TestMethod()]
[ExpectedException(typeof (ArgumentNullException))]
public void ConstructorTestNull()
{
    User target = new User(null);
}
```

```
[TestMethod()]
[ExpectedException(typeof (ArgumentException))]
public void ConstructorTestEmpty()
{
    User target = new User("");
}
```

```
[TestMethod()]
[ExpectedException(typeof (ArgumentNullException))]
public void ConstructorTestBlank()
{
    User target = new User(" ");
}
```

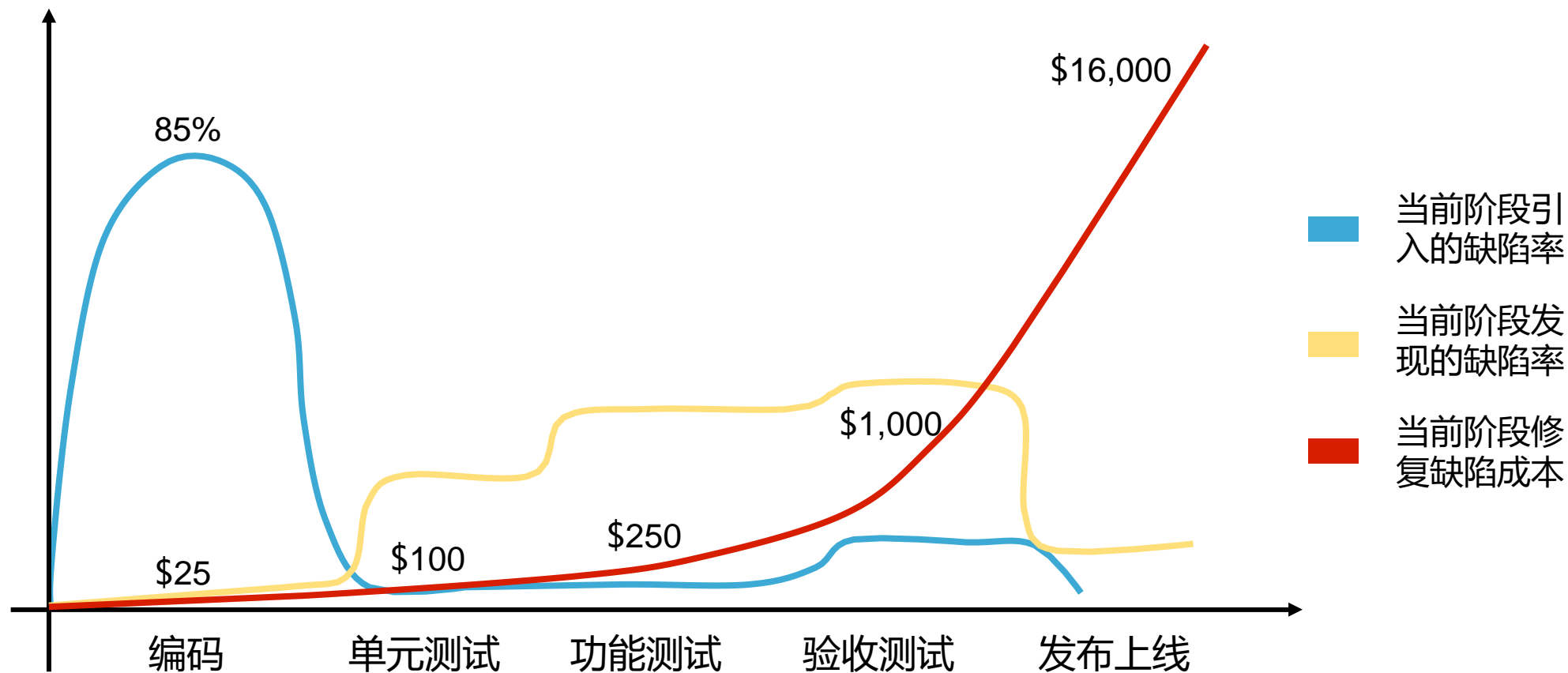


现实的开发问题



- ◆ 大量故障堆积在项目后期，项目后10%的工作占用了项目90%的时间。
- ◆ 故障难以定位，而且飘忽不定，开发和测试人员疲于奔命。

现实的开发问题



单元测试



要使城墙保持坚固，至少应该
保证每一块砖都是好的

单元是构造软件系统的基础，只有使每个单元得到足够的测试，系统的质量才能有可靠的保证，即单元测试是构筑产品质量的基石。

单元测试

单元测试（Unit Testing）是对软件中的最小可测试单元进行检查和验证。

验证
代码

设计
更好

文档化
行为

具有
回归性

好的单元测试

- 检查单元测试是否满足下面的条件
 - 单元测试应该在最基本的功能/参数上验证程序的正确性。
 - 最基本的单元——如在python/C++/C#/Java中的类
 - 单元测试要测试API中的每一个方法及每一个参数
 - 单元测试必须由最熟悉代码的人来写。
 - 单元测试过后，机器状态保持不变。
 - 建了临时的文件或目录，在数据库中创建或修改了记录，销毁

简单版单元测试

要写多少测试用例才够呢？

- Right-BICEP方法：

- **Right** - 结果是否正确？一致性、有序性、区间性、引用、耦合性、存在性（**格式错误、空值、不完整的值、意料之外的值、重复的值等**）
- **Border Condition** - 是否所有的边界条件都是正确的？
- **Inverse Relation**: 能查一下反向关联吗？
- **Cross check**: 能用其他手段交叉检查一下结果吗？
- **Error** - 你是否可以强制错误条件发生？
- **Performance** - 是否满足性能要求？

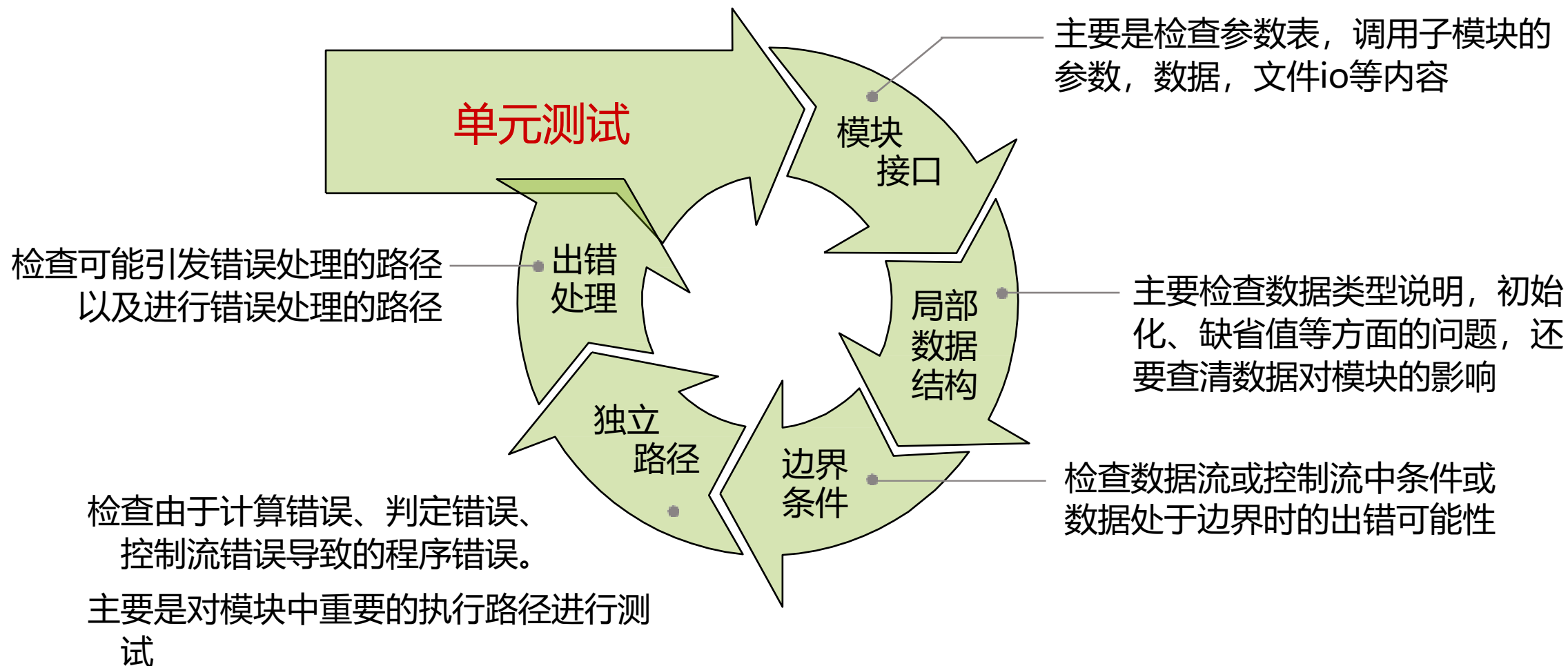
- 看大部分代码是否被覆盖了

内存耗光
磁盘用满
时钟出问题
网络不可用或者有问题
系统过载
受限的调色板
显示分辨率过高或者过低

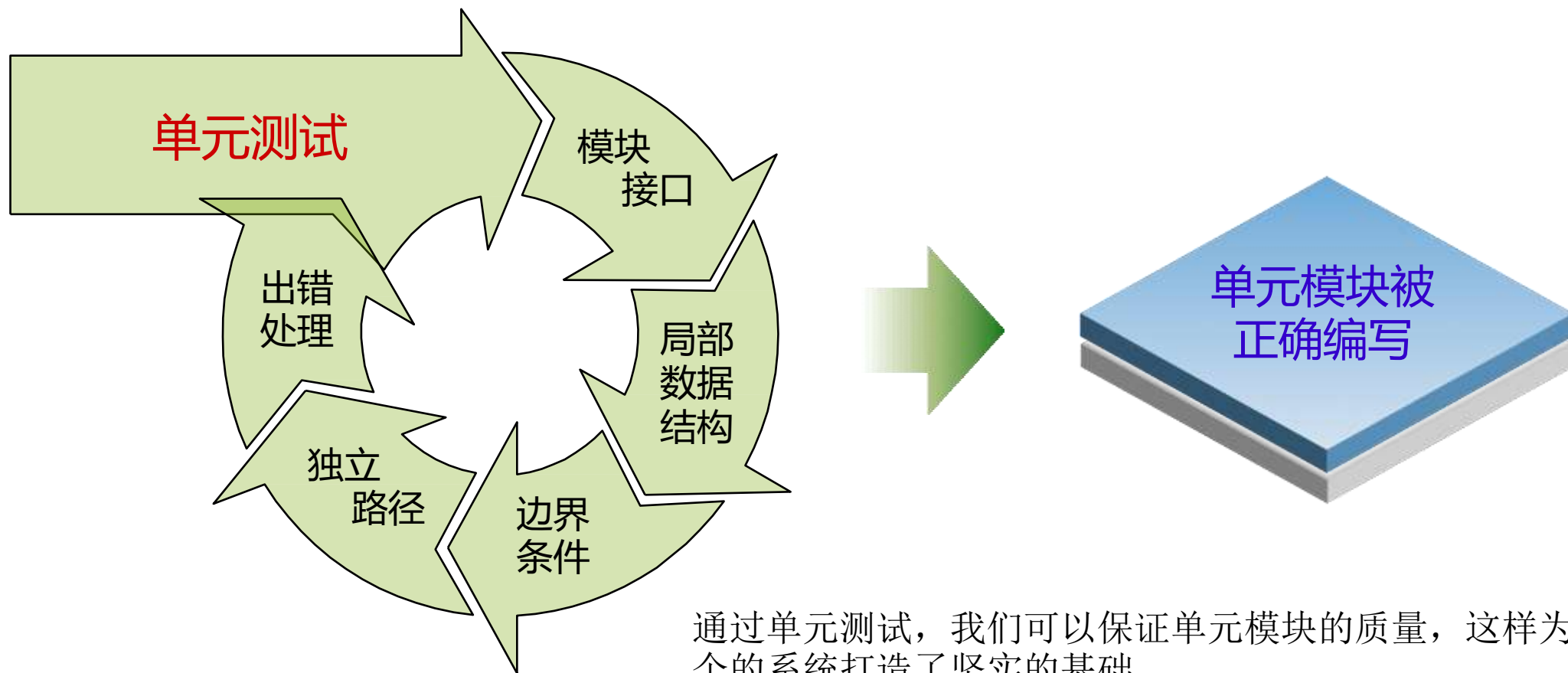


完整版单元测试

如果对模块运行时间有要求的话，还要专门进行关键路径测试，以确定最坏情况下运行的时间和影响模块运行时间的因素



单元测试内容



单元测试原则

快速的

单元测试应能快速运行，如果运行缓慢，就不会愿意频繁运行它。

独立的

单元测试应相互独立，某个测试不应为下一个测试设定条件。当测试相互依赖时，一个没通过就会导致一连串的失败，难以定位问题。

可重复的

单元测试应该是可以重复执行的，并且结果是可以重现的。

自我验证的

单元测试应该有**布尔输出**，无论是通过或失败，不应该查看日志文件或手工对比不同的文本文件来确认测试是否通过。

及时的

及时编写单元测试代码，应恰好在开发实际的单元代码之前。

单元测试质量

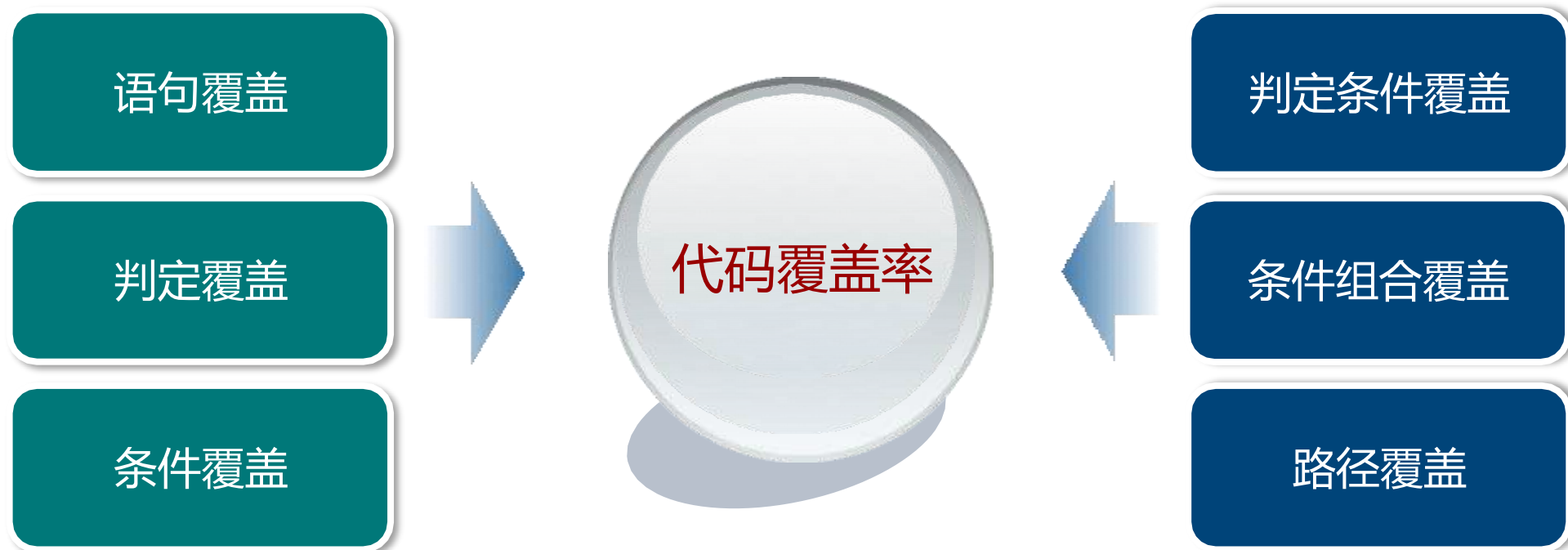
测试
通过率

测试通过率是指在测试过程中执行通过的测试用例所占比例，单元测试通常要求测试用例通过率达到100%。

测试
覆盖率

测试覆盖率是用来度量测试完整性的一个手段，通过覆盖率数据，可以了解测试是否充分以及弱点在哪里。代码覆盖率是单元测试的一个衡量标准，但也不能一味地去追求覆盖率。

单元测试质量



单元测试方法

静态测试：通过人工分析或程序正确性证明的方式来确认程序正确性。



动态测试：通过动态分析和程序测试等方法来检查和确认程序是否有问题。



测试用例的重要性



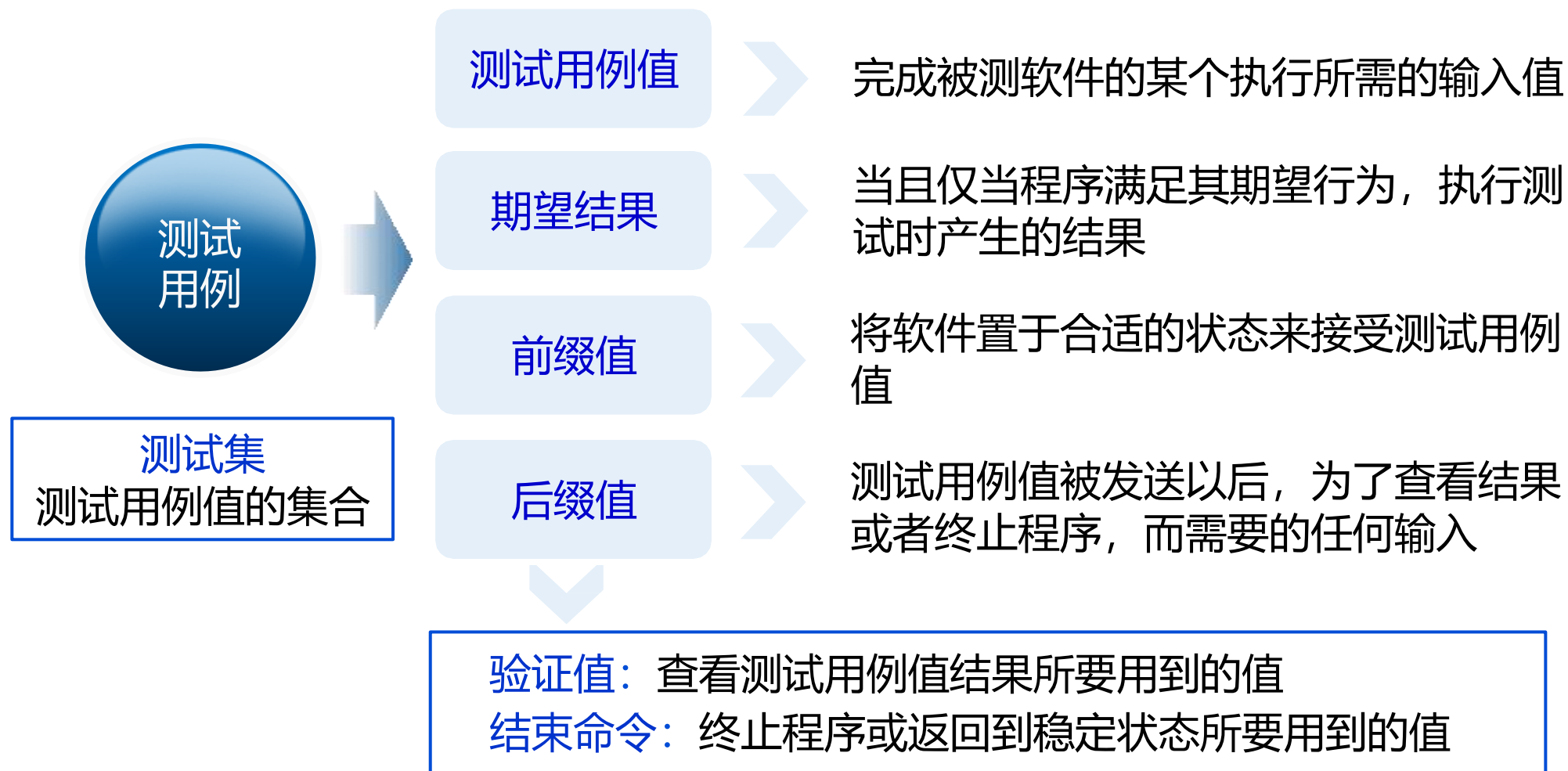
降低软件测试成本

保证测试工作质量

评估和检验测试效果

- 测试用例的设计是测试活动的基础和关键；
- 测试**不可能是完备的**，会受到时间的约束；
- 测试用例可以帮助我们分清先后主次，避免重复和遗漏
- 测试用例的通过率和软件缺陷的数量是检验软件质量的量化标准

测试用例的概念



测试用例的概念



- 测试用例值：电话号码
- 期望结果：接通（或未接通）
- 前缀值：电话开启并进入拨号界面
- 后缀值：按下“呼叫”或“取消”按钮

测试用例设计的要求

测试用例设计

具有代表性和典型性

寻求系统设计和功能设计的弱点

既有正确输入也有错误或异常输入

考虑用户实际的诸多使用场景

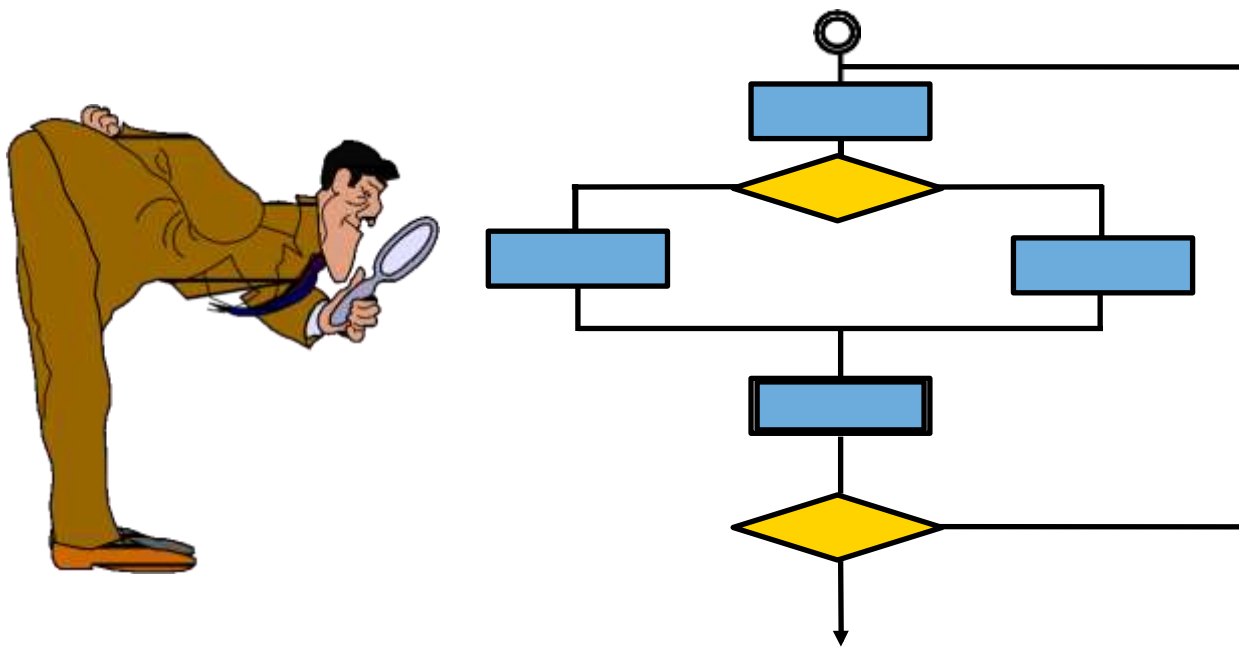
不同类型的测试

测试名称	测试内容
Unit Test	单元测试——在最低的功能/参数上验证程序的正确性
Functional Test	功能测试——验证模块的功能
Integration Test	集成测试——验证几个互相有依赖关系的模块的功能
Scenario Test	场景测试——验证几个模块是否能够完成一个用户场景
System Test	系统测试——对于整个系统功能的测试
Alpha/Beta Test	外部软件测试人员（Alpha/Beta 测试员）在实际用户环境中对软件进行全面的测试

白盒测试方法

单元测试方法

白盒测试 (White Box Testing)：又称结构测试，它把测试对象看做一个透明的盒子，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。



测试覆盖标准

- **测试需求**：测试需求是软件制品的一个特定元素，测试用例必须满足或覆盖这个特定元素。
- **覆盖标准**：一个覆盖标准是一条规则，或者是将测试需求施加在一个测试集上的一组规则。
- **测试覆盖**：给定一个覆盖标准 C 和相关的测试需求集合 TR ，欲使一个测试集合 T 满足 C ，当且仅当对于测试需求集合 TR 中的每一条测试需求 tr ，在 T 中至少存在一个测试 t 可以满足 tr 。
- **覆盖程度**：给定一个测试需求集合 TR 和一个测试集合 T ，覆盖程度就是 T 满足的测试需求数占 TR 总数的比例。

测试覆盖标准



软心糖豆：6 种口味和 4 种颜色

柠檬味（黄色）、开心果味（绿色）、梨子味（白色）
哈密瓜味（橙色）、橘子味（橙色）、杏味（黄色）

- ✧ 可以用什么覆盖标准来选择糖豆测试？
- ✧ 哪一种覆盖标准更好？为什么？
- ✧ 应该考虑哪些因素来选择覆盖标准？



第一处理测试需求的满意程度。第二是生成测试用例的难易程度。第三是用测试发现缺陷的能力

白盒测试技术

白盒测试是将测试对象看做一个透明的盒子，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。

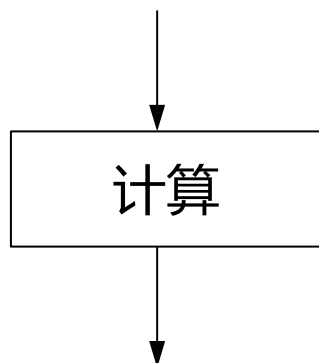


控制流图

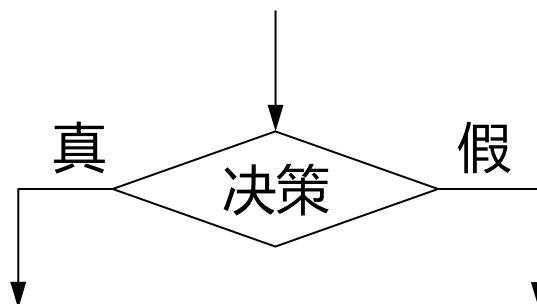
控制流图 (CFG, Control Flow Graph) 是一个过程或程序的抽象表示。

控制流图的基本符号：

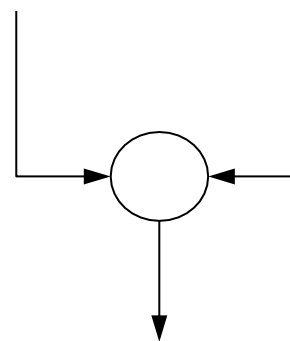
- 矩形代表了连续的顺序计算，也称基本块
- 节点是语句或语句的一部分，边表示语句的控制流



顺序计算



判断节点



合并节点

