持续集成单元测试方法与应用

专 业 的 IT 培 训 专 家

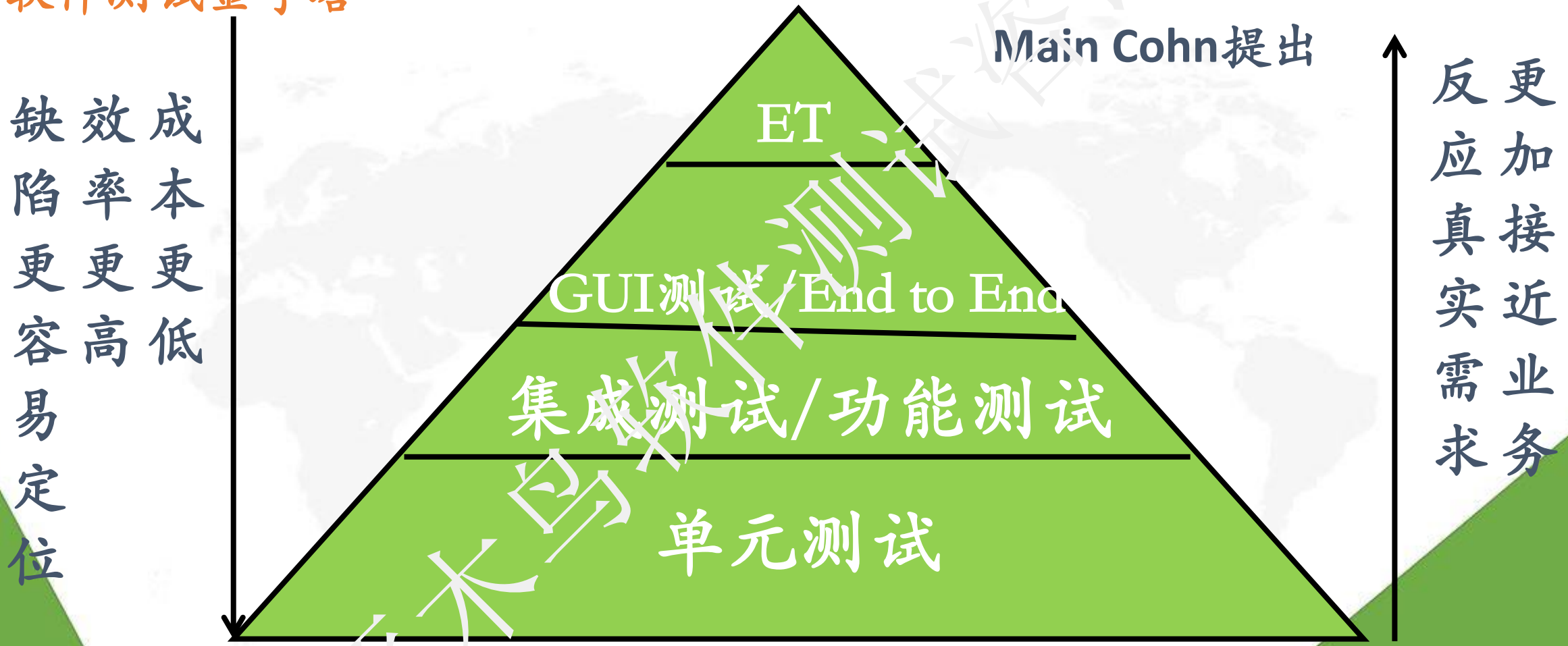
顾 翔

持续集成单元测试方法与应用

- 测试用例的设计
- 单元测试JUnit框架特性及使用
- 使用Stub进行测试
- Mock技术
- 容器内的测试
- 测试用例执行及数据分析统计
- 持续集成自动化回归测试

测试用例的设计

软件测试金字塔

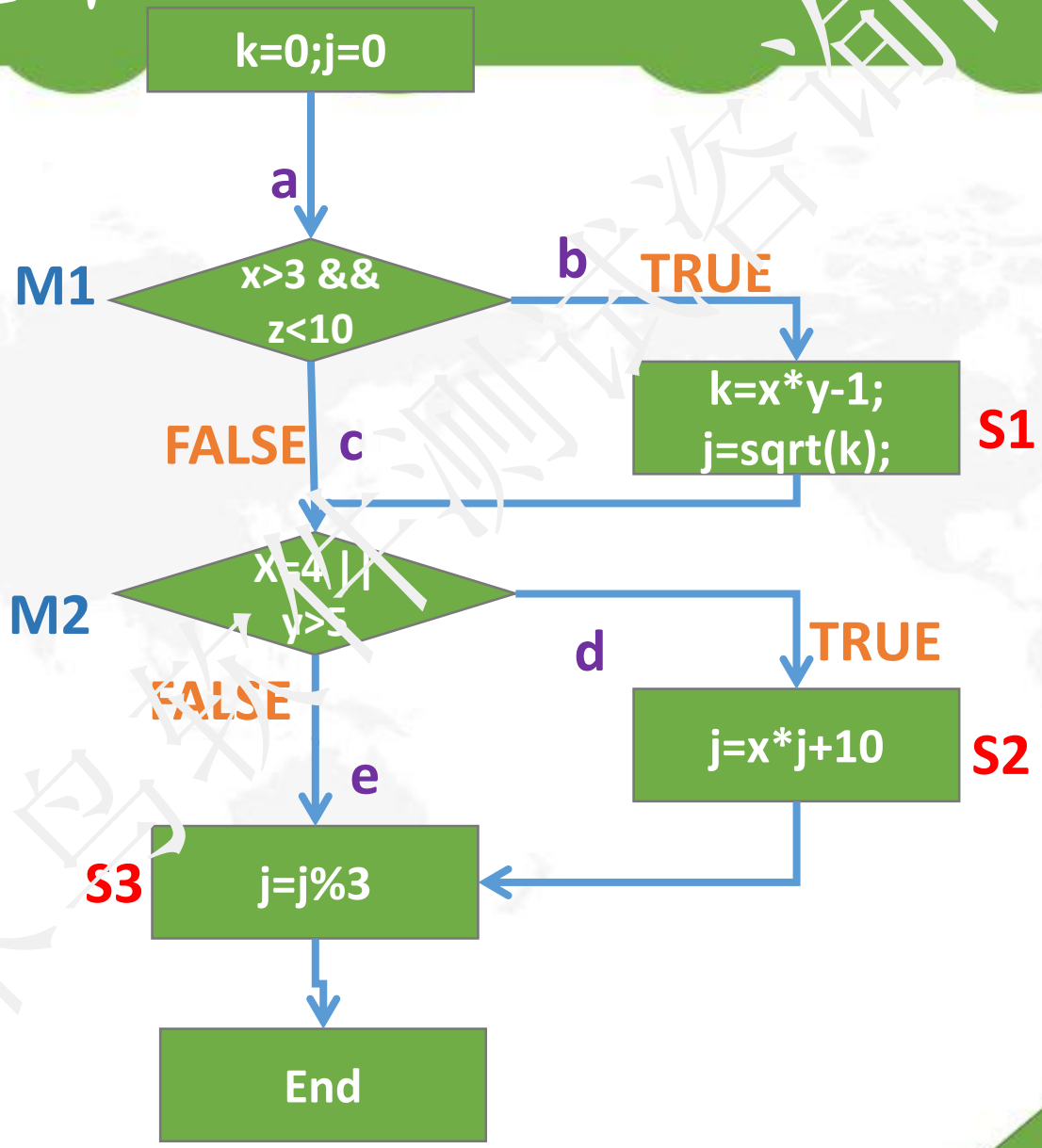


测试用例的设计

语句测试

案例分析

L1： a,c,e
L2： a,b,e
L3： a,c,d
L4： a,b,d



测试用例的设计

语句测试

- 语句测试在顺序语句：覆盖率最简单，只要把顺序语句中的每个语句都覆盖到。

```
int  f (int a, int b){
int c;
c=a+b;
return c;
}
```

语句覆盖率100%测试用例： f(1,2);

语句覆盖=被执行的语句数量/所有语句数量*100%

- 语句覆盖在没有 else的判断语句： 只要执行if语句中的内容就可以了

```
int  f (int a){
int b=0;
if (a>0){
   b=1 ;
   }
return b;
}
```

语句覆盖率100%测试用例： f(1);

测试用例的设计

语句测试

- 语句测试在有 else的判断语句：即要执行if语句，也要执行else中的语句

```
int  f (int a){
int b=0;
 if (a>0){
    b=1;
 } else{
    b=2;
 }
return b;
}
```

语句覆盖率100%测试用例：f(1)，f(0)两个

- 语句测试在循环语句中：循环体内执行一次即可

```
int  f (int a){
for (i=0;i<=a;i++)
    printf(“hello”,s);
}
```

语句覆盖率100%测试用例：f(0);

测试用例的设计

语句测试

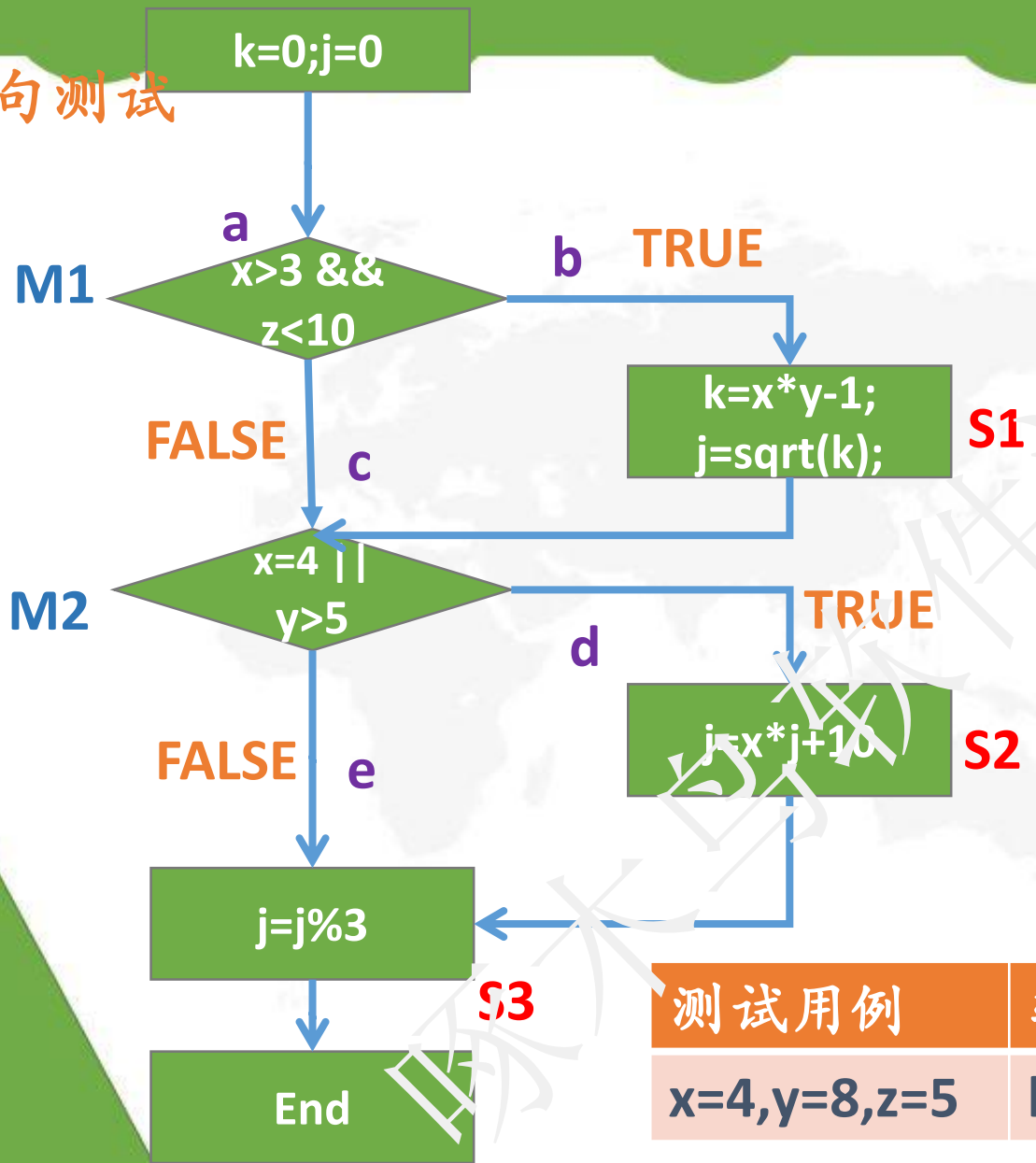
- 语句测试在多条件语句

```
int  f (int a){
        switch (a) {
        case:1 { f1(); break;}
        case:2 { f2(); break;}
        case:3 { f3(); break;}
        case:4 { f4(); break;}
}
```

语句覆盖率100%测试用例：f(1)，f(2)，f(3)，f(4)四个

测试用例的设计

语句测试



L1： a,c,e
L2： a,b,e
L3： a,c,d
L4： a,b,d

测试用例	输出	M1	M2	路径
x=4,y=8,z=5	k=31,j=0	TRUE	TRUE	L4

测试用例的设计

分支测试

分支测试：执行一个测试套件所能覆盖的分支。

分支测试没有 else的判断语句：即要执行if语句为true的情况，也要执行 if语句为false的情况

```
int  f (int a){
int b=0;
if (a>0){
  b=1;
}
return b;
}//分支覆盖率100%测试用例:f(1); f(0);
```

分支覆盖=被执行的分支数量/所有分支数量*100%

分支测试有 else的判断语句：即要执行if语句，也要执行else中的语句

```
int  f (int a){
Int b=0;
 if (a>0){
    b=1;
 } else{
    b=2;
 }
return b;
}
```

分支覆盖率100%测试用例:f(1); f(0);

可见分支覆盖达到100%，语句覆盖也达到100%

测试用例的设计

分支测试

- 分支测试在多条件语句

```
int  f (int a){
        switch (a) {
        case:1 { f1(); break;}
        case:2 { f2(); break;}
        case:3 { f3(); break;}
        case:4 { f4(); break;}
}
```

分支覆盖率100%测试用例：f(1)，f(2)，f(3)，f(4)，f(5) 五个

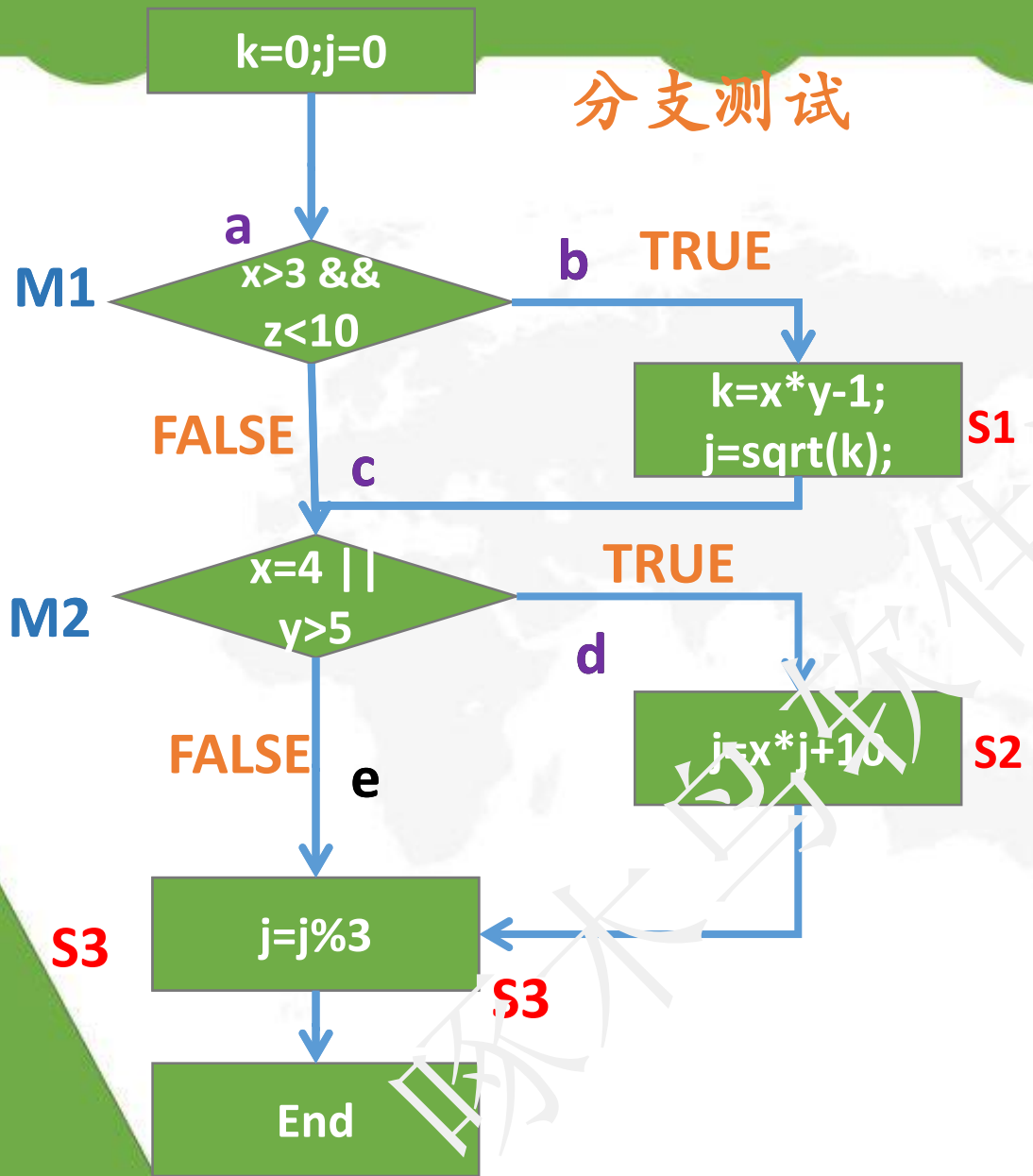
- 分支测试在循环语句中：判断语句必须True or False各执行一次

```
int  f (int a){
for (i=0;i<a;i++)
    printf("hello",s);
}
```

分支覆盖率100%测试用例：f(1);

测试用例的设计

分支测试



案例分析

L1： a,c,e
L2： a,b,e
L3： a,c,d
L4： a,b,d

如果y>5写成y<5,分支测试发现不了错

x=4 || y<5

测试用例	输出	M1	M2	路径
x=4,y=8,z=5	k=31,j=0	True	True	L4
x=2, y=5, z=11	k=0, j=0	False	False	L1
x=13,y=2,z=5	k=25, j=2	True	False	L2
x=4,y=11,z=6	k=0, j=2	False	True	L3

测试用例的设计

条件测试：执行测试套件(test suite)能够覆盖到的条件百分比。100%的条件测试要求测试到每一个条件语句真、假(true,false)的条件。

```
int f (int a, int b){
int c=0;
if ((a>0) &&(b>0)){
   c=1;
}else{
   c=2
}return c;
}
```

a>0	b>0	测试数据
T	T	a=1,b=1
T	F	a=1,b=0
F	T	a=0,b=1
F	F	a=0,b=0

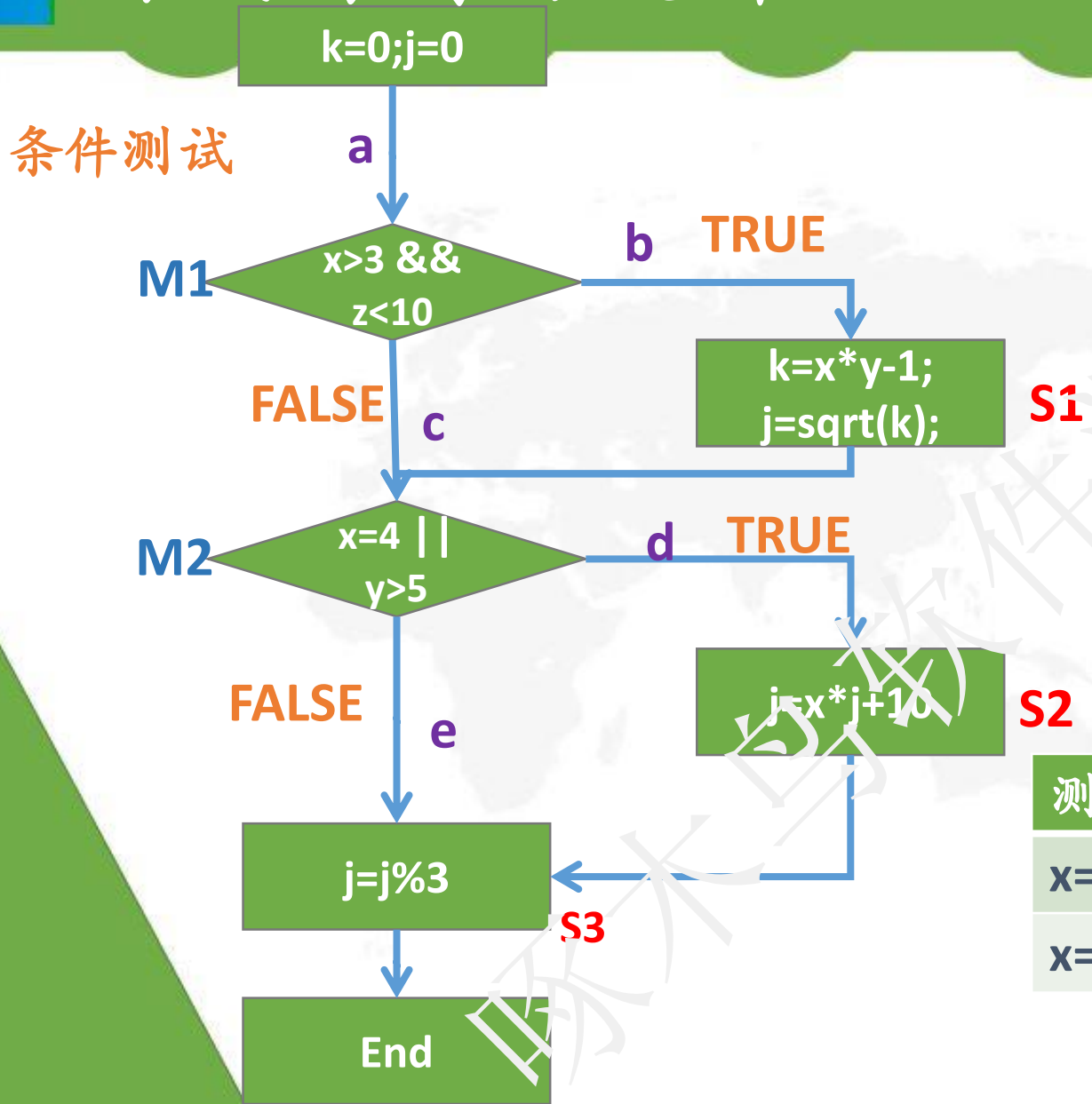
测试用例的设计

条件测试：要保证每个条件取True False各一次有时候是达不到的

```
Int f (int a, int b){
Int c=0;
If ((a>0) &&(a<5)){
   c=1;
}else{
   c=2
}return c;
}
```

a>0	a<5	测试数据
T	T	4
T	F	6
F	T	-1
F	F	?

测试用例的设计



案例分析

T1:x>3 T2:j<10 T3:x=4 T4:y>5

F1:x<=3 F2:j>=10 F3:x<>4 F4:y>=5

L1： a,c,e

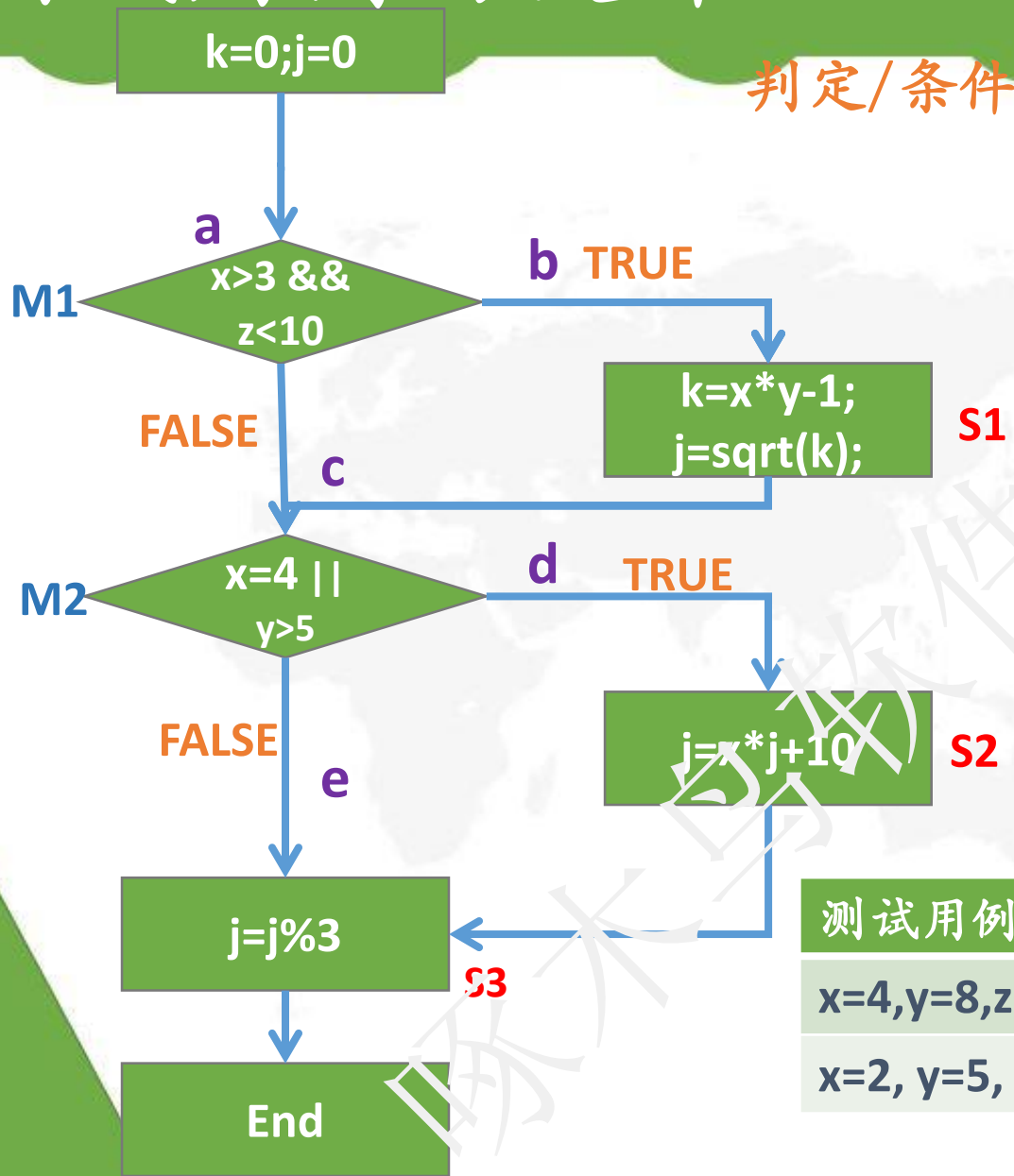
L2： a,b,e

L3： a,c,d

L4： a,b,d

测试用例	输出	原子条件	路径
x=4,y=2,z=11	k=0,j=0	T1 F2 T3 F4	L3
x=2, y=6, z=6	k=0,j=1	F1 T2 F3 T4	L3

测试用例的设计



判定/条件测试：所有条件测试一次，所有判定测试一次

条件测试

T1:x>3 T2:j<10 T3:x=4

T4:y>5

F1:x<=3 F2:j>=10 F3:x<>4

F4:y>=5

L1： a,c,e

L2： a,b,e

L3： a,c,d

L4： a,b,d

判定测试：

M1: (x>3 && z<10)

M2: (x=4 || y> 5)

测试用例	输出	原子条件	M1	M2	路径
x=4,y=8,z=5	k=31,j=0	T1 T2 T3 T4	T	T	L4
x=2, y=5, z=11	k=0, j=0	F1 F2 F3 F4	F	F	L1

测试用例的设计

MC/DC(修订的条件/分支测试)(Modified Condition/Decision Coverage)准则是一种实用的软件结构测试率测试准则,已被广泛地应用于软件验证和测试过程中.

condition 和 decision 的概念:

```
if （A || B && C）{
        Statement;
}
Else{
        Statement2;
```

A,B,C都是一个条件,而(A || B && C)叫一个Decision,如果是条件测试的话只需两个CASE就能测试,就是让这个decision为True和False各一次就能达到即为 TFT, FTF。如果是MC/DC的话就得四个case,而且只比条件数目多一个而已,怎么计算的呢?

定义:在每个判定中的每个条件都曾独立的影响判定的结果至少一次,(独立影响意思是在其他的条件不变的情况下,改变一个条件);

测试用例的设计

A || B && C

总结一句:每个条件对结果都独立起作用

- 比如A对结果起作用的话, B 必须为 false, C必须为 true -- TFT和 *FFT, 这样结果就独立受A的值影响. （A||0&&1）->(A||0)
- 同理如果B对结果独立起作用的话,A必须为false, C必须为 true, 两种情况B为true false各一. (0|| B &&1)-*FFT， *FTT
- 即为而C独立对结果起作用的话就是让(A or B) 为 true, 为了减少case, 上面的case 已经含有这样的case了,我们就取A为false,B为true, 这样C独体起作用的case为： *FTT和 FTF (0||1&& C)->(1&& C)
- 可以看出每个条件各走了一次true和false, 这样三个变量条件就会有六个case, 我们看出其中里面还有两个是重复的(FFT, FTF).
- 需要进一步补充说明的是，MC/DC测试的主要目的是为了防止在组合条件表达式中包含副作用(side effect)，如以下语句：

```
if (a() || b() || c()){ … }
```

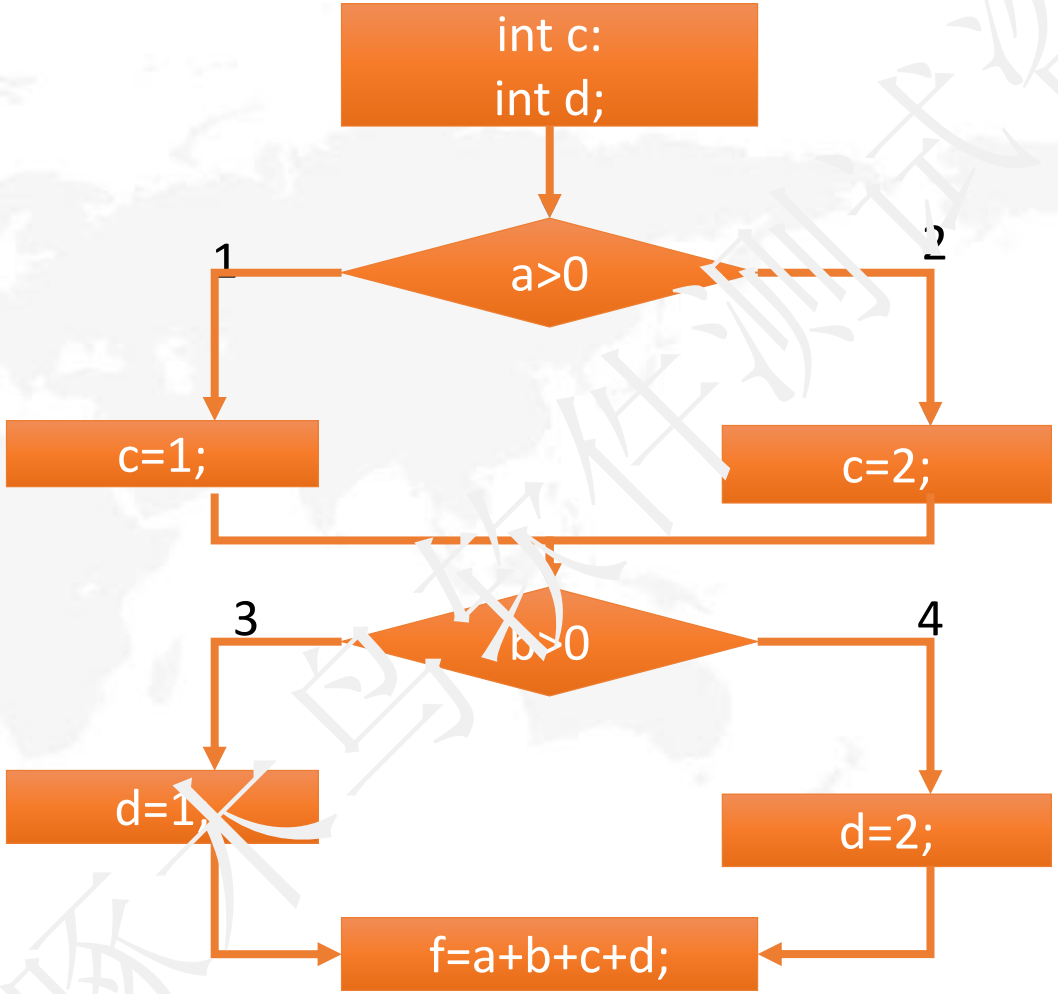
当b函数或C函数产生副作用时，MC/DC测试存在非常大的必要性。

原则上不应在组合条件表达式中调用产生副作用的函数。

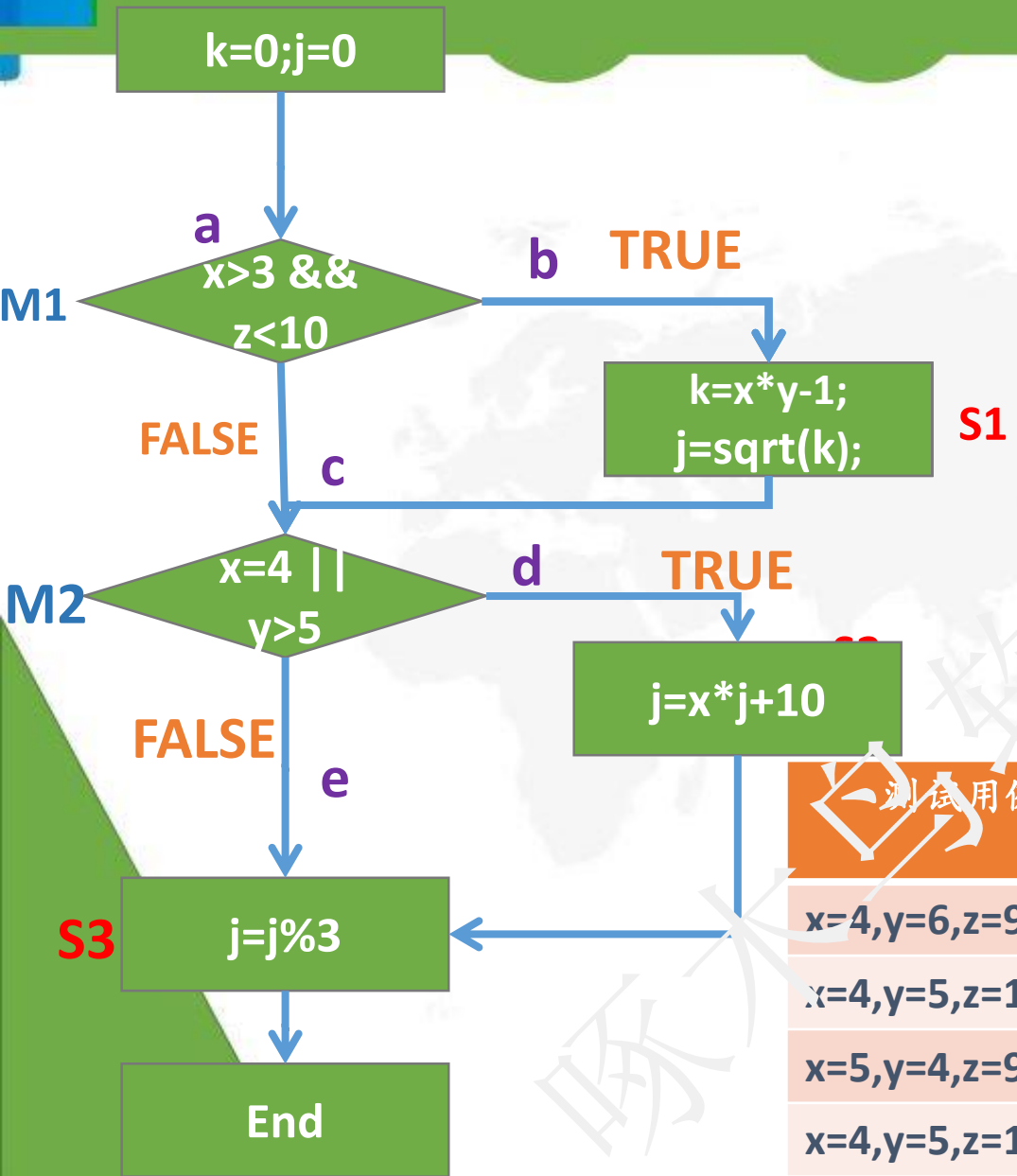
测试用例的设计

路径测试：又叫条件组合测试，所有条件组合至少执行一次

a	b	测试路径
1	1	1，3
1	0	1，4
0	1	2，3
0	0	2，4



测试用例的设计



条件测试

T1:x>3 T2:j<10 T3:x=4
T4:y>5
F1:x<=3 F2:j>=10
F3:x<>4 F4:y>=5

L1： a,c,e
L2： a,b,e
L3： a,c,d
L4： a,b,d

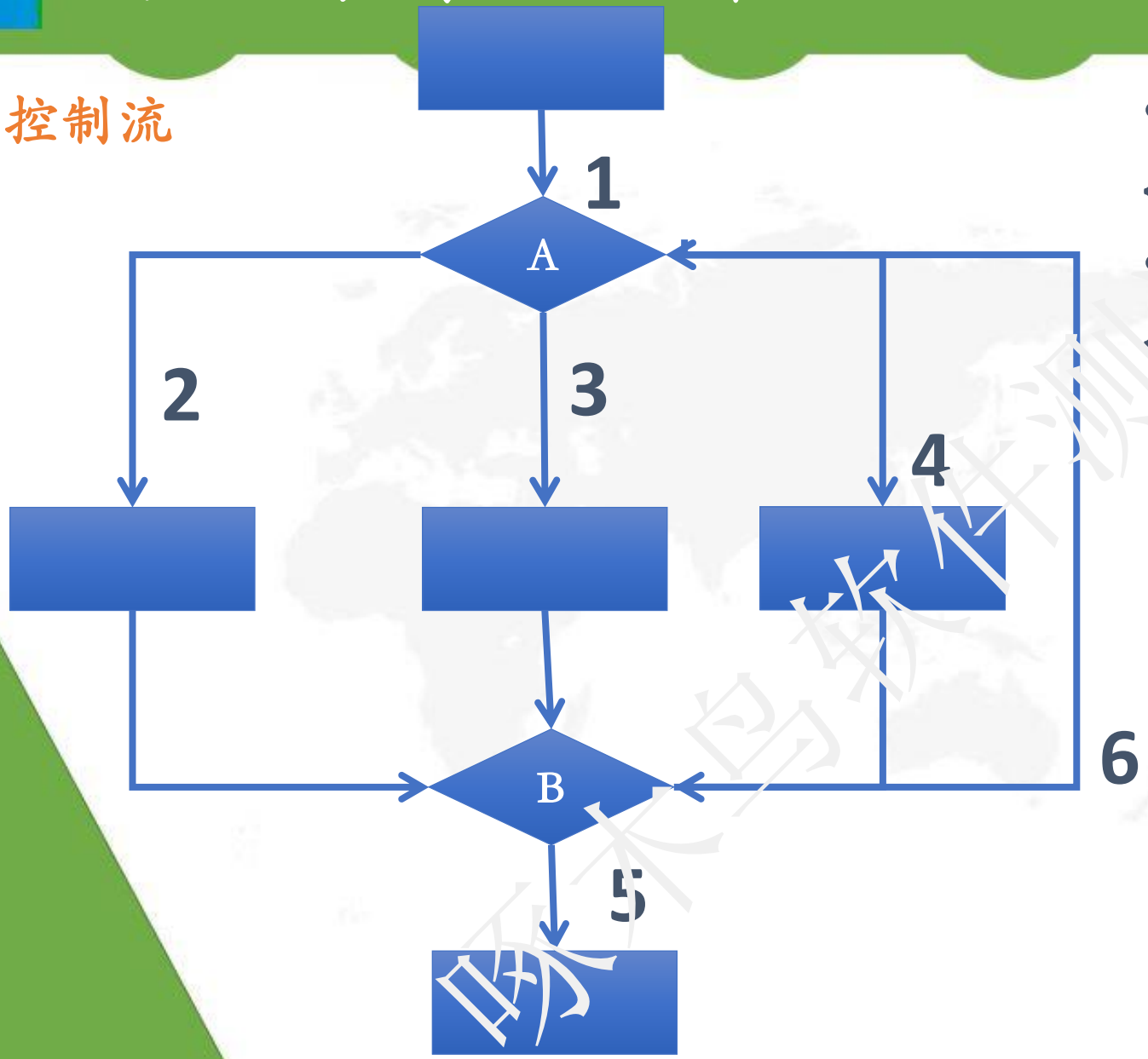
判定测试：

M1: (x>3 && z<10)
M2: (x=4 || y> 5)