```

FindMean(float *mean, FILE *fp)
{
    float sum = 0.0, score = 0.0;
    int num = 0;

    fscanf(fp, "%f", &score); /* Read and parse into score */
    while (!EOF(fp)) {
        if (score > 0.0) {
            sum += score;
            num++;
        }
        fscanf(fp, "%f", &score);
    }
    /* Compute the mean and print the result */
    if (num > 0) {
        *mean = sum/num;
        printf("The mean score is %f \n", mean);
    } else
        printf("No scores found in file\n");
}

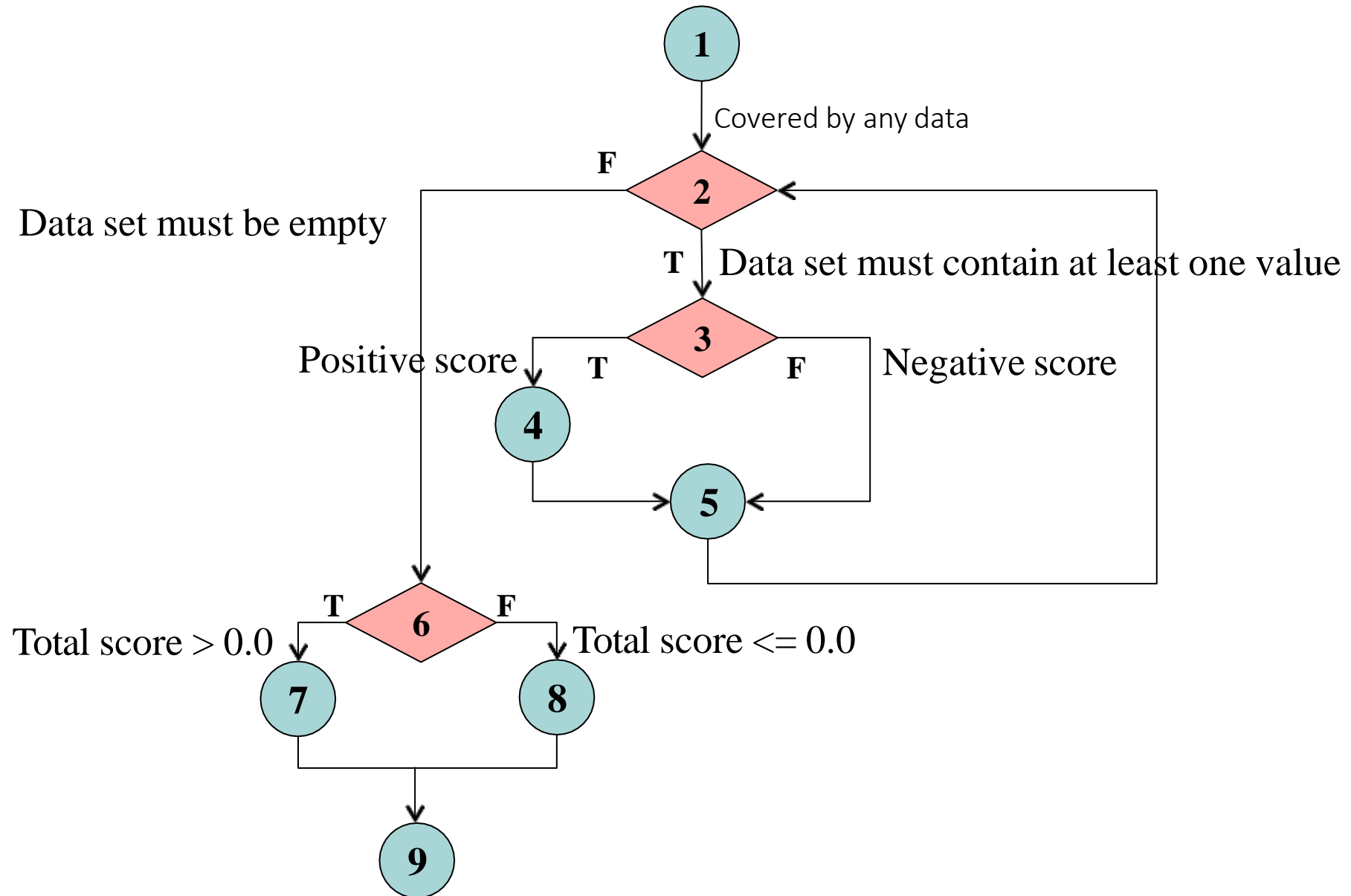
```

FindMean(float *mean, FILE *fp)

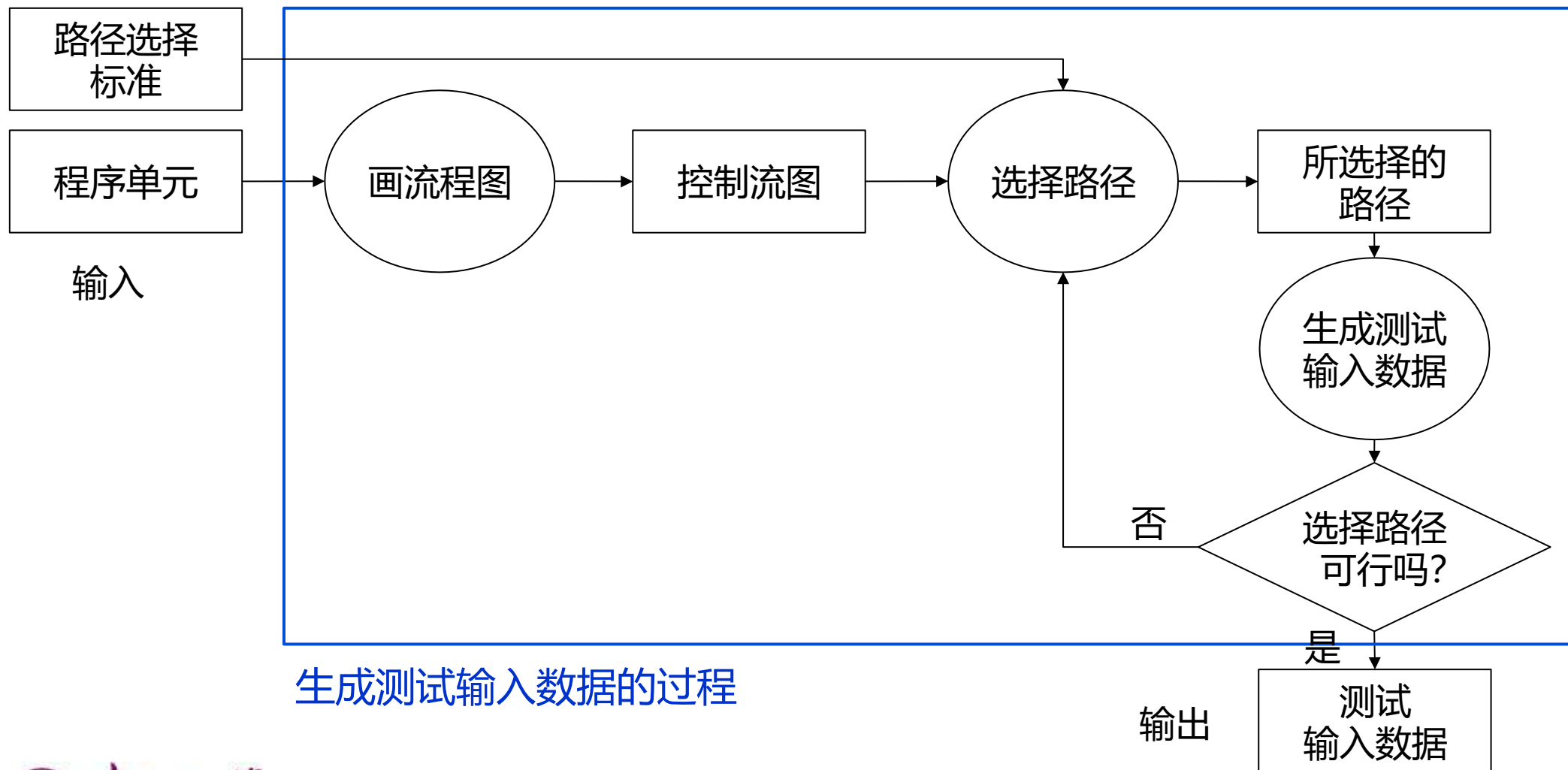
```
① float sum = 0.0, score = 0.0;
   int num = 0;

   fscanf(fp, "%f", &score); /* Read and parse into score */
② while (!EOF(fp)) {
    ③ if (score > 0.0) {
        ④ sum += score;
        num++;
    }
    ⑤ fscanf(fp, "%f", &score);
}
/* Compute the mean and print the result */
⑥ if (num > 0) {
    ⑦ *mean = sum/num;
    printf("The mean score is %f \n", mean);
} else
    ⑧ printf("No scores found in file\n");
}
```

基本块



基于控制流的测试



生成测试输入数据的过程

输出

测试
输入数据

代码覆盖标准

代码覆盖率描述的是代码被测试的比例和程度，通过代码覆盖率可以得知哪些代码没有被覆盖，从而进一步补足测试用例。



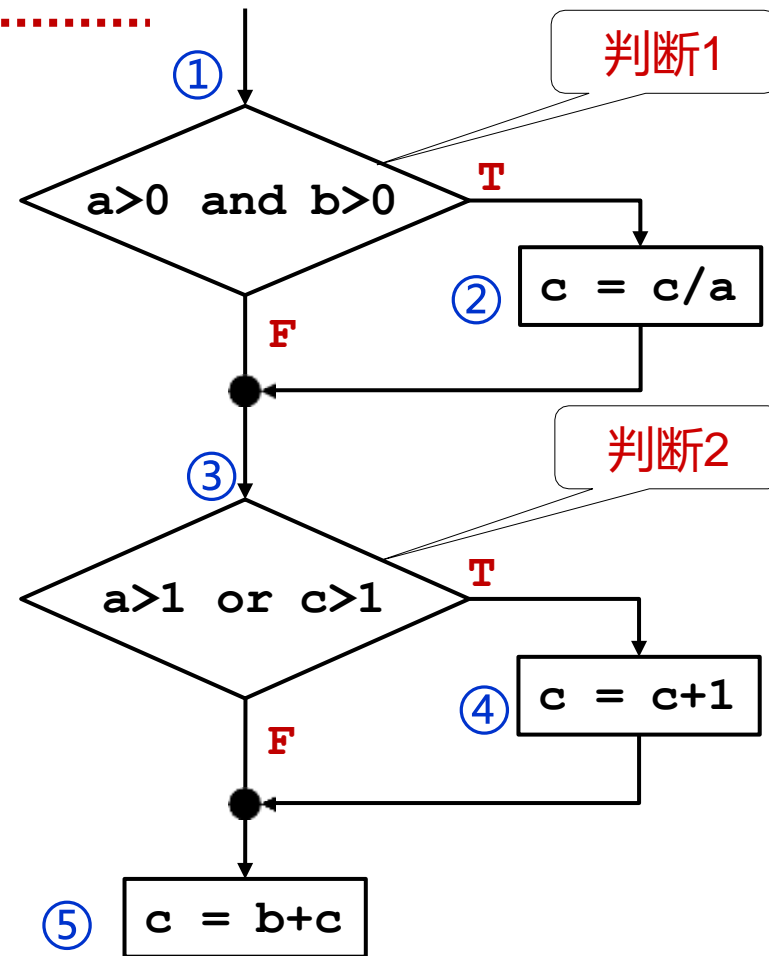
示例描述