测试用例	原子条件 x>3 ， z<10	M1 x>3 &&z<10	原子条件 x=4 ， y>5	M2 x=4 ||y>5	覆盖路径
x=4,y=6,z=9	T1 T2	True	T3 T4	True	L4
x=4,y=5,z=10	T1 F2	False	T3 F4	True	L3
x=5,y=4,z=9	T1 T2	True	F3 F4	False	L2
x=4,y=5,z=10	F1 F2	False	F3 F4	False	L1

测试用例的设计

控制流



- 对经过A点进行排序：

{1,2} {1,3} {1,4} {6,2} {6,3} {6,4}

- 对经过B点进行排序：

{2,5} {3,6} {4,6} {2,5} {3,5} {4,5}

测试用例的设计

控制流

{1,2,5}
{1,3,5}
{1,4,5}
{1,2,6,2,5}
{1,3,6,4,6,3,5}

排序

{1,2} {1,3} {1,4} {2,5} {2,6} {3,5} {3,6} {4,5} {4,6} {6,2} {6,3} {6,4}

1 开始 5结束

- 挑选：{1,2} {2,5}->{1,2,5}

{1,2} {1,3} {1,4} {2,5} {2,6} {3,5} {3,6} {4,5} {4,6} {6,2} {6,3} {6,4}

- 挑选：{1,3} {3,5} ->{1,3,5}

{1,2} {1,3} {1,4} {2,5} {2,6} {3,5} {3,6} {4,5} {4,6} {6,2} {6,3} {6,4}

- 挑选：{1,4} {4,5} ->{1,4,5}

{1,2} {1,3} {1,4} {2,5} {2,6} {3,5} {3,6} {4,5} {4,6} {6,2} {6,3} {6,4}

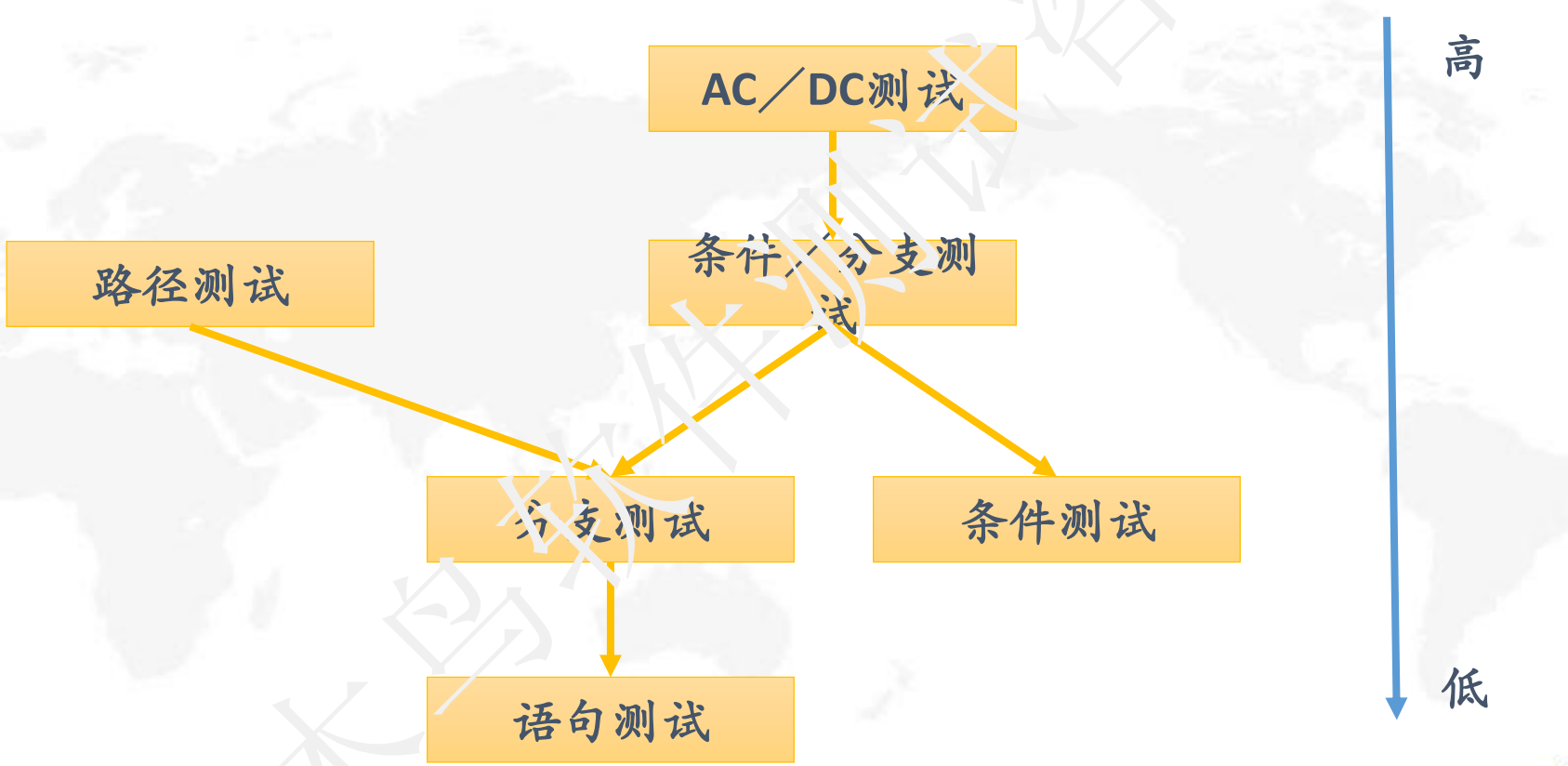
- 选择：{1,2} {2,6} {6,2} {2,5}->{1,2,6,2,5}

{1,2} {1,3} {1,4} {2,5} {2,6} {3,5} {3,6} {4,5} {4,6} {6,2} {6,3} {6,4}

- 选择：{1,3} {3,6} {6,4} {4,6} {6,3} {3,5}->{1,3,6,4,6,3,5}

{1,2} {1,3} {1,4} {2,5} {2,6} {3,5} {3,6} {4,5} {4,6} {6,2} {6,3} {6,4}

测试用例的设计



测试用例的设计

白盒测试还包括静态评审和基于函数的测试（有的地方也认为是函数级别的黑盒测试），在这里不做详细介绍。

白盒测试可以在单元、集成测试中执行，也可以在系统测试与验收测试中执行。

- 单元、集成测试：XUnit系列，如：JavaUnit、CPPUnit、unittest；
- 系统、验收测试：QAC/QAC++，星云测试工具。

持续集成单元测试方法与应用

- 测试用例的设计
- 单元测试JUnit框架特性及使用
- 使用Stub进行测试
- Mock技术
- 容器内的测试
- 测试用例执行及数据分析统计
- 持续集成自动化回归测试

单元测试JUnit框架特性及使用

- 什么是单元测试
- 测试驱动和测试桩
- 白盒测试和黑盒测试
- JAVA单元测试框架JUNIT介绍

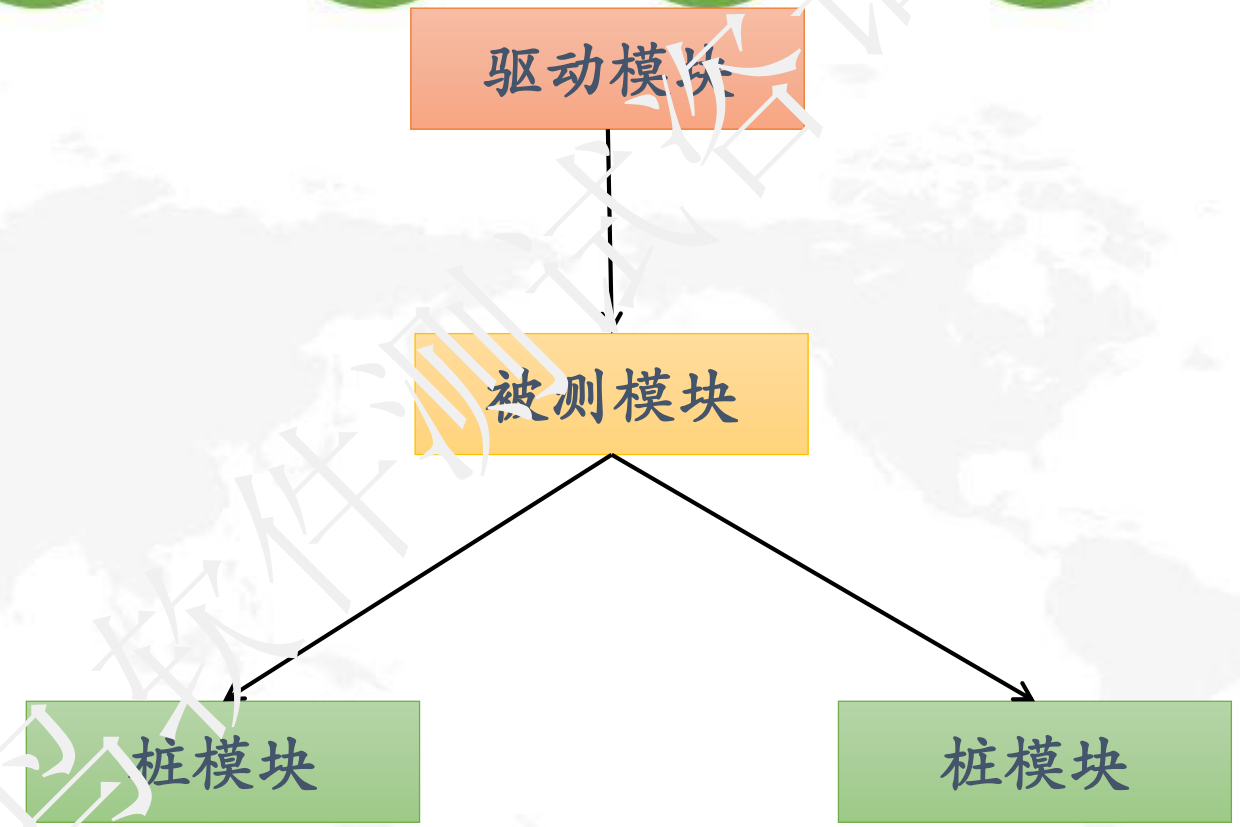
什么是单元测试

单元测试
测试组件功能
测试依据:

- 组件需求说明;
- 详细设计文档;
- 代码。

典型测试对象:

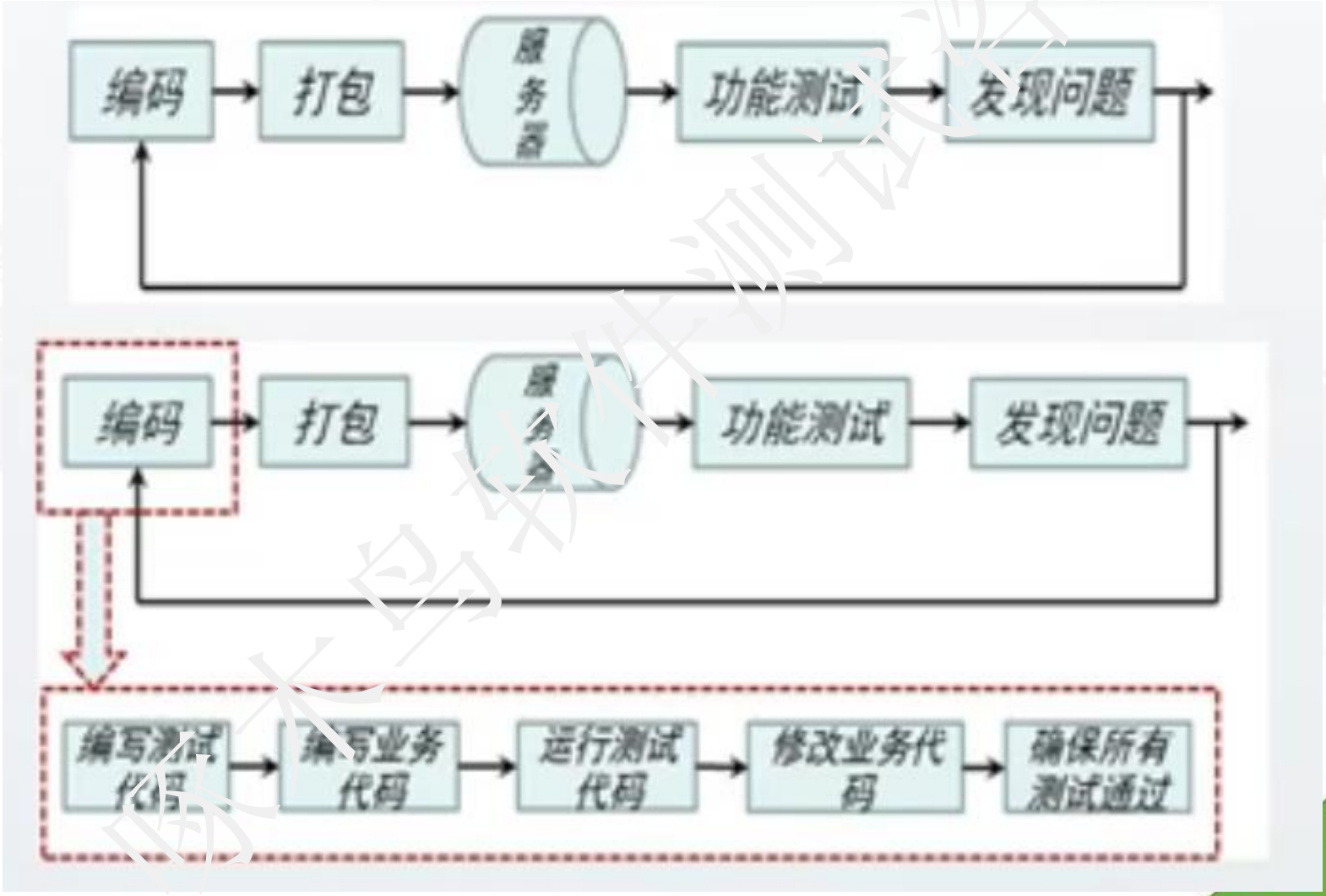
- 组件;
- 程序;
- 数据转换/移植程序;
- 数据库模型。



单元测试与编码同时进行

什么是单元测试

为什么要做单元测试



什么是单元测试

为什么要做单元测试?

无单元测试

有单元测试

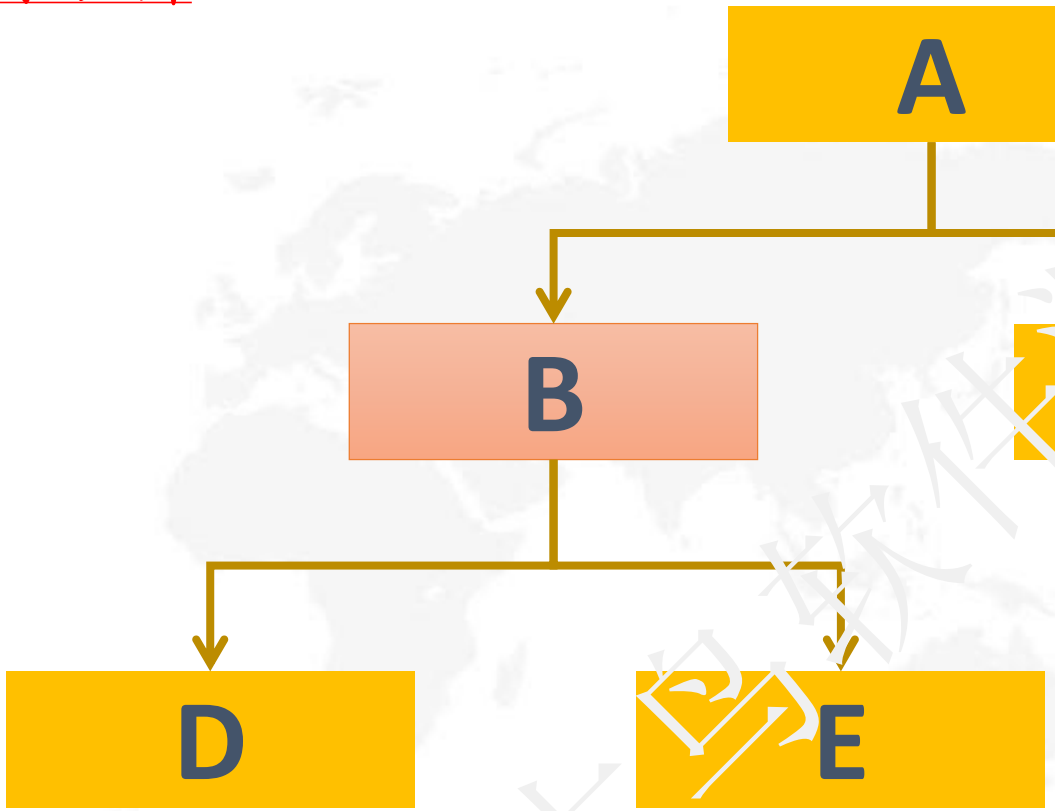
自动检查修改
是否引入错误

单元测试JUnit框架特性及使用

- 什么是单元测试
- 测试驱动和测试桩
- 白盒测试和黑盒测试
- JAVA单元测试框架JUNIT介绍

测试驱动和测试桩

案例分析



如何对函数B进行单元测试

```
int  B (int a, int b){
      int x= D (a);
      int y= E(b);
      return x+y;
}
```

测试驱动和测试桩

```
@Test
public void testB (){
    int a=3;
    int b=5;
    int c=B(a,b)
}
```

测试函数 testB ()为B的**驱动函数**

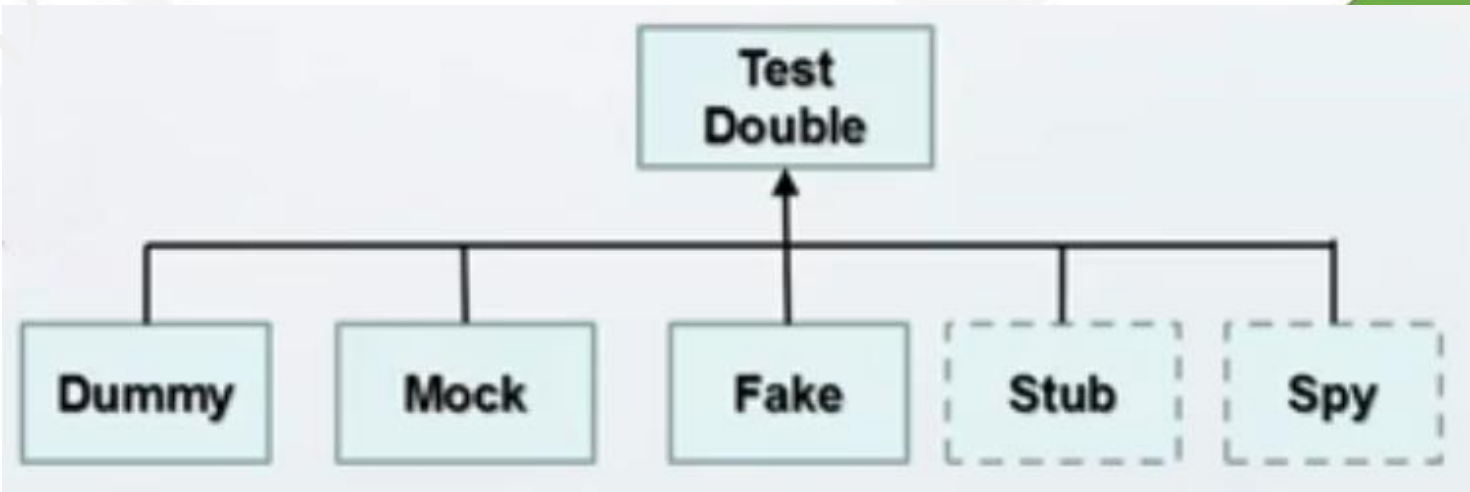
```
int  StubD(int x){
    return x+5;
}
int  StubE(int x){
    return x+6;
}
```

StubD, StubE为桩函数

B函数改造为

```
Int B (int a, int b){
    int x= StubD (a);
    int y= StubE(b);
    return x+y;
}
```

关于桩的其他技术：



测试驱动和测试桩

Java单元测试最好工具JUnit

比如某个函数要实现a+b的平均数

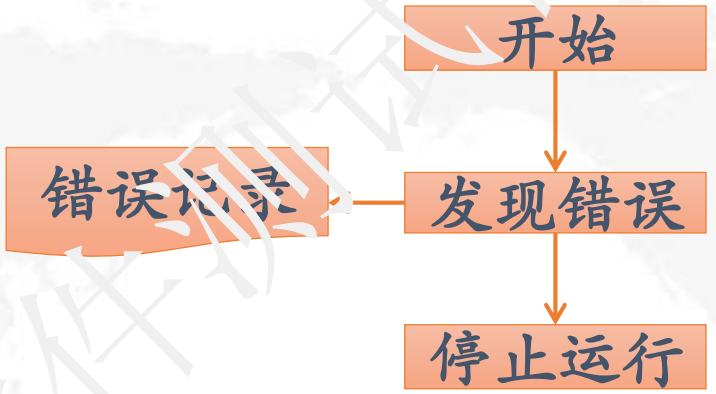
产品代码:

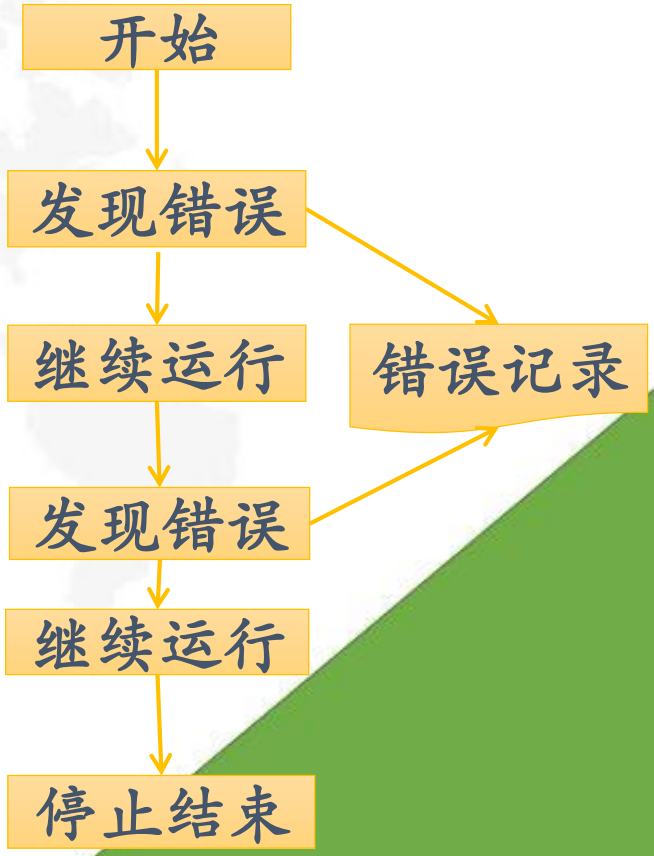
```
int ave(a,b){
...
}
```

测试函数

```
testave(){
    return(ave(4,8),6);
    return(ave(3,4),3.5);
}
```

如返回True则测试通过，否则测试不通过

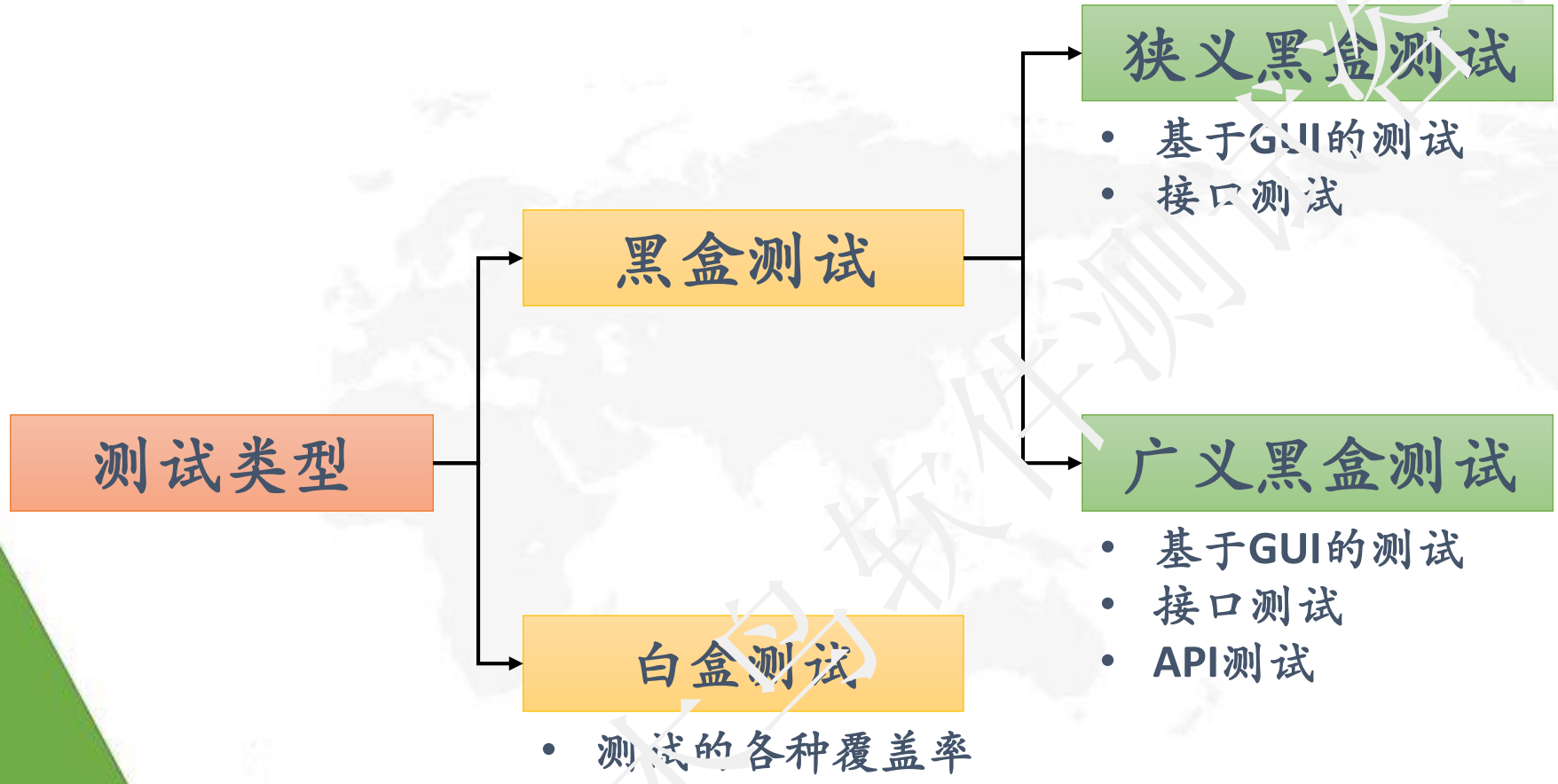




单元测试JUnit框架特性及使用

- 什么是单元测试
- 测试驱动和测试桩
- 白盒测试和黑盒测试
- JAVA单元测试框架JUNIT介绍

白盒测试和黑盒测试


测试类型
黑盒测试
狭义黑盒测试
基于GUI的测试
接口测试
广义黑盒测试
基于GUI的测试
接口测试
API测试
白盒测试
测试的各种覆盖率

单元测试JUnit框架特性及使用

- 什么是单元测试
- 测试驱动和测试桩
- 白盒测试和黑盒测试
- JAVA单元测试框架JUNIT介绍

JAVA单元测试框架JUNIT介绍

- 基本的JUNIT测试框架
- JUNIT断言
- JUNIT测试的高级技巧
- MAVEN+testNG介绍
- 练习

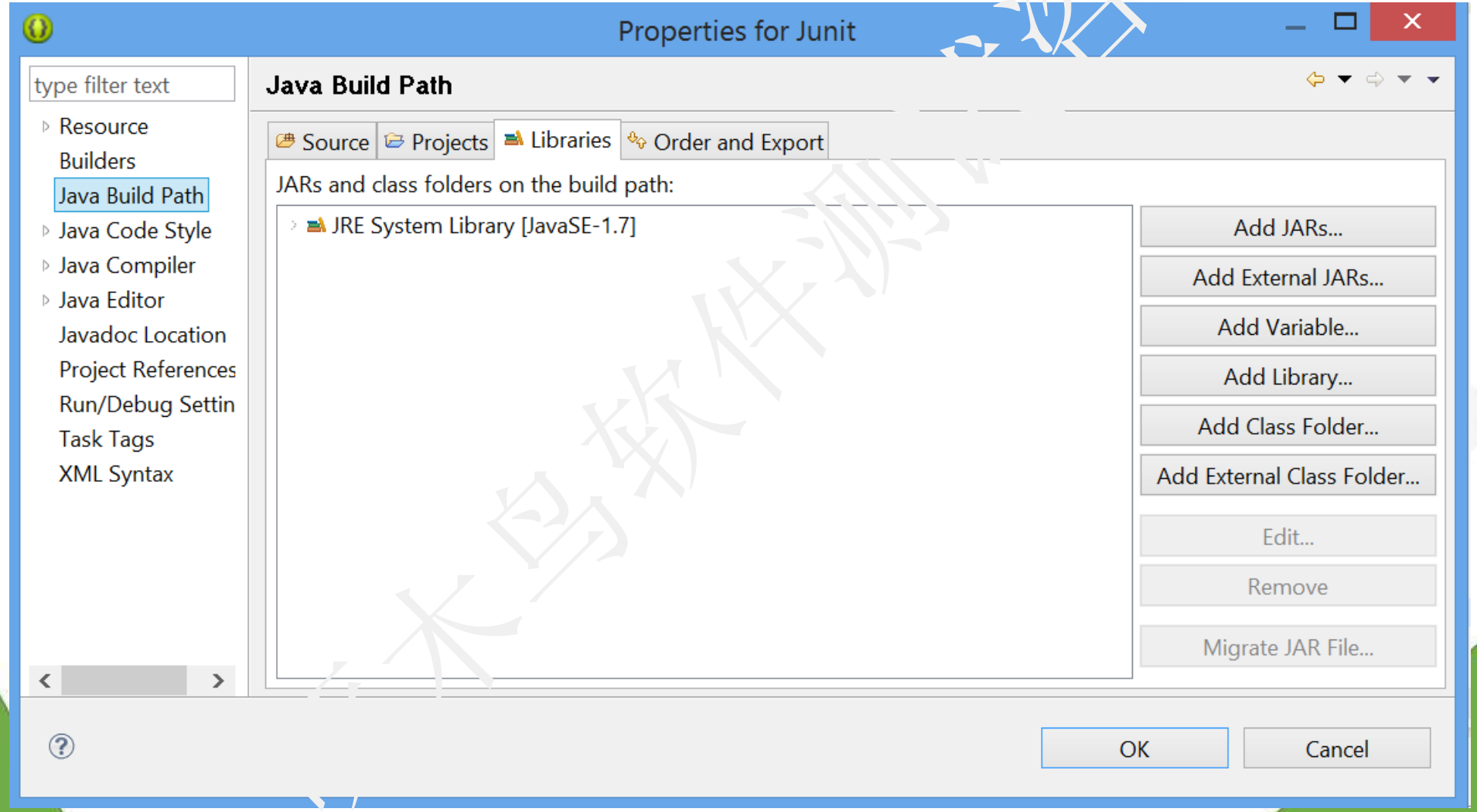
基本的JUNIT测试框架

新建一个项目，在这个项目中我们编写一个Calculator类

```
package com.jerry;

public class Calculator {
   private static int result; // 静态变量，用于存储运行结果
   public void add(int n) {
      result = result + n;
   }
   public void substract(int n) {
      result = result - 1;  //Bug: 正确的应该是result =result-n
   }
   public void multiply(int n) {
   }      // 此方法尚未写好
   public void divide(int n) {
      result = result / n;
   }
   public void square(int n) {
      result = n * n;
   }
   public void squareRoot(int n) {
      for (; ;) ;          //Bug : 死循环
   }
   public void clear() {    // 将结果清零
      result = 0;
   }
   public int getResult() {
      return result;
   }
}
```