```
double func1(int a, int b, double c)
{
    if (a>0 && b>0) {
        c = c/a;
    }

    if (a>1 || c>1) {
        c = c+1;
    }

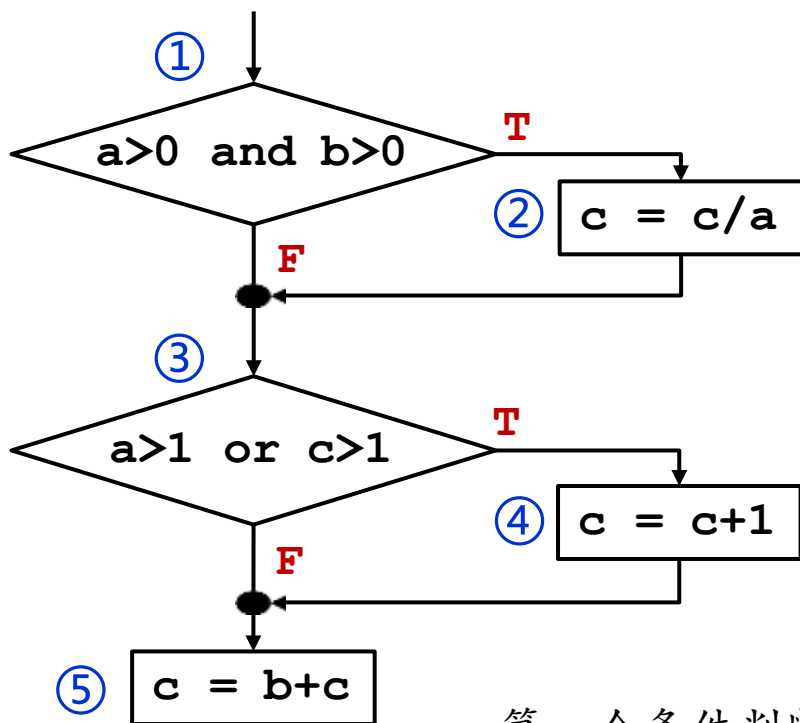
    c = b+c;

    return c;
}
```



语句覆盖

程序中的每个可执行语句至少被执行一次。



输入: $a=2, b=1, c=6$

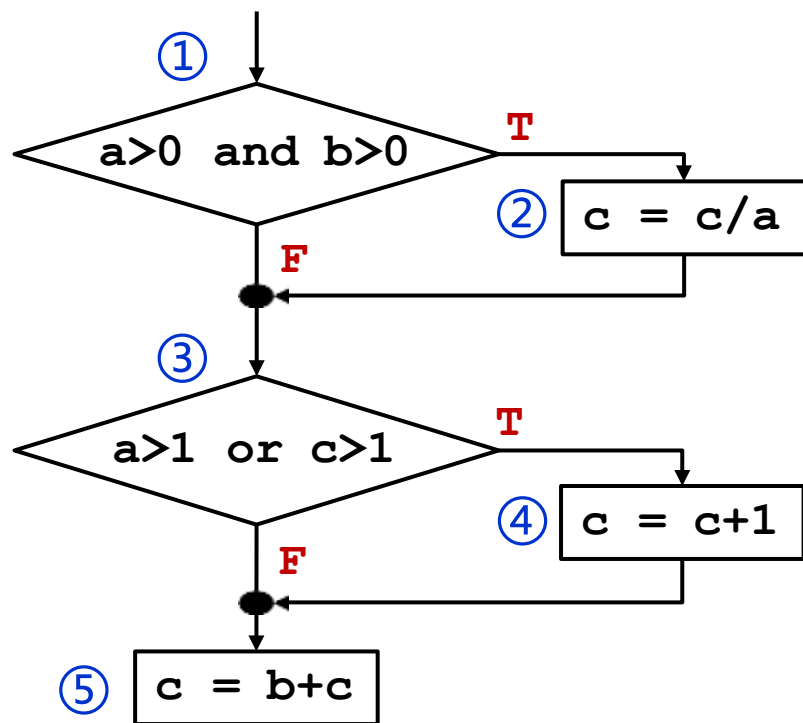
第一个条件判断中与被错误地写成或, 显然这个测试用例是无法检查出来的

语句覆盖

- 测试用例：
- 输入：a=2, b=1, c=6
- 程序中三个可执行语句均被执行一次，满足语句覆盖标准。
- 问题分析：
- 测试用例虽然覆盖可执行语句，但无法检查判断逻辑是否存在问题，例如 第一个条件判断中 “&&” 被错误地写成 “||”。
- 语句覆盖是最弱的逻辑覆盖准则。

判定覆盖（分支覆盖）

程序中每个判断的取真和取假分支至少经历一次，即判断真假值均被满足。



输入: $a=2, b=1, c=6$

输入: $a=-2, b=-1, c=-3$

判定覆盖（分支覆盖）

程序中每个判断的取真和取假分支至少经历一次，即判断真假值均被满足。

测试用例：

输入：a=2, b=1, c=6, 覆盖判断1的 T分支和判断2的 T分支

输入：a=-2, b=-1, c=-3, 覆盖判断1的 F分支和判断2的 F分支

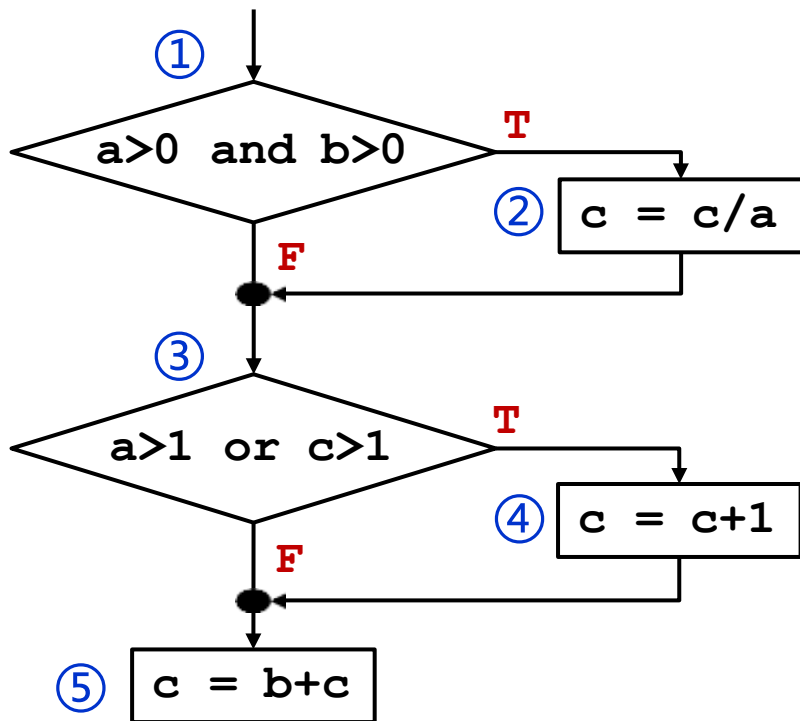
问题分析：

由于大部分判定语句是由多个逻辑条件组合而成，若仅判断其整个最终结果，而忽略每个条件的取值情况，必然会遗漏部分测试路径。

判定覆盖具有比语句覆盖更强的测试能力，但仍是弱的逻辑覆盖。

条件覆盖

每个判断中每个条件的可能取值至少满足一次。



输入: $a=2, b=-1, c=-2$

覆盖 $a > 0, b \leq 0, a > 1, c \leq 1$ 条件

输入: $a=-1, b=2, c=3$

覆盖 $a \leq 0, b > 0, a \leq 1, c > 1$ 条件

条件覆盖

每个判断中每个条件的可能取值至少满足一次。

测试用例：

输入： $a=2, b=-1, c=-2$ ，覆盖 $a>0, b\leq 0, a>1, c\leq 1$ 条件

输入： $a=-1, b=2, c=3$ ，覆盖 $a\leq 0, b>0, a\leq 1, c>1$ 条件

问题分析：

条件覆盖不一定包含判定覆盖，例如上面测试用例就没有覆盖判断1的T分支和判断2的F分支。

条件覆盖只能保证每个条件至少有一次为真，而没有考虑整个判定结果。

判定条件覆盖

判断中所有条件的可能取值至少执行一次，且所有判断的可能结果至少执行一次。

测试用例：

输入： $a=2, b=1, c=6$ ，覆盖 $a>0, b>0, a>1, c>1$ 且判断均为T

输入： $a=-1, b=-2, c=-3$ ，覆盖 $a\leq 0, b\leq 0, a\leq 1, c\leq 1$ 且判断均为F

问题分析：

判定条件覆盖能够同时满足判定、条件两种覆盖标准。

没有考虑条件的各种组合情况。

条件组合覆盖

判断中每个条件的所有可能取值组合至少执行一次，并且每个判断本身的结果也至少执行一次。

组合编号	覆盖条件取值	判定条件取值	判定-条件组合
1	$a > 0, b > 0$	判断1取T	$a > 0, b > 0$, 判断1取T
2	$a > 0, b \leq 0$	判断1取F	$a > 0, b \leq 0$, 判断1取F
3	$a \leq 0, b > 0$	判断1取F	$a \leq 0, b > 0$, 判断1取F
4	$a \leq 0, b \leq 0$	判断1取F	$a \leq 0, b \leq 0$, 判断1取F
5	$a > 1, c > 1$	判断2取T	$a > 1, c > 1$, 判断2取T
6	$a > 1, c \leq 1$	判断2取T	$a > 1, c \leq 1$, 判断2取T
7	$a \leq 1, c > 1$	判断2取T	$a \leq 1, c > 1$, 判断2取T
8	$a \leq 1, c \leq 1$	判断2取F	$a \leq 1, c \leq 1$, 判断2取F

条件组合覆盖

测试用例	覆盖条件	覆盖路径	覆盖组合
输入: $a=2, b=1, c=6$	$a>0, b>0$ $a>1, c>1$	1-2-4	1, 5
输入: $a=2, b=-1, c=-2$	$a>0, b\leq 0$ $a>1, c\leq 1$	1-3-4	2, 6
输入: $a=-1, b=2, c=3$	$a\leq 0, b>0$ $a\leq 1, c>1$	1-3-4	3, 7
输入: $a=-1, b=-2, c=-3$	$a\leq 0, b\leq 0$ $a\leq 1, c\leq 1$	1-3-5	4, 8

问题分析:

条件组合覆盖准则满足判定覆盖、条件覆盖和判定条件覆盖准则。

覆盖了所有组合，但覆盖路径有限，上面示例中1-2-5没覆盖。

路径覆盖

覆盖程序中的所有可能的执行路径。

测试用例	覆盖条件	覆盖路径	覆盖组合
输入: $a=2, b=1, c=6$	$a>0, b>0$ $a>1, c>1$	1-2-4	1, 5
输入: $a=1, b=1, c=-3$	$a>0, b>0$ $a\leq 1, c\leq 1$	1-2-5	1, 8
输入: $a=-1, b=2, c=3$	$a\leq 0, b>0$ $a\leq 1, c>1$	1-3-4	3, 7
输入: $a=-1, b=-2, c=-3$	$a\leq 0, b\leq 0$ $a\leq 1, c\leq 1$	1-3-5	4, 8

路径覆盖

问题分析：前面的测试用例完全覆盖所有路径，但没有覆盖所有条件组合。
下面结合条件组合和路径覆盖两种方法重新设计测试用例：

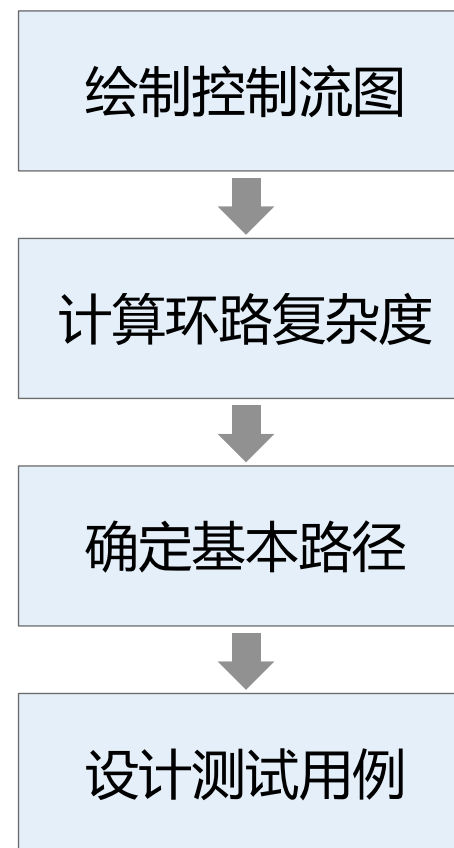
测试用例	覆盖条件	覆盖路径	覆盖组合
输入：a=2, b=1, c=6	a>0, b>0 a>1, c>1	1-2-4	1, 5
输入：a=1, b=1, c=-3	a>0, b>0 a<=1, c<=1	1-2-5	1, 8
输入：a=2, b=-1, c=-2	a>0, b<=0 a>1, c<=1	1-3-4	2, 6
输入：a=-1, b=2, c=3	a<=0, b>0 a<=1, c>1	1-3-4	3, 7
输入：a=-1, b=-2, c=-3	a<=0, b<=0 a<=1, c<=1	1-3-5	4, 8

如何看待测试覆盖率

- 覆盖率数据只能代表测试过哪些代码，不能代表是否测试好这些代码。
- 较低的测试覆盖率能说明所做的测试还不够，但反之不成立。
- 路径覆盖 > 判定覆盖 > 语句覆盖
- 测试人员不能盲目追求代码覆盖率，而应该想办法设计更好的测试用例。
- 测试覆盖率应达到多少需要考虑软件整体的覆盖率情况以及测试成本。

基本路径测试

基本路径测试是在程序控制流图基础上，通过分析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。

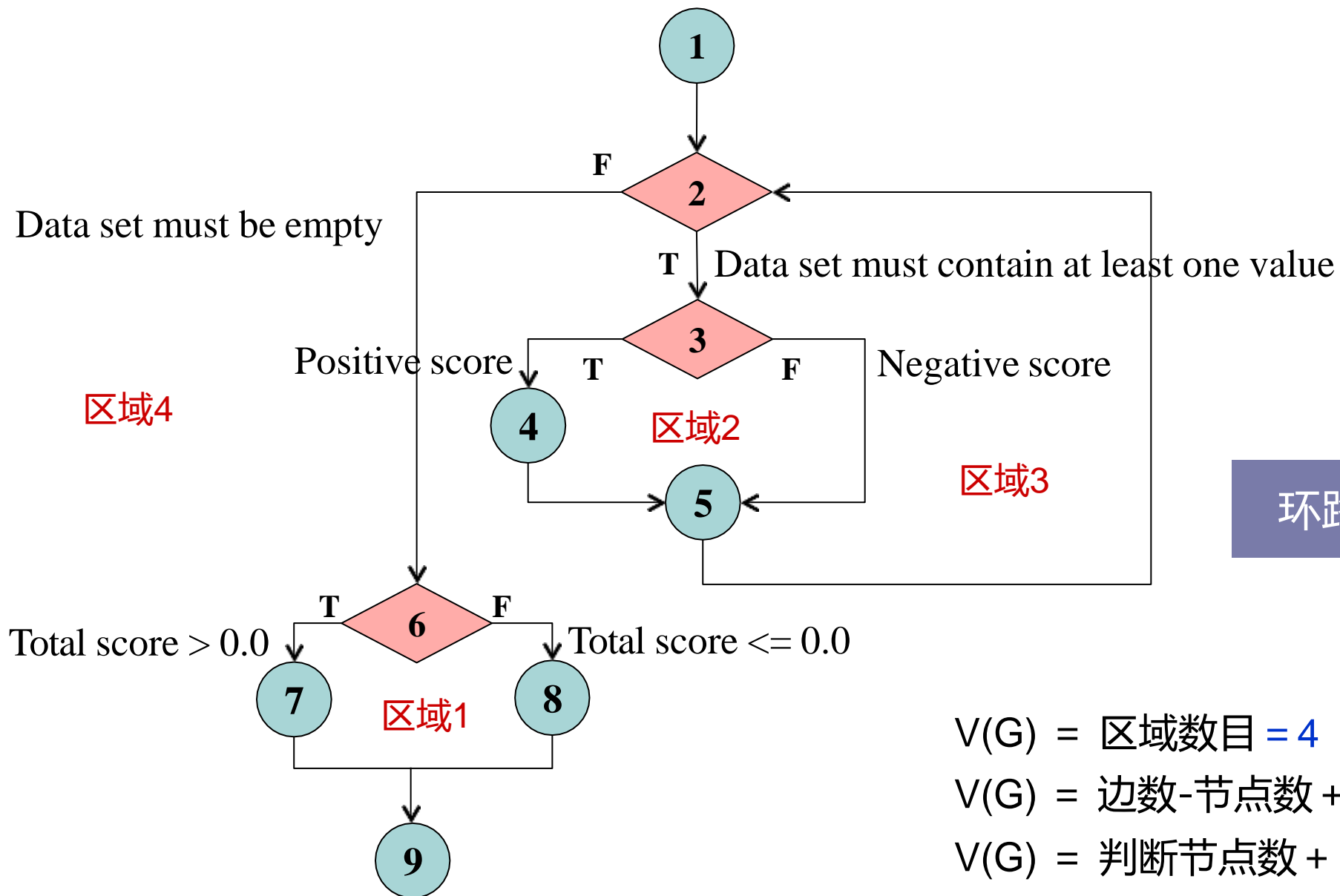