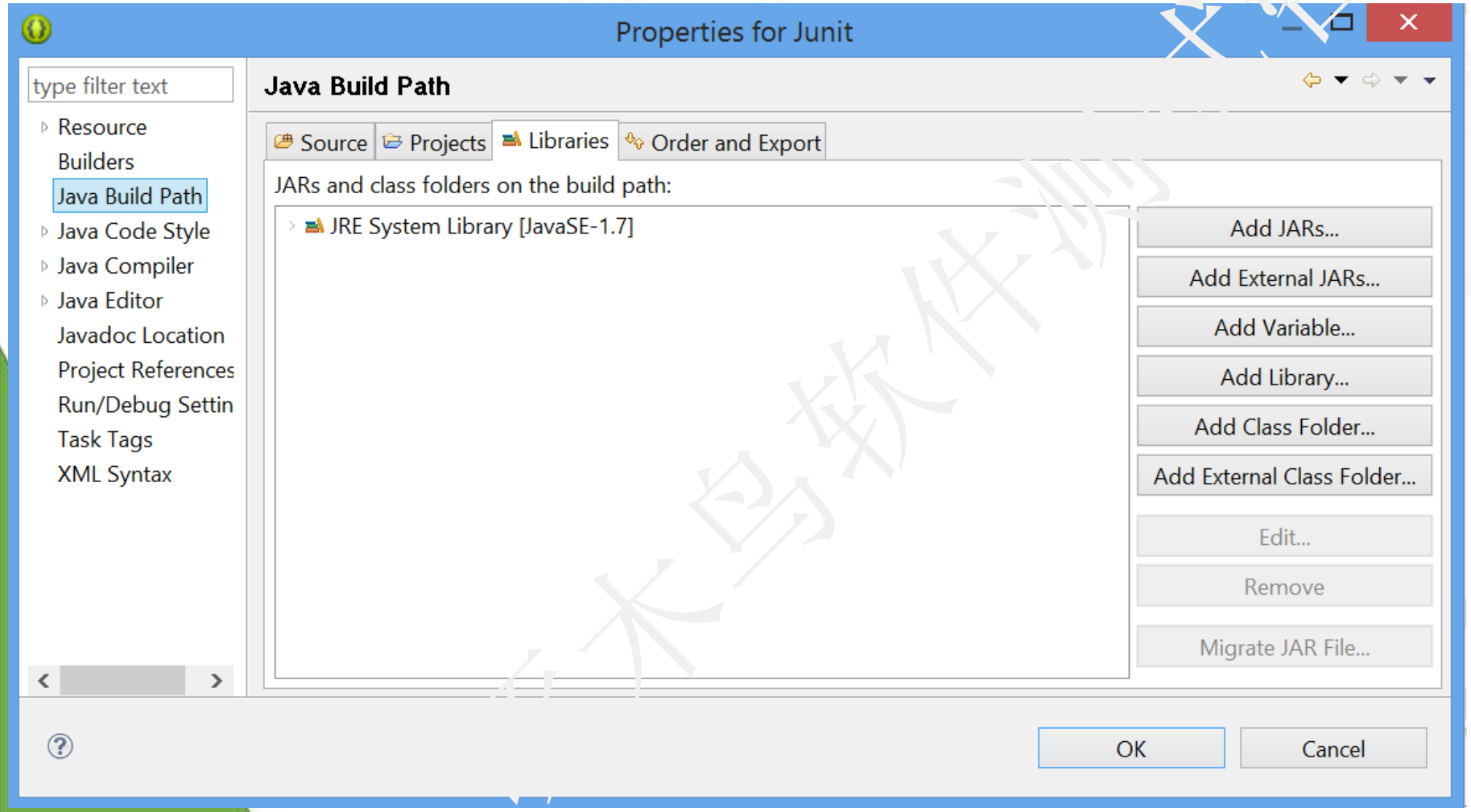
基本的JUNIT测试框架

将JUnit4单元测试包引入这个项目：在该项目上点右键，点“属性”



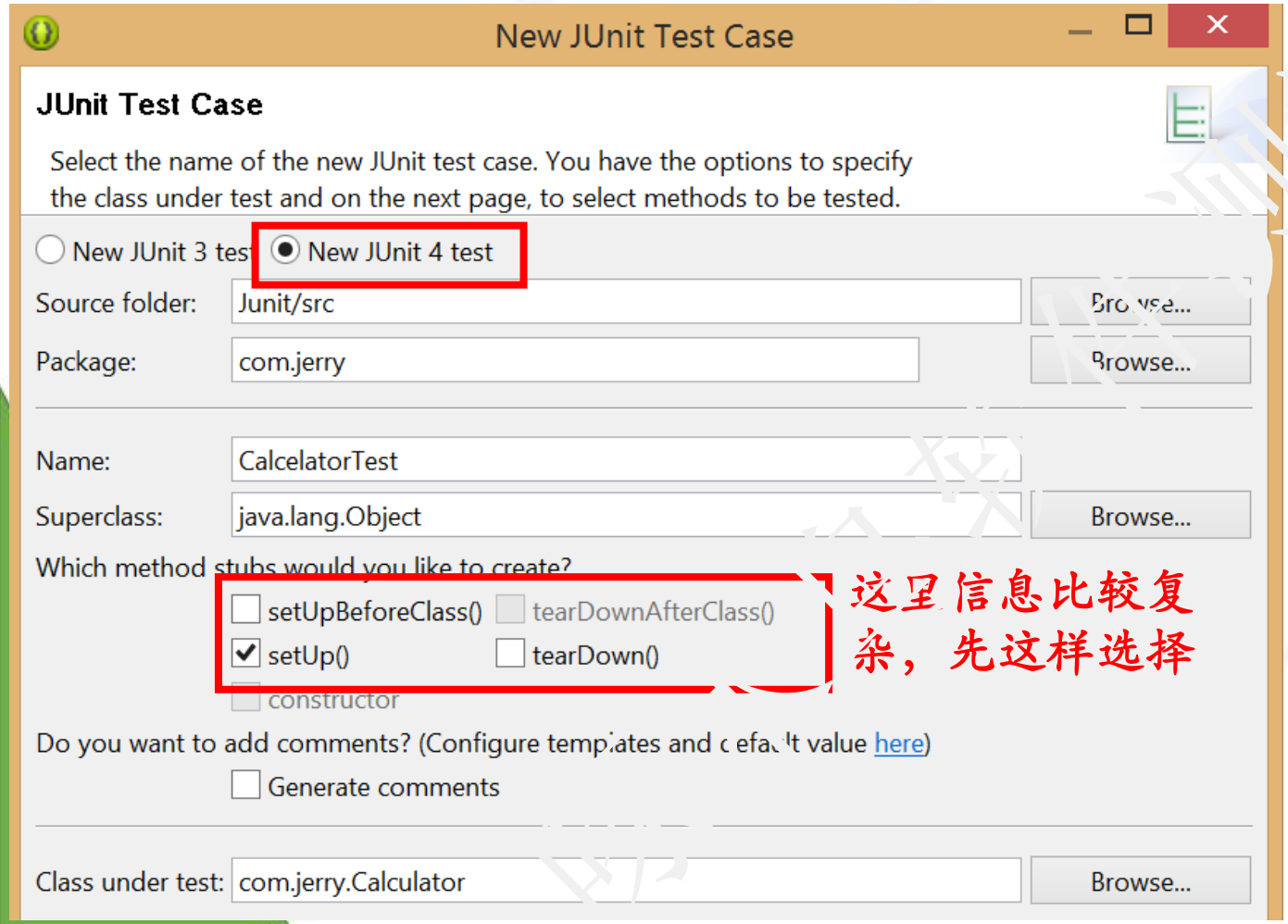
基本的JUNIT测试框架

在弹出的属性窗口中，首先在左边选择“Java Build Path”，然后到右上选择“Libraries”标签，之后在最右边点击“Add Library...”按钮



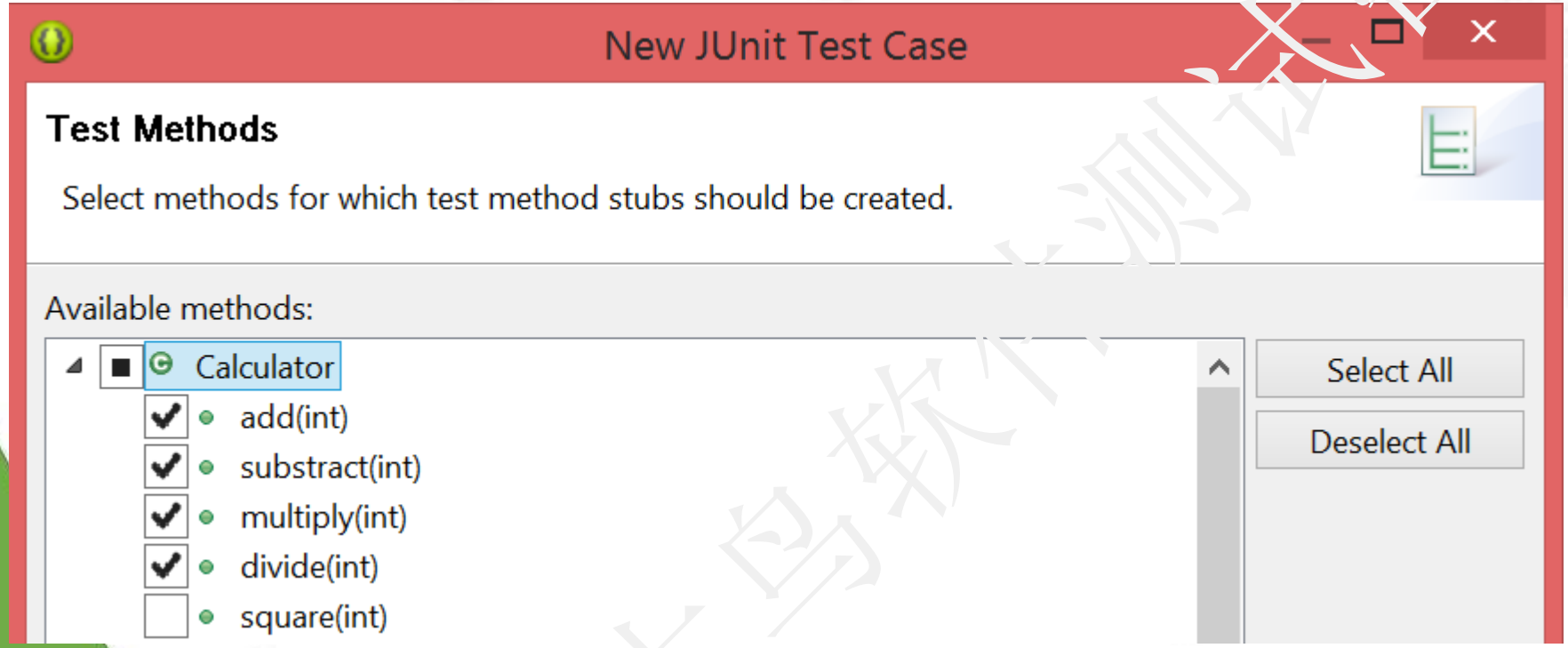
基本的JUNIT测试框架

生成JUnit测试框架：在Eclipse的Package Explorer中用右键点击该类弹出菜单，选择“New a JUnit4 Test Case”，在弹出的对话框中，进行相应的选择，如下图所示。



基本的JUNIT测试框架

点击“下一步”后，系统会自动列出你这个类中包含的方法，选择你要进行测试的方法。此例中，我们仅对“加、减、乘、除”四个方法进行测试。



基本的JUNIT测试框架

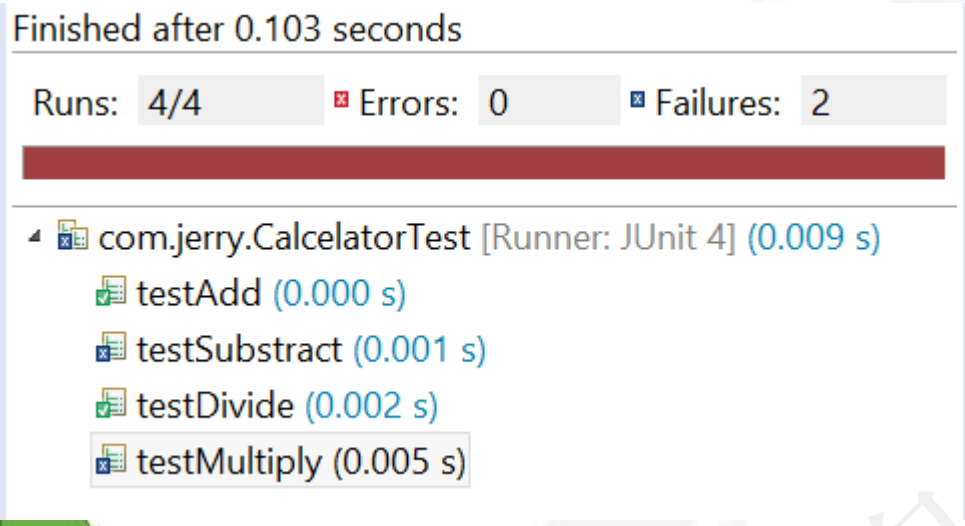
之后系统会自动生成一个新类CalculatorTest，里面包含一些空的测试用例。你只需要将这些测试用例稍作修改即可使用。

```
package com.jerry;

import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;
import org.junit.Ignore;

public class CalcelatorTest {
private static Calculator calculator = new Calculator();
@Before
public void setUp() throws Exception {
      calculator.clear();
}
public void testAdd() {
      calculator.add(2);
      calculator.add(3);
      assertEquals(5, calculator.getResult());
}
```

```
@Test
public void testSubstract() {
      calculator.add(10);
      calculator.substract(2);
      assertEquals(8, calculator.getResult());
}
@Test
public void testMultiply() {
      fail("Not yet implemented");
}
@Test
public void divide(int n) throws
java.lang.ArithmeticException{
            result = result/n;
      }
```

基本的JUNIT测试框架

运行测试代码：按照上述代码修改完毕后，我们在CalculatorTest类上点右键，选择“Run As a JUnit Test”来运行我们的测试用例了。

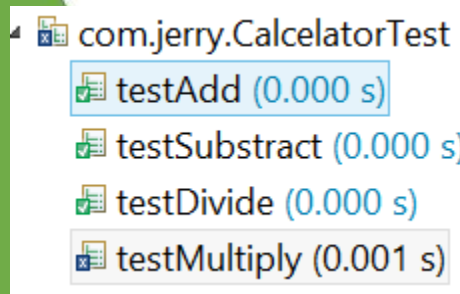


减法是有问题的所以没有通过，乘法没有完成所以也没有通过。

我们把减法修改正确，

```
public void substract(int n) {
    result = result-n;
}
```

运行后如图：



基本的JUNIT测试框架

包含必要地Package

```
import static org.junit.Assert.*;
```

assertEquals是Assert类中的一系列的静态方法，一般的使用方式是Assert. assertEquals()，但是使用了静态包含后，前面的类名就可以省略了，使用起来更加的方便。

比如：

```
assertEquals(8, calculator.getResult());
```

测试类的声明

测试类是一个独立的类，没有任何父类。测试类的名字也可以任意命名，没有任何局限性。所以我们不能通过类的声明来判断它是不是一个测试类，它与普通类的区别在于它内部的方法的声明。

基本的JUNIT测试框架

创建一个待测试的对象

你要测试哪个类，那么你首先就要创建一个该类的对象。

```
private static Calculator calculator = new Calculator();
```

为了测试Calculator类，我们必须创建一个calculator对象。

基本的JUNIT测试框架

测试方法的声明

在测试类中，并不是每一个方法都是用于测试的，你必须使用“标注”来明确表明哪些是测试方法。“标注”也是JDK5的一个新特性，用在此处非常恰当。我们可以看到，在某些方法的前有@Before、@Test、@Ignore等字样，这些就是标注，以一个“@”作为开头。

基本的JUNIT测试框架

在方法的前面使用@Test标注，以表明这是一个测试方法。对于方法的声明也有如下要求：名字可以随便取，没有任何限制，但是返回值必须为void，而且不能有任何参数。如果违反这些规定，会在运行时抛出一个异常。

```
@Test
  public void testAdd() {
      calculator.add(2);
      calculator.add(3);
      assertEquals(5, calculator.getResult());
  }
```

- 在测试方法中调用几次add函数，初始值为0，先加2，再加3，我们期待的结果应该是5。如果最终实际结果也是5，则说明add方法是正确的，反之说明它是错的。
- assertEquals(5, calculator.getResult());就是来判断期待结果和实际结果是否相等，第一个参数填写期待结果，第二个参数填写实际结果，也就是通过计算得到的结果。这样写好之后，JUnit会自动进行测试并把测试结果反馈给用户。

JAVA单元测试框架JUNIT介绍

基本的JUNIT测试框架

JUNIT断言

JUNIT测试的高级技巧

MAVEN+testNG介绍

练习

JUNIT断言

1，assertEquals([String message],expected,actual)

message：可选，假如提供，将会在发生错误时报告这个消息

expected：是期望值，通常都是用户指定的内容。

actual：是被测试的代码返回的实际值。

例：assertEquals("equals","1","1");

2，assertEquals([String message],expected,actual,tolerance)

message：是个可选的消息，假如提供，将会在发生错误时报告这个消息。

expected：是期望值，通常都是用户指定的内容。

actua：是被测试的代码返回的实际值。

toleranc：是误差参数，参加比较的两个浮点数在这个误差之内则会被认为是相等的。

例：assertEquals ("yes",6.6,13.0/2.0,0.5);

3，assertTrue ([String message],Boolean condition)

message:是个可选的消息，假如提供，将会在发生错误时报告这个消息。

condition:是待验证的布尔型值。

该断言用来验证给定的布尔型值是否为真，假如结果为假，则验证失败。

例： assertTrue("true",0==0);

JUNIT断言

4，assertFalse([String message],Boolean condition)

message:是个可选的消息，假如提供，将会在发生错误时报告这个消息。

condition:是待验证的布尔型值。

该断言用来验证给定的布尔型值是否为假，假如结果为真，则验证失败。

例： assertFalse("false",4==5);

5，assertNull([String message],Object object)

message:是个可选的消息，假如提供，将会在发生错误时报告这个消息。

object:是待验证的对象。

该断言用来验证给定的对象是否为null，假如不为null，则验证失败。

例： assertNull("null",null);

6，assertNotNull([String message],Object object)

message:是个可选的消息，假如提供，将会在发生错误时报告这个消息。

object:是待验证的对象。

该断言用来验证给定的对象是否为null，假如不为null，则验证失败。

该断言用来验证给定的对象是否为非null，假如为null，则验证失败。

例： assertNotNull("not null",new String());

JUNIT断言

7，assertSame ([String message], expected,actual)

message:是个可选的消息，假如提供，将会在发生错误时报告这个消息。

expected:是期望值。

actual:是被测试的代码返回的实际值。

该断言用来验证expected参数和actual参数所引用的是否是同一个对象，假如不是，则验证失败。

例： assertSame("same",6,3+3);

8，assertNotSame ([String message], expected,actual)

message:是个可选的消息，假如提供，将会在发生错误时报告这个消息。

expected:是期望值。

actual:是被测试的代码返回的实际值。

该断言用来验证expected参数和actual参数所引用的是否是不同对象，假如所引用的对象相同，则验证失败。

例： assertNotSame("not same",5,4+2);

9，fail([String message])

message是个可选的消息，假如提供，将会在发生错误时报告这个消息。

该断言会使测试立即失败，通常用在测试不能达到的分支上（如异常）。

JAVA单元测试框架JUNIT介绍

基本的JUNIT测试框架

JUNIT断言

JUNIT测试的高级技巧

MAVEN+testNG介绍

练习

JUNIT测试的高级技巧

@BeforeClass 和 @AfterClass

有一个类是负责对大文件（超过 500 兆）进行读写，他的每一个方法都是对文件进行操作。换句话说，在调用每一个方法之前，我们都要打开一个大文件并读入文件内容，这绝对是一个非常耗费时间的操作。如果我们使用 @Before 和 @After，那么每次测试都要读取一次文件，效率及其低下。

@BeforeClass 和 @AfterClass 两个 Fixture 来帮我们实现这个功能。从名字上就可以看出，用这两个 Fixture 标注的函数，只在测试用例初始化时执行 @BeforeClass 方法，当所有测试执行完毕之后，执行 @AfterClass 进行收尾工作。在这里要注意一下，每个测试类只能有一个方法被标注为 @BeforeClass 或 @AfterClass，并且该方法必须是 Public 和 Static 的。

JUNIT测试的高级技巧

防止超时

比如程序里面存在死循环，如何处理。

```
public void  squareRoot( int  n) {
        for  (; ;) ;                    // Bug : 死循环
    }
```

要实现这一功能，只需要给 @Test 标注加--个参数即可

```
@Test(timeout=1000 ) //1000毫秒
     public   void  squareRoot() {
       calculator.squareRoot( 4 );
       assertEquals( 2 , calculator.getResult());

   }
```

JUNIT测试的高级技巧

测试异常

方案一：可以在任何JUnit版本中运行

```
@Test
        public void TheFirstDivideByZeroThrowsException() {
            try {
                    calculator.add(8);
                    calculator.divide(0);
                fail("测试除数为0失败");
            } catch (java.lang.ArithmeticException anArithmeticException) {
                    assertTrue(anArithmeticException.getMessage().contains("/ by zero"));
            }
        }
```

JUNIT测试的高级技巧

测试异常

方案二：JUnit4中运行

```
@Test(expected = java.lang.ArithmeticException.class)
        public void TheSecondDivideByZeroThrowsException(){
                calculator.add(8);
                calculator.divide(0);
        }
```

JUNIT测试的高级技巧

测试异常

方案三：JUnit4加强版

```
@Rule
public ExpectedException thrown = ExpectedException.none();

@Test
public void TheThirdDivideByZeroThrowsException()
        throws java.lang.ArithmeticException {
        thrown.expect(java.lang.ArithmeticException.class);
        thrown.expectMessage("/ by zero");
        calculator.add(8);
        calculator.divide(0);
}
```

JUNIT测试的高级技巧

Runner（运行器）

把测试代码提交给JUnit 框架后，框架通过Runner来运行如何来运行你的代码。

不设定，为默认的，也可以设置

```
import  org.junit.internal.runners.TestClassRunner;
import  org.junit.runner.RunWith;
// 使用了系统默认的TestClassRunner，与下面代码完全一样
public   class  CalculatorTest  {
  …
 }
@RunWith(BlockJUnit4ClassRunner.class )
 public   class  CalculatorTest  {
 …
 }
```

JUNIT测试的高级技巧

参数化测试

测试一下“计算一个数的平方”这个函数，暂且分三类：正数、0、负数。

测试代码如下：

```
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;
public class AdvancedTest {
private static Calculator calculator = new Calculator();
@Before
public void clearCalculator() {
        calculator.clear();    }
```

JUNIT测试的高级技巧

```
@Test
 public  void  square1() {
   calculator.square( 2 );
   assertEquals( 4 , calculator.getResult());
}

@Test
 public  void  square2() {
   calculator.square( 0 );
   assertEquals( 0 , calculator.getResult());
}
```

```
@Test
  public  void  square3() {
    calculator.square( - 3 );
    assertEquals( 9 , calculator.getResult());
  }
}
```

可以用参数化来取代

JUNIT测试的高级技巧

```
import  static  org.junit.Assert.assertEquals;
import  org.junit.Test;
 import  org.junit.runner.RunWith;
import  org.junit.runners.Parameterized;
import  org.junit.runners.Parameterized.Parameters;
import  java.util.Arrays;
import  java.util.Collection;

 @RunWith(Parameterized. class )
  public   class  SquareTest  {
 private   static  Calculator calculator  =  new Calculator();
private   int  param;
private   int  result;
@Parameters
     public   static  Collection data() {
        return  Arrays.asList( new  Object[][] {
             { 2 , 4 },
             { 0 , 0 },
             {-3 , 9 },
        });
     }

// 构造函数，对变量进行初始化
      public  void SquareTest( int  param,  int
result)  {
          this .param=param;
          this .result =result;
     }
     @Test
public   void  square()  {
         calculator.square(param);
         assertEquals(result,
calculator.getResult());
     }
 }
```

JUNIT测试的高级技巧

打包测试

```
import  org.junit.runner.RunWith;
import  org.junit.runners.Suite;

@RunWith(Suite. class )
@Suite.SuiteClasses( {
      CalculatorTest. class ,
      SquareTest. class
      } )
public   class  AllCalculatorTests {
}
```

JAVA单元测试框架JUNIT介绍

基本的JUNIT测试框架

JUNIT断言

JUNIT测试的高级技巧

MAVEN+testNG介绍

练习

MAVEN+testNG介绍

TestNG

TestNG是一个测试框架，其灵感来自JUnit和NUnit的，但引入了一些新的功能，使其功能更强大，使用更方便。TestNG是一个开源自动化测试框架；TestNG表示下一代（Next Generation）。 TestNG是类似于JUnit（特别是JUnit 4），但它不是一个JUnit扩展。它的灵感来源于JUnit。它的目的是优于JUnit的，尤其是当测试集成的类。 TestNG的创造者是Cedric Beust（塞德里克·博伊斯特）

MAVEN+testNG介绍

TestNG 常用注解

注解	描述
@BeforeSuite	注解的方法将只运行一次，运行所有测试前此套件中。
@AfterSuite	注解的方法将只运行一次此套件中的所有测试都运行之后。
@BeforeClass	注解的方法将只运行一次先行测试在当前类中的方法调用。
@AfterClass	注解的方法将只运行一次后已经运行在当前类中的所有测试方法。
@BeforeTest	注解的方法将被运行之前的任何测试方法属于内部类的 <test>标签的运行。
@AfterTest	注解的方法将被运行后，所有的测试方法，属于内部类的<test>标签的运行。
@BeforeGroups	组的列表，这种配置方法将之前运行。此方法是保证在运行属于任何这些组第一个测试方法，该方法被调用。
@AfterGroups	组的名单，这种配置方法后，将运行。此方法是保证运行后不久，最后的测试方法，该方法属于任何这些组被调用。

MAVEN+testNG介绍

TestNG 常用注解

注解	描述
@BeforeMethod	注解的方法将每个测试方法之前运行。
@AfterMethod	被注释的方法将被运行后，每个测试方法。
@DataProvider	标志着一个方法，提供数据的一个测试方法。注解的方法必须返回一个Object[] []，其中每个对象[]的测试方法的参数列表中可以分配。该@Test 方法，希望从这个DataProvider的接收数据，需要使用一个dataProvider名称等于这个注解的名字。
@Factory	作为一个工厂，返回TestNG的测试类的对象将被用于标记的方法。该方法必须返回Object[]。
@Listeners	定义一个测试类的监听器。
@Parameters	介绍如何将参数传递给@Test方法。
@Test	标记一个类或方法作为测试的一部分。

MAVEN+testNG介绍

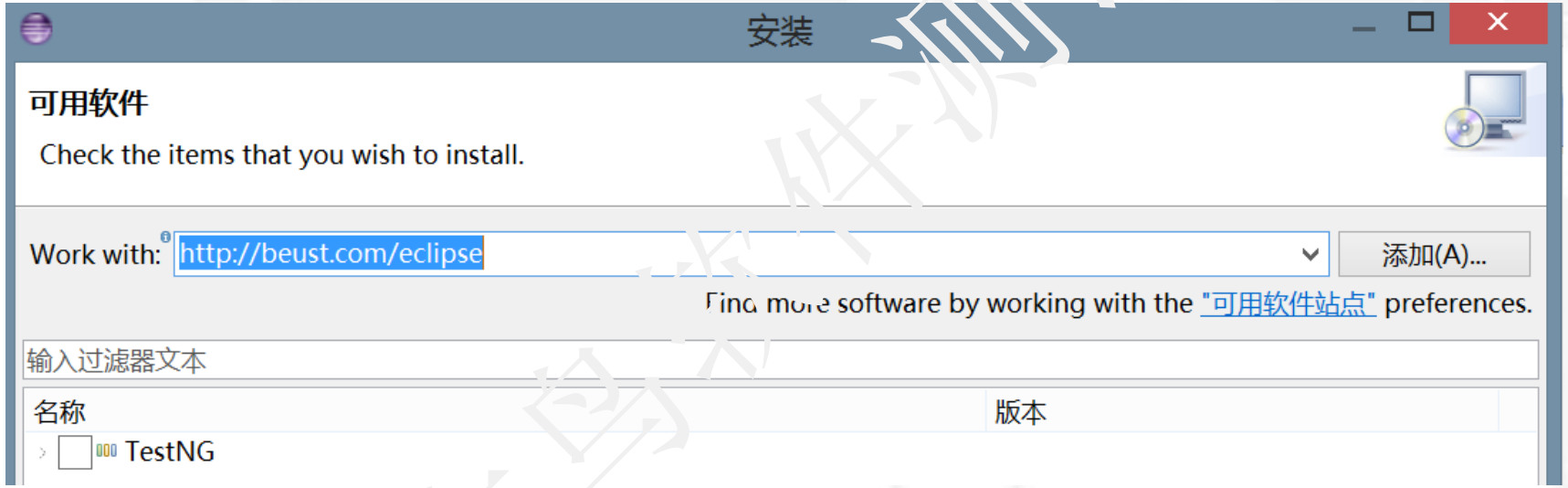
Maven

管理jar包

MAVEN+testNG介绍

TestNG安装

在 Eclipse 中，点击 Help -> Install new software，在 add 栏中输入http://beust.com/eclipse 或http://m2eclipse.sonatype.org/sites/m2e



按下一步直到安装完，在线安装会有点很慢

MAVEN+testNG介绍

MAVEN安装

下载地址：http://maven.apache.org/download.cgi

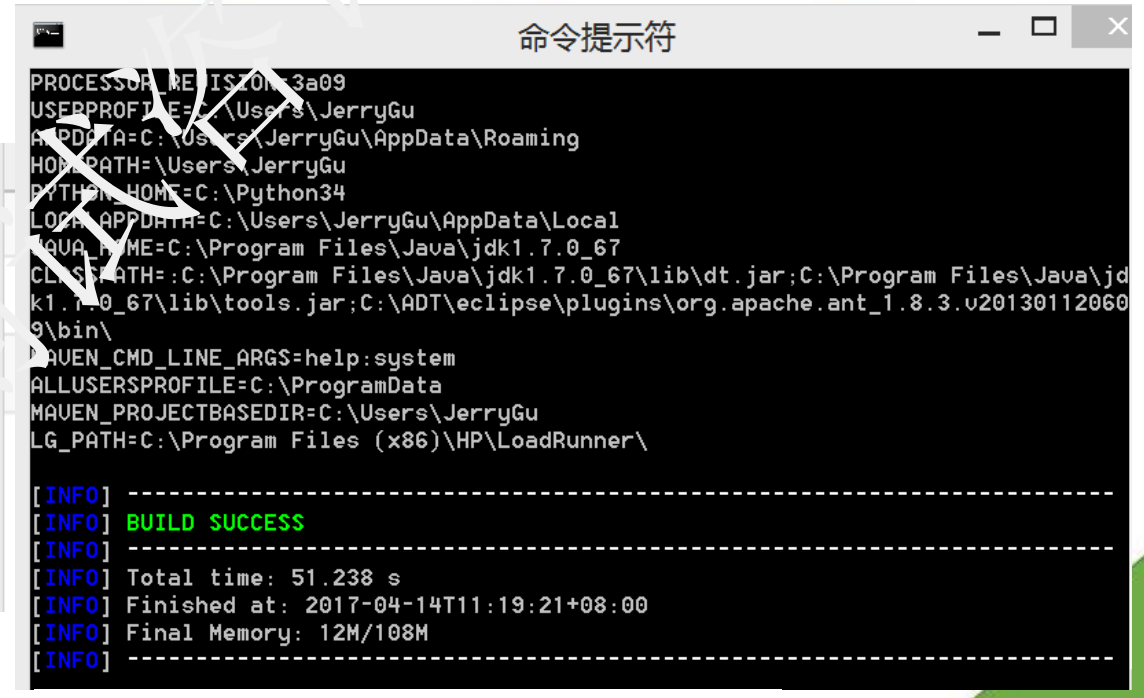
	Link	Checksum	Signature
Binary tar.gz archive	apache-maven-3.5.0-bin.tar.gz	apache-maven-3.5.0-bin.tar.gz.md5	apache-maven-3.5.0-bin.tar.gz.asc
Binary zip archive	apache-maven-3.5.0-bin.zip	apache-maven-3.5.0-bin.zip.md5	apache-maven-3.5.0-bin.zip.asc
Source tar.gz archive	apache-maven-3.5.0-src.tar.gz	apache-maven-3.5.0-src.tar.gz.md5	apache-maven-3.5.0-src.tar.gz.asc
Source zip archive	apache-maven-3.5.0-src.zip	apache-maven-3.5.0-src.zip.md5	apache-maven-3.5.0-src.zip.asc

解压后把文件放在本地目录下，设置环境变量

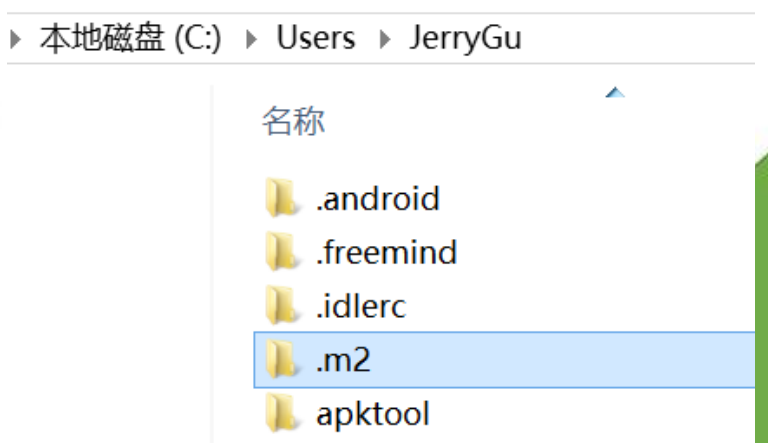
MAVEN+testNG介绍

MAVEN安装

运行 >mvn -v



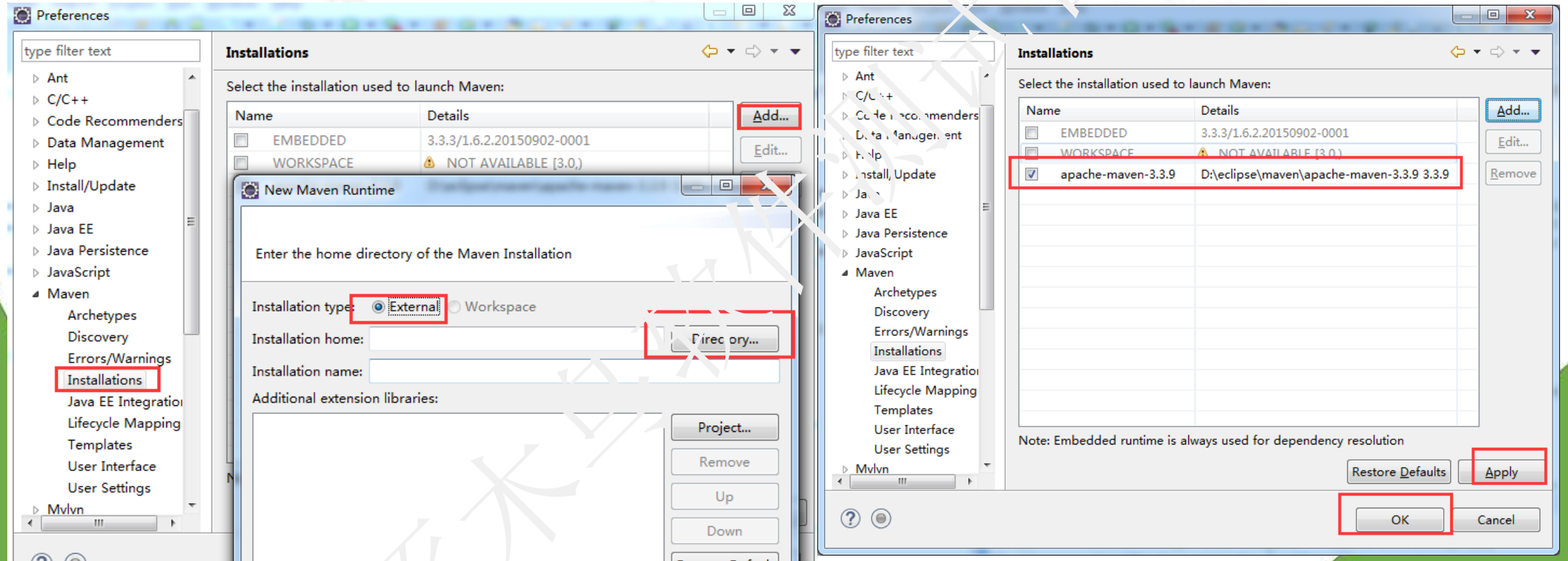
Maven安装成功后，第一次命令行输入 >mvn help:system会在C:\Users\用户\下生成.m2文件，默认放下载的jar包，叫maven仓库。