```

FindMean(float *mean, FILE *fp)
{
    float sum = 0.0, score = 0.0;
    int num = 0;

    fscanf(fp, "%f", &score); /* Read and parse into score */
    while (!EOF(fp)) {
        if (score > 0.0) {
            sum += score;
            num++;
        }
        fscanf(fp, "%f", &score);
    }
    /* Compute the mean and print the result */
    if (num > 0) {
        *mean = sum/num;
        printf("The mean score is %f \n", mean);
    } else
        printf("No scores found in file\n");
}

```



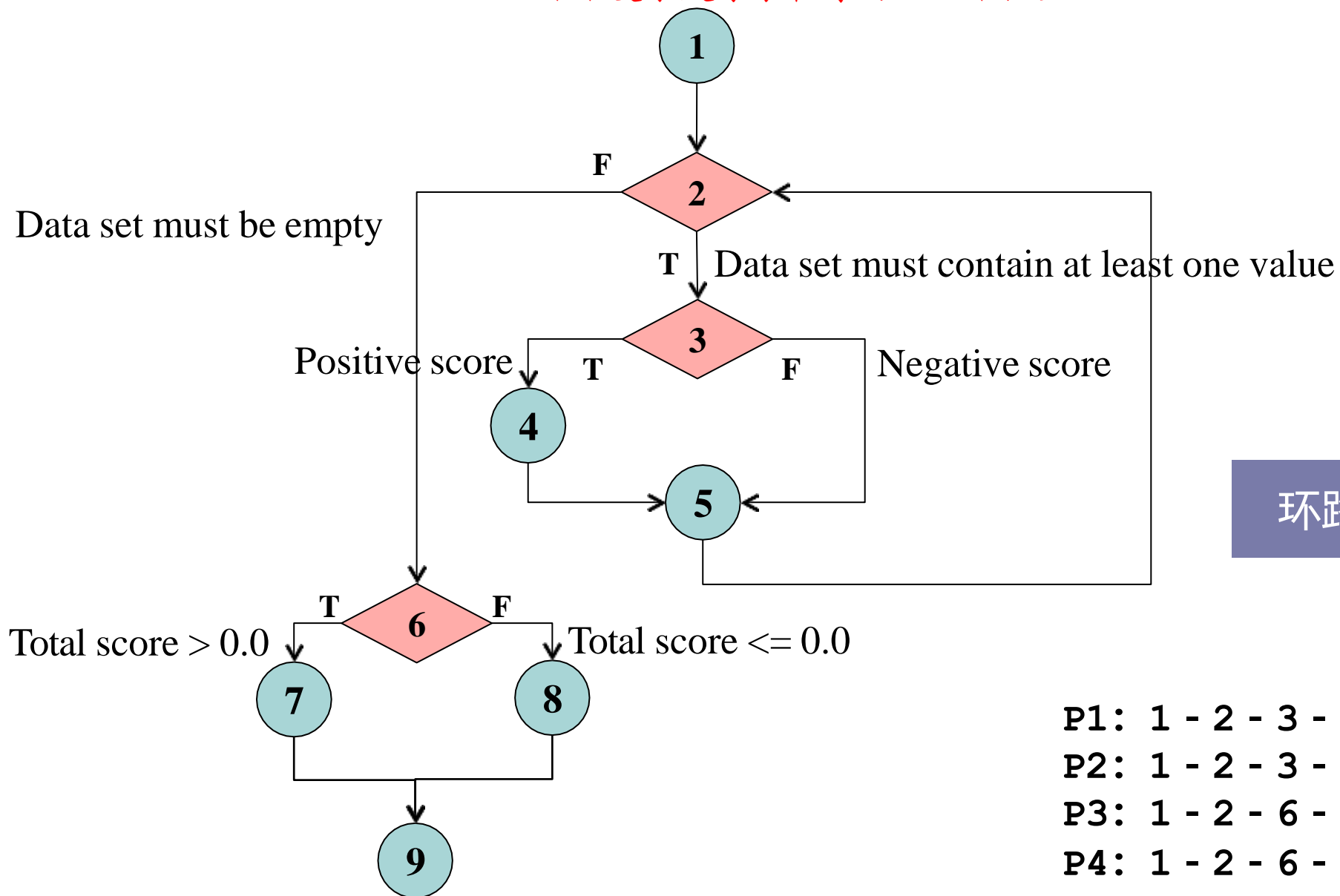
环路复杂性: 4

$$V(G) = \text{区域数目} = 4$$

$$V(G) = \text{边数} - \text{节点数} + 2 = 11 - 9 + 2 = 4$$

$$V(G) = \text{判断节点数} + 1 = 3 + 1 = 4$$

基本独立路径集：一条独立路径和其他独立路径相比，至少要包含一条新的边
环路复杂度等于程序的独立路径数



环路复杂性：4

P1: 1 - 2 - 3 - 4 - 5 - 2 - ...

P2: 1 - 2 - 3 - 5 - 2 - ...

P3: 1 - 2 - 6 - 7 - 9

P4: 1 - 2 - 6 - 8 - 9

示例：基本路径测试

选择合适的输入数据，形成测试用例集

- ① 输入：fp≠NULL，文件有数据{60.0,100.0,-1.0,110.0}，覆盖路径P1
输出：打印信息 "The mean score is 90.0"
- ② 输入：fp≠NULL，文件有数据{0.0,-1.0,75.0}，覆盖路径P2
输出：打印信息 "The mean score is 75.0"
- ③ 输入：fp≠NULL，文件无任何数据，覆盖路径P3
输出：该路径不可达
- ④ 输入：fp≠NULL，文件无任何数据，覆盖路径P4
输出：打印信息 "No scores found in file"

有一些独立路径往往不是孤立的，它是正常控制流的一部分，比如P3就是p1和2的一部分

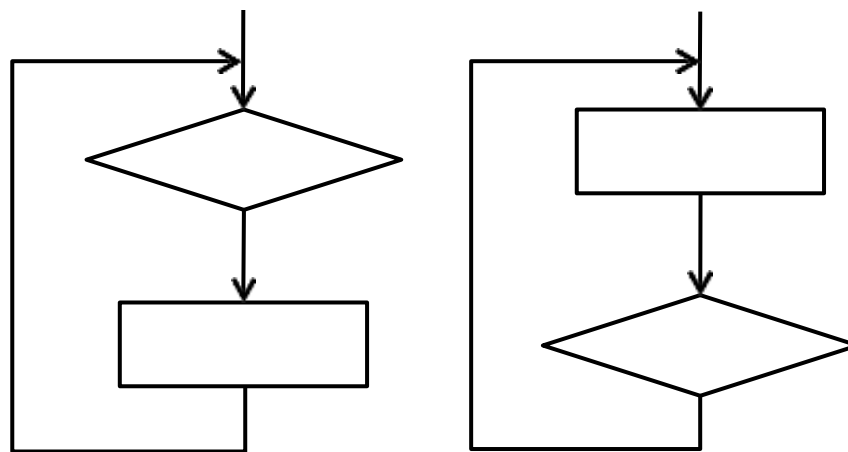
循环测试

循环测试

- 目的：检查循环结构的有效性
- 类型：简单循环、嵌套循环、串接循环和非结构循环

简单循环（次数为 n ）

- 完全跳过循环
- 只循环 1 次
- 只循环 2 次
- 循环 m ($m < n$) 次
- 分别循环 $n-1, n, n+1$ 次



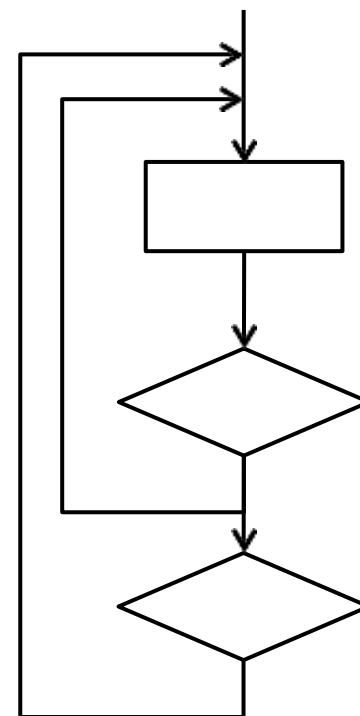
循环测试

嵌套循环

- 从最内层循环开始，所有外层循环次数设为最小值；
- 对最内层循环按照简单循环方法进行测试；
- 由内向外进行下一个循环的测试，本层循环的所有外层循环仍取最小值，而由本层循环嵌套的循环取某些“典型”值；
- 重复上一步的过程，直到测试完所有循环。

串接循环

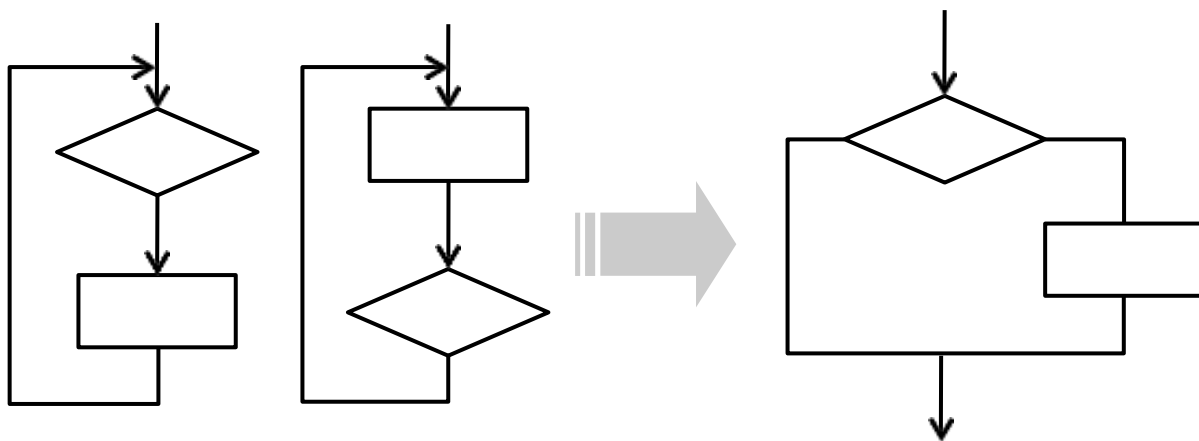
- 独立循环：分别采用简单循环的测试方法；
- 依赖性循环：采用嵌套循环的测试方法。



循环测试

Z路径覆盖下的循环测试

- 这是路径覆盖的一种变体，将程序中的**循环结构简化为选择结构**的一种路径覆盖。
- 循环简化的目的是限制循环的次数，无论循环的形式和循环体实际执行的次数，简化后的循环测试**只考虑执行循环体一次和零次（不执行）**两种情况。



在循环简化的思路下，循环与判定分支的效果是一样的，即循环要么执行、要么跳过。

白盒测试是根据程序的（ ）来设计测试用例。

提交

- ☐ A 功能
- ☐ B 性能
- ☒ C 内部逻辑
- ☐ D 内部数据

关于测试覆盖率，下面的（ ）说法是错误的。

提交

- ☒ A 测试覆盖率是度量代码质量的一种手段
- ☐ B 测试覆盖率是度量测试完整性的一种手段
- ☐ C 测试覆盖率意味着有多少代码经过测试
- ☐ D 不要盲目地追求100%测试覆盖率

在下面列举的测试覆盖中，（ ）是最强的逻辑覆盖准则。

- ☐ A 语句覆盖
- ☐ B 条件覆盖
- ☐ C 判定覆盖
- ☒ D 条件组合覆盖

提交

条件覆盖要求（ ）。

提交

- ☒ A 每个判定中每个条件的所有取值至少满足一次
- ☐ B 每个判定至少取得一次“真”值和一次“假”值
- ☐ C 每个判定中每个条件的所有可能取值组合至少满足一次
- ☐ D 每个可执行语句至少执行一次

一个判定中的复合条件表达式为 $(A > 2) \text{ or } (B \leq 1)$ ，为了达到100%条件覆盖率，至少需要设计（ ）测试用例。

提交

- ☐ A 1
- ☒ B 2
- ☐ C 3
- ☐ D 4

() 要求每个判定中所有条件的可能取值至少执行一次，而且每个判定的可能结果也至少执行一次。

提交

A

判定覆盖

B

条件覆盖

C

判定条件覆盖

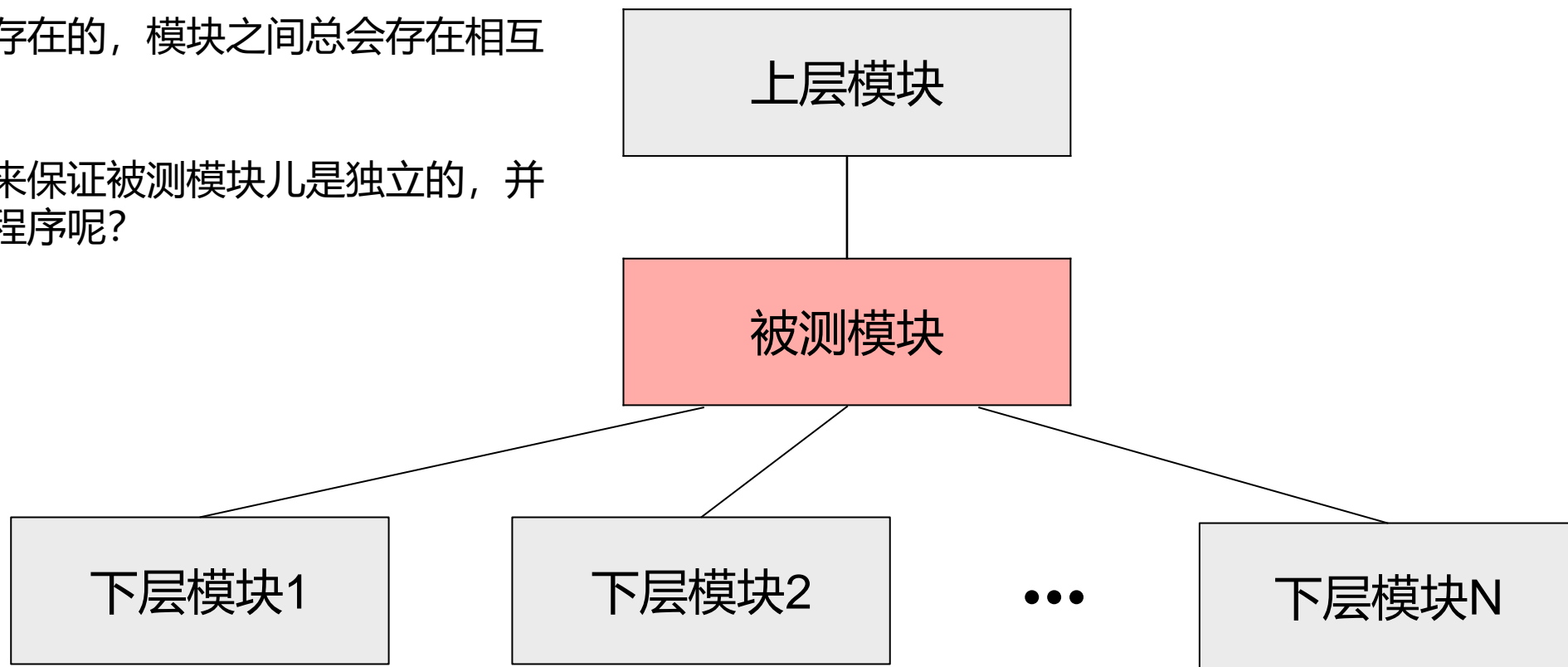
D

条件组合覆盖

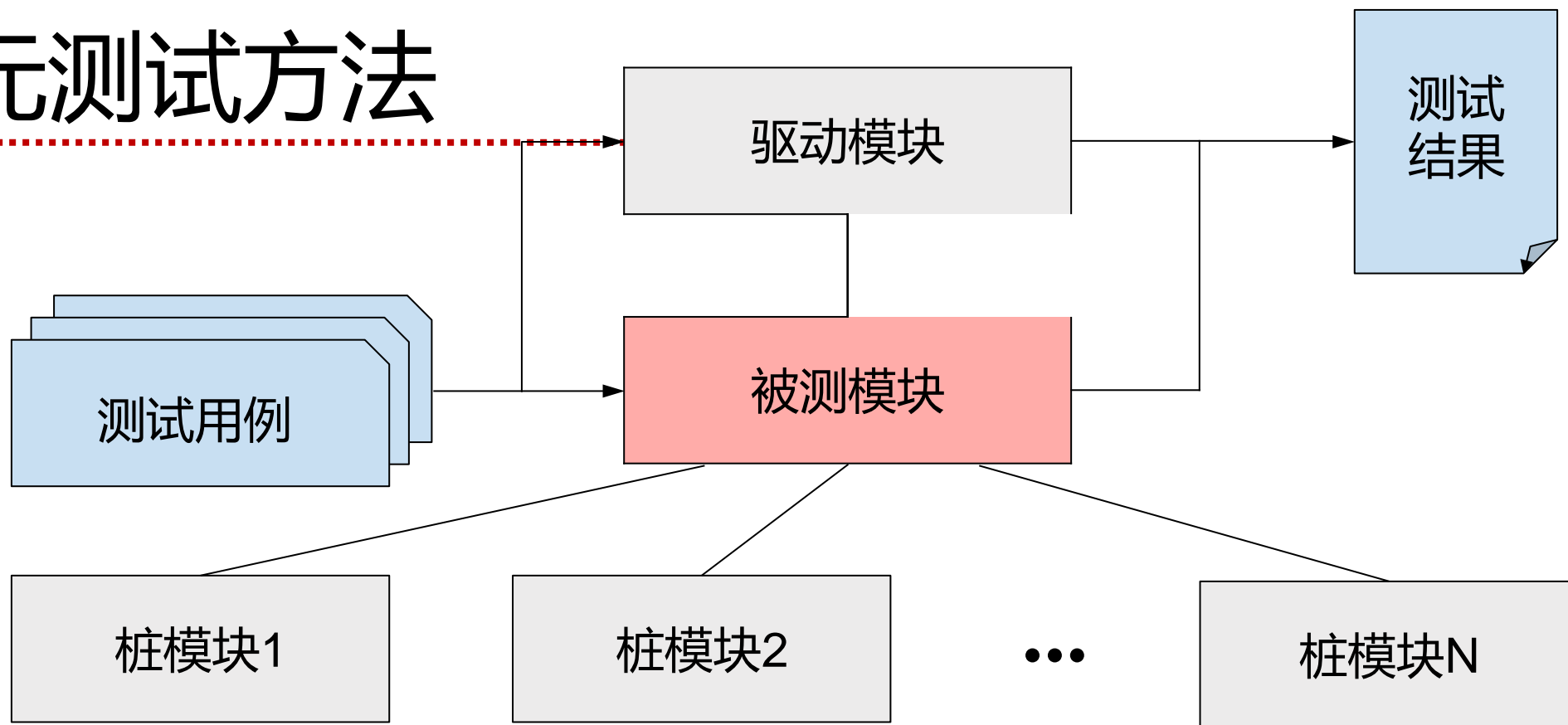
单元测试方法

一个单元模块并不是独立存在的，模块之间总会存在相互调用的关系，

在单元测试的时候，如何来保证被测模块是独立的，并且能够构成一个可运行的程序呢？



单元测试方法

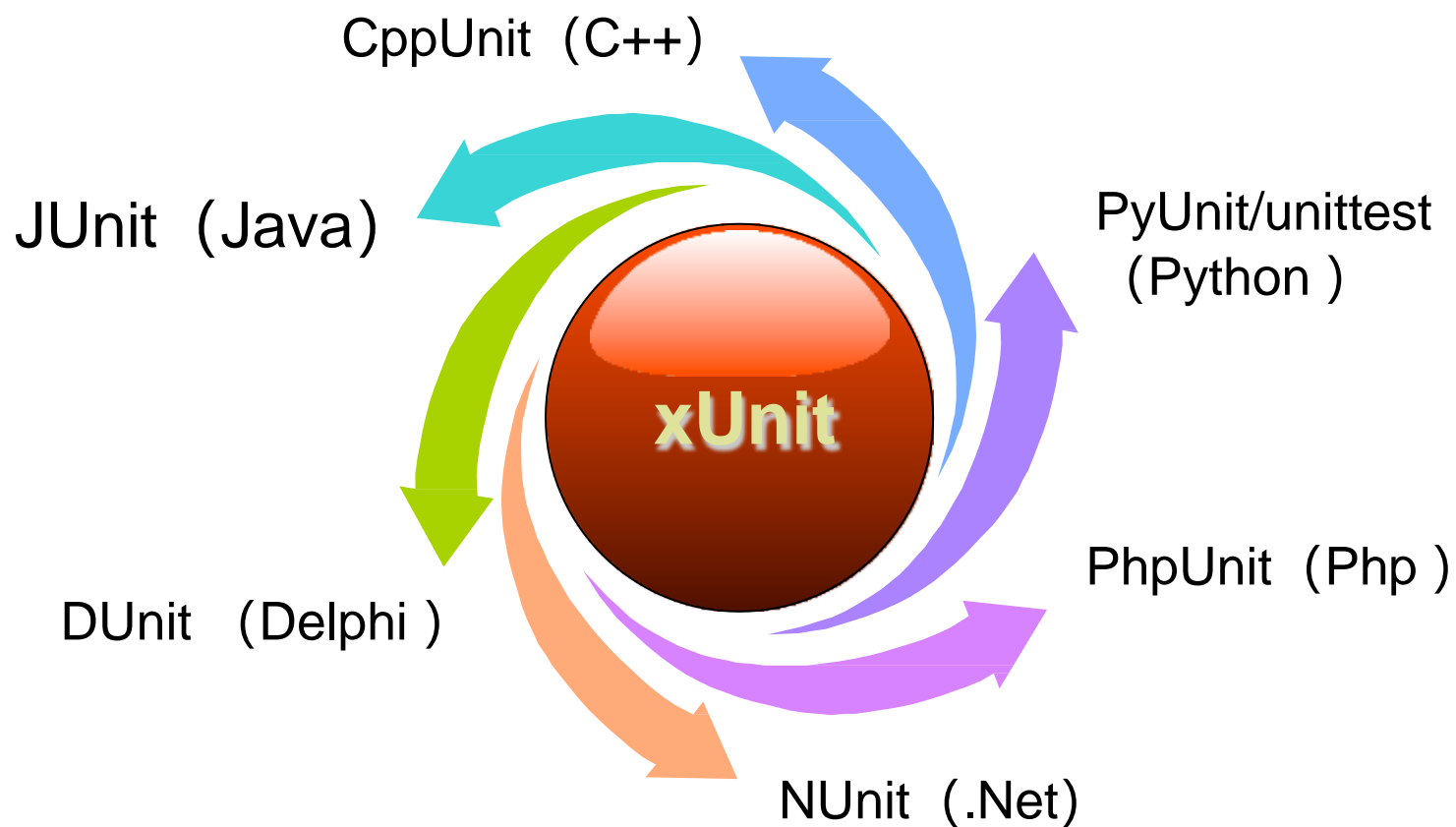


开发驱动模块和桩模块将被测模块进行隔离，并帮助完成单元测试。