MAVEN+testNG介绍

MAVEN安装

打开eclipse，选择window-preferences-maven-Installations。

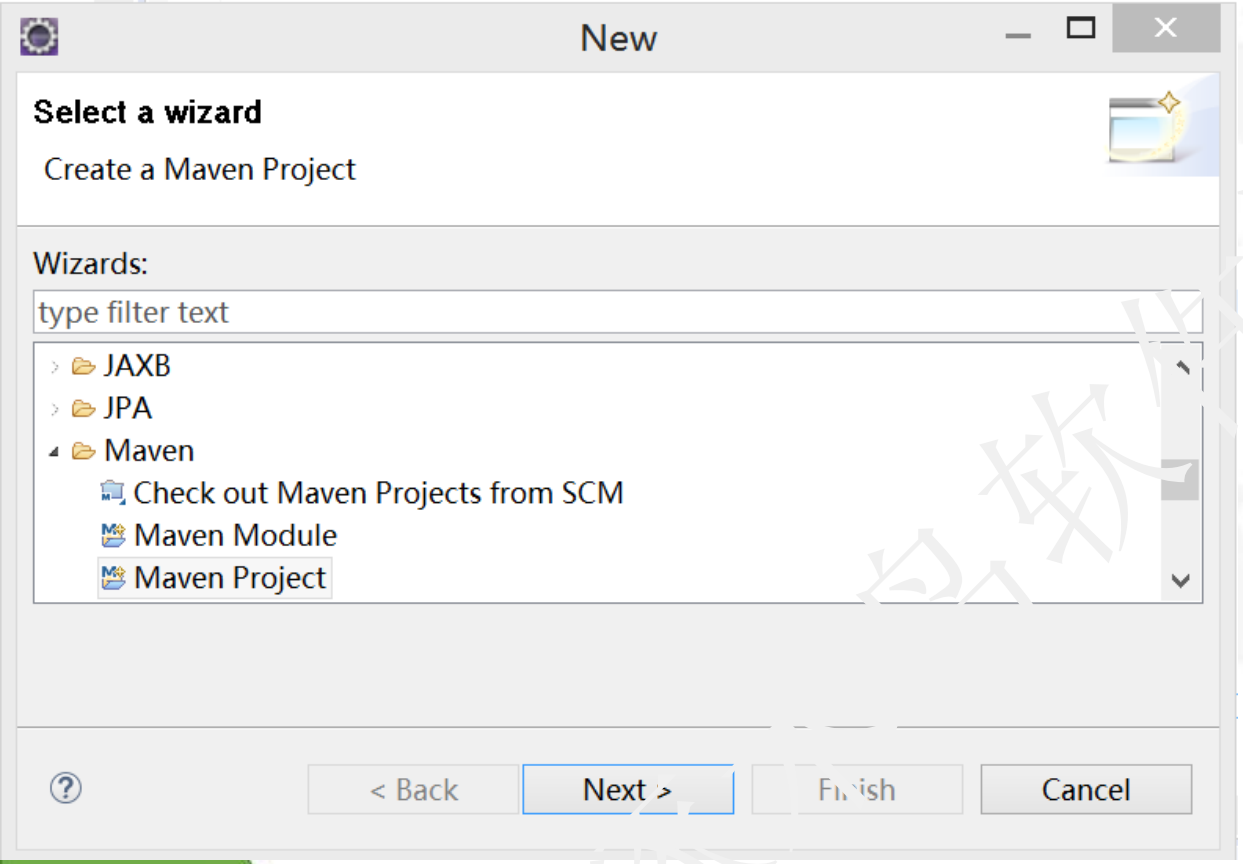


选中外部的maven，点击apply，再点击OK。

MAVEN+testNG介绍

新建测试项目

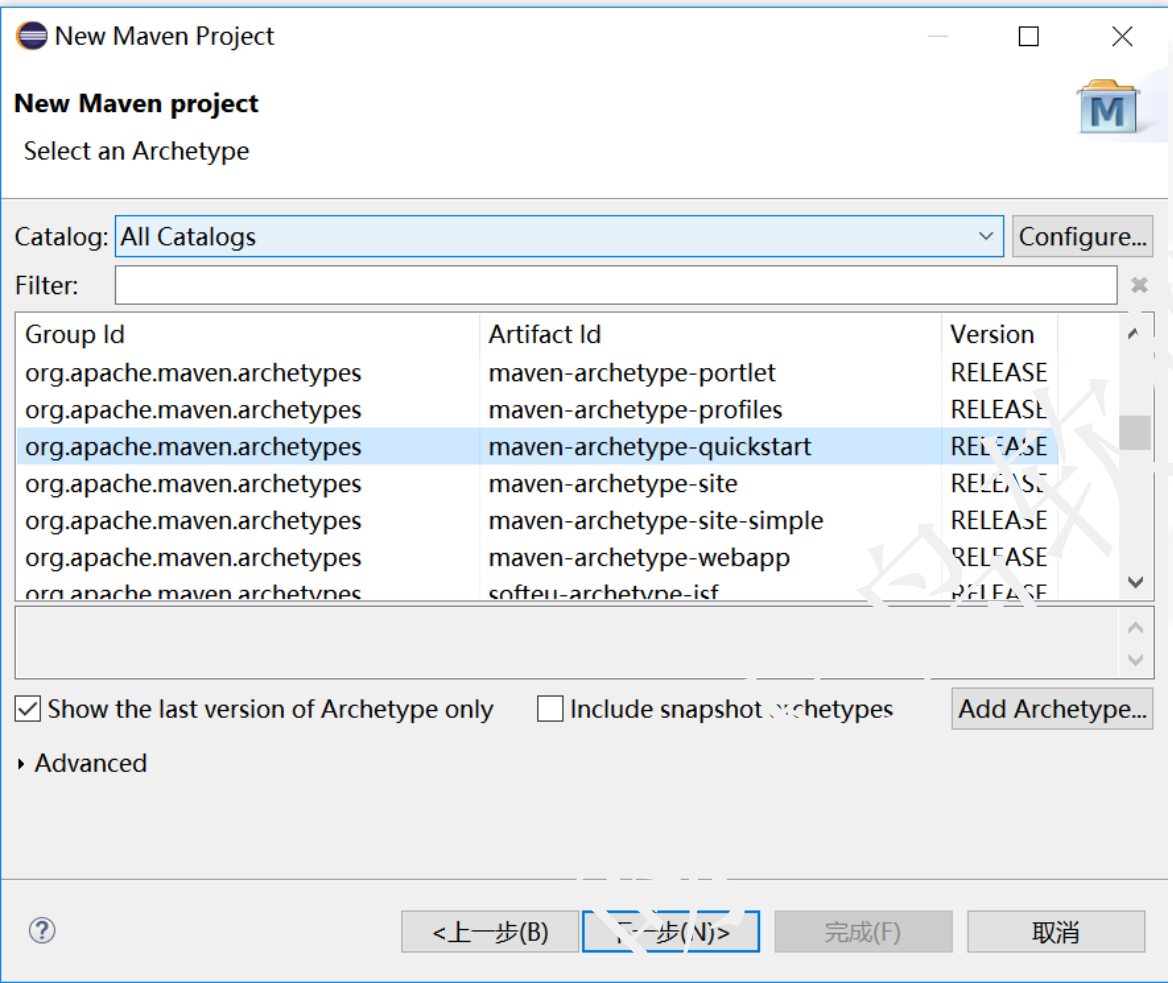
打开eclipse-file-new-other， 选中maven project,点击next



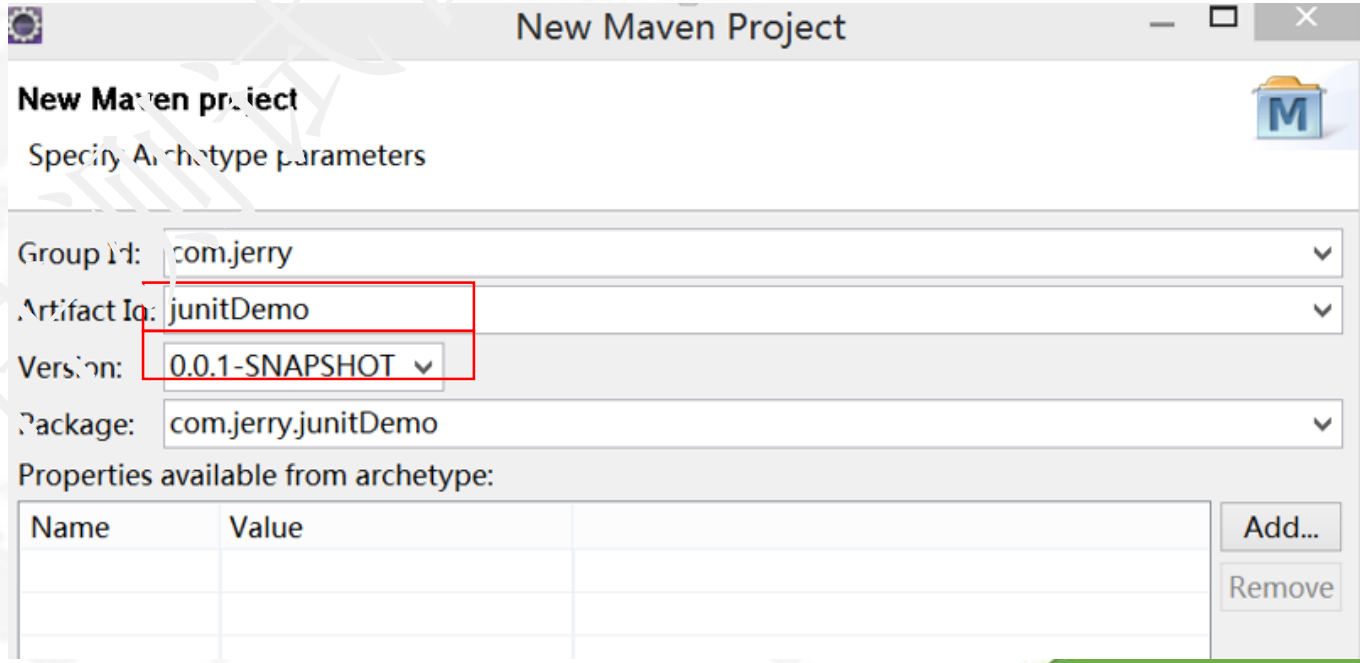
MAVEN+testNG介绍

新建测试项目

选中maven-archetype-quickstart，点击next



输入group ID和Artifact ID，点击finish。



MAVEN+testNG介绍

新建测试项目

生成项目后，打开项目中的pom.xml（此配置文件，管理maven的jar包）

删除配置文件中的

```
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
```

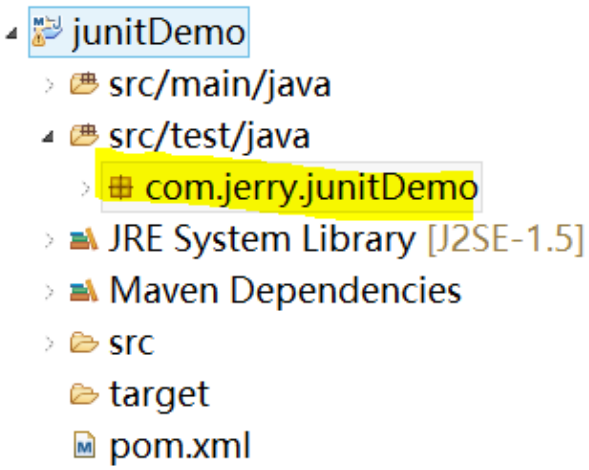
```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>
  <dependency>
<groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.12.4</version>
     </dependency>
 <dependency>
        <groupId>TestNG</groupId>
        <artifactId>TestNG</artifactId>
        <version>6.8.5</version>
 </dependency>
```

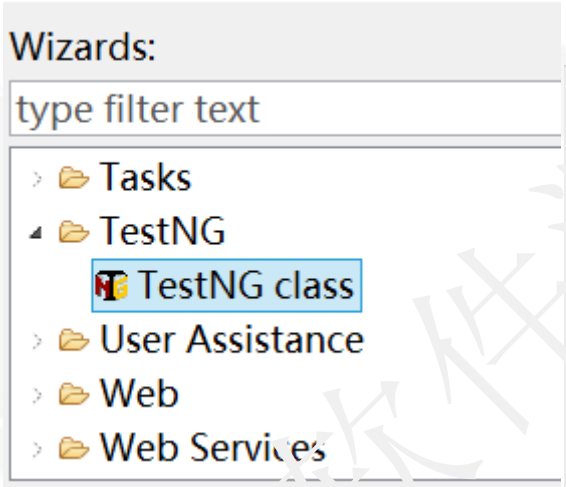
MAVEN+testNG介绍

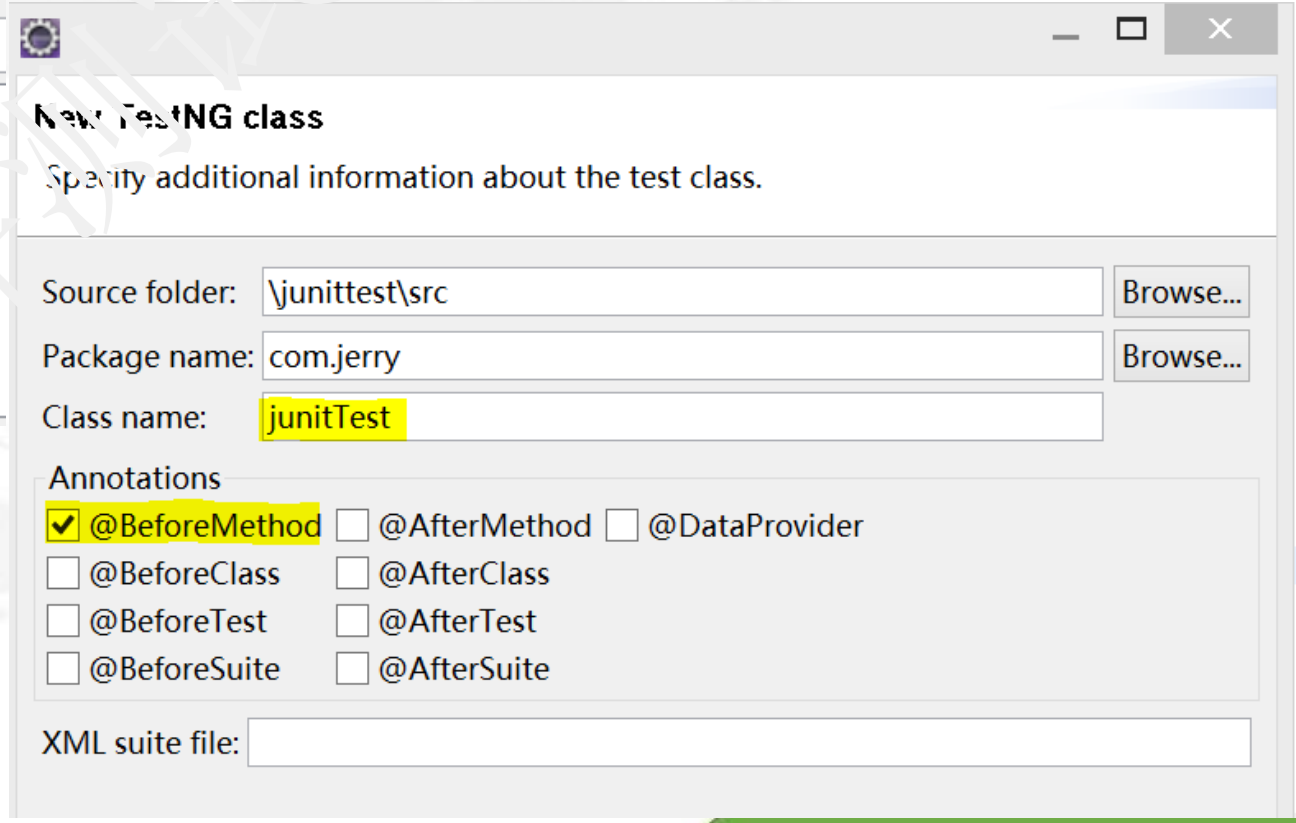
书写测试程序

选择testNG class，点击next

填写class名，选中BforeMethod，点击finish







建立Calculator.java

MAVEN+testNG介绍

书写测试程序

```
package com.jerry.junitDemo;

import org.testng.annotations.Test;
import com.jerry.junitDemo.Calculator;
import org.testng.annotations.BeforeMethod;
import static org.junit.Assert.assertEquals;

public class junitTest {
        private static Calculator calculator = new Calculator();

        @BeforeMethod
         public void beforeMethod() {
                calculator.clear();
         }

@Test
 public void testadd() {
                calculator.add(2);
   calculator.add(3);
   assertEquals(5, calculator.getResult());
}

@Test
        public void testSubstract() {
                calculator.add(5);
   calculator.substract(3);
   assertEquals(2, calculator.getResult());
        }

        @Test
        public void testMultiply() {
                calculator.add(3);
   calculator.multiply(2);
   assertEquals(6, calculator.getResult());
        }
```

MAVEN+testNG介绍

书写测试程序

```
        }

        @Test
        public void testDivide() {
                calculator.add(9);
    calculator.divide(3);
    assertEquals(3, calculator.getResult());
        }
}
```

MAVEN+testNG介绍

运行

```
Exception in thread "main" java.lang.NoClassDefFoundError: com/beust/jcommander/ParameterException
        at java.lang.Class.getDeclaredMethods0(Native Method)
        at java.lang.Class.privateGetDeclaredMethods(Class.java:2701)
        at java.lang.Class.privateGetMethodRecursive(Class.java:3048)
        at java.lang.Class.getMethod0(Class.java:3018)
        at java.lang.Class.getMethod(Class.java:1784)
        at sun.launcher.LauncherHelper.validateMainClass(LauncherHelper.java:544)
        at sun.launcher.LauncherHelper.checkAndLoadMain(LauncherHelper.java:526)
Caused by: java.lang.ClassNotFoundException: com.beust.jcommander.ParameterException
        at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:331)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
        ... 7 more
```

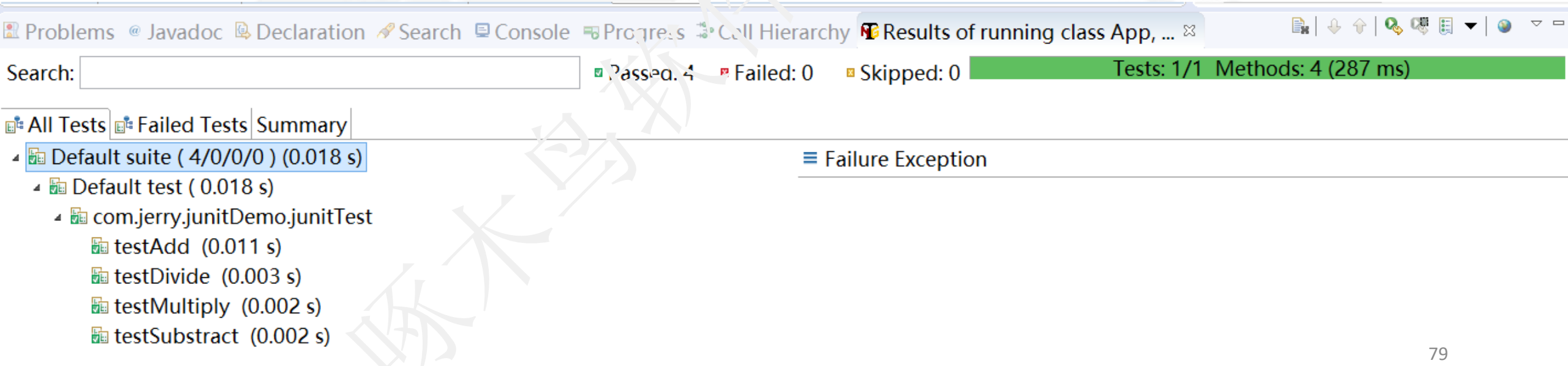
MAVEN+testNG介绍

运行

在pol.xml中加入

```
<dependency>
        <groupId>com.beust</groupId>
        <artifactId>jcommander</artifactId>
        <version>1.48</version>
 </dependency>
```

JCommander 是一个非常小的Java 类库，用来解析命令行参数。



JAVA单元测试框架JUNIT介绍

基本的JUNIT测试框架

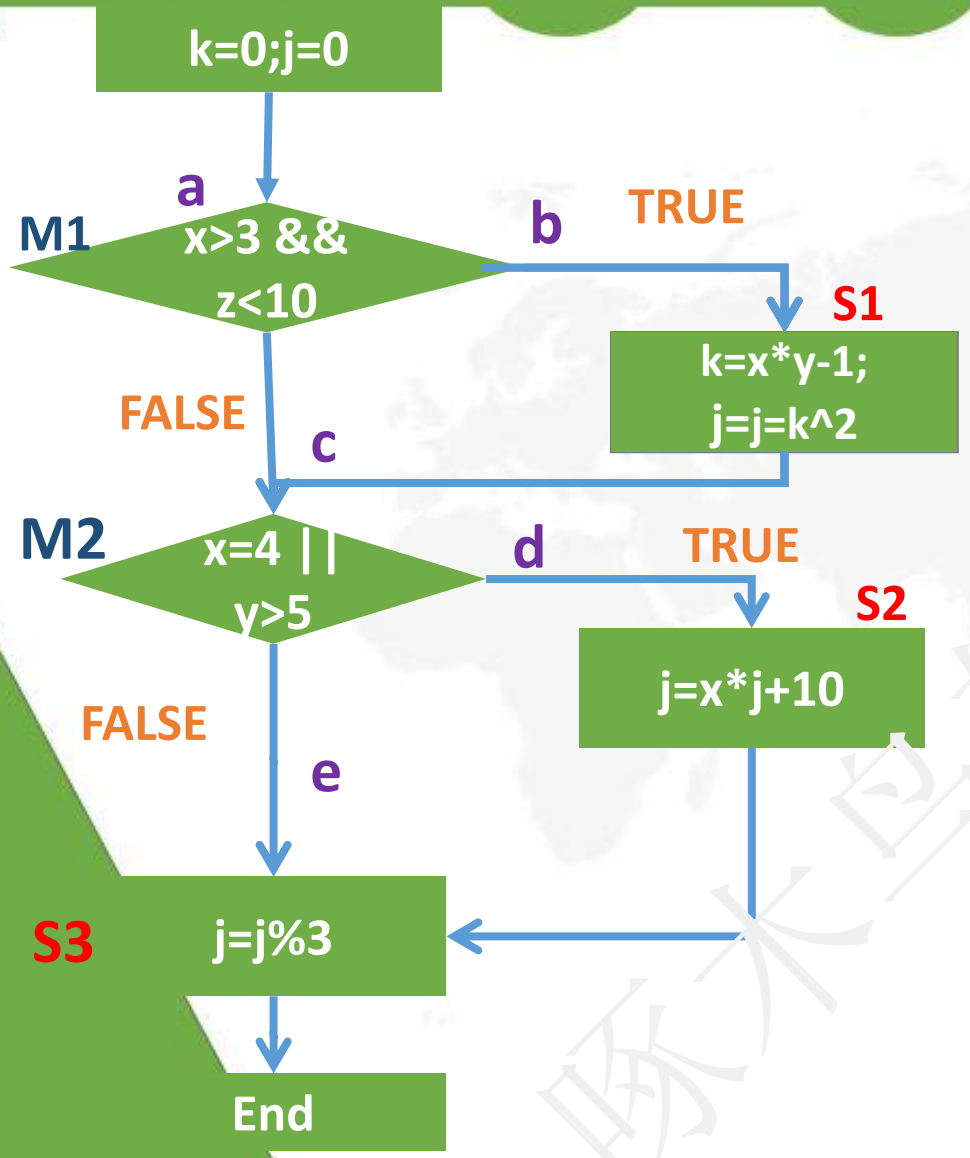
JUNIT断言

JUNIT测试的高级技巧

MAVEN+testNG介绍

练习

练习



测试用例	k+j
x=4,y=6,z=9	25
x=4, y=5, z=10	1
x=5,y=4,z=3	20
x=4,y=5,z=10	1

1. 根据流程图，书写产品代码；
2. 根据测试用例，书写测试代码（用参数）；
3. 用Ant实现运行测试代码，并且形成测试报告；
4. 用MAVEN+testNG的方法来编写测试用例

持续集成单元测试方法与应用

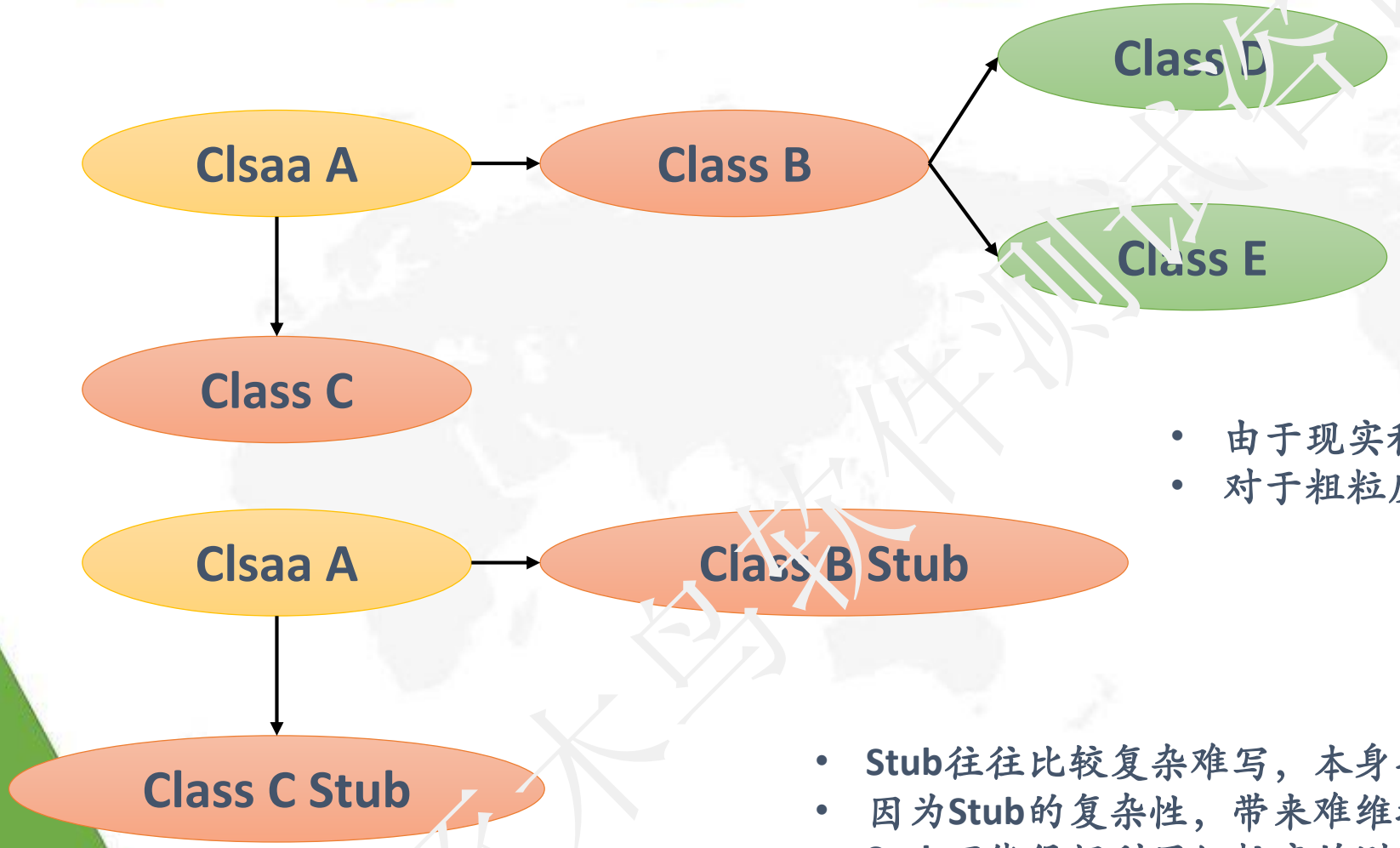
- 测试用例的设计
- 单元测试JUnit框架特性及使用
- 使用Stub进行测试
- Mock技术
- 容器内的测试
- 测试用例执行及数据分析统计
- 持续集成自动化回归测试

使用Stub进行测试

简介

练习

简介



- 由于现实程序比较复杂，难以调试
- 对于粗粒度的程序可以用Stub

- Stub往往比较复杂难写，本身也要调试
- 因为Stub的复杂性，带来难维护性
- Stub不能很好利用细粒度的测试

简介

Account.java

```
package com.jerry;
public class Account{
   private String accountId;
   private long balance;
   public Account( String accountId, long initialBalance ) {//构造函数
      this.accountId = accountId; //编号
      this.balance = initialBalance;} //初始化金额

   public void debit( long amount ){  //出钱
      this.balance -= amount;}
  public void credit( long amount ) { //收钱
      this.balance += amount;
   }
   public long getBalance(){  //获得当前手金额
      return this.balance;}}
```

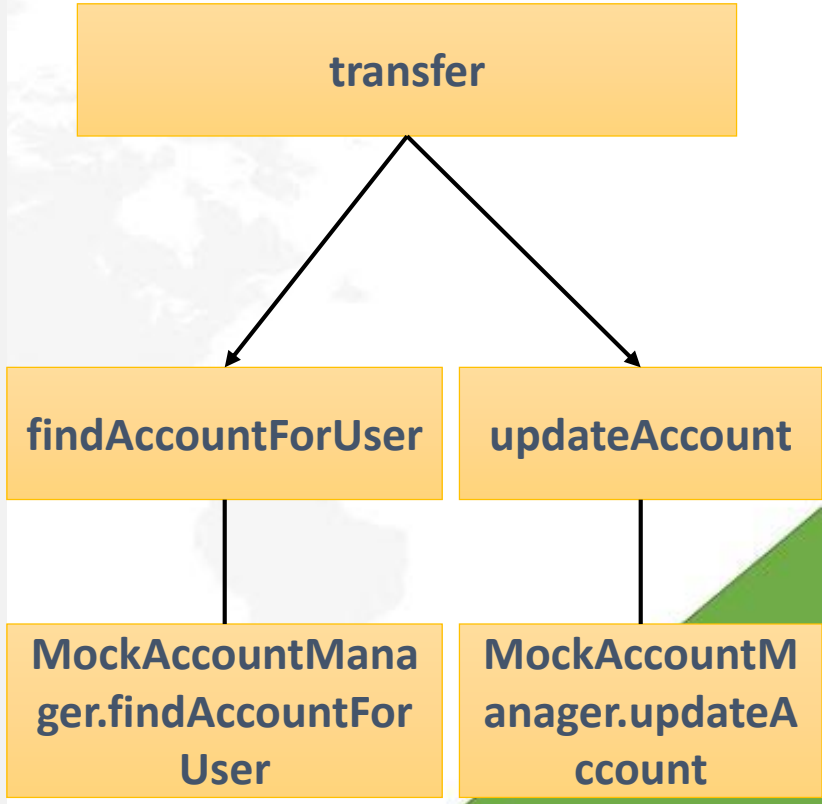
AccountManager.java（接口类）

```
package com.jerry;

public interface AccountManager
{
   //根据用户ID获得Account类
   Account findAccountForUser(String userId );
   //更新当前金额
   void updateAccount(Account account );
}
```

简介

AccountService.java

```
//一个设置客户经理实现从账户senderId到账户 beneficiaryId的方法。
//senderId:转出方Id
//beneficiaryId:  收益方Id
//amount:金额
public void transfer( String senderId, String beneficiaryId, long amount )
{
    //初始化转出方与收益方，findAccountForUser为接口类方法
    Account sender = this.accountManager.findAccountForUser( senderId );
    Account beneficiary = this.accountManager.findAccountForUser( beneficiaryId );

    //转入和接受
    sender.debit( amount );
    beneficiary.credit( amount );
    //更新，updateAccount为接口类方法
    this.accountManager.updateAccount( sender );
    this.accountManager.updateAccount( beneficiary );
  }
}
```



简介

stubAccountManager.java

```
package com.jerry;
import java.util.HashMap;
public class stubAccountManager implements AccountManager{
        private HashMap<String,Account> accounts = new HashMap<String,Account>();
        //添加Acount
        public void addAcount(String userId,Account account){
                this.accounts.put(userId,account);
        }
        //实现接口类AccountManager的findAccountForUser方法
        public Account findAccountForUser(String userId){
                return this.accounts.get(userId);
        }
        //实现接口类AccountManager的updateAccount方法
        public void updateAccount(Account account){
                //do nothing
        }
}
```

简介

AccountService.java

```
package com.jerry;

public class AccountService
{
   //使用的帐户管理器实现
   private AccountManager accountManager;

   //设置帐户管理器实现的设置方法
   public void setAccountManager( AccountManager manager )
   {
      this.accountManager = manager;
   }
```

简介

TestAccountService.java

```
package Account.com.jerry;
import static org.junit.Assert.*;
import org.junit.Test;
public class TestAccountService {
        @Test
        public void testTransferOK() {
                StubAccountManager stubAccountManager = new StubAccountManager(); //定义MockAccountManager类
                Account sendAccount = new Account("1",200); //定义收钱方和出钱方两个Account
                Account beneficiaryAccount = new Account("2",100);
                stubAccountManager.addAcount("1", sendAccount); //初始化收钱方和出钱方HashMap
                stubAccountManager.addAcount("2", beneficiaryAccount);
                AccountService accountService = new AccountService(); //初始化AccountService类
                accountService.setAccountManager(stubAccountManager); //初始化AccountManager
                accountService.transfer("1","2",50); //转钱

                assertEquals(150,sendAccount.getBalance()); //判断转换后收付方金额是否正确
                assertEquals(150,beneficiaryAccount.getBalance());
        }
}
```

使用Stub进行测试

简介

练习

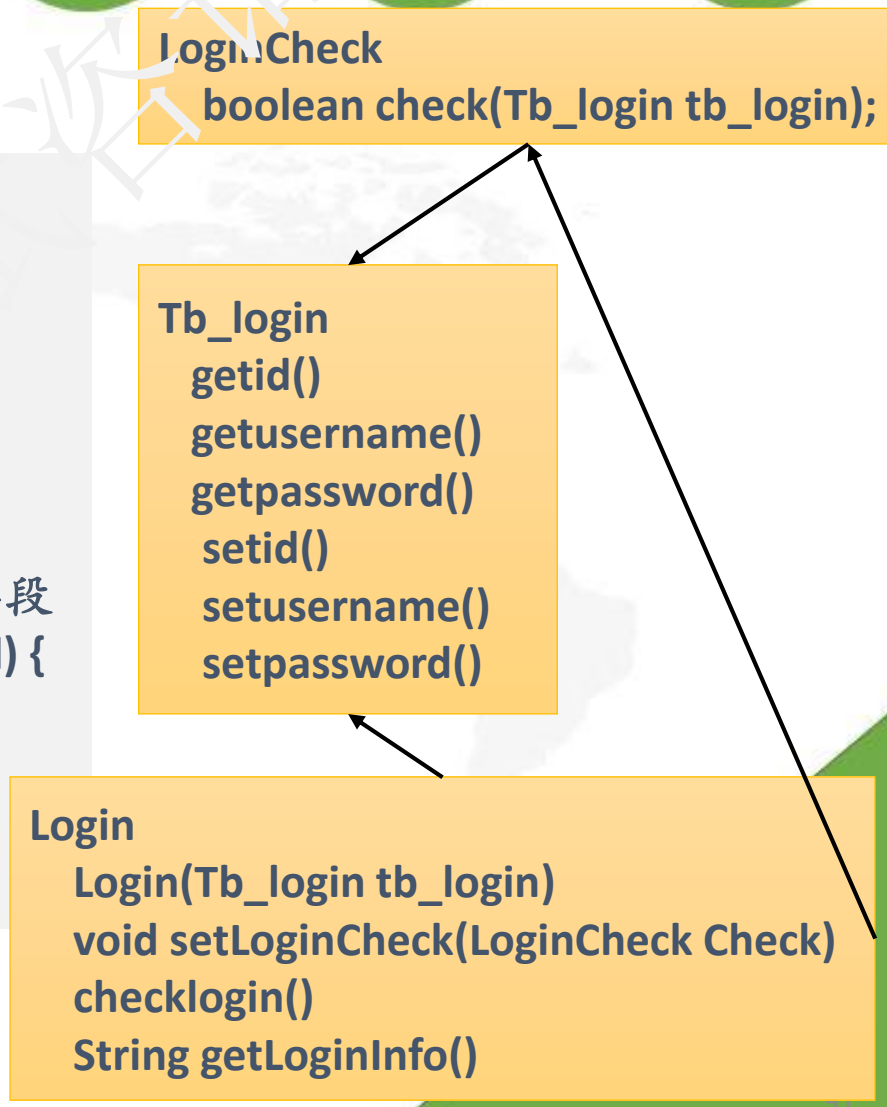
练习

对下面程序书写Stub方法并用JUnit测试

Tb_login

```
package com.jerry;
public class Tb_login {
        private int id;// 编号
        private String username;// 用户名
        private String password;// 密码
        public Tb_login() {// 默认构造函数
                super();}
        // 定义有参构造函数，用来初始化收入信息实体类中的各个字段
                public Tb_login(int id, String username, String password) {
                        super();
                        this.id = id;// 为收入编号赋值
                        this.username = username;// 为用户名赋值
                        this.password = password; }// 为密码
```

构建Stub实现LoginCheck，测试Login String getLoginInfo() 方法



练习

```
public int getid()// 设置编号的可读属性{
        return id;}
public void setid(int id)// 设置编号的可写属性{
        this.id = id;}
public String getusernme()// 设置用户名可读属性{
        return username;}
public void setusername(String username)// 设置用户名的可写属性{
        this.username = username;}
public String getpassword()// 设置密码可读属性{
        return password;}
public void setpassword(String password)// 设置密码的可写属性{
        this.password = password;}
}
```

练习

```
public int getid()// 设置编号的可读属性{
        return id;}
public void setid(int id)// 设置编号的可写属性{
        this.id = id;}
public String getusernme()// 设置用户名可读属性{
        return username;}
public void setusername(String username)// 设置用户名的可写属性{
        this.username = username;}
public String getpassword()// 设置密码可读属性{
        return password;}
public void setpassword(String password)// 设置密码的可写属性{
        this.password = password;}
}
```

LoginCheck

```
package com.jerry;
public interface LoginCheck {
        boolean check(tb_login tb_login);
}
```

练习

Login.java

```
package com.jerry;

public class Login {
        private LoginCheck loginCheck;
        private Tb_login tb_login;

        Login(Tb_login tb_login){
                this.tb_login = tb_login;}
        public void setLoginCheck(LoginCheck Check){
                this.loginCheck = Check;}
        public boolean checklogin(){
                return loginCheck.check(this.tb_login);}
        public String getLoginInfo() {
                if (checklogin()){
                        return "登录成功";
                }else{
                        return "登录失败";
                }}}
```

```
package com.jerry;

public interface LoginCheck {
        boolean check(Tb_login tb_login);
}
```

持续集成单元测试方法与应用

- 测试用例的设计
- 单元测试JUnit框架特性及使用
- 使用Stub进行测试
- Mock技术
- 容器内的测试
- 测试用例执行及数据分析统计
- 持续集成自动化回归测试