驱动模块用于替代上层的模块，他去调用被测模块，并且判断被测模块的返回值是否与测试用例的预期结果相符。

桩模块则用于替代下层的调用模块，他要模拟的返回所替代模块儿的各种可能的返回值。

单元测试之xUnit



单元测试一般需要借助工具来编写测试代码，目前最流行的是针对不同编程语言的unit系列工具，大家可以在以后的开发中自己学习和使用。

单元测试之xUnit

xUnit 通常适用于以下场景的测试

- 单个函数、一个类或者几个功能相关类的测试
- 尤其适用于纯函数测试或者接口级别的测试



xUnit 无法适用于复杂场景的测试

- 被测对象依赖关系复杂，甚至无法简单创建出这个对象
- 对于一些失败场景的测试
- 被测对象中涉及多线程合作
- 被测对象通过消息与外界交互的场景



单元测试之Mock

• **Mock测试**是在测试过程中对于某些不容易构造或者不容易获取的对象，用一个 虚拟的对象（即Mock对象）来创建以便测试的方法。

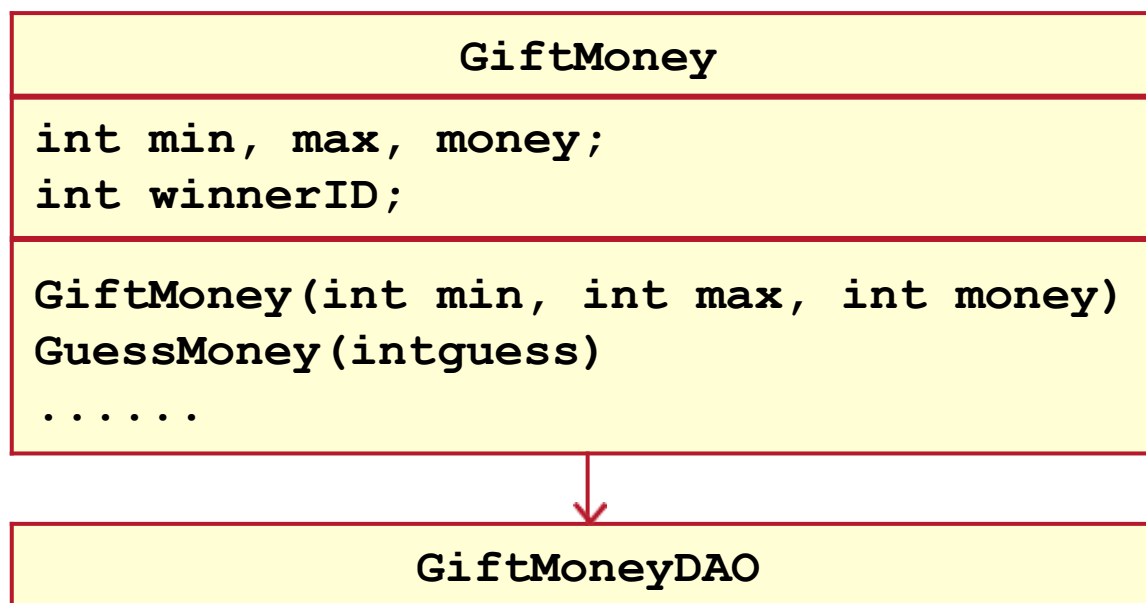


- 真实对象具有不可确定的行为（产生不可预测的结果）
- 真实对象很难被创建（如具体的Web容器）
- 真实对象的某些行为很难触发（如网络错误）
- 真实情况令程序的运行速度很慢
- 真实对象有用户界面
- 测试需要询问真实对象它是如何被调用的
- 真实对象实际上并不存在

单元测试之Mock

举例：支付宝接龙红包通过猜金额的小游戏方式，实现朋友之间的互动并领取春节红包。这种情况应如何测试？

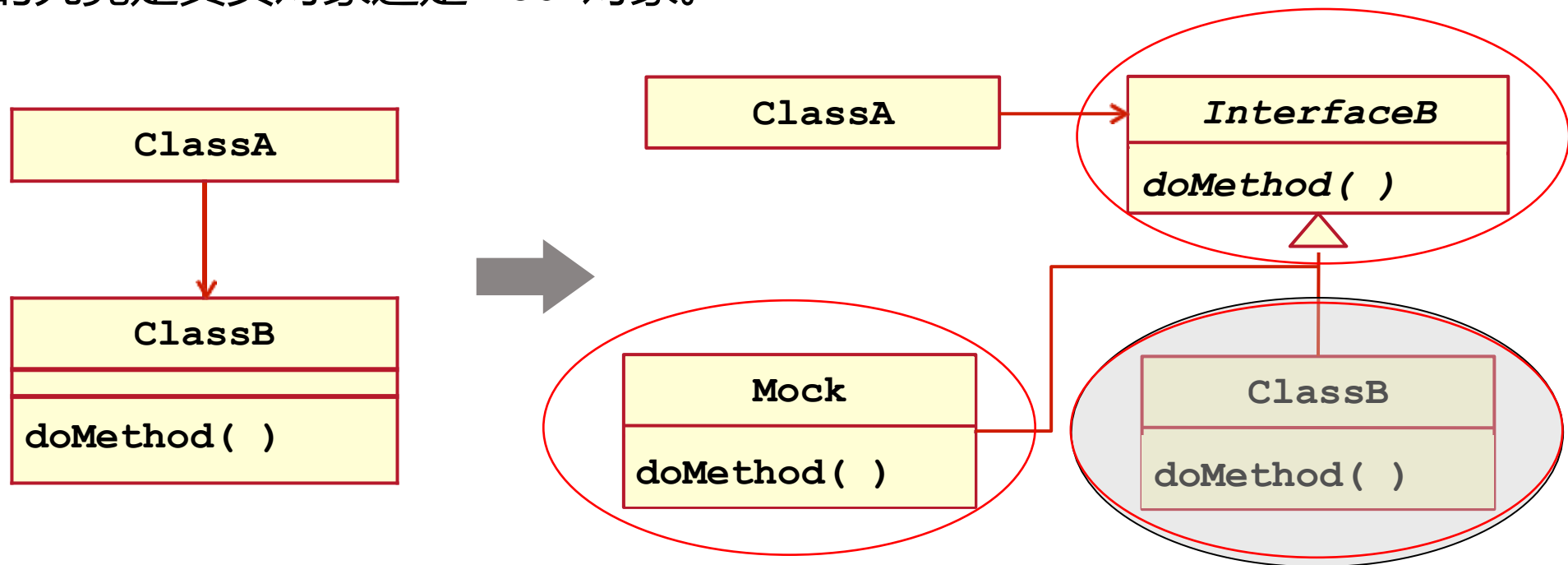
新春红包



Gift money这类允许设定红包的金额和上下限，并且将所差的金额和设定金额进行比较，由于数据是存放在数据库中的，所以这个类就需要调用数据库存取访问的方法，那么这种情况下应该如何进行测试？

单元测试之Mock

关键：需要应用**针对接口的编程技术**，即被测试的代码通过接口来引用对象，再使用Mock对象模拟所引用的对象及其行为，因此被测试模块并不知道它所引用的究竟是真实对象还是Mock对象。



下面的（ ）是错误的。

提交

- ☐ A 静态测试是不运行被测程序，仅通过检查和阅读等手段来发现程序中的错误
- ☐ B 动态测试是实际运行被测程序，通过检查运行的结果来发现程序中的错误
- ☐ C 动态测试可能是黑盒测试，也可能是白盒测试
- ☒ D 白盒测试是静态测试，黑盒测试是动态测试

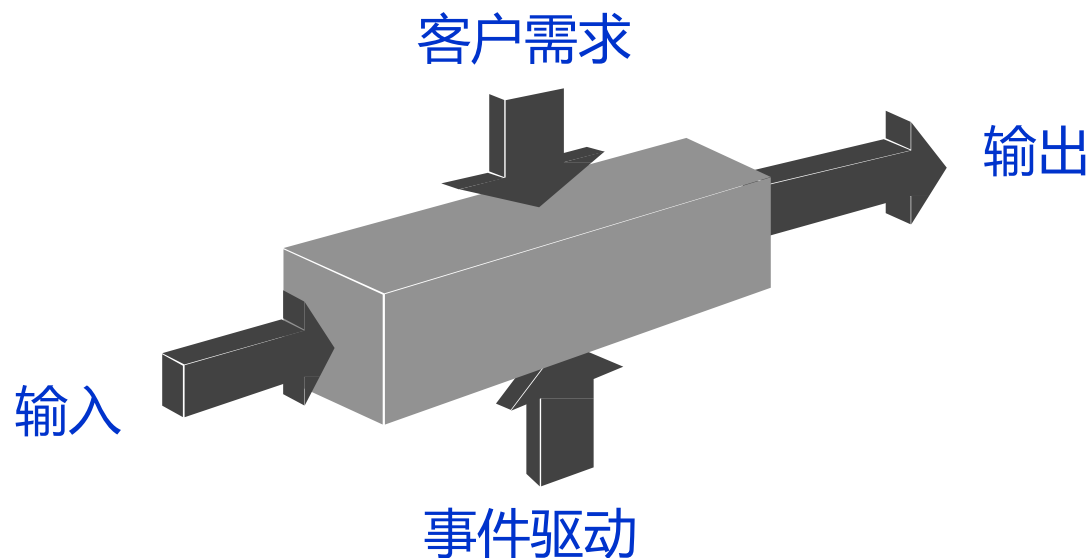
思考：在微信抢票应用中，程序通过直接调用系统函数来获得当前的系统时间。如果使用Mock方法进行测试，应该如何重构程序？

正常使用主观题需2.0以上版本雨课堂

黑盒测试技术

单元测试方法

黑盒测试 (Black Box Testing)：又称功能测试，它将测试对象看做一个黑盒子，完全不考虑程序内部的逻辑结构和内部特性，只依据程序的需求规格说明书，检查程序的功能是否符合它的功能说明。



只是根据需求规格说明设计有代表性的测试输入，再通过测试执行的输出结果来检查程序的功能是不是符合规格说明

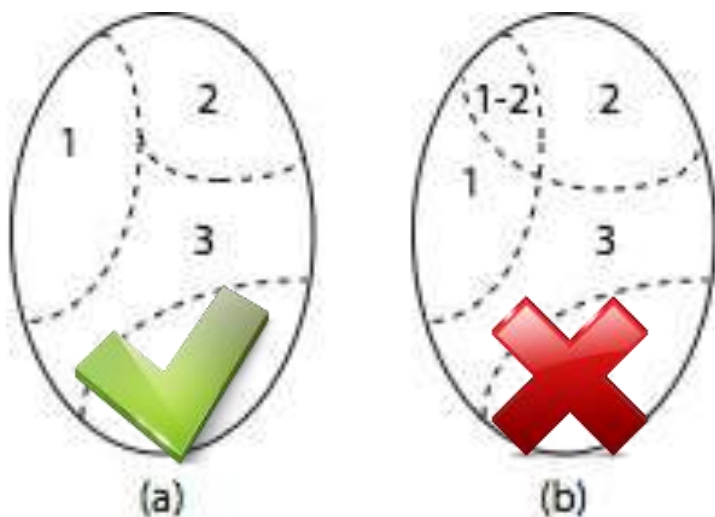
黑盒测试技术

黑盒测试是将测试对象看做一个黑盒子，完全不考虑程序内部的逻辑结构和内部特性，只依据程序的需求规格说明书，检查程序的功能是否符合它的功能说明。



等价类划分

等价类划分是将输入域划分成尽可能少的若干子域，在划分中要求每个子域两两互不相交，每个子域称为一个等价类。

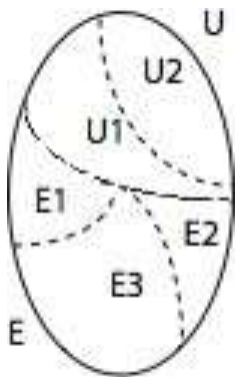


- 同一输入域的等价类划分可能不唯一
- 只需从每一个等价类中选取一个输入作为测试用例
- 对于相同的等价类划分，不同测试人员选取的测试用例集可能是不同的

等价类类型

有效等价类是对规格说明有意义、合理的输入数据构成的集合，能够检验程序是否实现了规格说明中预先规定的功能和性能。

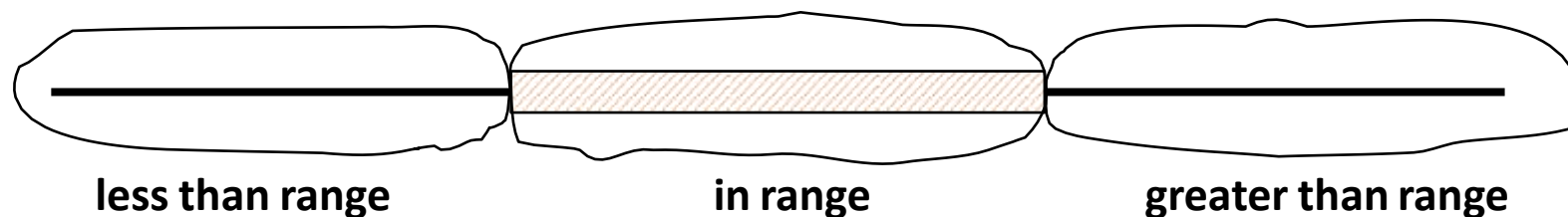
无效等价类是对规格说明无意义、不合理的输入数据构成的集合，以检查程序是否具有一定的容错性。



- E 表示所有正常和合法的输入
- U 表示所有异常和非法的输入

变量的等价类

取值范围：在输入条件规定了取值范围的情况下，可以确定一个有效等价类和两个无效等价类。



举例：程序的输入参数 x 是小于100大于10的整数。

1个有效等价类： $10 < x < 100$

2个无效等价类： $x \leq 10$ 和 $x \geq 100$

变量的等价类

字符串：在规定了输入数据必须遵守的规则情况下，可确定一个有效等价类（符合规则）和若干个无效等价类（从不同角度违反规则）。

举例：姓名是长度不超过20的非空字符串，且只由字母组成，数字和其他字符都是非法的。

1个有效等价类：满足了上述所有条件的字符串

3个无效等价类：

- 空字符串
- 长度超过20的字符串
- 包含了数字或其它字符的字符串（长度不超过20）

变量的等价类

枚举：若规定输入数据是一组值（假定N个），并且程序要对每一个输入值分别处理，可确定N个有效等价类和一个无效等价类。

举例：某程序根据不同的学历分别计算岗位工资，其中学历可以是专科、本科、硕士、博士等四种类型。

4个有效等价类：专科、本科、硕士、博士

1个无效等价类：其他学历

如果将专科、本科、硕士、博士按一种方式计算岗位工资，这时应如何划分等价类？

变量的等价类

数组：数组是一组具有相同类型的元素的集合，数组长度及其类型都可以作为等价类划分的依据。

举例：假设某程序的输入是一个整数数组 `int oper[3]`

1个有效等价类：所有合法值的数组，如 `{-10, 20}`

2个无效等价类：

- 空数组
- 所有大于期望长度的数组，如 `{-9, 0, 12, 5}`

如果对数组元素有其他附加约束，例如数组`oper`元素的取值范围是`[-3, 3]`，则需要增加相应的等价类。

变量的等价类

复合数据类型：复合数据类型是包含两个或两个以上相互独立的属性的输入数据，在进行等价类划分时需要考虑输入数据的每个属性的合法和非法取值。

举例：