Mock技术

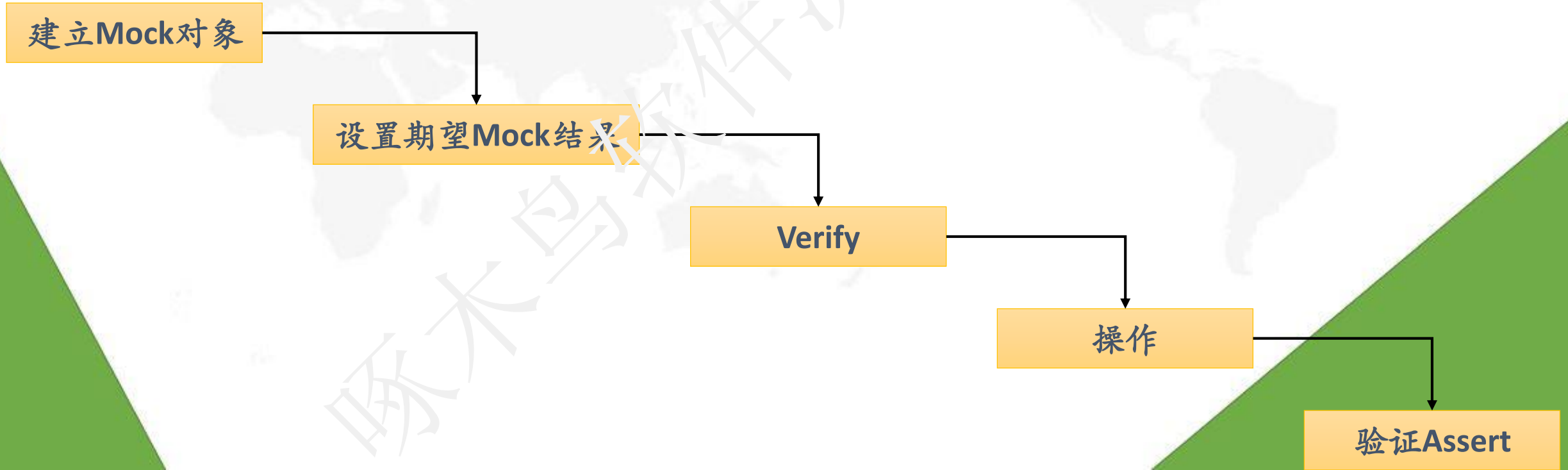
- Mock基本技术
- EasyMock技术
- JMock技术
- Mockito技术

Mock基本技术

mock测试就是在测试过程中，对于某些不容易构造或者不容易获取的对象，用一个虚拟的对象来创建以便测试的测试方法。

在Java阵营中主要的Mock测试工具有Jmock、MockCreator、Mockrunner、EasyMock、MockMaker、Mockito等，在微软的.Net阵营中主要是Nmock、.NetMock等。

Stub是完全模拟一个外部依赖， 而Mock还可以用来判断测试通过还是失败



Mock技术

- Mock基本技术
- EasyMock技术
- jMock技术
- Mockito技术

Mock技术

- EasyMock 简介
- 使用EasyMock 进行单元测试
- 对Mock对象的行为进行验证
- 在EasyMock中使用参数匹配器
- 特殊的Mock对象类型
- 案例分析
- 练习

EasyMock 简介

EasyMock便于无缝地创建模拟对象。它使用Java反射，以创造为给定接口的模拟对象。模拟对象是什么，只不过是代理的实际实现。考虑如：股票服务的情况下，它返回一个股票价格的详细信息。在开发过程中，实际的库存服务不能被用于获得实时数据。因此，我们需要一个虚拟的股票实施服务。简易模拟可以很容易理解顾名思义这样。

```
import static org.easymock.EasyMock.createMock;
import static org.easymock.EasyMock.replay;
import static org.easymock.EasyMock.expect;
import static org.easymock.EasyMock.verify;
```

EasyMock的好处

不用手写

没有必要通过自己编写的模拟对象。

异常支持

支持例外/异常。

重构安全

重构接口方法的名称或重新排序的参数不会破坏测试代码在运行时创建。

命令检查支持

支持检查命令方法调用。

返回值支持

支持返回值。

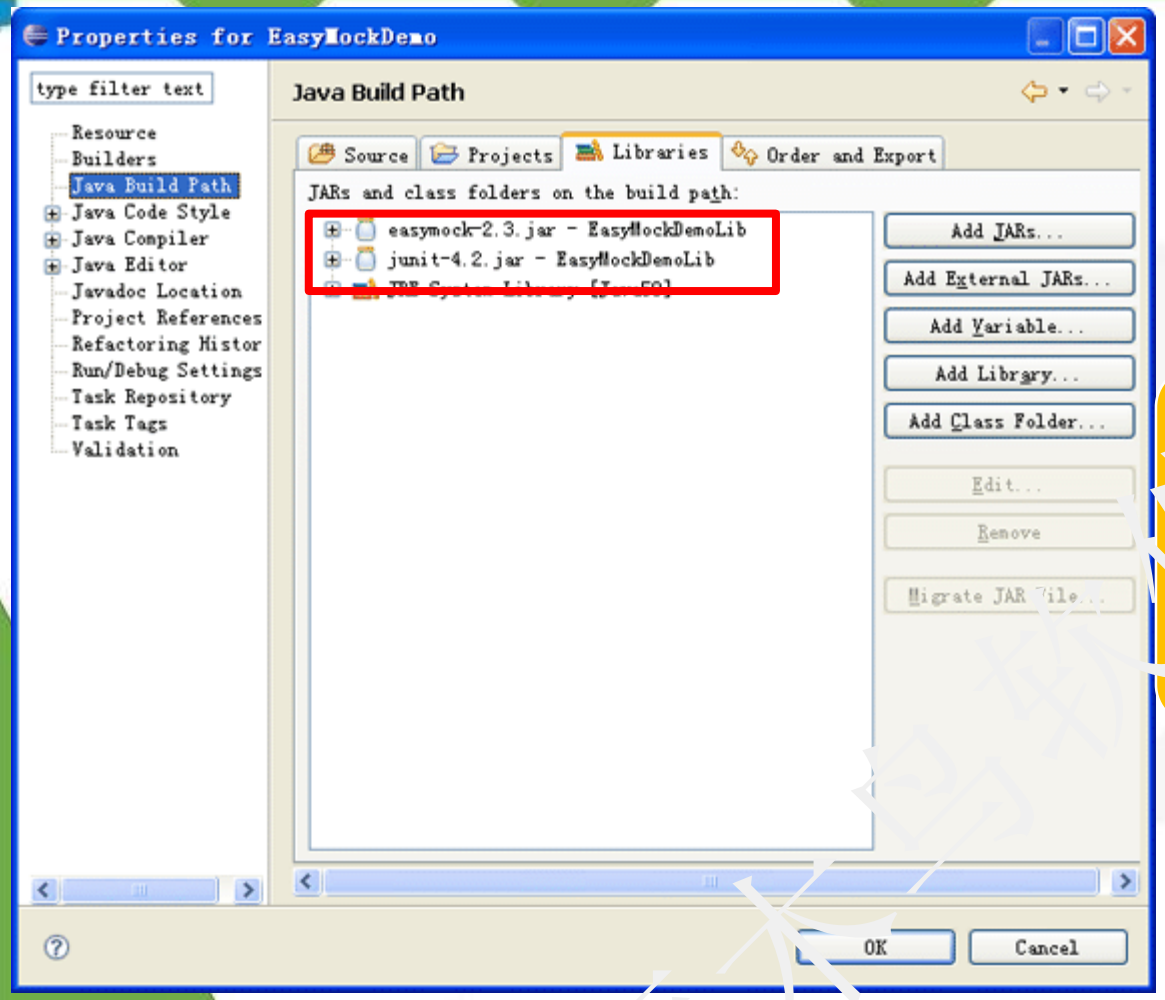
注释支持

支持使用注解创建

Mock技术

- EasyMock 简介
- 使用EasyMock 进行单元测试
- 对Mock对象的行为进行验证
- 在EasyMock中使用参数匹配器
- 特殊的Mock对象类型
- 案例分析
- 练习

使用EasyMock 进行单元测试



使用 EasyMock 生成 Mock 对象

设定 Mock 对象的预期行为和输出

将 Mock 对象切换到 Replay 状态

调用 Mock 对象方法进行单元测试

对 Mock 对象的行为进行验证

使用EasyMock 进行单元测试

使用 EasyMock 生成 Mock 对象

```
public interface java.sql.ResultSet {
......
public abstract java.lang.String getString(int arg0) throws java.sql.SQLException;
public abstract double getDouble(int arg0) throws java.sql.SQLException,
......
}
```

创建mockResultSet

```
ResultSet mockResultSet = createMock(ResultSet.class);
```

createMock 是 org.easymock.EasyMock 类所提供的静态方法

```
IMocksControl control = EasyMock.createControl();
java.sql.Connection mockConnection = control.createMock(Connection.class);
java.sql.Statement mockStatement = control.createMock(Statement.class);
java.sql.ResultSet mockResultSet = control.createMock(ResultSet.class);
```

IMocksControl 的对象，该对象能创建并管理多个 Mock 对象。

使用EasyMock 进行单元测试

设定 Mock 对象的预期行为和输出

添加 Mock 对象行为的过程通常可以分为以下3步：

1. 对 Mock 对象的特定方法作出调用；
2. 通过 org.easymock.EasyMock 提供的静态方法 expectLastCall 获取上一次方法调用所对应的 IExpectationSetters 实例；
3. 通过 IExpectationSetters 实例设定 Mock 对象的预期输出。

设定预期返回值

EasyMock 2.3 中，对 Mock 对象行为的添加和设置是通过接口 IExpectationSetters 来实现的。Mock 对象方法的调用可能产生两种类型的输出：

① 产生返回值；
② 抛出异常。

接口 IExpectationSetters 提供了多种设定预期输出的方法，其中和设定返回值相对应的是 andReturn 方法：

```
IExpectationSetters<T> andReturn(T value);
```

希望方法 mockResult.getString(1) 的返回值为 "My return value"

```
expect(mockResultSet.getString(1)).andReturn("My return value");
```

使用EasyMock 进行单元测试

设定 Mock 对象的预期行为和输出

```
mockStatement.executeQuery("SELECT * FROM sales_order_table");
expect(mockStatement.executeQuery("SELECT * FROM sales_order_table")).andStubReturn(mockResultSet);
```

```
void andStubReturn(Object value); //总是返回相同的值
```

设定预期异常抛出

```
IExpectationSetters<T> andThrow(Throwable throwable);
```

```
void andStubThrow(Throwable throwable); //返回期望的异常
```

设定预期方法调用次数

```
IExpectationSetters<T>times(int count);
```

```
expect(mockResultSet.getString(1)). andReturn("My return value").times(3); //getString 方法在测试过程中被调用3次，期间的返回值都是 "My return value"
```

使用EasyMock 进行单元测试

设定 Mock 对象的预期行为和输出

非准确调用次数的方法：

- **times(int minTimes, int maxTimes)**：该方法最少被调用 minTimes 次，最多被调用 maxTimes 次。
- **atLeastOnce()**：该方法至少被调用一次。
- **anyTimes()**：该方法可以被调用任意次。

```
expect(mockResultSet.close()).times(3, 5); //mockResultSet.close()方法调用3到5次
```

将 Mock 对象切换到 Replay 状态

如果 Mock 对象是通过 org.easymock.EasyMock 类提供的静态方法 createMock 生成的

```
ResultSet mockResultSet = createMock(ResultSet.class)
```

```
replay(mockResultSet);
```

如果 Mock 对象是通过 IMocksControl 接口提供的 createMock 方法生成的

```
java.sql.Connection mockConnection = control.createMock(Connection.class);
```

```
control.replay(); //对前面mockConnection、mockStatement 和 mockResultSet 等3个 Mock 对象都切换成 Replay 状态
```

调用 Mock 对象方法进行单元测试

使用EasyMock 进行单元测试

对 Mock 对象的行为进行验证。

如果 Mock 对象是由 org.easymock.EasyMock 类提供的 createMock 静态方法生成的，那么我们同样采用 EasyMock 类的静态方法 verify 进行验证

```
verify(mockResultSet);
```

如果Mock对象是有 IMocksControl 接口所提供的 createMock 方法生成的，那么采用该接口提供的 verify 方法

```
control.verify();
```

Mock 对象的重用

为了避免生成过多的 Mock 对象，EasyMock 允许对原有 Mock 对象进行重用。

如果 Mock 对象是由 org.easymock.EasyMock 类提供的 createMock 静态方法生成的，那么该 Mock 对象的可以用 EasyMock 类的静态方法 reset 重新初始化；

```
reset(mockResultSet);
```

如果 Mock 方法是由 IMocksControl 实例的 createMock 方法生成的，那么该 IMocksControl 实例方法 reset 的调用将会把所有该实例创建的 Mock 对象重新初始化。

```
control.reset();
```

重新初始化之后，Mock 对象的状态将被置为 Record 状态

Mock技术

- EasyMock 简介
- 使用EasyMock 进行单元测试
- 在EasyMock中使用参数匹配器
- 特殊的Mock对象类型
- 案例分析
- 练习

在EasyMock中使用参数匹配器

EasyMock 预定义的参数匹配器

EasyMock 对参数的匹配默认使用 equals() 方法进行比较

```
expect(mockStatement.executeQuery("SELECT * FROM sales_order_table")).andStubReturn(mockResultSet);
```

如果

```
expect(mockStatement.executeQuery("Select * FROM sales_order_table")).andStubReturn(mockResultSet);
```

认为两个参数不匹配，从而造成 Mock 对象的预期方法不被调用

在EasyMock中使用参数匹配器

EasyMock 预定义的参数匹配器

匹配方法	解释
anyObject	表示任意输入值都与预期值相匹配
aryEq(X value)	通过Arrays.equals()进行匹配，适用于数组对象
isNull()	当输入值为Null时匹配
notNull()	当输入值不为Null时匹配
same(X value)	当输入值和预期值是同一个对象时匹配
lt(X value), leq(X value), geq(X value), gt(X value)	当输入值小于、小等于、大等于、大于预期值时匹配，适用于数值类型
startsWith(String prefix), contains(String substring), endsWith(String suffix)	当输入值以预期值开头、包含预期值、以预期值结尾时匹配，适用于String类型
matches(String regex)	当输入值与正则表达式匹配时匹配，适用于String类型

在EasyMock中使用参数匹配器

自定义参数匹配器

自己的参数匹配器 SQLEquals

要定义新的参数匹配器，需要实现 org.easymock.IArgumentMatcher 接口。其中matches(Object actual) 方法应当实现输入值和预期值的匹配逻辑，而在 appendTo(StringBuffer buffer) 方法中，可以添加当匹配失败时需要显示的信息。

```
public class SQLEquals implements IArgumentMatcher {
 private String expectedSQL = null;
 public SQLEquals(String expectedSQL) {
  this.expectedSQL = expectedSQL;
 }
 ......
 public boolean matches(Object actualSQL) {
  if (actualSQL == null && expectedSQL == null)
    return true;
  else if (actualSQL instanceof String)
    return expectedSQL.equalsIgnoreCase((String) actualSQL);
  else
    return false;
 }}
```

在实现了 IArgumentMatcher 接口之后，我们需要写一个静态方法将它包装一下。这个静态方法的实现需要将 SQLEquals 的一个对象通过 reportMatcher 方法报告给EasyMock

在EasyMock中使用参数匹配器

自定义参数匹配器

自己的参数匹配器 SQLEquals

```
public static String sqlEquals(String in) {
  reportMatcher(new SQLEquals(in));
  return in;
}
```

```
mockStatement.executeQuery(sqlEquals("SELECT * FROM sales_order_table"));
expectLastCall().andStubReturn(mockResultSet);
```

```
executeQuery("select * from sales_order_table")
```

认为是相同的

Mock技术

- EasyMock 简介
- 使用EasyMock 进行单元测试
- 在EasyMock中使用参数匹配器
- 特殊的Mock对象类型
- 案例分析
- 练习

特殊的Mock对象类型

Strick Mock 对象

如果 Mock 对象是通过 EasyMock.createMock() 或是 IMocksControl.createMock() 所创建的，那么在进行 verify 验证时，方法的调用顺序是不进行检查的。如果要创建方法调用的先后次序敏感的 Mock 对象（Strick Mock），应该使

```
ResultSet strickMockResultSet = createStrickMock(ResultSet.class);
```

```
IMocksControl control = EasyMock.createStrictControl();
ResultSet strickMockResultSet = control.createMock(ResultSet.class);
```

Nice Mock 对象

使用 createMock() 创建的 Mock 对象对非预期的方法调用默认的行为是抛出 AssertionError，如果需要一个默认返回0，null 或 false 等"无效值"的 "Nice Mock" 对象，可以通过 EasyMock 类提供的 createNiceMock() 方法创建。类似的，你也可以用 IMocksControl 实例来创建一个 Nice Mock 对象

```
ResultSet strickMockResultSet = createNiceMock (ResultSet.class);
```

```
IMocksControl control = EasyMock.createNiceControl();
ResultSet strickMockResultSet = control.createMock(ResultSet.class);
```

Mock技术

- EasyMock 简介
- 使用EasyMock 进行单元测试
- 在EasyMock中使用参数匹配器
- 特殊的Mock对象类型
- 案例分析
- 练习

案例分析

转账案例分析

```
import static org.easymock.EasyMock.expect;
import static org.easymock.EasyMock.verify;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class TestAccountServiceEasyMock {

    private AccountManager mockAccountManager;

    @Before
    public void setUp()
    {
        //初始化easyMock
      mockAccountManager = createMock("mockAccountManager", AccountManager.class );
    }
```

案例分析

转账案例分析

```
@Test
   public void testTransferOk()
   {
      //初始化senderAccount和beneficiaryAccount
      Account senderAccount = new Account( "1", 200 );
      Account beneficiaryAccount = new Account( "2", 100 );

      //建立senderAccount 和beneficiaryAccount 对应的mockAccountManager
      mockAccountManager.updateAccount( senderAccount );
      mockAccountManager.updateAccount( beneficiaryAccount );

      //用EasyMock的expect设置期望，用replay方法调用到Replay 状态
      expect( mockAccountManager.findAccountForUser( "1" ) ).andReturn( senderAccount );
      expect( mockAccountManager.findAccountForUser( "2" ) ).andReturn( beneficiaryAccount );
      replay( mockAccountManager );

      AccountService accountService = new AccountService();
      accountService.setAccountManager( mockAccountManager );
      accountService.transfer( "1", "2", 50 );
```

案例分析

转账案例分析

```
        assertEquals( 150, senderAccount.getBalance() );
        assertEquals( 150, beneficiaryAccount.getBalance() );
    }

    @After
    public void tearDown()
    {
        verify( mockAccountManager );
    }

}
```

Mock技术

- EasyMock 简介
- 使用EasyMock 进行单元测试
- 在EasyMock中使用参数匹配器
- 特殊的Mock对象类型
- 案例分析
- 练习

练习

对Stub知识的后面习题使用EasyMock来完成

Mock技术

- Mock基本技术
- EasyMock技术
- JMock技术
- Mockito技术

JMock技术

- JMock 简介
- 案例分析
- 练习

JMock 简介

JMock是一个使用模拟对象机制测试Java代码的开发包。模拟对象（Mock Object）可以取代真实对象的位置，用于测试一些与真实对象进行交互或依赖于真实对象的功能，模拟对象的背后目的就是创建一个轻量级的、可控制的对象来代替测试中需要的真实对象，模拟真实对象的行为和功能，方便我们的测试。JMock就是这种机制的实现，使用JMock我们可以快速创建模拟对象，定义交互过程中的约束条件等，同时JMock也是易扩展的，你可以很方便添加自定义的需求

```
import org.jmock.integration.junit4.JMock;
import org.jmock.integration.junit4.JUnit4Mockery;
import org.jmock.Expectations;
import org.jmock.Mockery;
```

JMock 简介

先写一个接口类

```
package com.jerry;

public interface IMathfun {

    public int abs(int num);
}
```

普通测试方法

```
package com.jerry;;
import jmust.demo.Cobertura.Inter.IMathfun;

public class TestJunit4 {
    private IMathfun util;
    public TestJunit4(IMathfun util) {
        this.util = util;
    }
    public int cal(int num) {
        return 10 * util.abs(num);
    }
}
```

JMock 简介

测试方法

```
package com.jerry;

import org.jmock.Expectations;
import org.jmock.Mockery;
import org.jmock.integration.junit4.JUnit4Mockery;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import jmust.demo.Cobertura.Inter.IMathfun;
import jmust.demo.Cobertura.Service.TestJunit4;
import junit.framework.TestCase;

public class Junit4Test extends TestCase{
    private Mockery context = new JUnit4Mockery();
    private IMathfun math = null;
    private TestJunit4 test = null;
```

JMock 简介

测试方法

```
@Before
public void setUp() throws Exception{
  super.setUp();
   math = context.mock(IMathfun.class);
   test = new TestJunit4(math);
   context.checking(new Expectations(){
       {
          exactly(1).of(math).abs(-20);will(returnValue(20));
       }
   });
}
@After
public void setDown(){
}
@Test
public void test(){
   assertEquals(200, test.cal(-20));
}}
```

JMock 简介

表示	解释
oneof	调用只需要一次和一次
exactly(times).of	调用准确地期望n次。
atLeast(times).of	调用至少需要n次.
atMost(times).of	调用最多需要n次。
between(min, max).of	调用至少在最小次数和最大次数下进行。
allowing	允许任意多次 调用但没有发生.
ignoring	和allowing一样。允许或忽略应选择使测试代码清楚地表达意图。
never	根本不需要调用。这是用来使测试更明确，更容易理解

JMock技术

JMock 简介

案例分析

练习

案例分析

```
package com.jerry;

import static org.junit.Assert.assertEquals;

import org.jmock.integration.junit4.JMock;
import org.jmock.integration.junit4.JUnit4Mockery;
import org.jmock.Expectations;
import org.jmock.Mockery;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;

@RunWith(JMock.class)
public class TestAccountServiceJMock
{
    /**
     * The mockery context that we use to create our mocks.
     */
    private Mockery context = new JUnit4Mockery();
```

案例分析

```
/**
 * The mock instance of the AccountManager to use.
 */
private AccountManager mockAccountManager;
@Before
public void setUp()
{
   mockAccountManager = context.mock( AccountManager.class );
}
@Test
public void testTransferOk()
{
   final Account senderAccount = new Account( "1", 200 );
   final Account beneficiaryAccount = new Account( "2", 100 );
   context.checking( new Expectations()
   {
      {
         oneOf( mockAccountManager ).findAccountForUser( "1" );
         will( returnValue( senderAccount ) );
         oneOf( mockAccountManager ).findAccountForUser( "2" );
```

案例分析

```
            oneOf( mockAccountManager ).updateAccount( senderAccount );
            oneOf( mockAccountManager ).updateAccount( beneficiaryAccount );
        }
    } );

    AccountService accountService = new AccountService();
    accountService.setAccountManager( mockAccountManager );
    accountService.transfer( "1", "2", 50 );

    assertEquals( 150, senderAccount.getBalance() );
    assertEquals( 150, beneficiaryAccount.getBalance() );
  }
}
```

JMock技术

JMock 简介

案例分析

练习

练习

对Stub知识的后面习题使用JMock来完成

Mock技术

- Mock基本技术
- EasyMock技术
- JMock技术
- Mockito技术

Mockito技术

- Mockito 简介
- 案例分析
- 练习

Mockito 简介

Mockito是mocking框架，它让你用简洁的API做测试。而且Mockito简单易学，它可读性强和验证语法简洁。

Mockito资源

官网：

http://mockito.org

API文档：

http://docs.mockito.googlecode.com/hg/org/mockito/Mockito.html

项目源码：

https://github.com/mockito/mockito

使用场景

- 提前创建测试，TDD（测试驱动开发）
- 团队可以并行工作
- 你可以创建一个验证或者演示程序
- 为无法访问的资源编写测试
- Mock 可以交给用户
- 隔离系统

Mockito 简介

添加maven依赖

```
<dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-all</artifactId>
        <version>1.9.5</version>
        <scope>test</scope>
</dependency>
```

添加jUnit依赖

```
<dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
</dependency>
```

添加引用

```
import static org.mockito.Mockito.*;
import static org.junit.Assert.*;
```

验证行为

```
@Test
   public void verify_behaviour(){
      //模拟创建一个List对象
      List mock = mock(List.class);
      //使用mock的对象
      mock.add(1);
      mock.clear();
      //验证add(1)和clear()行为是否发生
      verify(mock).add(1);
      verify(mock).clear();
   }
```

Mockito 简介

模拟所期望的结果

```
@Test
   public void when_thenReturn(){
      //mock一个Iterator类
      Iterator iterator = mock(Iterator.class);
      //预设当iterator调用next()时第一次返回hello，第n（n>=2）次都返回world
      when(iterator.next()).thenReturn("hello").thenReturn("world");
      //使用mock的对象
      String result = iterator.next() + " " + iterator.next() + " " + iterator.next();
      //验证结果
      assertEquals("hello world world",result);
   }
```

Mockito 简介

RETURNS_SMART_NULLS

RETURNS_SMART_NULLS实现了Answer接口的对象，它是创建mock对象时的一个可选参数，mock(Class,Answer)。在创建mock对象时，有的方法我们没有进行stubbing，所以调用时会返回Null这样在进行操作是很可能抛出NullPointerException。如果通过RETURNS_SMART_NULLS参数创建的mock对象在没有调用stubbed方法时会返回SmartNull。例如：返回类型是String，会返回"";是int，会返回0；是List，会返回空的List。另外，在控制台窗口中可以看到SmartNull的友好提示。

```
@Test
   public void returnsSmartNullsTest() {
      List mock = mock(List.class, RETURNS_SMART_NULLS);
      System.out.println(mock.get(0));

      //使用RETURNS_SMART_NULLS参数创建的mock对象，不会抛出NullPointerException异常。另外控制台窗口会提示信息“SmartNull returned by unstubbed get() method on mock”
      System.out.println(mock.toArray().length);
   }
```

Mockito 简介

RETURNS_DEEP_STUBS

RETURNS_DEEP_STUBS参数程序会自动进行mock所需的对象，方法deepstubsTest和deepstubsTest2是等价的

```
@Test
  public void deepstubsTest(){
     Account account=mock(Account.class,RETURNS_DEEP_STUBS);
     when(account.getRailwayTicket().getDestination()).thenReturn("beijing");
     account.getRailwayTicket().getDestination();
     verify(account.getRailwayTicket()).getDestination();
     assertEquals("Beijing",account.getRailwayTicket().getDestination());
  }
  @Test
  public void deepstubsTest2(){
     Account account=mock(Account.class);
     RailwayTicket railwayTicket=mock(RailwayTicket.class);
     when(account.getRailwayTicket()).thenReturn(railwayTicket);
     when(railwayTicket.getDestination()).thenReturn("Beijing");
     account.getRailwayTicket().getDestination();
     verify(account.getRailwayTicket()).getDestination();
     assertEquals("Beijing",account.getRailwayTicket().getDestination());
```

合在一起写

Mockito 简介

```
public class RailwayTicket{
   private String destination;
   public String getDestination() {
      return destination;
   }

   public void setDestination(String destination) {
      this.destination = destination;
   }
}

public class Account{
   private RailwayTicket railwayTicket;

   public RailwayTicket getRailwayTicket() {
      return railwayTicket;
   }
```

Mockito 简介

模拟方法体抛出异常

```
@Test(expected = RuntimeException.class)
  public void doThrow_when(){
     List list = mock(List.class);
     doThrow(new RuntimeException()).when(list).add(1);
     list.add(1);
  }
```

使用注解来快速模拟

在上面的测试中我们在每个测试方法里都mock了一个List对象，为了避免重复的mock，是测试类更具有可读性，我们可以使用下面的注解方式来快速模拟对象

```
public class MockitoExample2 {
   @Mock
   private List mockList;

//在基类中添加初始化mock的代码，没有这段代码会返回null
    public MockitoExample2(){
       MockitoAnnotations.initMocks(this);
    }

   @Test
   public void shorthand(){
      mockList.add(1);
      verify(mockList).add(1);
   }
}
```

Mockito 简介

或者使用建立运行器：MockitoJUnitRunner

```
@RunWith(MockitoJUnitRunner.class)
public class MockitoExample2 {
    @Mock
    private List mockList;

    @Test
    public void shorthand(){
        mockList.add(1);
        verify(mockList).add(1);
    }
}
```

参数匹配

```
@Test
public void with_arguments(){
    Comparable comparable = mock(Comparable.class);
    //预设根据不同的参数返回不同的结果
    when(comparable.compareTo("Test")).thenReturn(1);
    when(comparable.compareTo("Omg")).thenReturn(2);
    assertEquals(1, comparable.compareTo("Test"));
    assertEquals(2, comparable.compareTo("Omg"));
    //对于没有预设的情况会返回默认值
    assertEquals(0, comparable.compareTo("Not stub"));
}
```

Mockito 简介

匹配自己想要的任意参数

```
@Test
   public void with_unspecified_arguments(){
      List list = mock(List.class);
      //匹配任意参数
      when(list.get(anyInt())).thenReturn(1);
      when(list.contains(argThat(new IsValid()))).thenReturn(true);
      assertEquals(1, list.get(1));
      assertEquals(1, list.get(999));
      assertTrue(list.contains(1));
      assertTrue(!list.contains(3));
   }

   private class IsValid extends ArgumentMatcher<List>{
      @Override
      public boolean matches(Object o) {
         return o == 1 || o == 2;
      }
   }
```

Mockito 简介

如果你使用了参数匹配，那么所有的参数都必须通过matchers来匹配，如下代码：

```
@Test
   public void all_arguments_provided_by_matchers(){
      Comparator comparator = mock(Comparator.class);
      comparator.compare("nihao","hello");
      //如果你使用了参数匹配，那么所有的参数都必须通过matchers来匹配
      verify(comparator).compare(anyString(),eq("hello"));
      //下面的为无效的参数匹配使用
      //verify(comparator).compare(anyString(),"hello");
   }
```

Mockito 简介

自定义参数匹配

```
@Test
  public void argumentMatchersTest(){
    //创建mock对象
    List<String> mock = mock(List.class);

    //argThat(Matches<T> matcher)方法用来应用自定义的规则，可以传入任何实现Matcher接口的实现类。
    when(mock.addAll(argThat(new IsListofTwoElements()))).thenReturn(true);
    mock.addAll(Arrays.asList("one","two","three"));
    //IsListofTwoElements用来匹配size为2的List，因为例子传入List为三个元素，所以此时将失败。
    verify(mock).addAll(argThat(new IsListofTwoElements()));
  }
  class IsListofTwoElements extends ArgumentMatcher<List>
  {
    public boolean matches(Object list)
    {
      return((List)list).size()==2;
    }
  }
```

Mockito 简介

捕获参数来进一步断言

```
@Test
  public void capturing_args(){
    PersonDao personDao = mock(PersonDao.class);
    PersonService personService = new PersonService(personDao);

    ArgumentCaptor<Person> argument = ArgumentCaptor.forClass(Person.class);
    personService.update(1,"jack");
    verify(personDao).update(argument.capture());
    assertEquals(1,argument.getValue().getId());
    assertEquals("jack",argument.getValue().getName());
  }
   class Person{
    private int id;
    private String name;

    Person(int id, String name) {
      this.id = id;
      this.name = name;
    }
```

Mockito 简介

```
    public int getId() {
        return id;
    }
    public String getName() {
        return name;
    }}
interface PersonDao{
    public void update(Person person);
}
class PersonService{
    private PersonDao personDao;

    PersonService(PersonDao personDao) {
        this.personDao = personDao;
    }
    public void update(int id,String name){
        personDao.update(new Person(id,name));
    }}
```

Mockito 简介

使用方法预期回调接口生成期望值（Answer结构）

```
@Test
   public void answerTest(){
      when(mockList.get(anyInt())).thenAnswer(new CustomAnswer());
      assertEquals("hello world:0",mockList.get(0));
      assertEquals("hello world:999",mockList.get(999));
   }

   private class CustomAnswer implements Answer<String>{
      @Override
      public String answer(InvocationOnMock invocation) throws Throwable {
         Object[] args = invocation.getArguments();
         return "hello world:"+args[0];
      }
   }
```

Mockito 简介

也可使用匿名内部类实现

```
@Test
  public void answer_with_callback(){
    //使用Answer来生成我们我们期望的返回
    when(mockList.get(anyInt())).thenAnswer(new Answer<Object>() {
      @Override
      public Object answer(InvocationOnMock invocation) throws Throwable {
        Object[] args = invocation.getArguments();
        return "hello world:"+args[0];
      }
    });
    assertEquals("hello world:0",mockList.get(0));
    assertEquals("hello world:999",mockList.get(999));
  }
```

Mockito 简介

修改对未预设的调用返回默认期望

```
@Test
  public void unstubbed_invocations(){
    //mock对象使用Answer来对未预设的调用返回默认期望值
    List mock = mock(List.class,new Answer() {
      @Override
      public Object answer(InvocationOnMock invocation) throws Throwable {
        return 999;
      }
    });
    //下面的get(1)没有预设，通常情况下会返回NULL，但是使用了Answer改变了默认期望值
    assertEquals(999, mock.get(1));
    //下面的size()没有预设，通常情况下会返回0，但是使用了Answer改变了默认期望值
    assertEquals(999,mock.size());
  }
```

Mockito 简介

用spy监控真实对象

- Mock不是真实的对象，它只是用类型的class创建了一个虚拟对象，并可以设置对象行为
- Spy是一个真实的对象，但它可以设置对象行为
- InjectMocks创建这个类的对象并自动将标记@Mock、@Spy等注解的属性值注入到这个中

```
@Test(expected = IndexOutOfBoundsException.class)
   public void spy_on_real_objects(){
      List list = new LinkedList();
      List spy = spy(list);
      //下面预设的spy.get(0)会报错，因为会调用真实对象的get(0)，所以会抛出越界异常
      //when(spy.get(0)).thenReturn(3);

      //使用doReturn-when可以避免when-thenReturn调用真实对象api
      doReturn(999).when(spy).get(999);
      //预设size()期望值
      when(spy.size()).thenReturn(100);
      //调用真实对象的api
      spy.add(1);
      spy.add(2);
      assert Equals(100,spy.size());
       assertEquals(1,spy.get(0));
       assertEquals(2,spy.get(1));
       verify(spy).add(1);
       verify(spy).add(2);
       assertEquals(999,spy.get(999));
       spy.get(2);}
```

Mockito 简介

真实的部分mock

```
@Test
   public void real_partial_mock(){
      //通过spy来调用真实的api
      List list = spy(new ArrayList());
      assertEquals(0,list.size());
      A a  = mock(A.class);
      //通过thenCallRealMethod来调用真实的api
      when(a.doSomething(anyInt())).thenCallRealMethod();
      assertEquals(999,a.doSomething(999));
   }


   class A{
      public int doSomething(int i){
         return i;
      }
   }
```

重置mock

```
@Test
   public void reset_mock(){
      List list = mock(List.class);
      when(list.size()).thenReturn(10);
      list.add(1);
      assertEquals(10,list.size());
      //重置mock，清除所有的互动和预设
      reset(list);
      assertEquals(0,list.size());
   }
```

Mockito 简介

验证确切的调用次数

```
@Test
   public void verifying_number_of_invocations(){
      List list = mock(List.class);
      list.add(1);
      list.add(2);
      list.add(2);
      list.add(3);
      list.add(3);
      list.add(3);
      //验证是否被调用一次，等效于下面的times(1)
      verify(list).add(1);
      verify(list,times(1)).add(1);
      //验证是否被调用2次
      verify(list,times(2)).add(2);
      //验证是否被调用3次
      verify(list,times(3)).add(3);
      //验证是否从未被调用过
      verify(list,never()).add(4);
      //验证至少调用一次
      verify(list,atLeastOnce()).add(1);
      //验证至少调用2次
      verify(list,atLeast(2)).add(2);
      //验证至多调用3次
      verify(list,atMost(3)).add(3);
   }
```

Mockito 简介

连续调用

```
@Test(expected = RuntimeException.class)
  public void consecutive_calls(){
    //模拟连续调用返回期望值，如果分开，则只有最后一个有效
    when(mockList.get(0)).thenReturn(0);
    when(mockList.get(0)).thenReturn(1);
    when(mockList.get(0)).thenReturn(2);
    when(mockList.get(1)).thenReturn(0).thenReturn(1).thenThrow(new RuntimeException());
    assertEquals(2,mockList.get(0));
    assertEquals(2,mockList.get(0));
    assertEquals(0,mockList.get(1));
    assertEquals(1,mockList.get(1));
    //第三次或更多调用都会抛出异常
    mockList.get(1);
  }
```

Mockito 简介

验证执行顺序

```
@Test
   public void verification_in_order(){
      List list = mock(List.class);
      List list2 = mock(List.class);
      list.add(1);
      list2.add("hello");
      list.add(2);
      list2.add("world");
      //将需要排序的mock对象放入InOrder
      InOrder inOrder = inOrder(list,list2);
      //下面的代码不能颠倒顺序，验证执行顺序
      inOrder.verify(list).add(1);
      inOrder.verify(list2).add("hello");
      inOrder.verify(list).add(2);
      inOrder.verify(list2).add("world");
   }
```

确保模拟对象上无互动发生

```
@Test
   public void verify_interaction(){
      List list = mock(List.class);
      List list2 = mock(List.class);
      List list3 = mock(List.class);
      list.add(1);
      verify(list).add(1);
      verify(list,never()).add(2);
      //验证零互动行为
      verifyZeroInteractions(list2,list3);
   }
```

Mockito 简介

找出冗余的互动(即未被验证到的)

```
@Test(expected = NoInteractionsWanted.class)
   public void find_redundant_interaction(){
      List list = mock(List.class);
      list.add(1);
      list.add(2);
      verify(list,times(2)).add(anyInt());
      //检查是否有未被验证的互动行为，因为add(1)和add(2)都会被上面的anyInt()验证到，所以下面的代码会通过
      verifyNoMoreInteractions(list);

      List list2 = mock(List.class);
      list2.add(1);
      list2.add(2);
      verify(list2).add(1);
      //检查是否有未被验证的互动行为，因为add(2)没有被验证，所以下面的代码会失败抛出异常
      verifyNoMoreInteractions(list2);
   }
```

Mockito技术

Mockito 简介

案例分析

练习

Mockito 简介

```
package com.jerry;

import static org.junit.Assert.*;
import static org.mockito.Mockito.*;

import org.junit.Test;

public class TestAccountServiceMockito {
@Test
public void test() {
      AccountManager mockAccountManager = mock(AccountManager.class);
     Account senderAccount = new Account( "1", 200 );
     Account beneficiaryAccount = new Account( "2", 100 );

      mockAccountManager.updateAccount( senderAccount );
      mockAccountManager.updateAccount( beneficiaryAccount );

      when(mockAccountManager.findAccountForUser("1")).thenReturn( senderAccount );
      when(mockAccountManager.findAccountForUser("2")).thenReturn( beneficiaryAccount )
```

Mockito 简介

```
        verify(mockAccountManager).findAccountForUser("1");
        verify(mockAccountManager).findAccountForUser("2");

        AccountService accountService = new AccountService();
        accountService.setAccountManager( mockAccountManager );
        accountService.transfer( "1", "2", 50 );

        assertEquals( 150, senderAccount.getBalance() );
        assertEquals( 150, beneficiaryAccount.getBalance() );
            }

}
```

Mockito技术

Mockito 简介

案例分析

练习

练习

对Stub知识的后面习题使用Mockito来完成

持续集成单元测试方法与应用

- 测试用例的设计
- 单元测试JUnit框架特性及使用
- 使用Stub进行测试
- Mock技术
- 容器内的测试
- 测试用例执行及数据分析统计
- 持续集成自动化回归测试

容器内的测试

- 表示层测试HttpUnit
- 表示层测试selenium
- 数据层DBUnit的测试

表示层测试HttpUnit

简介

HttpUnit 是基于JUnit构建的一个开源测试框架，专门针对Web应用的测试，解决使用JUnit框架无法对远程Web内容进行测试的弊端。

工作原理

HttpUnit通过模拟浏览器的行为，包括提交表单（form）、处理页面框架（frames）、基本的http验证、cookies及页面跳转（redirects）处理等。通过HttpUnit提供的功能，用户可以方便的和服务器端进行信息的交互，将返回的网页内容作为普通文本、XML Dom对象或者是作为链接、页面框架、图像、表单、表格等的集合进行处理，然后使用JUnit框架进行测试，还可以导向一个新的页面，然后进行新页面的处理，这个功能使你可以处理一组在一个操作链中的页面。

特征

HttpUnit因为关注的是这些控件的内容，而不管页面的表现形式（layout），所以不管表现形式如何变化，都不影响已确定的测试的可重用性。

表示层测试HttpUnit

如何使用httpunit处理页面的内容

WebConversation类是HttpUnit框架中最重要的类，WebConversation可以被看作是“浏览器”

```
//新创建一个“浏览器”对象
WebConversation wc = new WebConversation();
// WebRequest类，用于模仿客户的“请求”，通过它可以向服务器发送信息。
WebRequest req = new GetMethodWebRequest( http://www.sqalab.com );
// WebResponse类，用于模仿浏览器获取服务器端的响应信息。
WebResponse resp = wc.getResponse ( req );
```

获取指定页面的内容

通过 getText 直接获取页面的所有内容

```
// 建立一个“浏览器”实例
WebConversation wc = new WebConversation();
// 将指定URL的请求传给wc，然后获取相应的响应
WebResponse wr = wc.getResponse( "http://www.sqalab.com" );
// 用wc的getText方法获取相应的全部内容
System.out.println( wr.getText() );
```

表示层测试HttpUnit

增加参数通过Get方法访问页面

```
// 建立一个WebConversation实例
WebConversation wc = new WebConversation();
// 向指定的URL发出请求
WebRequest req = new GetMethodWebRequest( "http://www.srmlab.com/search" );
// 给请求加上参数
req.setParameter("keyword","httpunit");
// 获取响应对象
WebResponse resp = wc.getResponse( req );
// 用getText方法获取相应的全部内容
System.out.println( resp.getText() );
```

表示层测试HttpUnit

增加参数通过Post方法访问页面

```
//建立一个WebConversation实例
WebConversation wc = new WebConversation();
//向指定的URL发出请求
WebRequest req = new PostMethodWebRequest( "http://www.sqalab.com/search" );
//给请求加上参数
req.setParameter("keyword","httpunit");
//获取响应对象
WebResponse resp = wc.getResponse( req );
//用getText方法获取相应的全部内容
System.out.println( resp.getText() );
```

表示层测试HttpUnit

处理页面的链接（links）

```
// 建立一个WebConversation实例
WebConversation wc = new WebConversation();
// 获取响应对象
WebResponse resp = wc.getResponse( "http://www.sqalab.com/index.html" );
// 获得页面链接对象
WebLink link = resp.getLinkWith( "应用 HttpUnit 进行Web测试" );
// 模拟用户单击事件
link.click();
// 获得当前的响应对象
WebResponse nextLink = wc.getCurrentPage();
// 用getText方法获取相应的全部内容，并打印
System.out.println(nextLink.getText() );
```

```
<a href=" http://www.sqalab.com/article/html/article_59.html ">应用 HttpUnit 进行Web测试 </a>
```

表示层测试HttpUnit

处理页面的表格（table）

用 resp.getTables() 方法获取页面所有的表格对象。将它们依次出现在页面中的顺序保存在一个数组里。

```
// 创建一个WebConversation对象
WebConversation wc = new WebConversation();
// 设置HTTP代理服务器地址和端口
wc.setProxyServer( "proxy.pvgl.sap.corp", 8080 );
// 新建一个URL请求对象req
WebRequest req = new GetMethodWebRequest("http://httpunit.sourceforge.net/doc/cookbook.html");
// 发出一个请求req,并取得它相对应的响应resp
WebResponse resp = wc. (req);
// 获得响应的页面中的 TablegetResponse
WebTable[] tables = resp.getTables();
// 取出第一个 table
WebTable table = tables[0];
// 从 2 * 2 的 table 取出cell里的值
for ( int i=0 ; i<3 ; i++ ) {
        for ( int j=0 ; j<2 ; j++ )
                System.out.println(table.getCellAsText(i,j).trim());}
```

表示层测试HttpUnit

处理页面的表单（form）

用resp.getForms()方法获取页面所有的表单对象。他们依照出现在页面中的顺序保存在一个数组里面。

```
// 建立一个WebConversation实例
WebConversation wc = new WebConversation();
// 获取响应对象
WebResponse resp = wc.getResponse( "http://www.sqalab.com/article/html/article_59.html" );
// 获得对应的表单对象
WebForm webForm = resp.getForms()[0];
// 获得表单中所有控件的名字
String[] pNames = webForm.getParameterNames();
int i = 0;
int m = pNames.length;
// 循环显示表单中所有控件的内容
while(i<m){
System.out.println("第"+(i+1)+"个控件的名字是"+pNames[i]+",里面的内容是"+webForm.getParameterValue(pNames[i]));
++i;
}
```

表示层测试HttpUnit

如何使用httpunit进行测试

```
// 建立一个WebConversation实例
WebConversation wc = new WebConversation();
// 获取响应对象
WebResponse resp = wc.getResponse( "http://www.sqalab.com/article/html/article_59.html" );
// 获得对应的表格对象
WebTable webTable = resp.getTables()[0];
// 将表格对象的内容传递给字符串数组
String[][] datas = webTable.asText();
// 对表格内容进行测试
String expect = "中文";
// 增加 assertEquals 判断
Assert.assertEquals(expect,datas[0][0]);
```

表示层测试HttpUnit

测试Servlet的内部行为

```
// 创建Servlet的运行环境
ServletRunner sr = new ServletRunner();
// 向环境中注册Servlet
sr.registerServlet( "InternalServlet",  InternalServlet.class.getName() );
// 创建访问Servlet的客户端
ServletUnitClient sc = sr.newClient();
// 发送请求
WebRequest request = new GetMethodWebRequest( "http://localhost/InternalServlet" );
request.setParameter("pwd", "pwd");
// 获得该请求的上下文环境
InvocationContext ic = sc.newInvocation( request );
// 调用Servlet的非服务方法
InternalServlet is = (InternalServlet)ic.getServlet();
is.myMethod();
// 直接通过上下文获得request对象
System.out.println("request中获取的内容: "+ic.getRequest().getParameter("pw      d"));
// 直接通过上下文获得response对象，并且向客户端输出信息
ic.getResponse().getWriter().write("haha");
```

表示层测试HttpUnit

测试Servlet的内部行为

```
// 直接通过上下文获得session对象，控制session对象
// 给session赋值
ic.getRequest().getSession().setAttribute("username"，"timeson");
// 获取session的值
System.out.println("session中的值："+ic.getRequest().getSession().getAttribute      ("username"));
// 使用客户端获取返回信息，并且打印出来
WebResponse response = ic.getServletResponse();
System.out.println(response.getText());
```

表示层测试HttpUnit

案例分析

```
@Before
public void setUp(){
        wc = new WebConversation();
}
@Test
public void testEBusiness() throws IOException, SAXException {
        HttpUnitOptions.setScriptingEnabled(false);
        String username = "cindy";
        String password = "123456";
        WebRequest req = new PostMethodWebRequest("http://127.0.0.1:8000/login_action/");
        //给请求加上参数
        req.setParameter("username",username);
        req.setParameter("password",password);
        //获取响应对象
        WebResponse resp = wc.getResponse(req);
        //用getText方法获取相应的全部内容
        assertTrue(resp.getText().contains("购物车"));
}
```

表示层测试HttpUnit

案例分析

```
@Test
public void test3testing() throws IOException, SAXException {
        HttpUnitOptions.setScriptingEnabled(false);
        String CheckWord = "测试";
        WebResponse req = wc.getResponse("http://www.3testing.com/include/head.htm");
        WebLink link = req.getLinkWith("我的课程");
        link.click();
        WebResponse nextLink = wc.getCurrentPage();
        assertTrue(nextLink.getText().contains(CheckWord));
}
```

容器内的测试

- 表示层测试HttpUnit
- 表示层测试selenium
- 数据层DBUnit的测试

表示层测试 selenium

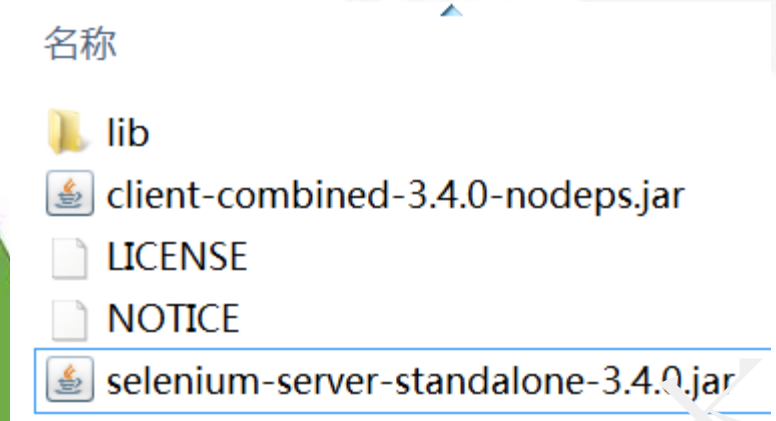
- 安装
- 对浏览器支持
- API介绍
- 案例
- 练习

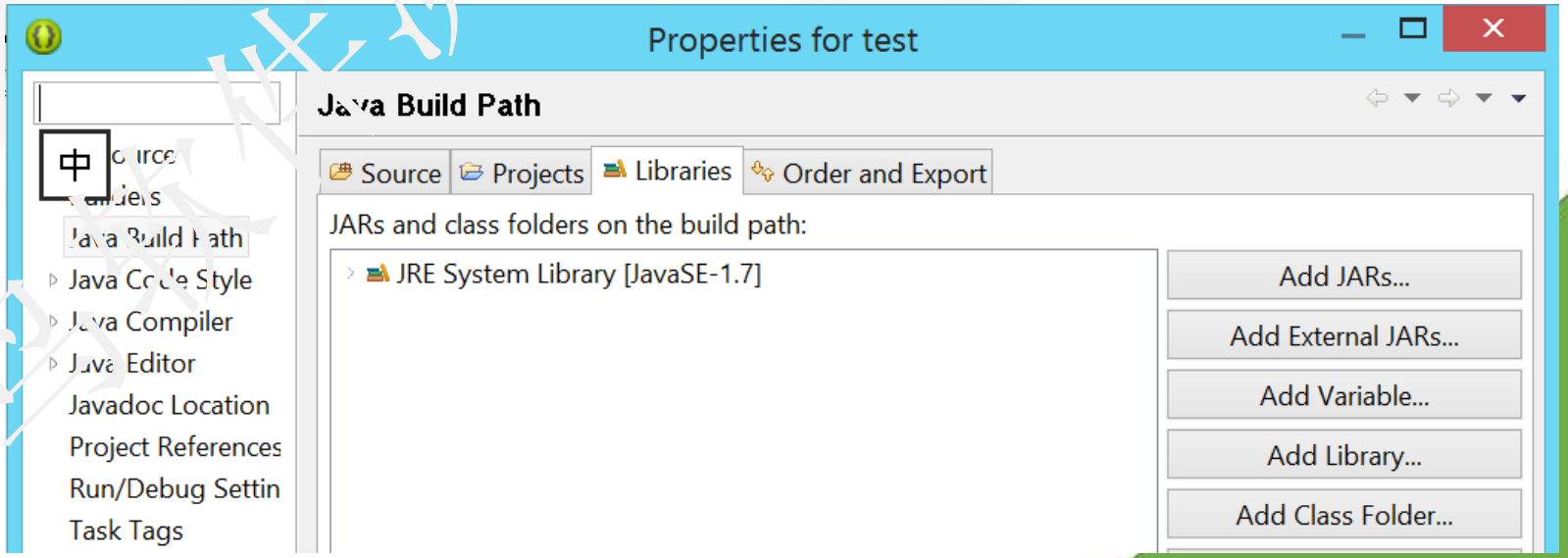
安装

- 到网上下载selenium3.4.0
- 解压 selenium-3.4.0.zip，放在本地一个非中文字符目录下，并且把selenium-server-standalone-3.4.0.jar放在该目录下。
- 根据Firefox类型，解压geckodriver-v0.16.1-win32.zip或geckodriver-v0.16.1-win64.zip到Firefox安装根目录下，比如：C:\Program Files (x86)\Mozilla Firefox\
- 根据IE类型，解压IEDriverServer_x64_3.4.0.zip或IEDriverServer_Win32_3.4.0.zip到IE安装根目录下，比如：C:\Program Files\Internet Explorer

安装

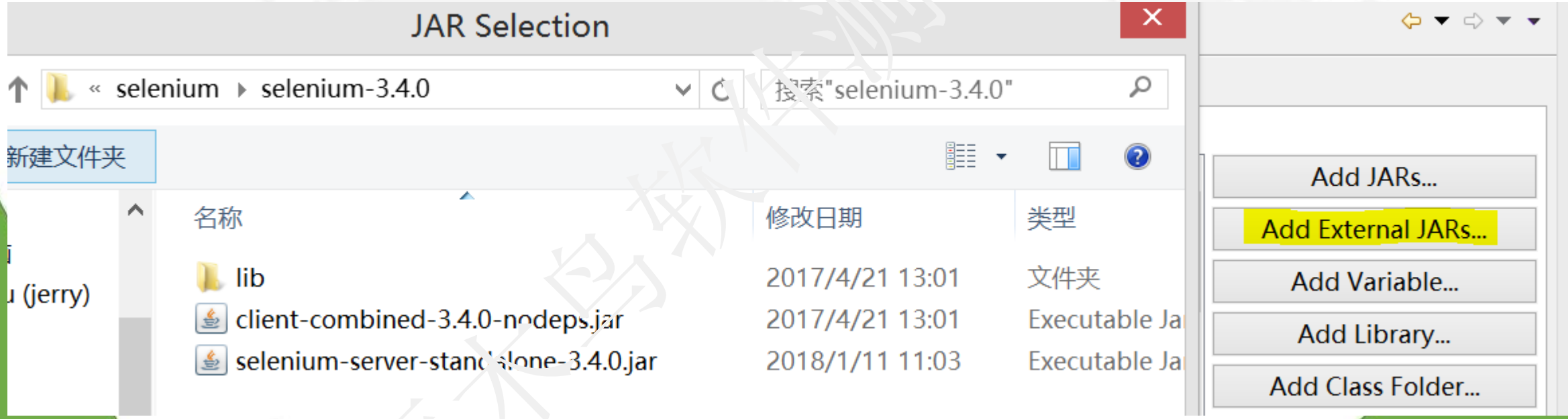
- 在Eclipse上建立Project、Package、Class
- 右击Project，在弹出菜单中点属性。
- 进入JavaBuild





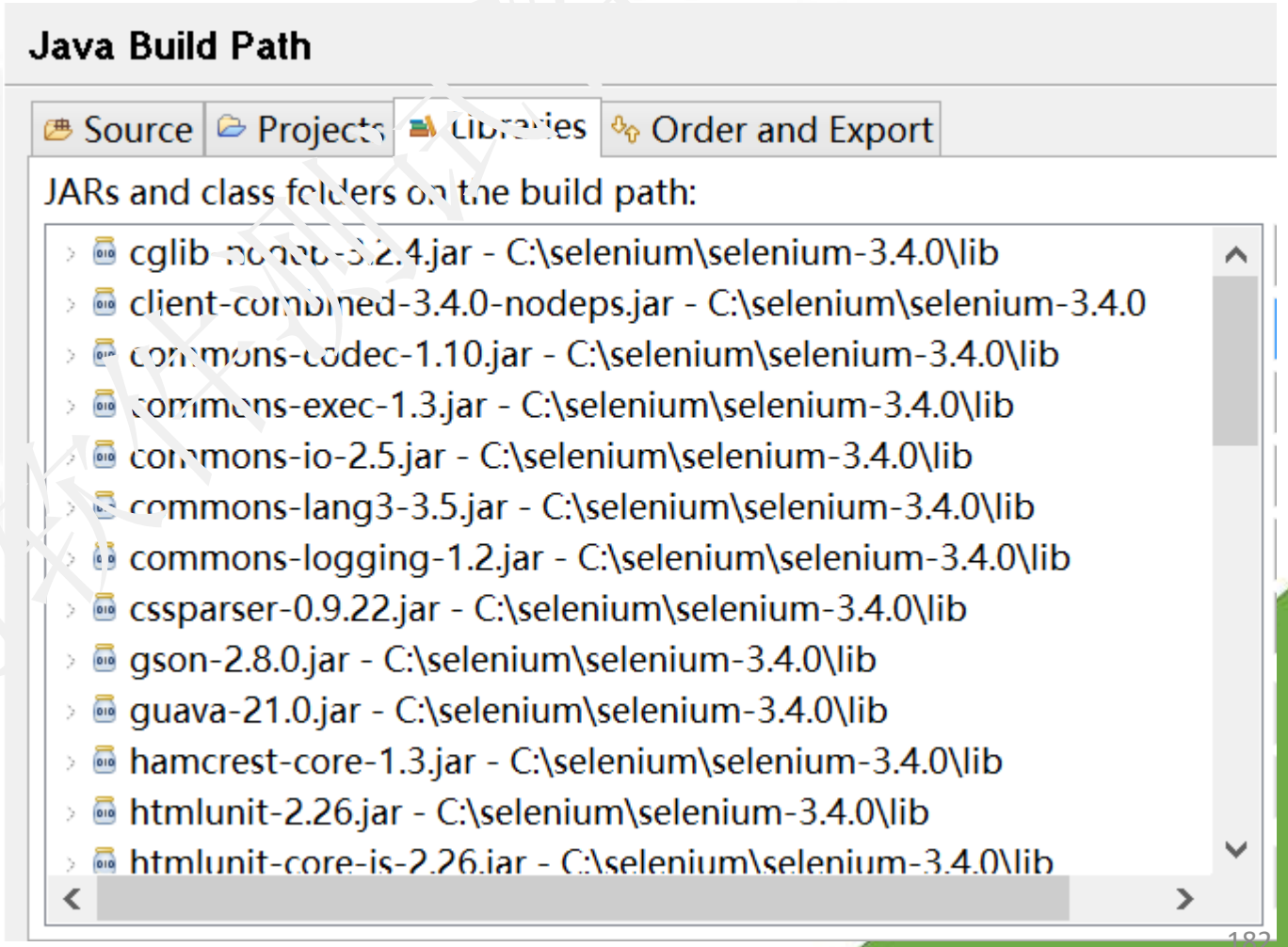
安装

- 点Add External JARs...
- 加入： client-combined-3.4.0-nodeps.jar和s selenium-server-standalone-3.4.0.jar



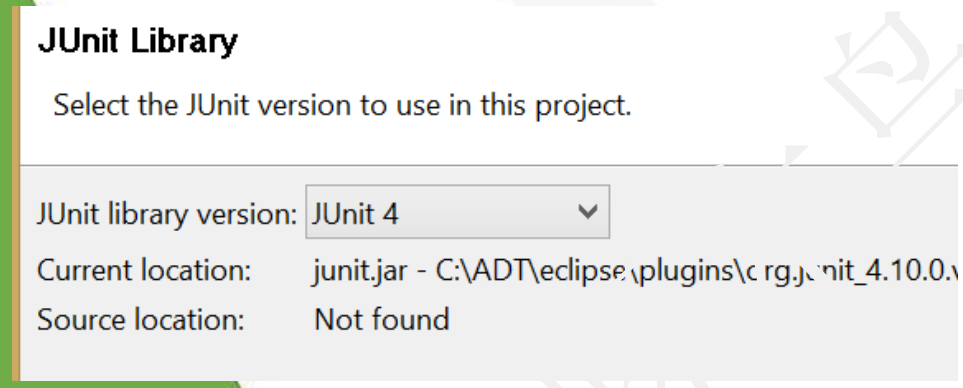
安装

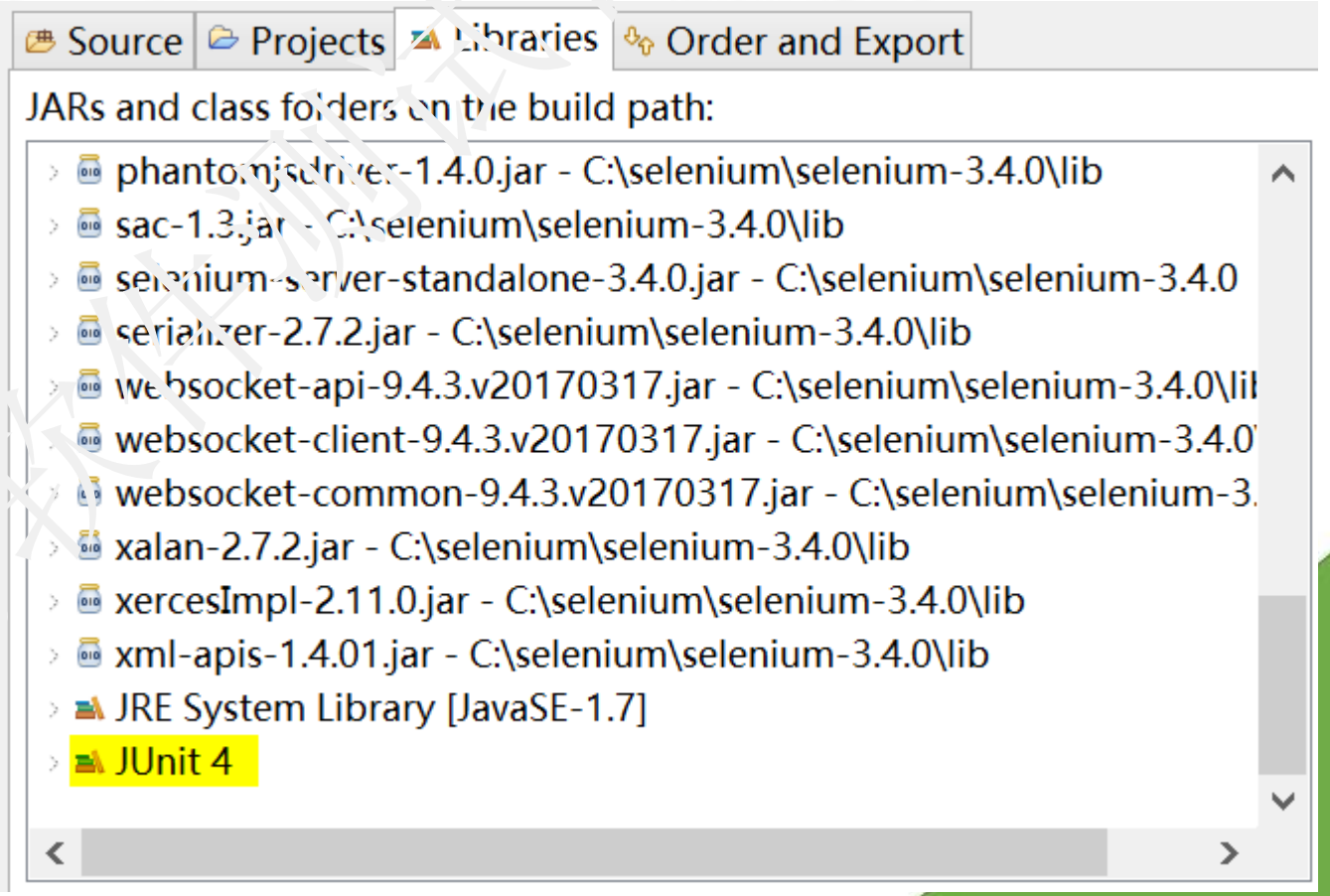
- 再次点Add External JARs...
- 进入libs目录
- 加入所有jar包



安装

- 点Add Libarary...
- 选择Junt
- 然后选择Junit4





安装

如何在安卓上使用

1, Setup Android emulator

a. Download the Android SDK

http://developer.android.com/sdk/index.html

Note that there is an emulator bug on Gingerbread (2.3.x) that might cause WebDriver to crash. My testing is on Ice Cream Sandwich (4.0.x)

b. Install Android SDK:

http://developer.android.com/sdk/installing.html

c. Start Android SDK Manager (SDK Manager.exe)

d. Select and install Packages online

e. Start AVD Manager.exe

f. Create an emulator

安装

如何在安卓上使用

2. Install the AndroidDriver APK by using platform-tools

a. list divce name:

```
>adb devices
```

b. download AndroidDriver APK:

http://code.google.com/p/selenium/downloads/list

c. install AndroidDriver APK：

```
>adb -s emulator-5554 -e install -r c:\android-server-2.9.apk
```

d. start the Android WebDriver application

```
>adb -s emulator-5554 shell am start -a android.intent.action.MAIN -n
org.openqa.selenium.android.apps/.MainActivity
```

e. setup the port forwarding in order to forward traffic from the host machine to the emulator

```
>adb -s emulator-5554 forward tcp:8080 tcp:8080
```

安装

注意：

- Firefox用版本必须大于为37的；
- IE根据32位或64位需要配置相应的IEDriverServer.exe文件
- FireFox根据32位或64位需要配置相应的geckodriver-v0.16.0-win64.exe文件
- Android用selenium-2.53.0

表示层测试 selenium

- 安装
- 对浏览器支持
- API介绍
- 案例
- 练习

对浏览器支持

1， HtmlUnit Driver

优点：HtmlUnit Driver不会实际打开浏览器，运行速度很快。对于用FireFox等浏览器来做测试的自动化测试用例，运行速度通常很慢，HtmlUnit Driver无疑是可以很好地解决这个问题。

缺点：它对JavaScript的支持不够好，当页面上有复杂JavaScript时，经常会捕获不到页面元素。

```
WebDriver driver = new HtmlUnitDriver();
```

对浏览器的支持

2， FireFox Driver

优点：FireFox Dirver对页面的自动化测试支持得比较好，很直观地模拟页面的操作，对JavaScript的支持也非常完善，基本上页面上做的所有操作FireFox Driver都可以模拟。

缺点：启动很慢，运行也比较慢，不过，启动之后Webdriver的操作速度虽然不快但还是可以接受的，建议不要频繁启停FireFox Driver。

```
// 如果你的 FireFox 没有安装在默认目录，那么必须在程序中设置
System.setProperty("webdriver.firefox.bin", "C:\\Program Files\\Mozilla Firefox\\firefox.exe");
System.setProperty("webdriver.gecko.driver" , "C:\\Program Files (x86)\\Mozilla Firefox\\geckodriver.exe" );
WebDriver driver = new FirefoxDriver();
```

Firefox profile的属性值是可以改变的，比如我们平时使用得非常频繁的改变useragent的功能，可以这样修改：

对浏览器的支持

```
FirefoxProfile profile = new FirefoxProfile();
profile.setPreference("general.useragent.override", "some UAstring");
WebDriver driver = new FirefoxDriver(profile);
```

3，IE的支持

优点：直观地模拟用户的实际操作，对JavaScript提供完善的支持。

缺点：是所有浏览器中运行速度最慢的，并且只能在Windows下运行，对CSS以及XPATH的支持也不够好。

```
System.setProperty("webdriver.ie.driver", "C:\\Program Files\\Internet Explorer\\IEDriverServer.exe");
driver = new InternetExplorerDriver();
```

4, 安卓的支持

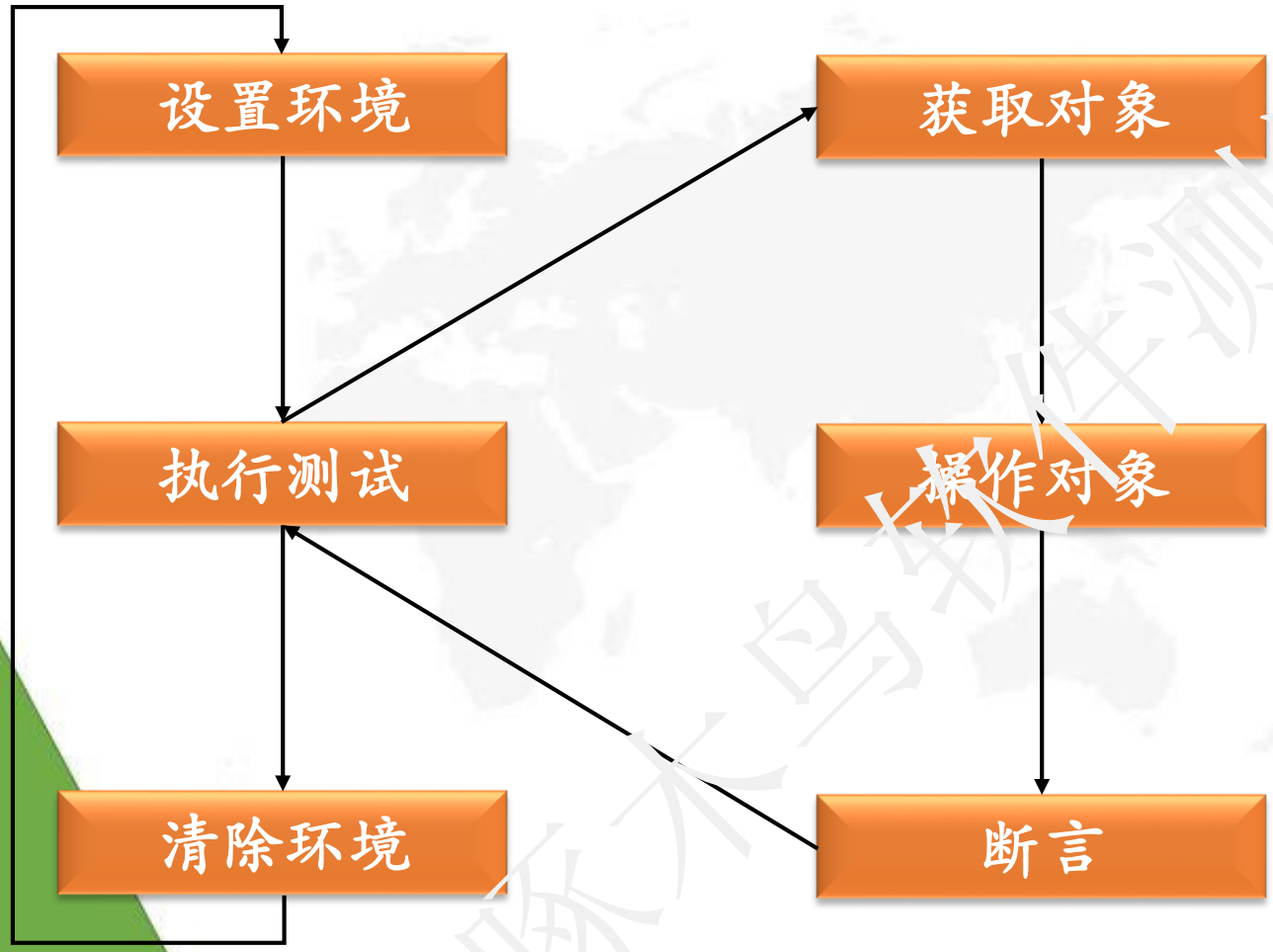
```
WebDriver driver = new AndroidDriver();
```

表示层测试 selenium

- 安装
- 对浏览器支持
- API介绍
- 案例
- 练习

API介绍

自动化测试综述



API介绍

定位

操作

断言

定位

如果不能定位到元素，就不要使用自动化测试了

```
<input type="text" name="passwd" id="passwd-id" />
WebElement element = driver.findElement(By.id("passwd-id"));
WebElement element = driver.findElement(By.name("passwd"));
<div class="cheese"><span>Cheddar</span></div>
WebElement cheeses = driver.findElements(By.className("cheese"));
<a href="http://www.google.com/search?q=cheese">软件测试</a>
WebElement cheese =driver.findElement(By.linkText("软件测试"));
WebElement cheese =driver.findElement(By.partialLinkText("测试"));
```

定位

```
<a href="http://www.baidu.com">百度</a>
<a href="http://www.taobao.com">淘宝</a>
```

```
WebElement Link = driver.findElement(By.tarName("a"));
List<WebElement> Links = driver.findElements(By.tarName("a"));
```

第一个获取第一个被匹配到的a标签连接；

第二个获取页面内所有被匹配到的a标签连接，被存储到List对象中。

定位

使用XPath定位

XPath： XML Path

1、使用绝对路径进行定位

```
<html>
<body>
<div id="div1">
<form action="login.jsp">
<input type="text"  name="username" maxlength="15">
<input type="text"  name="password" maxlength="10">
<input type="submit"  value="确定">
</form>
</div>
</body>
</html>
```

```
WebElement button =
driver.findElement(By.xpath("/html/body/div/form/input[@value='确定']"));
```

2、使用相对路径进行定位

```
WebElement button = driver.findElement(By.xpath("//input[@value='确定']"));
```

查找页面中，input的value为button的元素

定位

3、使用索引号来定位

```
WebElement button = driver.findElement(By.xpath("//input[2]"));
```

定位到第二次出现的input元素，在Firefox中不建议使用，在input数量变化的情况下不建议使用。

4、通过页面属性值来进行定位

```
<img alt="div-img" src="/image/dog.jpg">
```

```
WebElement img = driver.findElement(By.xpath("img[@alt='div-img']"));
```

5、通过页面模糊属性值来进行定位

```
WebElement img = driver.findElement(By.xpath("img[starts-with(@alt,'div')]"));
WebElement img = driver.findElement(By.xpath("img[contains(@alt, 'img')]"));
```