```
struct student {  
    string name;  
    string course[100];  
    int grade[100];  
}
```

对复合数据类型中的每个元素进行等价类划分，再将这些等价类进行组合，最终形成对软件整个输入域的划分。

等价类组合

测试用例生成：测试对象通常有多个输入参数，如何对这些参数等价类进行组合测试，来保证等价类的覆盖率，是测试用例设计首先需要考虑的问题。

所有有效等价类的代表值都集成到测试用例中，即**覆盖有效等价类的所有组合**。任何一个组合都将设计成一个有效的测试用例，也称**正面测试用例**。

无效等价类的代表值只能和其他有效等价类的代表值（随意）进行组合。因此，每个无效等价类将产生一个额外的无效测试用例，也称**负面测试用例**。

举例

```
enum Sex { Male = 1, Female };
```

```
struct person {  
    string name;  
  
    int age; // 0 < age < 200  
  
    Sex gender;  
  
    string friends[100];  
}
```

有效等价类和无效等价类?

举例：判断三角形类型

输入三个整数a、b、c，分别作为三角形的三条边，现通过一个程序判断这三条边构成的三角形类型，包括等边三角形、等腰三角形、一般三角形（特殊的还包括直角三角形）以及构不成三角形。

现在要求输入的三个整数a、b、c必须满足以下条件：

条件1： $1 \leq a \leq 200$

条件2： $1 \leq b \leq 200$

条件3： $1 \leq c \leq 200$

请使用等价类划分方法，设计该程序的测试用例。

举例：判断三角形类型

等价类划分：

- ① 按输入取值划分
- ② 按输出的几何特性划分

{等腰且非等边三角形，等边三角形，一般三角形，非三角形}



你会选择哪一种方法划分等价类



在多数情况下，等价类是根据输入域进行划分的，但并不是说不能从输出反过来划分

举例：判断三角形类型

标准等价类测试用例

序号	测试用例描述	输入参数			期望输出
		a	b	c	
1	$a > 0, b > 0, c > 0$ $a + b > c$ $b + c > a, a + c > b$ $a = b = c$	60	60	60	等边三角形
2	$a > 0, b > 0, c > 0$ $a + b > c$ $b + c > a, a + c > b$ $a = b \neq c$ 或 $b = c \neq a$ 或 $a = c \neq b$	60	60	50	等腰三角形
3	$a > 0, b > 0, c > 0$ $a + b > c$ $b + c > a, a + c > b$ $a \neq b \neq c$	30	40	50	一般三角形
4	$a > 0, b > 0, c > 0$ $a + b \leq c$ 或 $b + c \leq a$ 或 $a + c \leq b$	40	10	20	非三角形

边界值分析

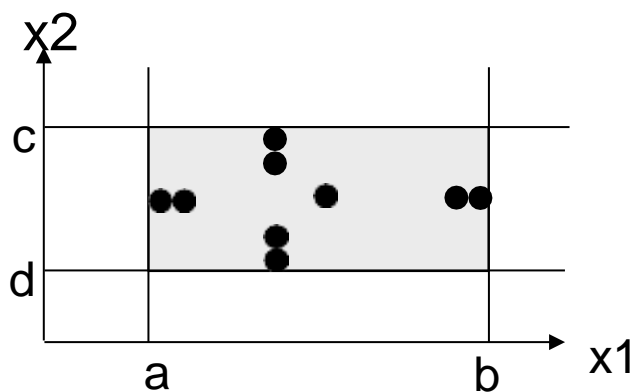
边界值分析是对输入或输出的边界值进行测试的一种方法，它通常作为等价类划分法的补充，这种情况下的测试用例来自等价类的边界。

- 先确定边界：通常输入或输出等价类的边界就是应该着重测试的边界情况。
- 选取正好等于、刚刚大于或刚刚小于边界的值作为测试数据，而不是选取等价类中的典型值或任意值。

实践表明：大多数故障往往发生在输入定义域或输出值域的边界上，而不是内部。因此，针对各种边界情况设计测试用例，通常会取得很好的测试效果。

边界值分析

- 基本思想：故障往往出现在程序输入变量的边界值附近
- 边界值分析法是基于可靠性理论中称为“单故障”的假设，即有两个或两个以上故障同时出现而导致失效的情况很少。
- 对程序中的每个变量重复：每次保留一个变量，让其余的变量取正常值，被保留的变量依次取 min、min+、nom、max- 和 max



$\langle x1_{nom}, x2_{min} \rangle$

$\langle x1_{nom}, x2_{min+} \rangle$

$\langle x1_{nom}, x2_{nom} \rangle$

$\langle x1_{nom}, x2_{max} \rangle$

$\langle x1_{nom}, x2_{max-} \rangle$

$\langle x1_{min}, x2_{nom} \rangle$

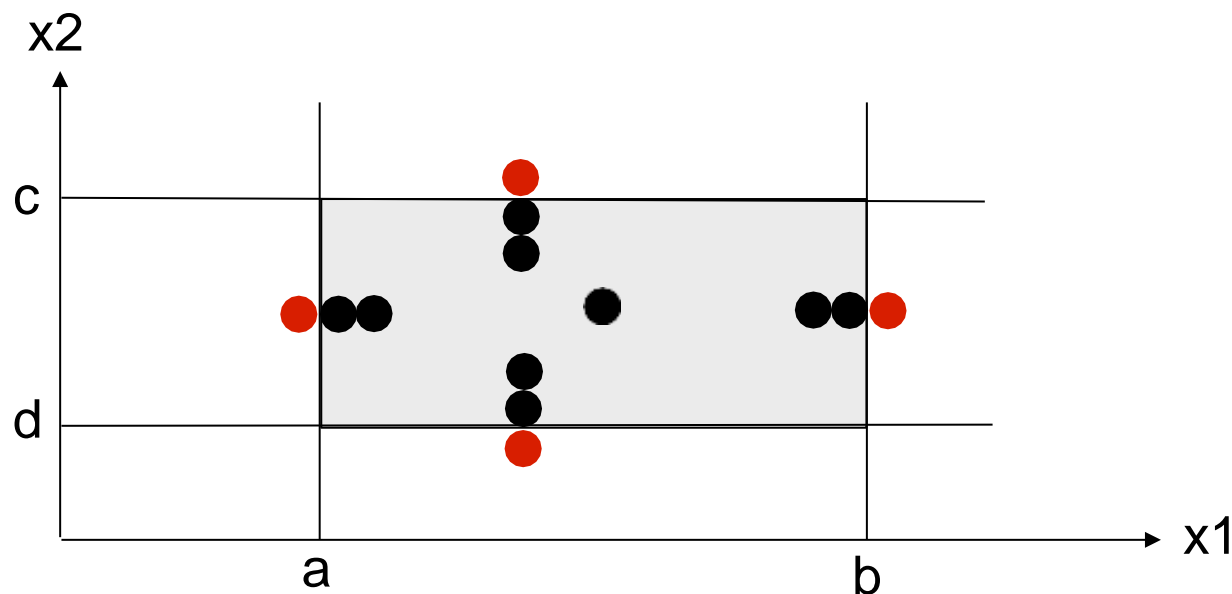
$\langle x1_{min+}, x2_{nom} \rangle$

$\langle x1_{max}, x2_{nom} \rangle$

$\langle x1_{max-}, x2_{nom} \rangle$

健壮性测试

健壮性测试是作为边界值分析的一个简单的扩充，它除了对变量的5个边界值分析取值外，还要增加一个略大于最大值（max+）以及略小于最小值（min-）的取值，检查超过极限值时系统的情况。



健壮性测试

输入项	边界值	测试用例的设计思路
字符	起始 - 1个字符 结束 + 1个字符	假设一个文本输入区域允许输入1到255个字符，输入1个和255个字符作为有效等价类；输入0个和256个字符作为无效等价类，这几个数值都属于边界条件值。
数值	最小值 - 1 最大值 + 1	假设某软件要求输入5位十进制整数值，可以使用10000作为最小值、99999作为最大值；然后使用刚好小于5位和大于5位的数值作为边界条件。
空间	小于空余空间一点 大于满空间一点	例如在用U盘存储数据时，使用比剩余磁盘空间大一点（几 KB）的文件作为边界条件。

举例：判断三角形类型 边界值分析？

标准等价类测试用例

序号	测试用例描述	输入参数			期望输出
		a	b	c	
1	$a > 0, b > 0, c > 0$ $a + b > c$ $b + c > a, a + c > b$ $a = b = c$	60	60	60	等边三角形
2	$a > 0, b > 0, c > 0$ $a + b > c$ $b + c > a, a + c > b$ $a = b \neq c$ 或 $b = c \neq a$ 或 $a = c \neq b$	60	60	50	等腰三角形
3	$a > 0, b > 0, c > 0$ $a + b > c$ $b + c > a, a + c > b$ $a \neq b \neq c$	30	40	50	一般三角形
4	$a > 0, b > 0, c > 0$ $a + b \leq c$ 或 $b + c \leq a$ 或 $a + c \leq b$	40	10	20	非三角形

举例：判断三角形类型

序号	测试用例描述	输入参数			期望输出
		a	b	c	
1	$a > 0, b > 0, c > 0$ $a + b > c$ $b + c > a, a + c > b$ $a = b = c$	60	60	60	等边三角形
2	$a > 0, b > 0, c > 0$ $a + b > c, b + c > a, a + c > b$ $a = b \neq c$ 或 $b = c \neq a$ 或 $a = c \neq b$	60	60	1	等腰三角形
3		60	60	2	
4		60	60	119	
5		60	1	60	
6		60	2	60	

举例：判断三角形类型

序号	测试用例描述	输入参数			期望输出
		a	b	c	
7	$a > 0, b > 0, c > 0$ $a + b > c$, $b + c > a, a + c > b$ $a = b \neq c$ 或 $b = c \neq a$ 或 $a = c \neq b$	60	119	60	等腰三角形
8		1	60	60	
9		2	60	60	
10		119	60	60	
11	$a > 0, b > 0, c > 0$ $a + b \leq c$ 或 $b + c \leq a$ 或 $a + c \leq b$	60	60	120	非三角形
12		60	120	60	
13		120	60	60	

举例：判断三角形类型

健壮等价类测试用例

序号	测试用例描述	输入参数			期望输出
		a	b	c	
1-4	有效等价类同前面（略）				
5	$a < 0$, $b > 0$, $c > 0$	-1	5	5	a 值越界
6	$a > 0$, $b < 0$, $c > 0$	5	-1	5	b 值越界
7	$a > 0$, $b > 0$, $c < 0$	5	5	-1	c 值越界
8	$a > 200$, $b > 0$, $c > 0$	201	5	5	a 值越界
9	$a > 0$, $b > 200$, $c > 0$	5	201	5	b 值越界
10	$a > 0$, $b > 0$, $c > 200$	5	5	201	c 值越界

错误推测法

错误推测法是人们根据经验或直觉推测程序中可能存在的各种错误，从而有针对性地编写检查这些错误的测试用例的方法。

- 软件缺陷具有空间聚集性，80%的缺陷常常存在于20%的代码中。因此，应当记住常常光临代码的高危多发“地段”，这样发现缺陷的可能性会大得多。
- 列举程序中所有可能的错误和容易发生错误的特殊情况，根据可能出现的错误情况选择测试用例。

80%
20

返校登记程序，需要登记姓名、学号、年龄、性别和体温，要求如下：

姓名：长度不超过10的字符串，只能由字母构成，不能有其它字符

学号：长度为7的字符串，只能由数字构成

年龄：大于0，小于等于100的整数

性别：男或者女

体温：20~50的数字

请分析如何对这个程序进行等价类划分、边界测试和健壮性测试

作答

正常使用主观题需2.0以上版本雨课堂

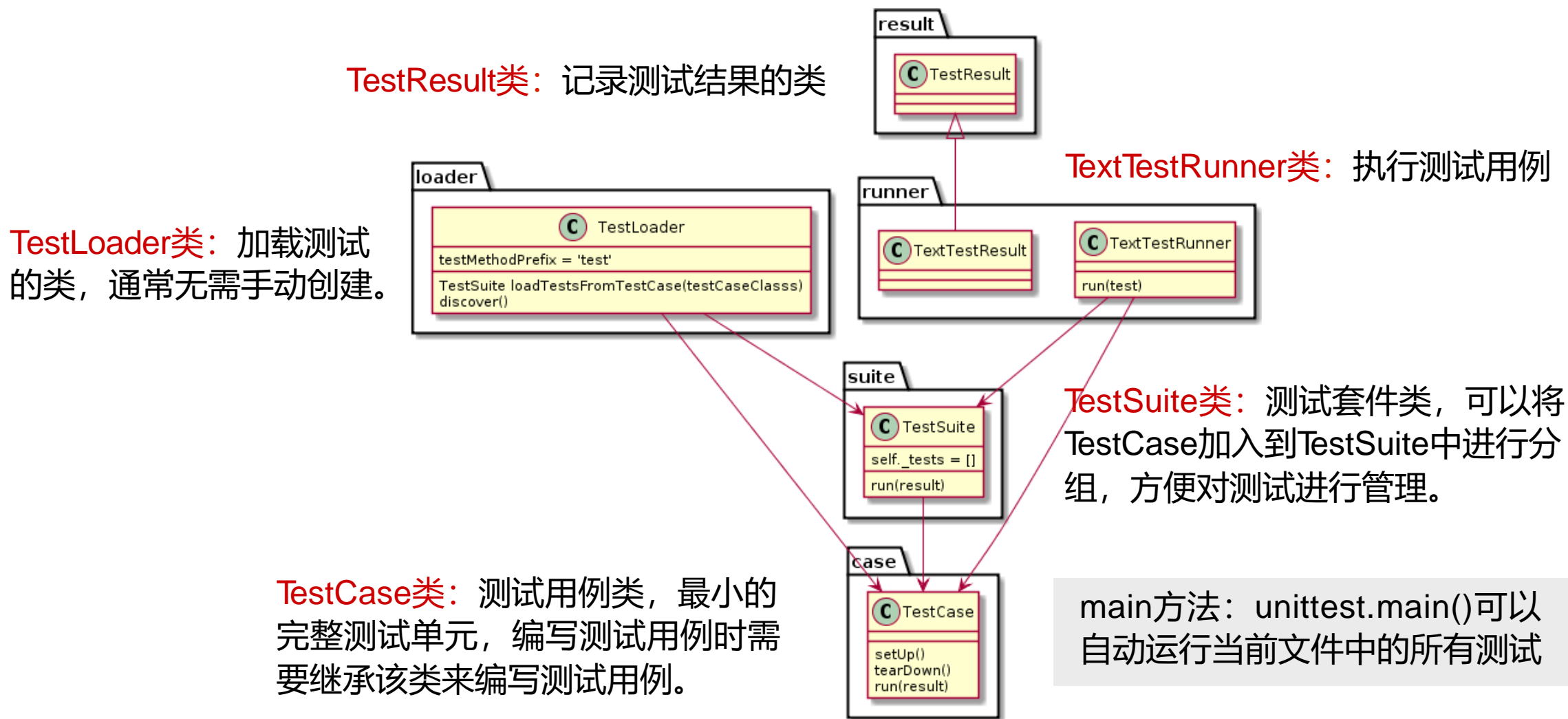
关于等价类划分，下面的（ ）说法是正确的。

提交

- ☐ A 将输入域的子域越少越好
- ☐ B 同一输入域的等价类划分是唯一的
- ☒ C 用同一等价类中的任意输入对软件进行测试，软件都输出相同的结果
- ☐ D 对于相同的等价类划分，不同测试人员选取的测试用例集是一样的

Python单元测试工具

Python单元测试之unittest



Python单元测试之unittest

测试过程中需要通过属性断言对结果进行判断，以验证结果是否满足需求。

- TestCase类提供了多种强大的断言方法，如assertTrue, assertFalse, assertEquals, assertNotEqual, assertIs等。

参考文档见 <https://docs.python.org/3/library/unittest.html>

- 这些断言方法可以在断言的同时加上一个message参数，这样可以使断言的意义明确而且方便维护，在测试失败时抛出可读的信息。

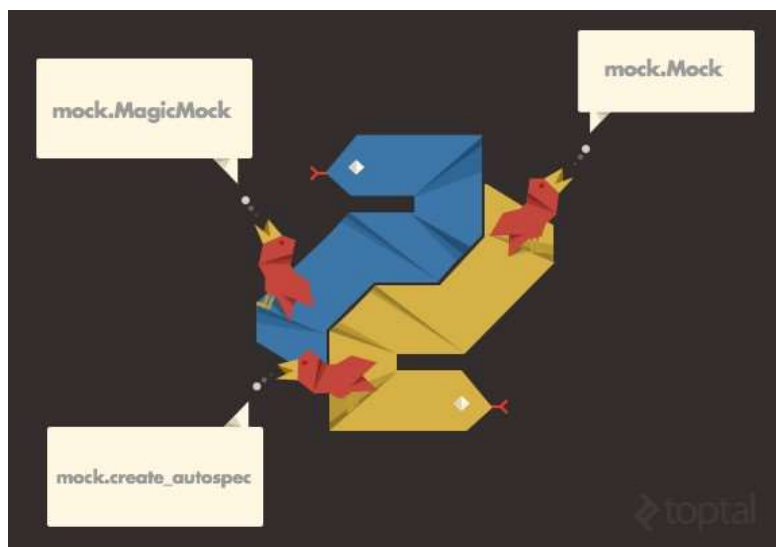


Python单元测试之unittest

- ① import unittest
- ② 定义一个继承自unittest.TestCase的测试用例类
- ③ 定义setUp和tearDown，在每个测试用例前后做一些辅助工作
- ④ 定义测试用例，名字以test开头
- ⑤ 一个测试用例应该只测试一个方面，测试目的和测试内容应很明确。主要是调用assertEqual、assertRaises等断言方法判断程序执行结果和预期值是否相符
- ⑥ 调用unittest.main()启动测试
- ⑦ 如果测试未通过，会输出相应的错误提示；如果测试全部通过则显示ok，添加-v参数显示详细信息。

Python单元测试之mock

Python 3.3开始内置了Mock工具包，可以使用mock对象替代掉指定的Python对象，以达到模拟对象的行为。



- **Mock类**：用于创建mock对象，当访问mock对象的某个属性时，mock对象会自动创建该属性。
- **MagicMock类**：Mock对象的子类，预先定义了操作符（如`__lt__`, `__len__`）。
- **patch装饰器**：可以将其作用在测试方法上，限定在当前测试方法中使用mock来替换真实对象。

Python单元测试之mock

属性断言： mock对象提供了一系列断言方法，可以在使用属性断言时判断程序对mock对象的调用是否符合预期。

- `assert_called_with`, `assert_called_once_with`, `assert_any_call`, `assert_has_calls`
- <https://docs.python.org/3/library/unittest.mock.html#the-mock-class>

行为控制： 通常程序需要从依赖对象的方法上取得返回值，mock对象也提供了一些途径对返回值进行控制。

- `return_value`: 固定返回值
- `side_effects`: 返回值的序列或自定义方法

Python单元测试之覆盖分析

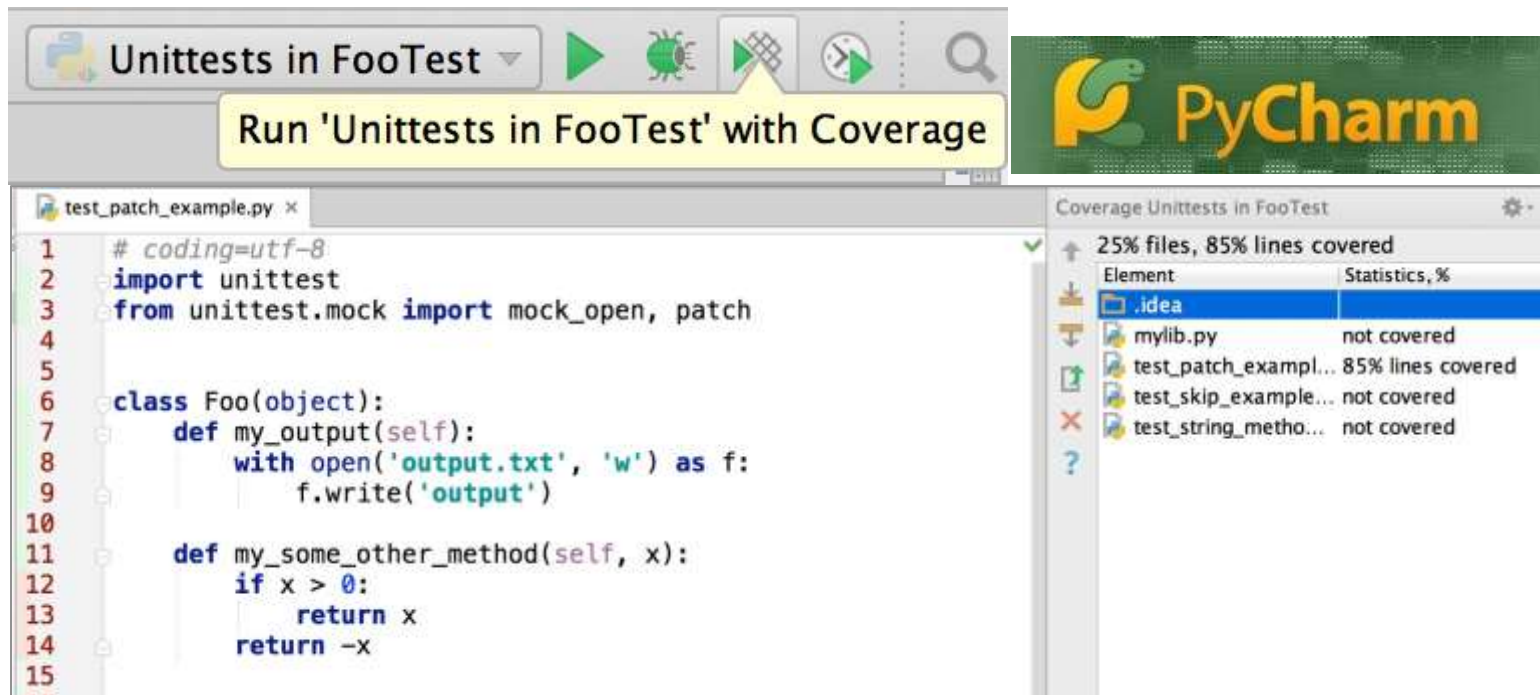
coverage.py是一个用来统计python程序代码覆盖率的工具，它使用起来非常简单，并且支持最终生成界面友好的html报告。

安装



`pip install -U coverage.py`

使用



案例：生命游戏

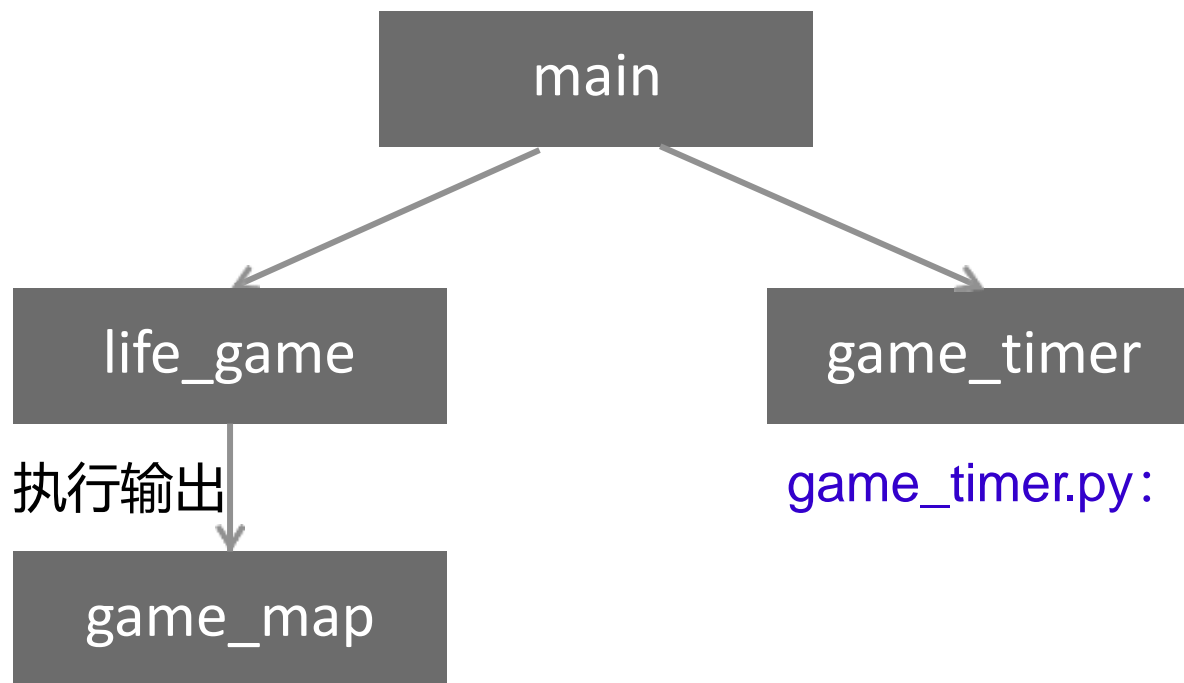


Python单元测试

- unittest
- mock
- coverage.py

案例：生命游戏

`main.py`: 生命游戏的主程序, 用户使用的入口



`life_game.py`: 游戏的 执行输出

`game_timer.py`: 定时器, 定时触发

`game_map.py`: 生命游戏地图, 包含了所需的底层操作

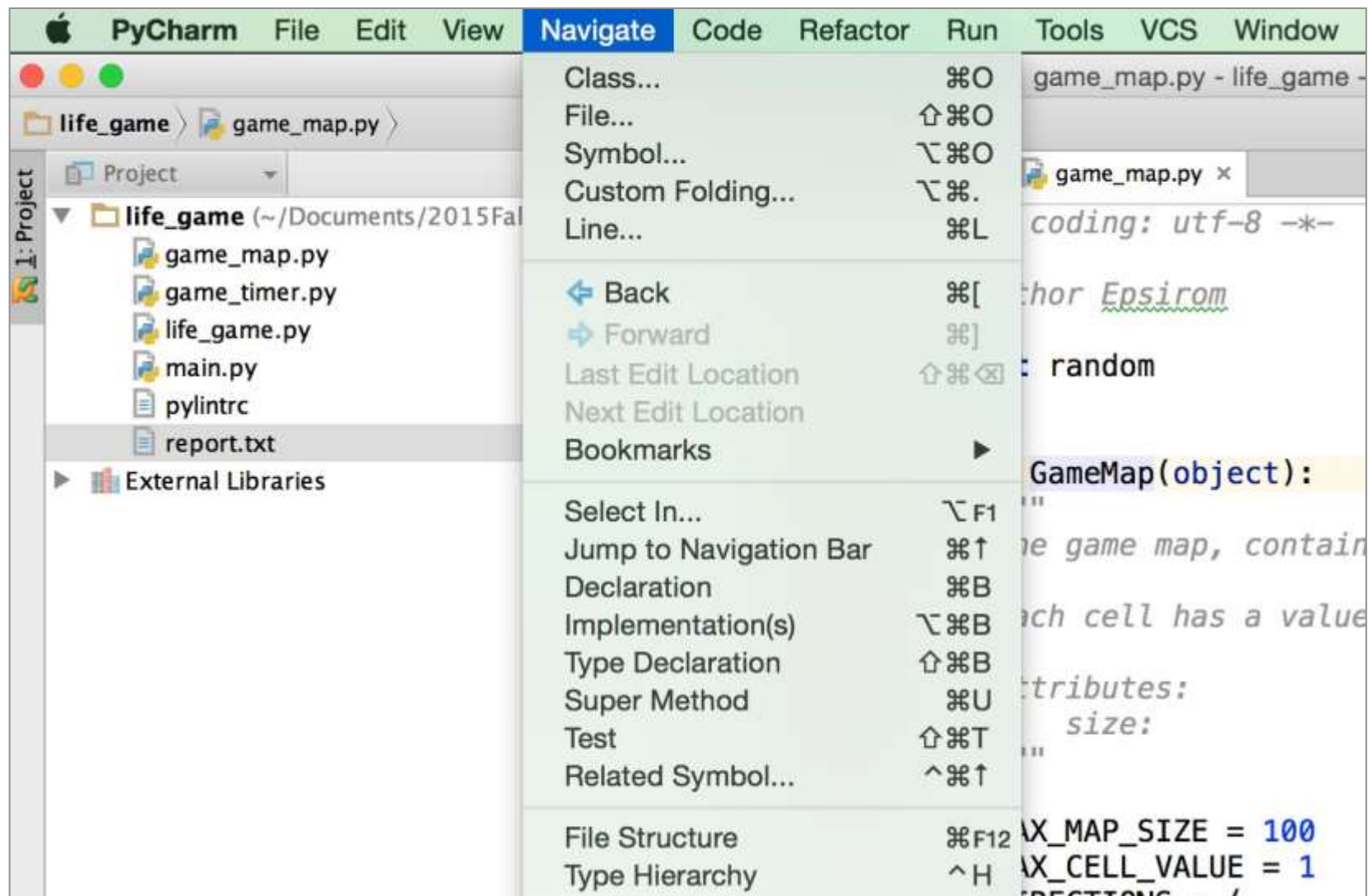
案例：生命游戏

game_map



- rows, cols: 表示地图的行数和列数
- reset: 以一定的概率设置地图的每个格子的状态
- get/set: 获取、设置地图的某个格子的状态
- get_neighbor_count: 获取一个格子的邻居数量
- get_neighbor_count_map: 获取每个格子的邻居数量
- set_map: 设置地图
- print_map: 打印地图

创建测试



创建测试