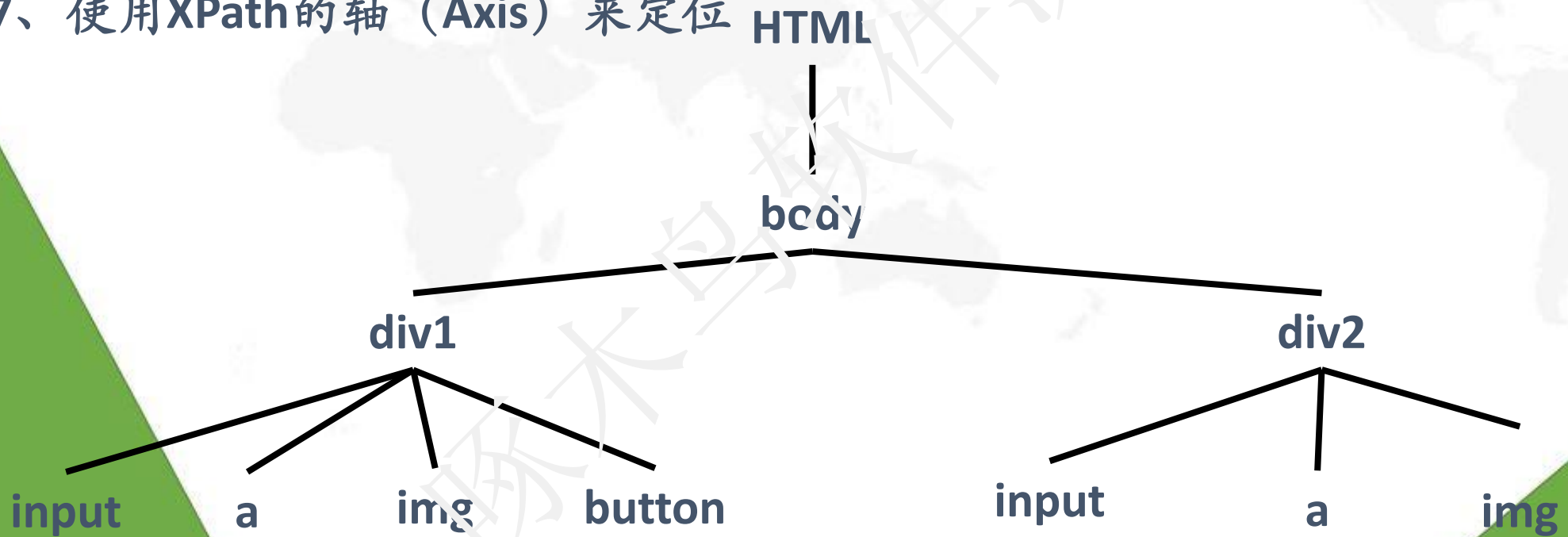
定位

6、使用文本来定位

```
<p>软件测试</p>
```

```
WebElement p= driver.findElement(By.xpath("//p[test()=软件测试]"));
WebElement p= driver.findElement(By.xpath("//p[contains(test()=测试)]"));
```

7、使用XPath的轴（Axis）来定位



定位

Xpath轴关键字	轴的定义说明	定位表达式实例	表达式解释
parent	选取当前节点的父节点	//img[@alt='div2-img2']/parent::div	查找到alt属性为div2-img2的img元素，并基于图片找到其上一级的div元素
child	选取当前节点的子节点	//div[@id='div1']/child::img	查找id为div1的div标签，基于当前div查找标签为img的子节点
ancestor	选取当前节点的所有上层节点	//img[@alt='div2-img2']/ancestor::div	查找alt属性为div2-img2的图片，基于当前图片找到其上级的div页面元素
descendant	选取当前节点所有下层节点	//div[@id='div2']/descendant::img	查找id属性为div2的div元素，在查找其下级所有节点中的img元素
following	选取当前节点之后显示的所有节点	//div[@id='div1']/following::img	查找到ID属性为div1的div元素，并基于div的位置找到它后面节点中的img元素
following-sibling	选取当前节点所有的平级节点	//img[@alt='div1-img1']/following-sibling::input	查找到alt属性为div1-img1的img页面元素并基于img的位置找到后续节点中的input元素
preceding	选取当前节点前面所有的节点	//img[@alt='div2-img2']/preceding::div	查找到alt属性为div2-img2的图片页面元素，并基于图片的位置找到它前面节点中的div元素
preceding-sibling	选取当前节点前面所有平级的节点	//img[@alt='div2-img2']/preceding-sibling::a[1]	查找到alt属性值为div2-img2的图片元素，基于图片位置找到它前面同级节点的第二个链接页面元素

定位

使用CSS定位

1、使用绝对路径进行定位

```
WebElement button =
driver.findElement(By.cssSelector("html>body>div>form>input[type='submit']"));
```

2、使用相对路径进行定位

```
WebElement button = driver.findElement(By.cssSelector("input[type='submit']"));
```

查找页面中，input的type为submit的元素

3、使用class名称和id属性进行定位

```
<input type="text" class="username" id="1">
```

```
WebElement button = driver.findElement(By.cssSelector("input.username"));
WebElement button = driver.findElement(By.cssSelector("input#1"));
```

定位

4、使用页面其他属性进行定位

```
<img alt="baidu" src="/image/baidu.jpg">
```

```
WebElement img = driver.findElement(By.cssSelector("img[alt='baudu']"));
WebElement img =
driver.findElement(By.cssSelector("img[alt='baudu'][src='/image/baidu.jpg']"));
```

5、使用页面属性值的一部分进行定位

```
<a href="http://www.china-esp.com">科普端IT学院</a>
```

```
WebElement a = driver.findElement(By.cssSelector("a[href^='http://www.c']")); //^表示开头
WebElement a = driver.findElement(By.cssSelector("a[href$='esp.com']")); //$表示结尾
WebElement a = driver.findElement(By.cssSelector("a[href*='china-esp']")); //*表示包含
```

定位

6、使用页面元素的子页面元素进行定位

```
<div id="div1">
<input id="name" type="text"></input>
```

```
WebElement input = driver.findElement(By.cssSelector("div#div1>input#name"));
```

7、使用伪类元素进行定位

```
WebElement input1 = driver.findElement(By.cssSelector("div#div1:first-child"));//第一个元素
WebElement a = driver.findElement(By.cssSelector("div#div1:nth-child(2)"));//第二个元素
WebElement input2=driver.findElement(By.cssSelector("div#div1:last-child"));//最后一个元素
WebElement input = driver.findElement(By.cssSelector("input:focus"));//聚焦元素
WebElement chexcbox1 = driver.findElement(By.cssSelector("input:enable"));//可操作元素
WebElement chexcbox2 = driver.findElement(By.cssSelector("input:checked"));//勾选元素
```

定位

8、查找同级页面下的兄弟元素

```
1:<div id="div1">
2:<input id="name" type="text"></input>
3:<a href="http://www.china-esp.com">科普瑞IT学院</a>
4:<img src=/image/rabit.png>
5:</div>
```

```
WebElement input = driver.findElement(By.cssSelector("div#div1 >input+a"));//定位到3
WebElement input = driver.findElement(By.cssSelector("div#div1 >input+a+img"));//定位到4
WebElement input = driver.findElement(By.cssSelector("div#div1 >input+*+img"));//定位到4
```

定位

XPath与CSS定位

CSS定位比XPath定位快，XPath定位跟强大

定位元素目标	XPath	CSS
所有元素	//*	*
所有的div元素	//div	div
所有的div元素的子元素	//div/*	div>*
根据id定位元素	//*[@id='div1']	#div1
根据class定位元素	//*[contains(@class, 'name')]	.name
拥有某个属性的元素	//*[@href]	*[href]
所有div元素的第一个元素	//div/*[1]	div>*:first-child
所有拥有子元素a的div元素	//div[a]	无法实现
input下的兄弟元素	//input/following-sibling::*[1]	Input+a

API介绍

定位

操作

断言

操作

1，如何对页面元素进行操作

1.1输入框（text field or textarea）

- 找到输入框元素：

```
WebElement element = driver.findElement(By.id("passwd-id"));
```

- 在输入框中输入内容：

```
element.sendKeys("test");
```

- 将输入框清空：

```
element.clear();
```

- 获取输入框的文本内容：

```
element.getText();
```

操作

1.2下拉选择框(Select)

找到下拉选择框的元素：

```
select select = new Select(driver.findElement(By.id("select")));
```

选择对应的选择项：

```
select.selectByVisibleText("mediaAgencyA");或
select.selectByValue("MA_ID_001");
```

不选择对应的选择项：

```
select.deselectAll();
select.deselectByValue("MA_ID_001");
select.deselectByVisibleText("mediaAgencyA");
```

或者获取选择项的值：

```
select.getAllSelectedOptions();
select.getFirstSelectedOption();
```

操作

1.3单选项(Radio Button)

- 找到单选框元素：

```
WebElement bookMode =driver.findElement(By.id("BookMode"));
```

- 选择某个单选项：

```
bookMode.click();
```

- 清空某个单选项：

```
bookMode.clear();
```

- 判断某个单选项是否已经被选择：

```
bookMode.isSelected();
```

操作

1.4 多选项(checkbox)

- 多选项的操作和单选的差不多：

```
WebElement checkbox =driver.findElement(By.id("myCheckbox"));

checkbox.click();

checkbox.clear();

checkbox.isSelected();

checkbox.isEnabled();
```

操作

1.5 按钮(button)

- 找到按钮元素:

```
WebElement saveButton = driver.findElement(By.id("save"));
```

- 点击按钮:

```
saveButton.click();
```

- 判断按钮是否enable:

```
saveButton.isEnabled ();
```

操作

1.6 左右选择框

也就是左边是可供选择项，选择后移动到右边的框中，反之亦然。例如：

```
Select lang = new Select(driver.findElement(By.id("languages")));
lang.selectByVisibleText(“English”);
WebElement addLanguage =driver.findElement(By.id("addButton"));
addLanguage.click();
```

1.7 弹出对话框(Popup dialogs)

```
Alert alert = driver.switchTo().alert();
alert.accept();
alert.dismiss();
alert.getText();
```

操作

1.8 表单(Form)

Form中的元素的操作和其它的元素操作一样，对元素操作完成后对表单的提交可以：

```
WebElement approve = driver.findElement(By.id("approve"));
approve.click();
```

或

```
approve.submit();
```

操作

1.9 上传文件 (Upload File)

- 上传文件的元素操作：

```
WebElement adFileUpload = driver.findElement(By.id("WAP-upload"));

String filePath = "C:\\test\\uploadfile\\media_ads\\test.jpg";

adFileUpload.sendKeys(filePath);
```

操作

1.10 Windows 和Frames之间的切换

- 一般来说，登录后建议是先进入默认frame：

```
driver.switchTo().defaultContent();
```

- 切换到某个Frames：

```
driver.switchTo().frame("leftFrame");
```

- 从一个Frames切换到另一个Frames：

```
driver.switchTo().frame("mainFrame");
```

- 切换到某个window：

```
driver.switchTo().window("windowName");
```

操作

1.11 拖拉(Drag and Drop)

```
WebElement element =driver.findElement(By.name("source"));
WebElement target = driver.findElement(By.name("target"));
(new Actions(driver)).dragAndDrop(element, target).perform();
```

1.12 导航 (Navigationand History)

- 打开一个新的页面：

```
driver.navigate().to("http://www.example.com");
```

- 通过历史导航返回原页面：

```
driver.navigate().forward();
driver.navigate().back();
```

操作

2，高级使用

2.1改变user agent

User Agent的设置是平时使用得比较多的操作：

```
FirefoxProfile profile = new FirefoxProfile();
profile.addAdditionalPreference("general.useragent.override","some UA string");
WebDriver driver = new FirefoxDriver(profile);
```

操作

2.2增加cookie

我们经常要对的值进行读取和设置。

```
// Now set the cookie. This one's valid for the entire domain

Cookie cookie = new Cookie("key", "value");

driver.manage().addCookie(cookie);
```

操作

2.3 获取cookie的值：

```
// And now output all the available cookies for the current URL
Set<Cookie> allCookies = driver.manage().getCookies();
for (Cookie loadedCookie : allCookies) {
  System.out.println(String.format("%s -> %s",loadedCookie.getName(),
loadedCookie.getValue()));
}
```

据某个cookie的name获取cookie的值：

```
driver.manage().getCookieNamed("mmsid");
```

操作

2.4 删除cookie:

```
// You can delete cookies in 3 ways
// By name
driver.manage().deleteCookieNamed("CookieName");
// By Cookie
driver.manage().deleteCookie(loadedCookie);
// Or all of them
driver.manage().deleteAllCookies();
```

操作

2.5 Web截图

如果用WebDriver截图是：

```
driver = webdriver.Firefox();

driver.save_screenshot("C:\error.jpg");
```

操作

2.6页面等待

Webdriver提供两种方法，一种是显性等待，另一种是隐性等待。

2.6.1显性等待:

```
WebDriver driver =new FirefoxDriver();
driver.get("http://somedomain/url_that_delays_loading");
WebDriverWait wait = new WebDriverWait(driver, 10);
wait.until(ExpectedConditions.titleContains("百度"));
wait.until(ExpectedConditions.presenceOfElementLocated(By.id("kw")));
```

操作

等待的条件	WebDriver方法
页面元素是否在页面上可用和可被单击	elementToBeClickable(By locator)
页面元素处于被选中状态	elementToBeSelected(WebElement element)
页面元素在页面中存在	presenceOfElementLocated(By locator)
在页面元素中是否包含特定的文本	textToBePresentInElement(By locator)
页面元素值	textToBePresentInElementValue(By locator, java.lang.String text)
标题 (title)	titleContains(java.lang.String title)

操作

2.6.2 隐性等待:

```
WebDriver driver = new FirefoxDriver();

driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

driver.get("http://somedomain/url_that_delays_loading");

WebElement myDynamicElement

=driver.findElement(By.id("myDynamicElement"));
```

API介绍

定位

操作

断言

断言

```
//判断元素是否存在
public Boolean isWebElementExist(WebDriver driver,By selector) {
        try {
                driver.findElement(selector);
                        return true;
                } catch(Exception e) {
                        e.printStackTrace();
                        driver.quit();
                        return false;
        }
}
```

返回页面标题

- driver.getTitle()

返回当前页面的url

- driver.getcurrentUrl

表示层测试 selenium

- 安装
- 对浏览器支持
- API介绍
- 案例
- 练习

案例

myWebTestUnit.java

```
public class myWebTestUnit {
        public static WebDriver driver=null;
        public static check mycheck=new check();
        public static String browser="IE";
        @Before
        public void setUp() throws Exception {
                driver=mycheck.checkBrower(browser);
        }

        @Test
        public void testBaidu() {
                String inputString="软件测试";
                // 进入Baidu
                driver.get("https://www.baidu.com");
                // 通过 id 找到 input 的 DOM
                WebDriverWait wait = new WebDriverWait(driver,10);
                wait.until(ExpectedConditions.presenceOfElementLocated(By.id("kw")));
                WebElement element = driver.findElement(By.id("kw"));
                element.clear();
```

案例

myWebTestUnit.java

```
            // 输入关键字
            element.sendKeys(inputString);
            //提交 input 所在的 form
             element.submit();
             //显示搜索结果页面的 title
             wait.until(ExpectedConditions.titleContains("百度搜索"));
            assertEquals(inputString+"_百度搜索",driver.getTitle());
}

@Test
public void test3testing() {
            String menu="我的介绍";
            String checktext="顾翔";
            driver.get("http://www.3testing.com");
            driver.switchTo().defaultContent();
            driver.switchTo().frame("head");
            WebDriverWait wait = new WebDriverWait(driver,10);
            wait.until(ExpectedConditions.presenceOfElementLocated(By.linkText(menu)));
            WebElement myinfomation =driver.findElement(By.linkText(menu));
```

案例

myWebTestUnit.java

```
                myinfomation.click();
                driver.switchTo().defaultContent();
                assertEquals(true,mycheck.isTextPresent(driver,checkText));
        }

        @After
        public void teardown() throws Exception {
                driver.close();
        }
}
```

表示层测试 selenium

- 安装
- 对浏览器支持
- API介绍
- 案例
- 练习

练习

测试eBusines登录

容器内的测试

- 表示层测试HttpUnit
- 表示层测试selenium
- 数据层DBUnit的测试

数据层DBUnit的测试

DBUnit 简介

案例分析

练习

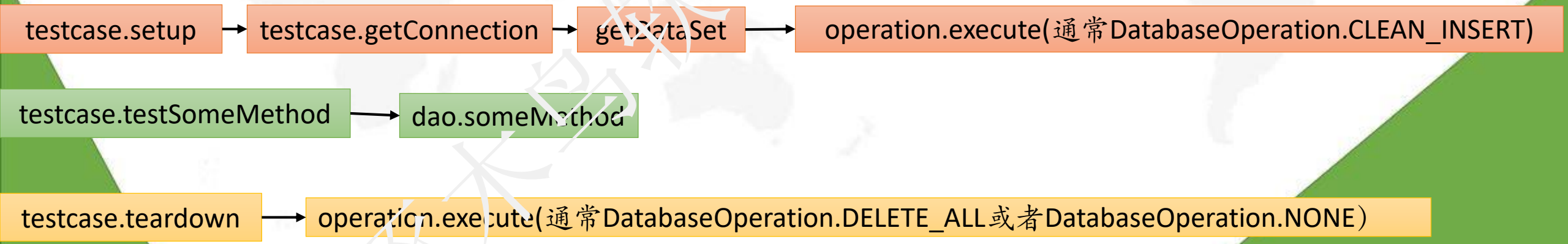
DBUnit 简介

DbUnit是一个基于JUnit扩展的数据库测试框架。它提供了大量的类对与数据库相关的操作进行了抽象和封装，虽然在80%的情况，你只需使用它极少的API。它通过使用用户自定义的数据集以及相关操作使数据库处于一种可知的状态，从而使得测试自动化、可重复和相对独立。虽然不用DbUnit也可以达到这种目的，但是我们必须为此付出代价（编写大量代码，测试及维护），既然有了这么优秀的开源框架，我们又何必再造轮子。

官网：http://dbunit.sourceforge.net/

下载地址：http://download.csdn.net/detail/sanfye/8992365

DbUnit的与单元测试相关的两个最重要的核心是org.dbunit.database.IDatabaseConnection 和 org.dbunit.dataset.IDataSet，前者是产品代码使用的数据库连接的一个简单的封装，后者是对单元测试人员自定义的数据集（通常以xml文件的形式存在，且xml文件的格式也有好几种）的封装



数据层DBUnit的测试

DBUnit 简介

案例分析

练习

DBUnit 简介

User.java

```
package com.jerry;

public class User {
  private long id;
  private String username;
  private String firstName;
  private String lastName;
  public long getId() {
    return id;
  }
  public void setId(long id) {
    this.id = id;
  }
  public String getUsername() {
    return username;
  }
  public void setUsername(String login) {
    this.username = login;
  }
  public String getFirstName() {
    return firstName;
  }
  public void setFirstName(String name) {
    this.firstName = name;
  }
  public String getLastName() {
    return lastName;
  }
  public void setLastName(String lastName) {
    this.lastName = lastName;
  }
}
```

UserDao.java

```
package com.jerry;
import java.sql.SQLException;
public interface UserDao {

  int addUser( User user ) throws SQLException;

  User getUserById(long id) throws SQLException;

}
```

DBUnit 简介

UserDaoJdbcImpl.java

```
package com.jerry;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class UserDaoJdbcImpl implements UserDao {

 private Connection connection;

 public void setConnection(Connection connection) {
  this.connection = connection;
 }                                          设置连接

 public Connection getConnection() {
  return connection;
 }                                          获得连接
```

DBUnit 简介

```
  public int addUser(User user) throws SQLException {      建立用户，返回主键
//    PreparedStatement pstmt = connection.prepareStatement("INSERT INTO users VALUES (?,?,?)",
Statement.RETURN_GENERATED_KEYS);
    connection.setAutoCommit(false);
    PreparedStatement pstmt = connection.prepareStatement("INSERT INTO users (username, first_name, last_name) VALUES
(?,?,?)");
    try {
      pstmt.setString(1, user.getUsername());
      pstmt.setString(2, user.getFirstName());
      pstmt.setString(3, user.getLastName());
      pstmt.executeUpdate();
      int id = getLastIdentity();
      connection.commit();
      return id;
    } finally {
      close(pstmt);
      connection.setAutoCommit(true);
    }
  }
```

DBUnit 简介

```
private int getLastIdentity() throws SQLException {        获得最大的主键
        PreparedStatement pstmt = connection.prepareStatement("select max(id) from users;");
 ResultSet rs = null;
 try {
   rs = pstmt.executeQuery();
   rs.next();
   int id = rs.getInt(1);
   return id;
 } finally {
   close(rs, pstmt);
 }
}

public User getUserById(long id) throws SQLException {    通过主键寻找用户记录
 PreparedStatement pstmt = connection.prepareStatement("SELECT * FROM users WHERE id = ?");
 ResultSet rs = null;
 User user = null;
 try {
   pstmt.setLong(1, id);
   rs = pstmt.executeQuery();
```

数据层DBUnit的测试

```
if ( rs.next() ) {
      user = new User();
      fill( user, rs );
    }
  } finally {
    close(rs, pstmt);
  }
  return user;
}

private void fill(User user, ResultSet rs) throws SQLException {   查询完毕，建立User记录
  user.setUsername(rs.getString("username"));
  user.setFirstName(rs.getString("first_name"));
  user.setLastName(rs.getString("last_name"));
  user.setId(rs.getLong("id"));
}

void createTables() throws SQLException {          建立表
  Statement stmt = connection.createStatement();
  try {
```

数据层DBUnit的测试

```
    stmt.execute("CREATE TABLE users(id INT NOT NULL AUTO_INCREMENT,username VARCHAR(10),first_name
VARCHAR(10),last_name VARCHAR(10),PRIMARY KEY (id));");
   } finally {
    close(stmt);
   }
  }

  void dropTables() throws SQLException {        删除表
            Statement stmt = connection.createStatement();
             try {
               stmt.execute("drop table users");
             } finally {
               close(stmt);
             }
  }

  private void close(ResultSet rs, Statement pstmt) {   关闭记录（rs）和执行语句（stmt）
   if ( rs != null ) {
    try {
     rs.close();
```

数据层DBUnit的测试

```
  } catch (SQLException e) {
      e.printStackTrace();
    }
   }
   close(pstmt);
  }

  private void close(Statement stmt) {
   try {
    stmt.close();
   } catch (SQLException e ) {
    e.printStackTrace();
   }
  }

}
```

执行语句（stmt）

数据层DBUnit的测试

测试

初始化

- 定义驱动名称，我们这里用的是com.mysql.jdbc.Driver
- 定义数据库用户名字符串变量
- 定义数据库密码字符串变量
- 定义数据库名称字符串变量
- 通过数据库用户名、数据库密码和数据库名称拼接数据库连接url
- 通过Class.forName(driverName).newInstance();建立驱动实例
- 获得数据库连接
- 建立DbUnit连接
- 建立数据库连接
- 建立数据库表

结束

- 清除数据库表
- 断开DbUnit连接
- 设置DbUnit连接变量为null

数据层DBUnit的测试

原理

数据层DBUnit的测试

AbstractDbUnitTestCase.java

```
package com.jerry;

import static org.junit.Assert.*;

public abstract class AbstractDbUnitTestCase {
 //根据XML表初始化数据
 public static final String USER_FIRST_NAME = "Jeffrey";
 public static final String USER_LAST_NAME = "Lebowsky";
 public static final String USER_USERNAME = "ElDuderino";
 //初始化私有变量
 protected static UserDaoJdbcImpl dao = new UserDaoJdbcImpl(); //定义被测对象
 protected static Connection connection; //定义数据库连接
 protected static HsqldbConnection dbunitConnection; //定义DbUnit数据库连接

 @BeforeClass //初始化，注意@BeforeClass必须为static方法
 public static void setupDatabase() throws Exception {
        String driverName="com.mysql.jdbc.Driver"; //定义驱动名称，我们这里用的是com.mysql.jdbc.Driver
        String userName="root"; //数据库用户名
        String userPasswd="123456"; //数据库密码
        String dbName="hr"; //数据库名称
```

数据层DBUnit的测试

```
  String url="jdbc:mysql://localhost/"+dbName+"?user="+userName+"&password="+userPasswd; //拼接数据库连接url
  Class.forName(driverName).newInstance(); //建立驱动实例
  connection = DriverManager.getConnection(url); //获得数据库连接
  dbunitConnection = new HsqldbConnection(connection,null); //建立DbUnit连接
  dao.setConnection(connection); //建立数据库连接
  dao.createTables(); //建立数据库表
}

@AfterClass
public static void closeDatabase() throws Exception {
  if ( dbunitConnection != null ) {
    dao.dropTables(); //删除表
    dbunitConnection.close(); //关闭DbUnit连接
    dbunitConnection = null; //设置DbUnit连接为空
  }
}
```

从XML数据集中读取数据和结构，实例化为IDataSet对象，返回的Idataset代表数据表里面将要存放的数据。

```
public static IDataSet getDataSet(String name) throws Exception {
  InputStream inputStream = new FileInputStream(name);
  assertNotNull("file " + name + " not found in classpath", inputStream );
```

数据层DBUnit的测试

```
  Reader reader = new InputStreamReader(inputStream);
  FlatXmlDataSet dataset = new FlatXmlDataSet(reader);
  return dataset;
}

//调用下面一个方法，获得数据集
public static IDataSet getReplacedDataSet(String name, long id) throws Exception {
  IDataSet originalDataSet = getDataSet(name); //获得XML数据，数据为originalDataSet（原始数据集）
  return getReplacedDataSet(originalDataSet, id); //返回并且调用下面一个方法
}

//将原始变量进行替换
public static IDataSet getReplacedDataSet(IDataSet originalDataSet, long id) throws Exception {
  ReplacementDataSet replacementDataSet = new ReplacementDataSet(originalDataSet); //构造函数
  replacementDataSet.addReplacementObject("[ID]", id); //定义被替换的内容，见user-token.xml， <users id="[ID]" [ID]->id
  replacementDataSet.addReplacementObject("[NULL]", null); //[NULL]->null
  return replacementDataSet;
}
```

<users id="[ID]"，可能某个记录的ID在数据库中不确定，可以忽略

数据层DBUnit的测试

```
public static User newUser() {  //建立新用户
            User user = new User();
            user.setFirstName(USER_FIRST_NAME);
            user.setLastName(USER_LAST_NAME);
            user.setUsername(USER_USERNAME);
            return user;
          }

 public static void assertUser(User user) {  //验证用户
            assertNotNull(user);
            assertEquals(USER_FIRST_NAME, user.getFirstName());
            assertEquals(USER_LAST_NAME, user.getLastName());
            assertEquals(USER_USERNAME, user.getUsername());
          }

}
```

数据层DBUnit的测试

DataBaseOperation操作

操作	解释
DatabaseOperation.UPDATE	这个操作将从测试数据源中读取的数据集的内容更新到数据库中，注意这个操作正确执行的前提是测试数据表已经存在，如果不存在这个测试用例将会失败
DatabaseOperation.INSERT	这个操作把从测试数据源中读取的数据集的内容插入到数据库中，注意这个操作正确执行的前提是测试数据表不存在，这个操作将新建数据表。如果测试数据表已经存在这个测试用例将会失败。另外，在执行这个操作的时候要特别注意数据集中数据表的顺序，否则可能会因为违反外键约束而造成测试用例失败
DatabaseOperation.DELETE	这个操作会从数据库中删除数据，注意，这个操作只删除数据集中存在的数据行而不是整个数据表中的数据
DatabaseOperation.DELETE_ALL	这个操作删除数据表中的所有行，注意，这个操作也只影响数据集中声明了的数据表，数据集中没有涉及到的数据表中的数据不会删除
DatabaseOperation.TRUNCATE	这个操作将删除数据集中声明的数据表，如果数据中有些表并没有在预定义的数据集中提到，这个数据表将不会被影响。注意，这个操作是按照相反的顺序执行的
DatabaseOperation.REFRESH	顾名思义，这个操作将把预定义数据集中的数据同步到数据库中，也就是说这个操作将更新数据数中已有的数据、插入数据库中没有的数据。数据库中已有的、但是数据集中没有的行将不会被影响。我们用一个产品数据库的拷贝进行测试的时候可以使用这个操作将预定义数据同步到产品数据库中

数据层DBUnit的测试

操作	解释
DatabaseOperation.CLEAN_INSERT	删除所有的数据表中的数据，然后插入数据集中定义的数据，如果你想保证数据库是受控的，这个比较喜欢
DatabaseOperation.NONE	不执行任何操作
CompositeOperation	将多个操作组合成一个，便以维护和重用
TransactionOperation	在一个事物内执行多个操作
IdentityInsertOperation	在使用MSSQL的时候，插入数据时IDENTITY列我们是没有办法控制的，用这个就可以控制了，只有在使用MSSQL的时候才会用得到

数据层DBUnit的测试

org.dbunit.Assertion的四种判断方案(建议使用第一个)

```
//从预期文件中读取预期结果集
IDataSet dataSet=new XmlDataSet(new FileInputStream(exceptFile));
//比较结果集
Assertion.assertEquals(buildBackUpDataSet(),dataSet);
```

```
//includeColumn是指  比较包含 后面String数组列 进行比较
//excludeColumn是指  比较排除 后面String数组列 进行比较
Assertion.assertEquals(this.includeColumn(this.buildExceptTableByName("config.page_home"), new String[]{"user_check"}),
this.includeColumn(this.buildTableByName("config.page_home"), new String[]{"user_check"}))
```

数据层DBUnit的测试

org.dbunit.Assertion的四种判断方案

```
//比较表数据 （以表的形式进行数据比较）
//从xml的结果集中获取需要的表数据
ITable iTable=dataSet.getTable("config.page_home");
//第一种方法 获取数据库中的表数据
ITable dbTable =this.buildTableByName("config.page_home");
//第二种方法 从结果集中获取所需要的表信息数据
ITable dbTable=buildBackUpDataSet().getTable("config.page_home");
//进行比较
Assertion.assertEquals(dbTable,iTable);
```

```
//通过表比较列
//通过表获取到表中所需要的列 并将用户所需要的列形成一个新的临时表
iTable=DefaultColumnFilter.excludedColumnsTable(iTable,new String[]{"user_check"});
dbTable=DefaultColumnFilter.includedColumnsTable(dbTable,new String[]{"user_check"});
Assertion.assertEquals(dbTable,iTable);
```

数据层DBUnit的测试

org.dbunit.Assertion方法

- static void assertEquals(IDataSet expectedDataSet, IDataSet actualDataSet)
- static void assertEquals(IDataSet expectedDataSet, IDataSet actualDataSet, FailureHandler failureHandler)
- static void assertEquals(ITable expectedTable, ITable actualTable)
- static void assertEquals(ITable expectedTable, ITable actualTable, Column[] additionalColumnInfo)
- static void assertEquals(ITable expectedTable, ITable actualTable, FailureHandler failureHandler)
- static void assertEqualsByQuery(IDataSet expectedDataset, IDatabaseConnection connection, String sqlQuery, String tableName, String[] ignoreCols)
- static void assertEqualsByQuery(ITable expectedTable, IDatabaseConnection connection, String tableName, String sqlQuery, String[] ignoreCols)
- static void assertEqualsIgnoreCols(IDataSet expectedDataset, IDataSet actualDataset, String tableName, String[] ignoreCols)
- static void assertEqualsIgnoreCols(ITable expectedTable, ITable actualTable, String[] ignoreCols)
- static void assertWithValueComparer(IDataSet expectedDataSet, IDataSet actualDataSet, FailureHandler failureHandler, ValueComparer defaultValueComparer, Map<String,Map<String,ValueComparer>> tableColumnValueComparers)
- static void assertWithValueComparer(IDataSet expectedDataSet, IDataSet actualDataSet, ValueComparer defaultValueComparer, Map<String,Map<String,ValueComparer>> tableColumnValueComparers)
- static void assertWithValueComparer(ITable expectedTable, ITable actualTable, Column[] additionalColumnInfo, ValueComparer defaultValueComparer, Map<String,ValueComparer> columnValueComparers)
- static void assertWithValueComparer(ITable expectedTable, ITable actualTable, FailureHandler failureHandler, ValueComparer defaultValueComparer, Map<String,ValueComparer> columnValueComparers)
- static void assertWithValueComparer(ITable expectedTable, ITable actualTable, ValueComparer defaultValueComparer, Map<String,ValueComparer> columnValueComparers)

数据层DBUnit的测试

```
package com.jerry;
...
public class UserDaoJdbcImplTest extends AbstractDbUnitTestCase{

 @Test
 public void testGetUserById() throws Exception { //验证通过ID获得用户
  IDataSet setupDataSet = getDataSet("test/com/jerry/user.xml"); //test/com/jerry/user.xml为XML文件的位置
  DatabaseOperation.CLEAN_INSERT.execute(dbunitConnection, setupDataSet);
  User user = dao.getUserById(1); //调用getUserById方法
  assertNotNull( user); //验证获得的数据不为空
  assertEquals( "Jeffrey", user.getFirstName() ); //验证FirstName
  assertEquals( "Lebowsky", user.getLastName() ); //验证LastName
  assertEquals( "ElDuderino", user.getUsername() ); //验证Username
 }

 @Test
 public void testAddUseIgnoringId() throws Exception {//验证添加用户忽略ID
  IDataSet setupDataSet = getDataSet("test/com/jerry/user.xml");  //test/com/jerry/user.xml为XML文件的位置
  DatabaseOperation.DELETE_ALL.execute(dbunitConnection, setupDataSet);
  User user = newUser();  //调用newUser方法
```

参数为“DbUnit数据库连接”和“XML文件内容”

参数为“DbUnit数据库连接”和“XML文件内容”

数据层DBUnit的测试

```
    int id = dao.addUser(user); //调用addUser方法
    assertTrue(id>0); //验证id是否大于0
    IDataSet expectedDataSet = getDataSet("test/com/jerry/user.xml"); //test/com/jerry/user.xml为XML文件，内容为期望值
    IDataSet actualDataSet = dbunitConnection.createDataSet(); //从数据库里获得数据
    Assertion.assertEqualsIgnoreCols( expectedDataSet, actualDataSet, "users", new String[] { "id" } ); //比对两个数据集忽略user的id列
  }
  @Test
  public void testGetUserByIdReplacingIds() throws Exception { //验证通过ID获得用户，替换XML中的id
    long id = 42;
    IDataSet setupDataset = getReplacedDataSet("test/com/jerry/user-token.xml", id );
    DatabaseOperation.INSERT.execute(dbunitConnection, setupDataset);
    User user = dao.getUserById(id); //调用getUserById()方法
    assertUser(user); //验证用户
  }
  @Test
  public void testAddUserReplacingIds() throws Exception {//验证添加用户，替换XML中的id
    IDataSet setupDataSet = getDataSet("test/com/jerry/user-token.xml");
    DatabaseOperation.DELETE_ALL.execute(dbunitConnection, setupDataSet);
```

数据层DBUnit的测试

```
  User user = newUser();
   int id = dao.addUser(user); //调用方法addUser()
   assertTrue(id>0); //验证id是否大于0
   IDataSet expectedDataSet = getReplacedDataSet(setupDataSet, id ); //替换id
   IDataSet actualDataSet = dbunitConnection.createDataSet(); //从数据库里获得数据
   Assertion.assertEquals( expectedDataSet, actualDataSet ); //比对两个数据集
 }

}
```

数据层DBUnit的测试

user-token.xml

```
<?xml version="1.0"?>
<dataset>
  <users id="[ID]" username="ElDuderino" first_name="Jeffrey" last_name="Lebowsky" />
</dataset>
```

user.xml

```
?xml version="1.0"?>
<dataset>
  <users id="1" username="ElDuderino" first_name="Jeffrey" last_name="Lebowsky" />
</dataset>
```

持续集成单元测试方法与应用

- 测试用例的设计
- 单元测试JUnit框架特性及使用
- 使用Stub进行测试
- Mock技术
- 容器内的测试
- 测试用例执行及数据分析统计
- 持续集成自动化回归测试