```
1  # -*- coding: utf-8 -*-
2  #
3  # @author Epsirom
4
5  import random
6
7
8  class GameMap(object):
9      """
10     The game map is a 2D array of cells.
11
12     Each cell has a value, 0 means it is a dead/empty cell.
13
14     Attributes:
15         size:
16         """
```

创建测试



Create test

Target directory

Test file name

Test class name

Test method

- ☐ test_rows
- ☐ test_cols
- ☐ test_reset
- ☐ test_get
- ☐ test_set
- ☐ test_get_neighbor_count
- ☐ test_get_neighbor_count_map
- ☐ test_set_map
- ☐ test_print_map

?

Cancel OK

创建测试



Create test

Target directory

Test file name

Test class name

Test method

- ☒ test_rows
- ☒ test_cols
- ☒ test_reset
- ☒ test_get
- ☒ test_set
- ☒ test_get_neighbor_count
- ☒ test_get_neighbor_count_map
- ☒ test_set_map
- ☒ test_print_map

?

Cancel OK

创建测试



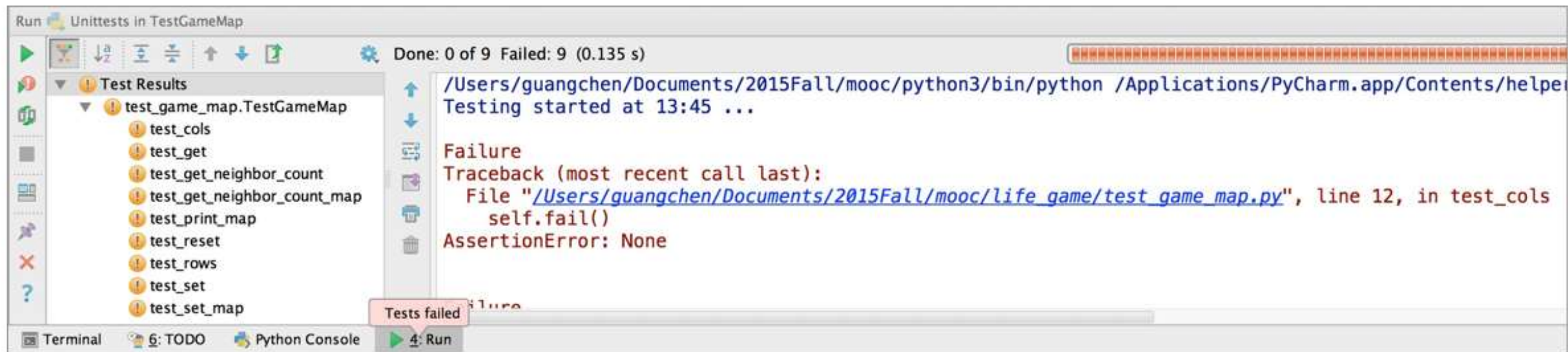
```
1  # coding=utf-8
2  from unittest import TestCase
3
4  __author__ = 'guangchen'
5
6
7  class TestGameMap(TestCase):
8      def test_rows(self):
9          self.fail()
10
11     def test_cols(self):
12         self.fail()
13
14     def test_reset(self):
15         self.fail()
16
17     def test_get(self):
18         self.fail()
19
20     def test_set(self):
21         self.fail()
22
23     def test_get_neighbor_count(self):
24         self.fail()
25
26     def test_get_neighbor_count_map(self):
27         self.fail()
28
29     def test_set_map(self):
30         self.fail()
31
32     def test_print_map(self):
33         self.fail()
34
```

运行测试



```
1 # coding=utf-8
2 from unittest import TestCase
3
4 __author__ = 'guangchen'
5
6
7 class TestGameMap(TestCase):
8     def test_run(self):
9         self.fail('Test Run')
10
11     def test_create(self):
12         self.fail('Test Create')
13
14     def test_remove(self):
15         self.fail('Test Remove')
16
17     def test_get(self):
18         self.fail('Test Get')
19
20     def test_set(self):
21         self.fail('Test Set')
22
23     def test_get(self):
24         self.fail('Test Get')
25
26     def test_get(self):
27         self.fail('Test Get')
28
29     def test_set(self):
30         self.fail('Test Set')
31
32     def test_remove(self):
33         self.fail('Test Remove')
34
```

- Copy Reference ⌘⇧C
- Paste ⌘V
- Paste from History... ⇧⌘V
- Paste Simple ⌘⇧V
- Column Selection Mode ⇧⌘8
- Find Usages ⌘F7
- Refactor ▶
- Folding ▶
- Go To ▶
- Generate... ⌘N
- Create 'Unittests in TestGameMap'...
- Run 'Unittests in TestGameMap'** ⇧⌘R
- Debug 'Unittests in TestGameMap' ⇧⌘D
- Run 'Unittests in TestGameMap' with Coverage
- Profile 'Unittests in TestGameMap'
- Local History ▶
- Git ▶
- Execute Line in Console ⌘⇧E
- Compare with Clipboard
- File Encoding
- Diagrams ▶
- Create Gist...



TestCase.fail() 无条件使当前测试失败

创建测试



```
1  # coding=utf-8
2  from unittest import TestCase
3
4  __author__ = 'guangchen'
5
6
7  class TestGameMap(TestCase):
8      def test_rows(self):
9          self.fail()
10
11     def test_cols(self):
12         self.fail()
13
14     def test_reset(self):
15         self.fail()
16
17     def test_get(self):
18         self.fail()
19
20     def test_set(self):
21         self.fail()
22
23     def test_get_neighbor_count(self):
24         self.fail()
25
26     def test_get_neighbor_count_map(self):
27         self.fail()
28
29     def test_set_map(self):
30         self.fail()
31
32     def test_print_map(self):
33         self.fail()
34
```

案例：生命游戏

- setUp方法：创建每个测试方法都需要的公共对象
- tearDown方法：销毁公共对象（如果需要的话），如数据库断开连接等

创建测试fixture



这里只需要setUp方法，并在其中创建一个GameMap待测对象

```
class TestGameMap(TestCase):  
    def setUp(self):  
        self.game_map = GameMap(4, 3)
```

案例：生命游戏

属性测试



测试 rows 和 cols

```
def test_rows(self):  
    self.assertEqual(4, self.game_map.rows, "Should get correct rows")  
  
def test_cols(self):  
    self.assertEqual(3, self.game_map.cols, "Should get correct cols")
```



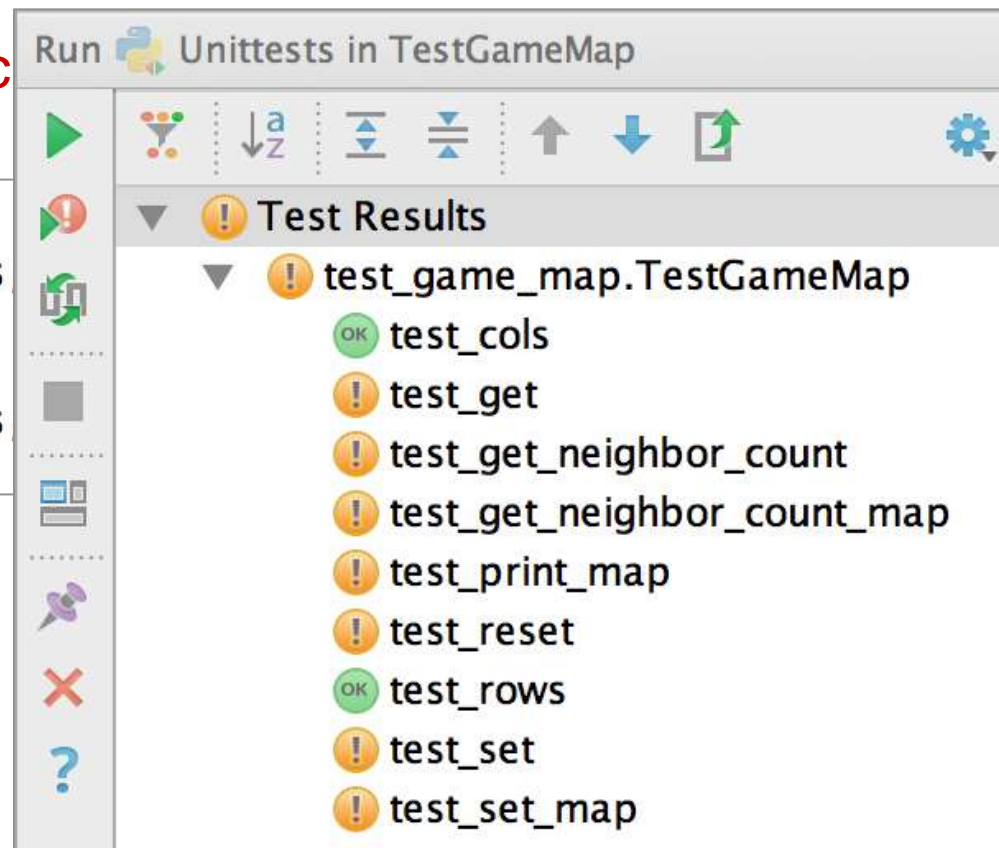
案例：生命游戏

属性测试



测试 rows 和 cols

```
def test_rows(self):  
    self.assertEqual(4, self.game_map.rows)  
  
def test_cols(self):  
    self.assertEqual(3, self.game_map.cols)
```



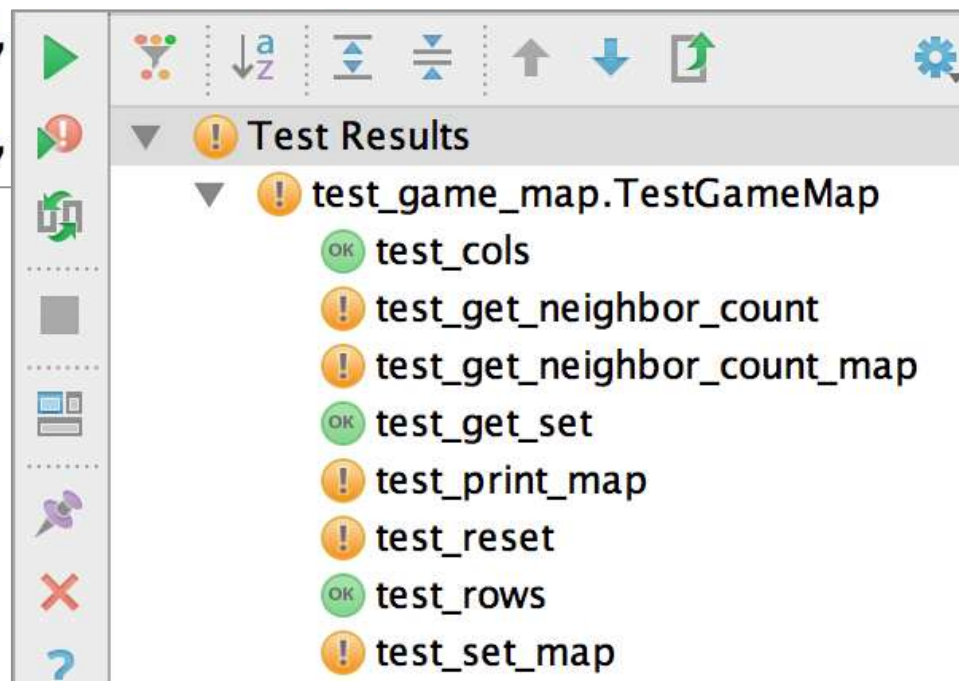
案例：生命游戏

方法测试



get/set: 两个方法相互联系, 合并为一个测试

```
def test_get_set(self):  
    self.assertEqual(0, self.game_map.get(0,  
    self.game_map.set(0, 0, 1)  
    self.assertEqual(1, self.game_map.get(0,
```



案例：生命游戏

方法测试



reset: 依赖概率, 需要进行mock

```
def reset(self, possibility=0.5):  
    """Reset the map with random data."""  
    if not isinstance(possibility, float):  
        raise TypeError("possibility should be float")  
    for row in self.cells:  
        for col_num in range(self.cols):  
            row[col_num] = 1 if random.random() < possibility else 0
```



案例：生命游戏

方法测试



reset: 依赖概率, 需要进行mock

```
def reset(self, possibility=0.5):  
    """Reset the map with random data."""  
    if not isinstance(possibility, float):  
        raise TypeError("possibility should be float")  
    for row in self.cells:  
        for col_num in range(self.cols):  
            row[col_num] = 1 if random.random() < possibility else 0
```

```
@patch('random.random', new=Mock(side_effect=[0.1, 0.5, 0.9]))  
def test_reset(self):  
    self.game_map.reset()  
    for i in range(0, 4):  
        self.assertEqual(1, self.game_map.cells[i][0])  
        for j in range(1, 3):  
            self.assertEqual(0, self.game_map.cells[i][j])
```

Test Results

test_game_map.TestGameMap

- OK test_cols
- ! test_get_neighbor_count
- ! test_get_neighbor_count_map
- OK test_get_set
- ! test_print_map
- OK test_reset
- OK test_rows
- ! test_set_map

方法测试




get_neighbor_count

```
def get_neighbor_count(self, row, col):  
    """Get count of neighbors in specific cell.  
  
    Args:  
        row: row number  
        col: column number  
  
    Returns:  
        Count of live neighbor cells  
    """  
    if not isinstance(row, int):  
        raise TypeError("row should be int")  
    if not isinstance(col, int):  
        raise TypeError("col should be int")  
    assert 0 <= row < self.rows  
    assert 0 <= col < self.cols  
    count = 0  
    for d in self.DIRECTIONS:  
        d_row = row + d[0]  
        d_col = col + d[1]  
        if d_row >= self.rows:  
            d_row -= self.rows  
        if d_col >= self.cols:  
            d_col -= self.cols  
        count += self.cells[d_row][d_col]  
    return count
```

方法测试

➔ `get_neighbor_count`

```
def test_get_neighbor_count(self):
    expected_value = [[8] * 3] * 4
    self.game_map.cells = [[1] * 3] * 4
    for i in range(0, 4):
        for j in range(0, 3):
            self.assertEqual(expected_value[i][j], (self.game_map.get_neighbor_count(i, j)), '(%d, %d)' % (i, j))
```

Run  Unittests in TestGameMap

Done: 4 of 8 Failed: 4 (95 ms)

Test Results

- test_game_map.TestGameMap
 - OK test_cols
 - test_get_neighbor_count
 - test_get_neighbor_count_map
 - OK test_get_set
 - test_print_map
 - OK test_reset
 - OK test_rows
 - test_set_map

Failure

Traceback (most recent call last):

File "[/Users/quangchen/Documents/2015Fall](#)"

self.assertEqual(expected_value[i][j],

AssertionError: 8 != 4 : (0, 0)

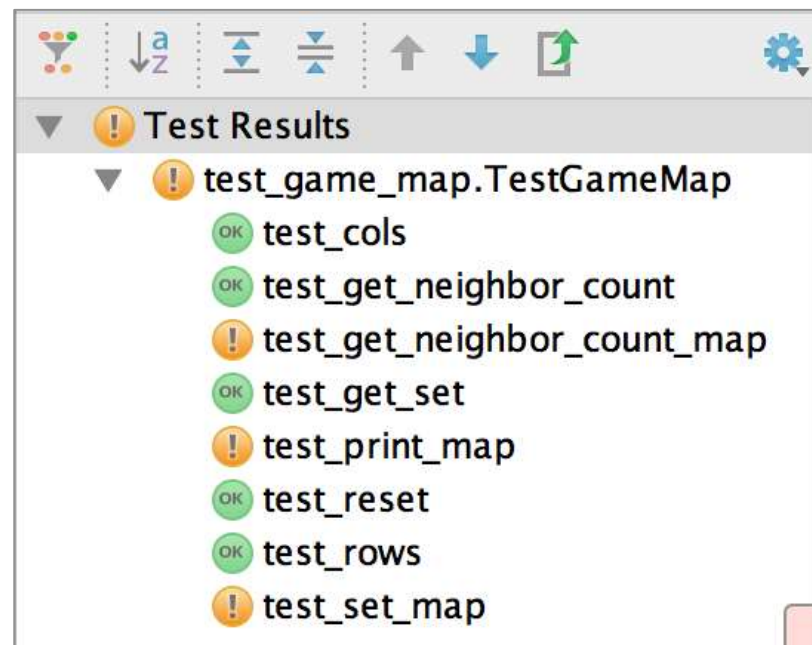
方法测试

➔ `get_neighbor_count`

```
count = 0
for d in self.DIRECTIONS:
    d_row = row + d[0]
    d_col = col + d[1]
    if d_row >= self.rows:
        d_row -= self.rows
    if d_col >= self.cols:
        d_col -= self.cols
    count += self.cells[d_row][d_col]
return count
```

```
DIRECTIONS = (
    (0, 1, ),
    (0, -1, ),
    (1, 0, ),
    (-1, 0, )
)
```

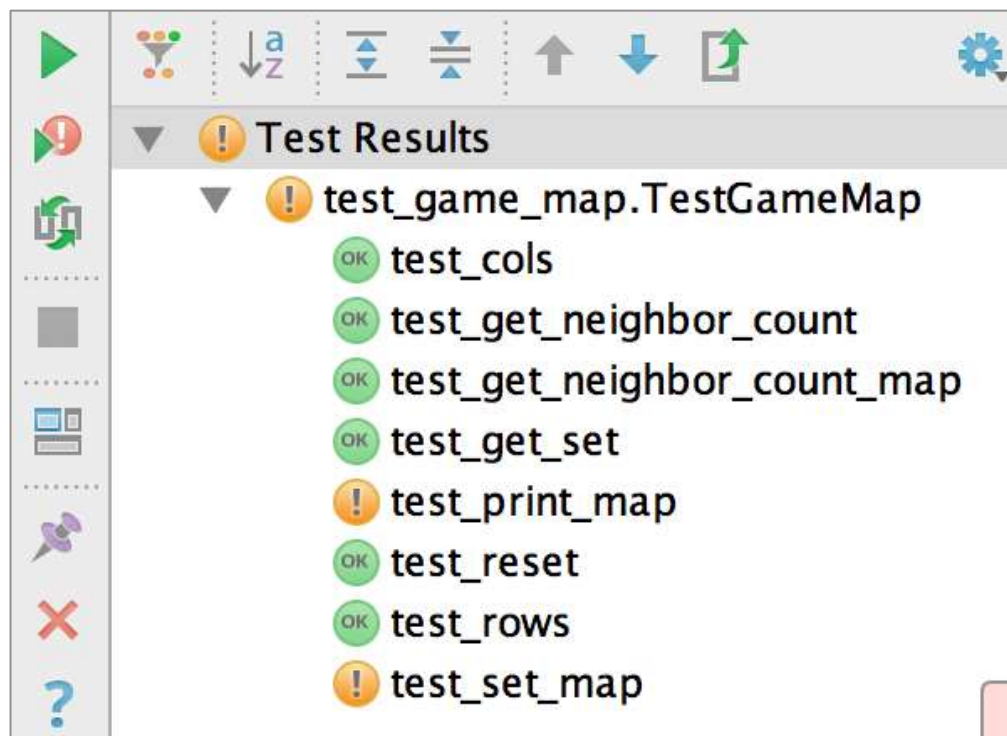
```
DIRECTIONS = (
    (0, 1, ),
    (0, -1, ),
    (1, 0, ),
    (-1, 0, ),
    (1, 1, ),
    (1, -1, ),
    (-1, 1, ),
    (-1, -1)
)
```



方法测试

➔ **get_neighbor_count_map**: 依赖 get_neighbor_count, 测试时对依赖方法进行mock, 保持测试的独立性。

```
@patch('game_map.GameMap.get_neighbor_count', new=Mock(return_value=8))
def test_get_neighbor_count_map(self):
    expected_value = [[8] * 3] * 4
    self.assertEqual(expected_value, self.game_map.get_neighbor_count_map())
```



案例：生命游戏

方法测试



set_map

```
def set_map(self, new_map):
    if not isinstance(new_map, list):
        raise TypeError("new_map should be list")
    assert len(new_map) == self.rows
    for row in new_map:
        if not isinstance(row, list):
            raise TypeError("rows in new_map should be list")
        assert len(row) == self.cols
        for cell in row:
            if not isinstance(cell, int):
                raise TypeError("cells in new_map should be int")
            assert 0 <= cell <= self.MAX_CELL_VALUE
    self.cells = new_map
```

案例：生命游戏

方法测试



set_map

```
def test_set_map(self):  
    self.assertRaises(TypeError,  
    self.assertRaises(Assertio  
    self.assertRaises(TypeError,  
    self.assertRaises(Assertio  
  
    self.game_map.set_map([[1]  
    self.assertEqual([[1] * 3]
```

The screenshot shows a test runner interface with a toolbar at the top containing icons for running tests, sorting, and other functions. Below the toolbar is a section titled "Test Results" with a dropdown arrow and an orange warning icon. Under this section, there is a tree view showing the test results for "test_game_map.TestGameMap". The results are as follows:

- test_cols: OK (green circle)
- test_get_neighbor_count: OK (green circle)
- test_get_neighbor_count_map: OK (green circle)
- test_get_set: OK (green circle)
- test_print_map: ! (orange circle with exclamation mark)
- test_reset: OK (green circle)
- test_rows: OK (green circle)
- test_set_map: OK (green circle)

On the right side of the interface, there is a vertical list of test results, partially visible, showing " * 3)", "4)", and " * 4)".

案例：生命游戏

方法测试



print_map

```
def print_map(self, cell_maps=None, sep=' '):  
    if not cell_maps:  
        cell_maps = ['0', '1']  
    if not isinstance(cell_maps, list) and not isinstance(cell_maps, dict):  
        raise TypeError("cell_maps should be list or dict")  
    if not isinstance(sep, str):  
        raise TypeError("sep should be string")  
    for row in self.cells:  
        print(sep.join([cell_maps[cell] for cell in row]))
```



案例：生命游戏

方法测试



print_map

```
def test_print_map(self):
    self.game_map.cells = [
        [0, 1, 1],
        [0, 0, 1],
        [1, 1, 1],
        [0, 0, 0]
    ]
    with patch('builtins.print') as mock:
        self.game_map.print_map()
        mock.assert_has_calls([
            call('0 1 1'),
            call('0 0 1'),
            call('1 1 1'),
            call('0 0 0'),
        ])
    ])
```

▼ OK Test Results

- ▼ OK test_game_map.TestGameMap
 - OK test_cols
 - OK test_get_neighbor_count
 - OK test_get_neighbor_count_map
 - OK test_get_set
 - OK test_print_map
 - OK test_reset
 - OK test_rows
 - OK test_set_map



覆盖率分析



game_map.py

84% lines covered

```
33     if not isinstance(rows, int):
34         raise TypeError("rows should be int")
35     if not isinstance(cols, int):
36         raise TypeError("cols should be int")
37     assert 0 < rows <= self.MAX_MAP_SIZE

54     if not isinstance(possibility, float):
55         raise TypeError("possibility should be float")
56     for row in self.cells:

62     if not isinstance(row, int):
63         raise TypeError("row should be int")
64     if not isinstance(col, int):
65         raise TypeError("col should be int")
66     assert 0 <= row < self.rows

72     if not isinstance(row, int):
73         raise TypeError("row should be int")
74     if not isinstance(col, int):
75         raise TypeError("col should be int")
76     if not isinstance(val, int):
77         raise TypeError("val should be int")
```

其他工具

- 其他测试框架

python nose: <https://nose.readthedocs.org/en/latest/>

pytest: <http://pytest.org/latest/>



- 其他Mock框架

<http://garybernhardt.github.io/python-mock-comparison/>

- Python测试工具大全（测试框架、Mock框架、Web测试工具等）

<https://wiki.python.org/moin/PythonTestingToolsTaxonomy>

谢谢大家！