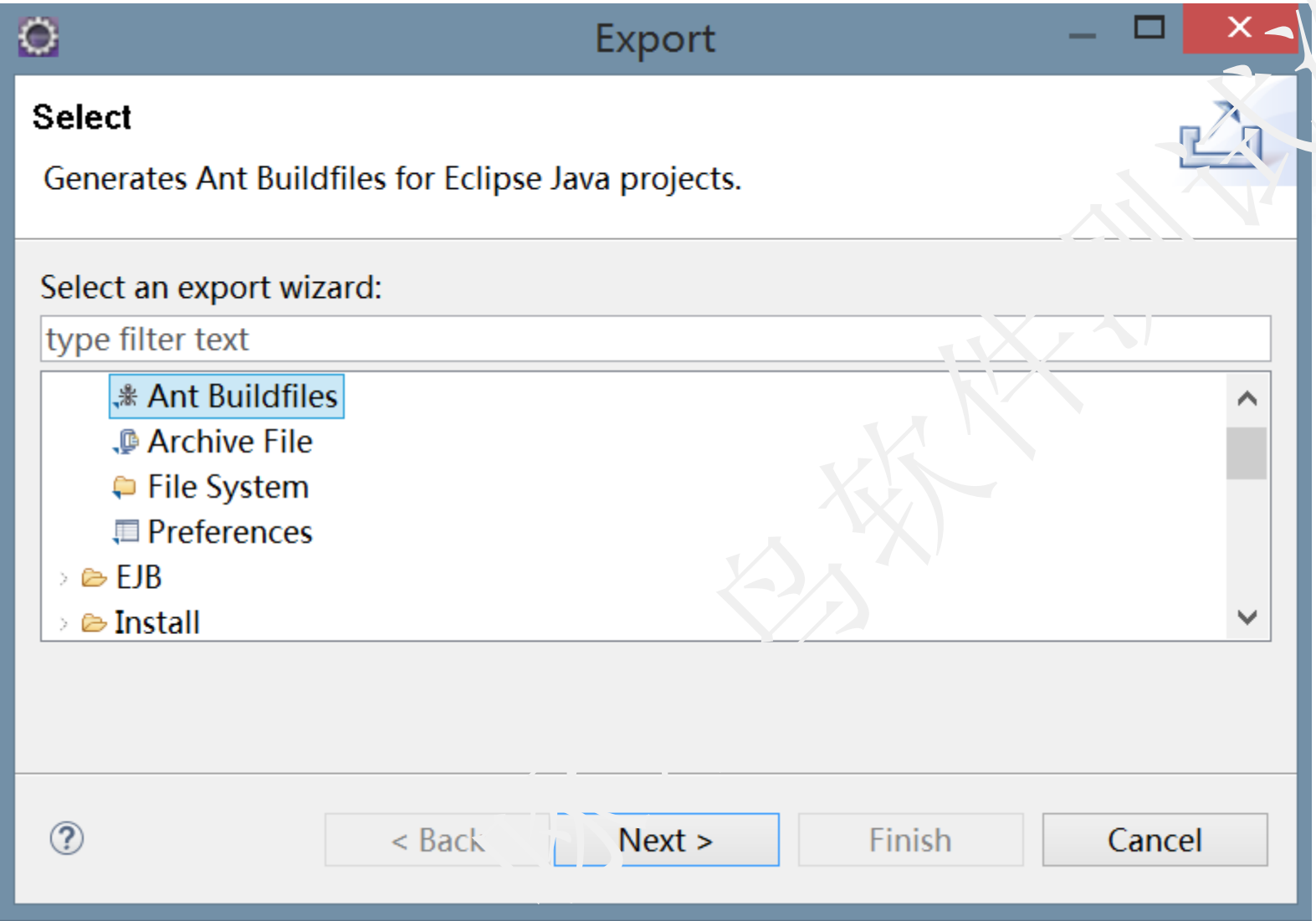
测试用例执行及数据分析统计

测试用例执行

数据分析统计

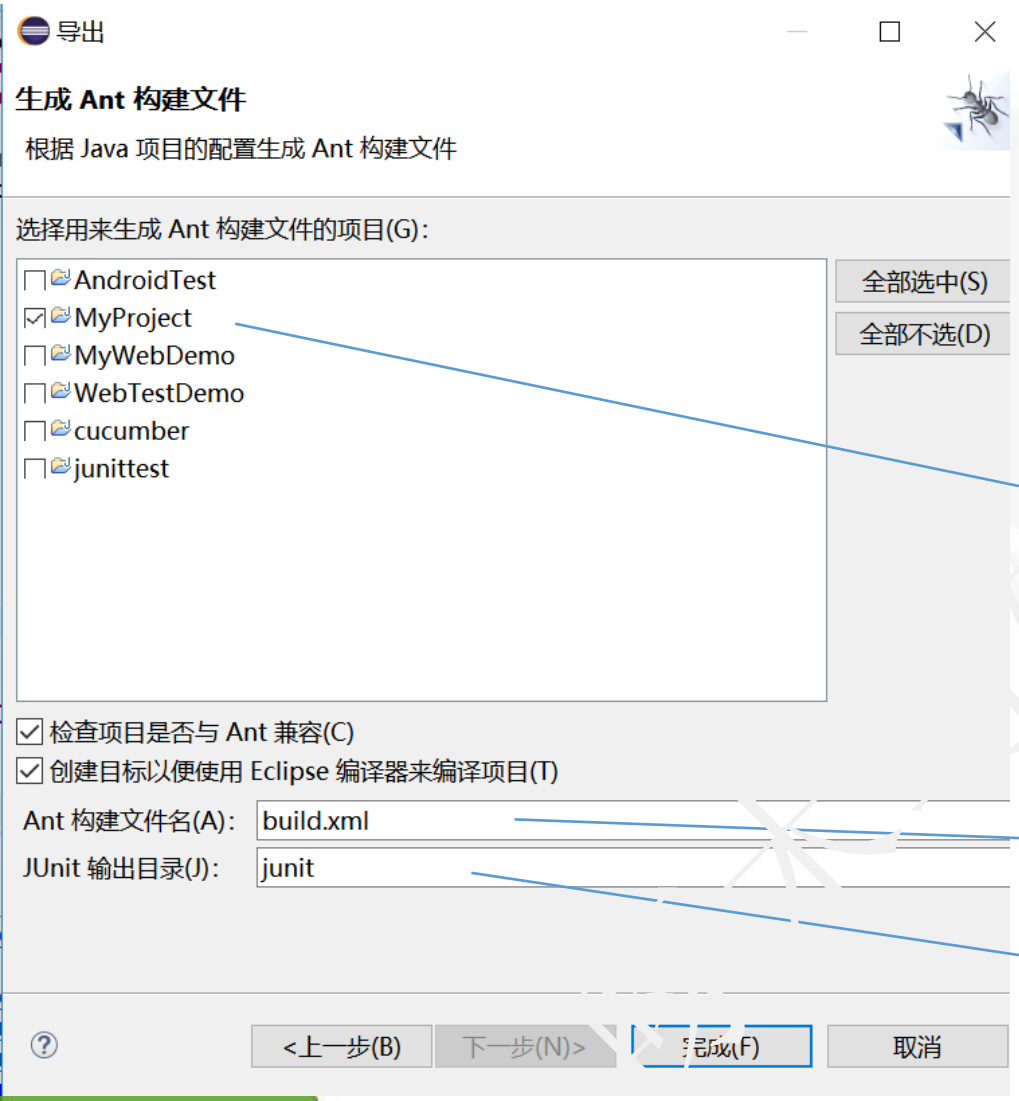
测试用例执行

在项目上右击鼠标->导出，选择Ant构建文件，点【Next】



测试用例执行

选择所需要的生成报告的project，然后点击【完成】



生成报告的project

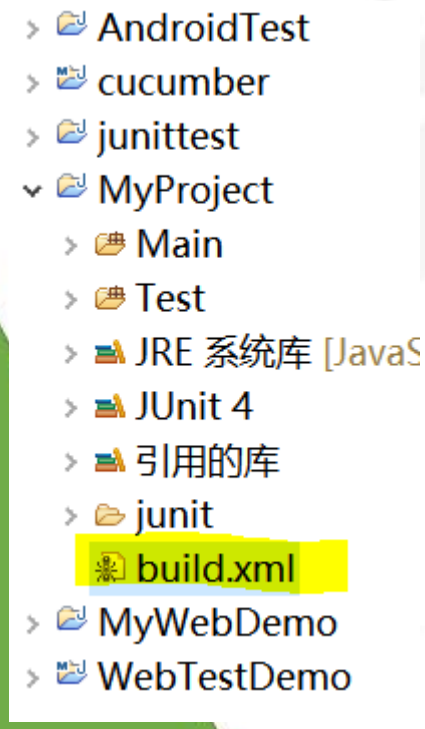
是生成ant文件的文件名

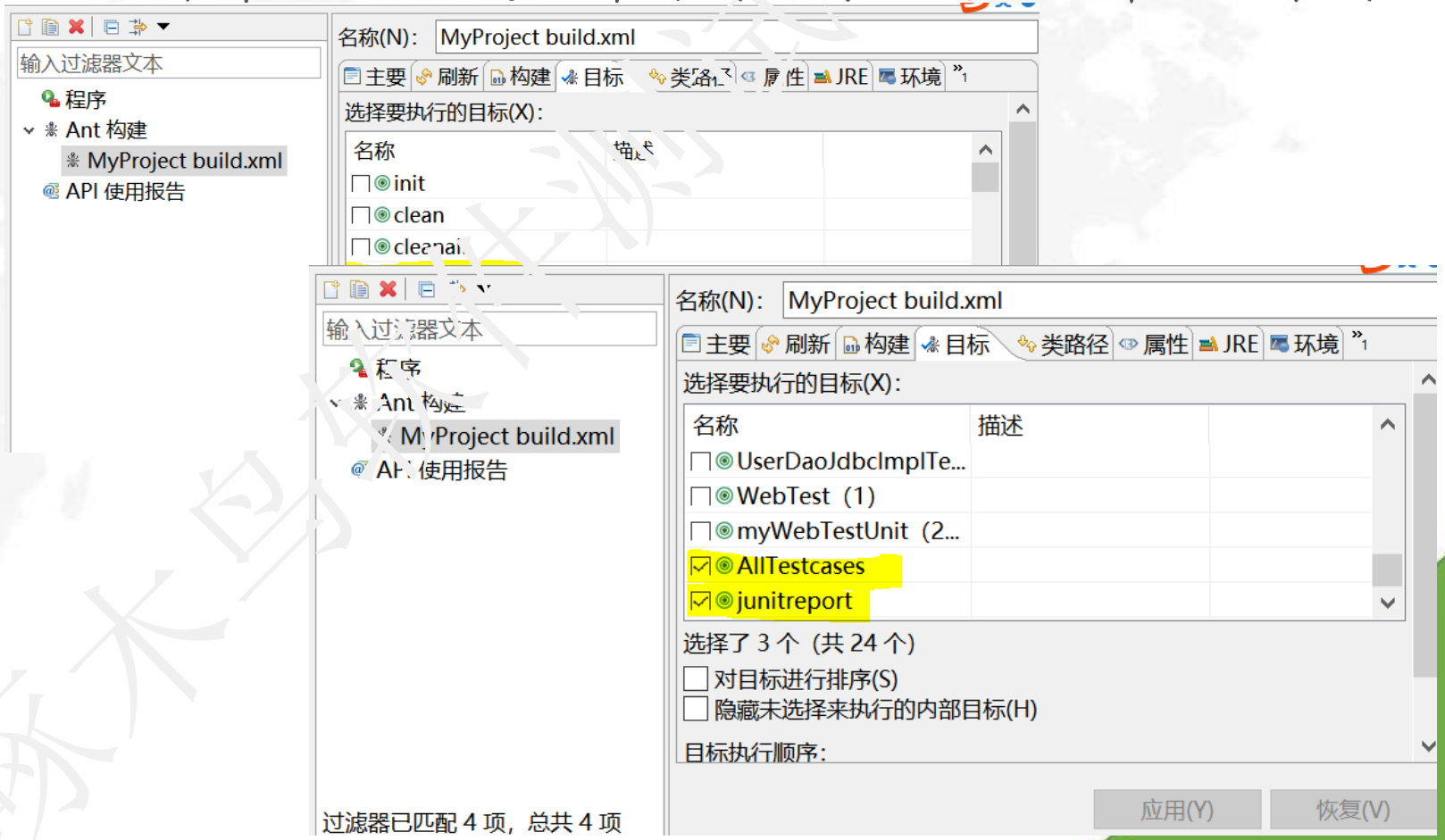
是生成测试报告的目录（需要建立以后手工建立）

测试用例执行

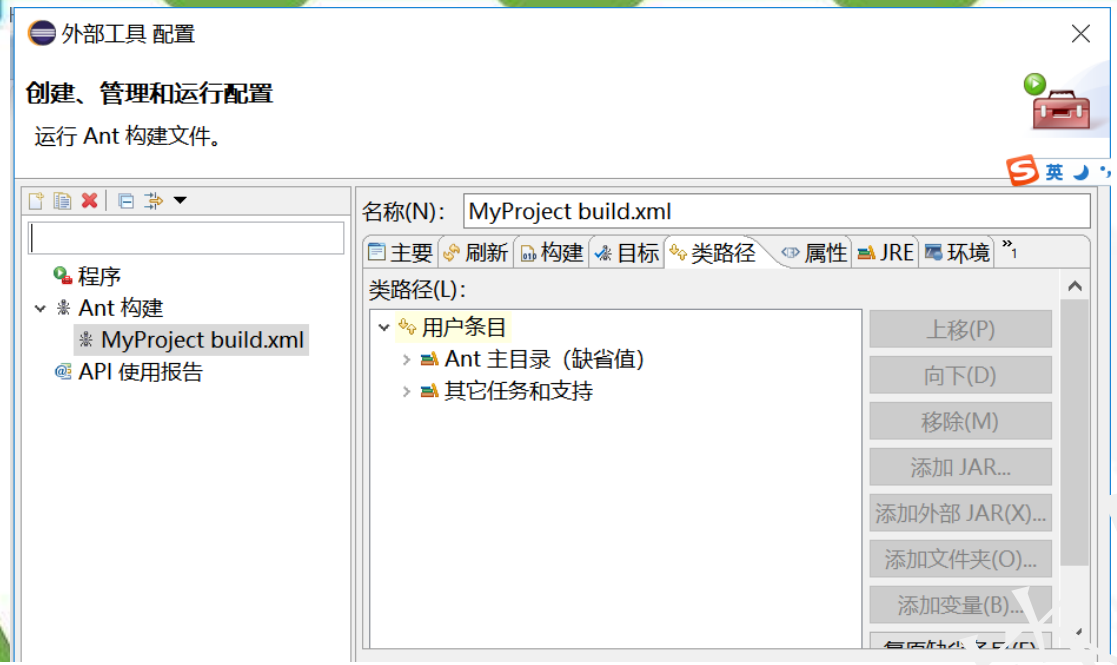
将在项目内生成build.xml（默认）文件和junit文件夹，如果junit文件夹不能自动生成可以手动生成。

右击build.xml，选择Run as->外部工具 配置,选择需要运行的测试文件。如果要生成测试报告选择junitreport

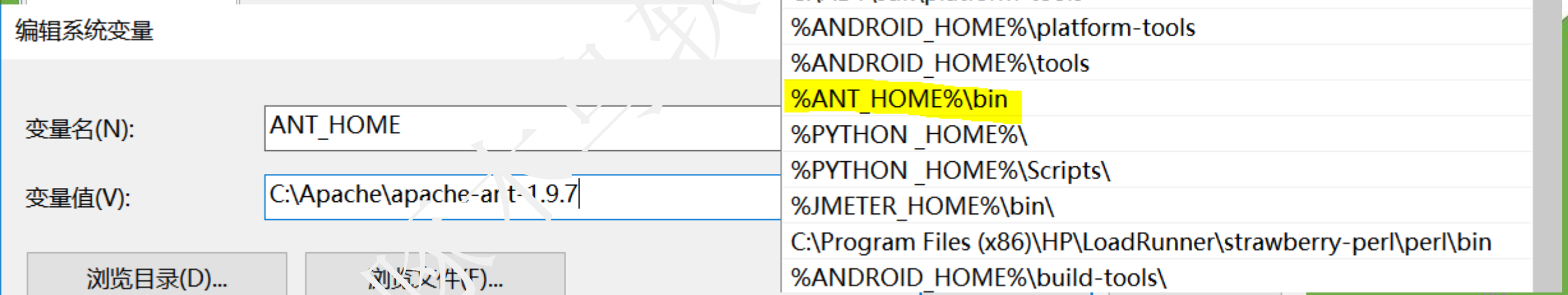




测试用例执行



检查类路径中Ant主目录的Ant版本是否大于1.9.7，若不是，通过Ant主目录（H）选择一个大于1.9.7版本的Ant工具（可以到网上下载）



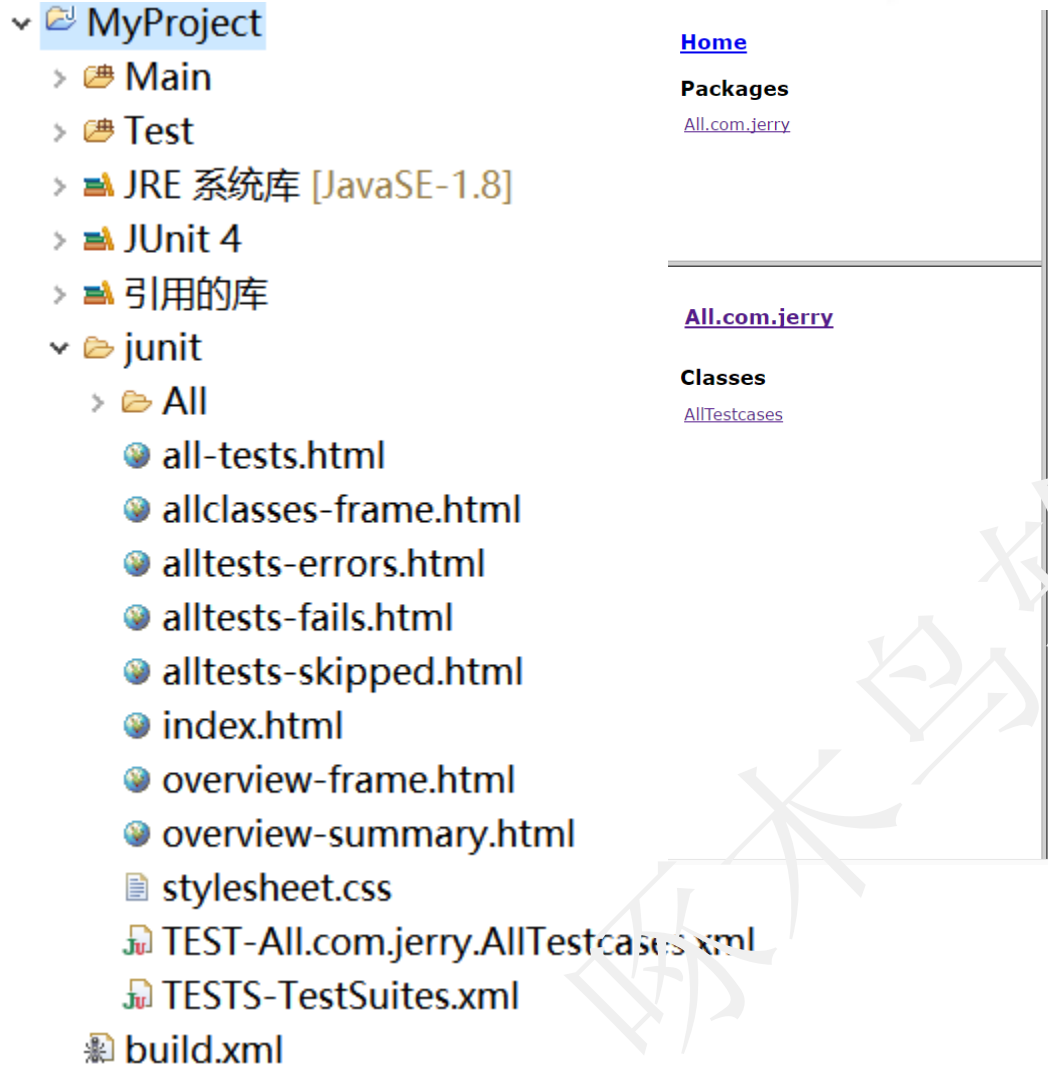
测试用例执行及数据分析统计

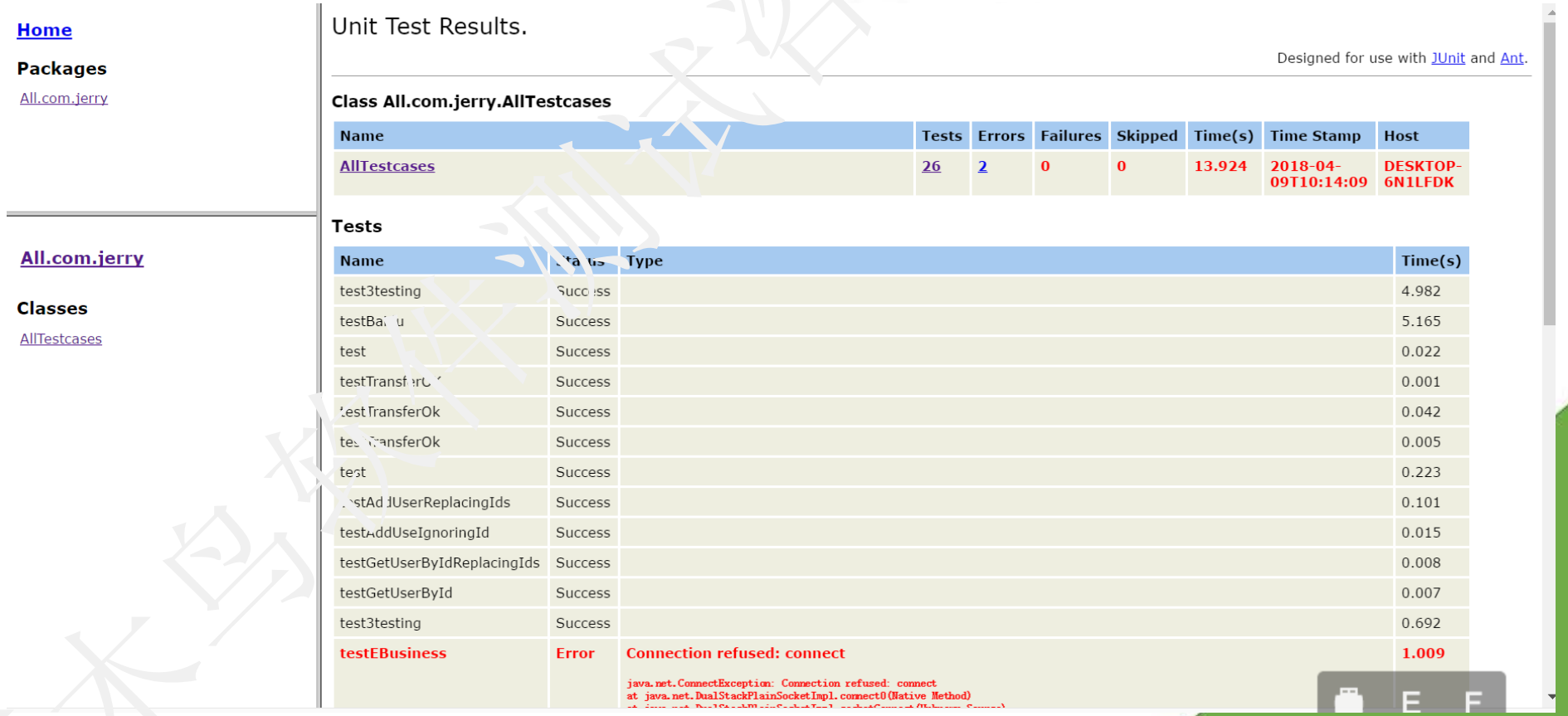
测试用例执行

数据分析统计

数据分析统计

点完成，开始运行Junit测试用例，在并且产生index.html测试报告在junit目录下。





数据分析统计

实现这个功能需要下载：javax.mail.jar ,org.apache.commons.lang_2.4.0.v201005080502.jar和log4j-1.2.17.jar

```
import javax.activation.DataHandler;
import javax.activation.FileDataSource;
import javax.mail.AuthenticationFailedException;
import javax.mail.Authenticator;
…
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeBodyPart;
import javax.mail.internet.MimeMessage;
import javax.mail.internet.MimeMultipart;
import javax.mail.internet.MimeUtility;

import org.apache.commons.lang.StringUtils;
import org.apache.log4j.LogManager;
import org.apache.log4j.Logger;
```

数据分析统计

```
/**
 * 实现邮件发送功能
 * @author Administrator
 **/
public class EmailSender {
        private static final Logger logger = LogManager.getLogger(EmailSender.class);
        private String host; // 服务器地址
        private String from; // 发件人
        private String to; // 收件人 多个收件人以,分隔
        private String title; // 主题
        private String content; // 内容
        private List<File> attachmentlist ; //附件集
        private String username; // 用户名
        private String password; // 密码
        private String sendEmployeeId;//发件人员工编号
        public String getSendEmployeeId() {
                return sendEmployeeId;}
```

数据分析统计

```
public void setSendEmployeeId(String sendEmployeeId) {
        this.sendEmployeeId = sendEmployeeId;
}
public String getHost() {
        return host;
}
public void setHost(String host) {
        this.host = host;
}
public String getFrom() {
        return from;
}
public void setFrom(String from) {
        this.from = from;
}
public String getTo() {
        return to;
}
public void setTo(String to) {
        this.to = to;
```

数据分析统计

```
public String getTitle() {
        return title;
}
public void setTitle(String title) {
        this.title = title;
}
public String getContent() {
        return content;
}
public void setContent(String content) {
        this.content = content;
}
public List<File> getAttachmentlist() {
        return attachmentlist;
}
public void setAttachmentlist(List<File> attachmentlist) {
        this.attachmentlist = attachmentlist;
}
public String getUsername() {
        return username;}
```

数据分析统计

```
public String getTitle() {
        return title;
}
public void setTitle(String title) {
        this.title = title;
}
public String getContent() {
        return content;
}
public void setContent(String content) {
        this.content = content;
}
public List<File> getAttachmentlist() {
        return attachmentlist;
}
public void setAttachmentlist(List<File> attachmentlist) {
        this.attachmentlist = attachmentlist;
}
public String getUsername() {
        return username;}
```

数据分析统计

```
public void setUsername(String username) {
        this.username = username;
}
public String getPassword() {
        return password;
}
public void setPassword(String password) {
        this.password = password;
}
public String getPort() {
        return port;
}
public void setPort(String port) {
        this.port = port;
}

private String port;
```

数据分析统计

```
public void setUsername(String username) {
        this.username = username;
}
public String getPassword() {
        return password;
}
public void setPassword(String password) {
        this.password = password;
}
public String getPort() {
        return port;
}
public void setPort(String port) {
        this.port = port;
}

private String port;
```

```
        public EmailSender(String host, String from, String to, String title,
                String content, List attachmentlist, String username, String password,String port) {
        this.host = host;
        this.from = from;
        this.to = to;
        this.title = title;
        this.content = content;
        this.attachmentlist = attachmentlist;
        this.username = username;
        this.password = password;
        this.port=port;
}

public EmailSender(String to, String title,
                String content, List attachmentlist) {
        this.to = to;
        this.title = title;
        this.content = content;
        this.attachmentlist = attachmentlist;
}
```

数据分析统计

```
/**
 * 发送邮件
 * @return 发送状态信息 index0：状态 0成功 1失败;index1：描述错误信息
 */
public String[] sendMail(){
        String[] result=new String[2];

        Session session=null;
        Properties props = System.getProperties();
        props.put("mail.smtp.host", host);
        props.put("mail.smtp.sendpartial", "true");
        props.put("mail.smtp.port", port);

        if(StringUtils.isBlank(username)){//不需要验证用户名密码
                session = Session.getDefaultInstance(props, null);
        }else{
                props.put("mail.smtp.auth", "true");
                EmailAuthenticator auth = new EmailAuthenticator(username, password);
                session = Session.getInstance(props, auth);
        }
```

数据分析统计

```
//设置邮件发送信息
try{
        // 创建邮件
        MimeMessage message = new MimeMessage(session);
        // 设置发件人地址
        message.setFrom(new InternetAddress(from));
        // 设置收件人地址（多个邮件地址）
        InternetAddress[] toAddr = InternetAddress.parse(to);
        message.addRecipients(Message.RecipientType.TO, toAddr);
        // 设置邮件主题
        message.setSubject(title);
        // 设置发送时间
        message.setSentDate(new Date());
        // 设置发送内容
        Multipart multipart = new MimeMultipart();
        MimeBodyPart contentPart = new MimeBodyPart();
        contentPart.setText(content);
        multipart.addBodyPart(contentPart);
        //设置附件
        if(attachmentlist!=null && attachmentlist.size()>0){
```

数据分析统计

```
                for(int i = 0 ; i < attachmentlist.size();i++){
                        MimeBodyPart attachmentPart = new MimeBodyPart();

                        FileDataSource source = new FileDataSource(attachmentlist.get(i));
                        attachmentPart.setDataHandler(new DataHandler(source));

attachmentPart.setFileName(MimeUtility.encodeWord(attachmentlist.get(i).getName(), "gb2312", null));
                        multipart addBodyPart(attachmentPart);
                }
        }
        message.setContent(multipart),

        //登录SMTP服务器
        if (StringUtils.isBlank(username)) {
                // 不需验证
                Transport.send(message);
        } else {
                // 需要验证
                Transport transport = session.getTransport("smtp");
                transport.connect();
```

数据分析统计

```
                transport.sendMessage(message, message.getAllRecipients());
                transport.close();
        }

        result[0]="0";
        result[1]="发送成功";

        logger.info("邮件发送成功!发送人："+from);

}catch(MessagingException mex){
        result[0]="1";
        result[1]="邮件服务器发生错误";

        if(mex instanceof AuthenticationFailedException){
                result[1]="用户名或密码错误";
        }
} catch (Exception e) {
        result[0]="1";
        result[1]="系统异常";
}
```

数据分析统计

```
                return result;}
        public static void main(String[] args){
                List list=new ArrayList();
                list.add(new File("C:\\myjava\\JUnit\\MyProject\\junit\\junit.rar"));
                EmailSender sender=new EmailSender("smtp.126.com","xianggu625@126.com","xianggu625@126.com","
发送测试报告","附件为 测试报告",list,"xianggu625@126.com","123456","25");
                String [] result = sender.sendMail();
                System.out.println(result[1]+"ffffffffffffffff");
        }}
/**
 * class MyAuthenticator用于邮件服务器认证 构造器需要用户名、密码作参数
 */
class EmailAuthenticator extends Authenticator {
        private String username = null;
        private String password = null;
        public EmailAuthenticator(String username, String password) {
                this.username = username;
                this.password = password;}
        public PasswordAuthentication getPasswordAuthentication() {
                return new PasswordAuthentication(username, password);}}
```

练习

练习使用Java发送Mail

持续集成单元测试方法与应用

- 测试用例的设计
- 单元测试JUnit框架特性及使用
- 使用Stub进行测试
- Mock技术
- 容器内的测试
- 测试用例执行及数据分析统计
- 持续集成自动化回归测试

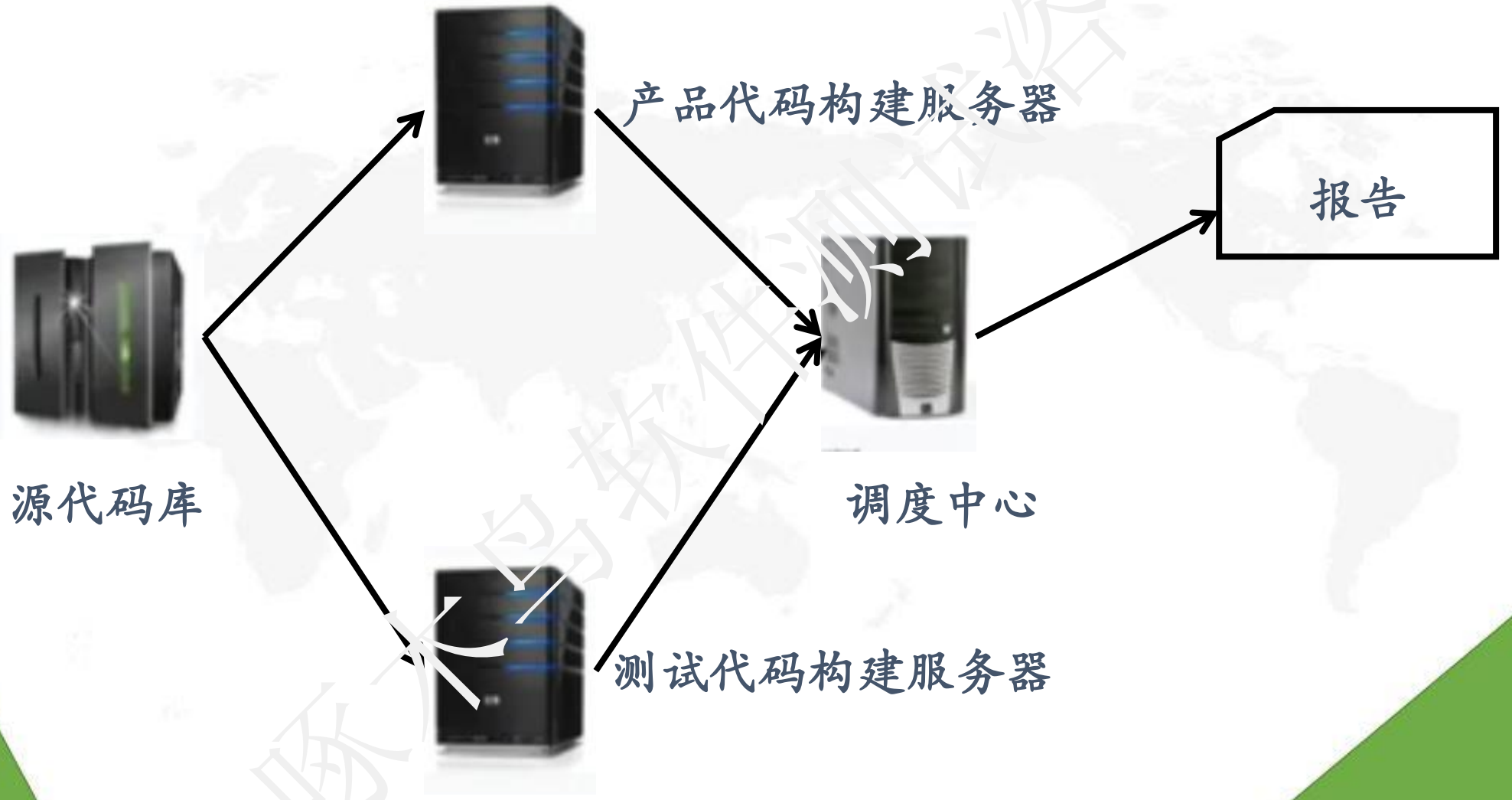
持续集成自动化回归测试

持续集成定义:

持续集成是一种软件开发实践，即团队开发成员经常集成他们的工作，通常每个成员每天至少集成一次，也就意味着每天可能会发生多次集成。每次集成都通过自动化的构建（包括编译、发布、自动化测试）来验证，从而尽快地发现集成错误。许多团队发现这个过程可以大大减少集成的问题，让团队能够更快地开发内聚的软件。

——Martin Fowler

持续集成自动化回归测试



持续集成自动化回归测试

安装

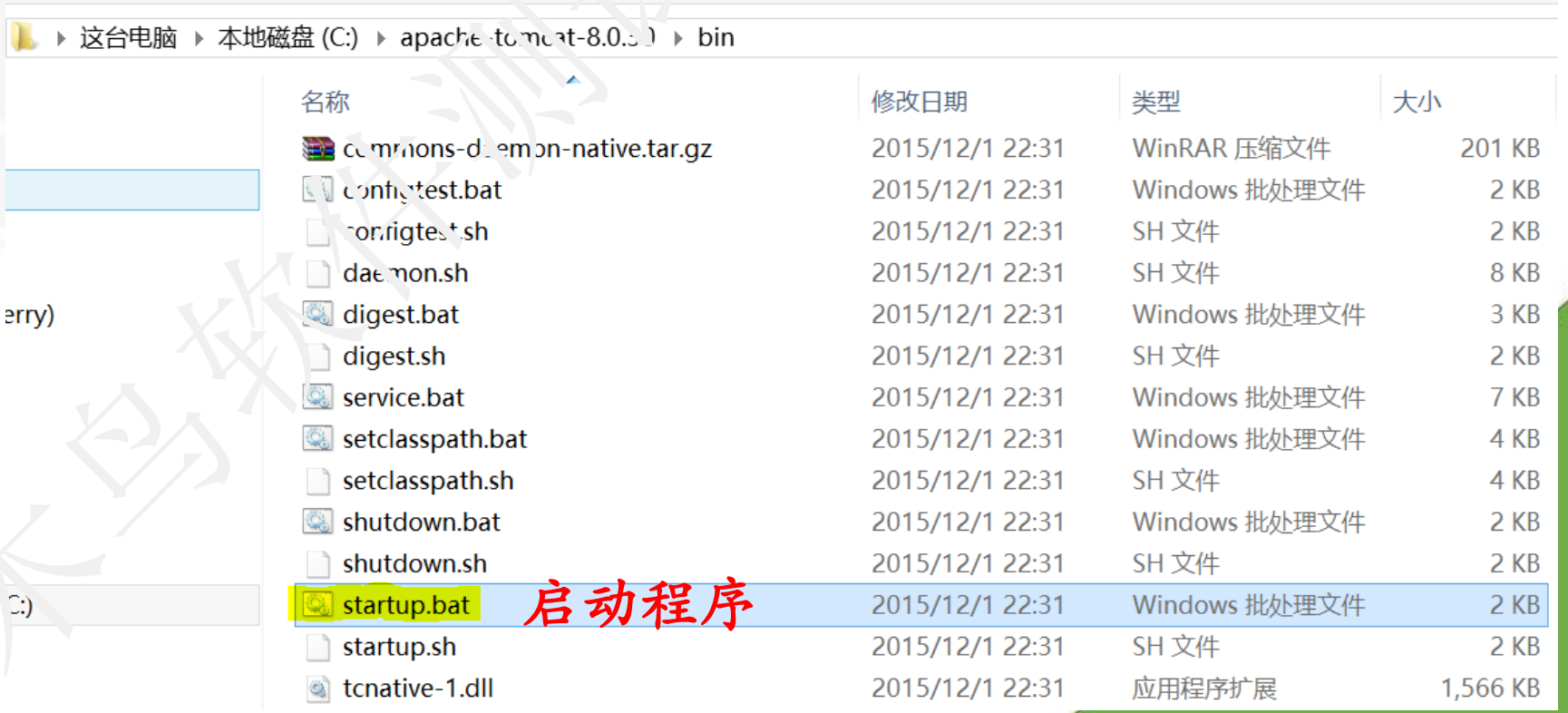
1、安装JDK

2、安装TomCat

http://tomcat.apache.org/

Documentation

- Tomcat 9.0
- Tomcat 8.5
- Tomcat 8.0
- Tomcat 7.0
- Tomcat Connectors
- Tomcat Native
- Wiki
- Migration Guide
- Presentations



持续集成自动化回归测试

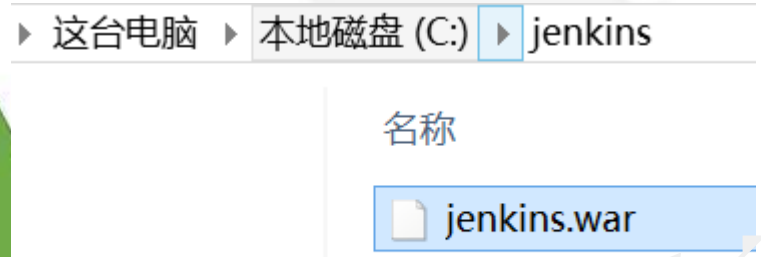
安装

3、安装Jenkins

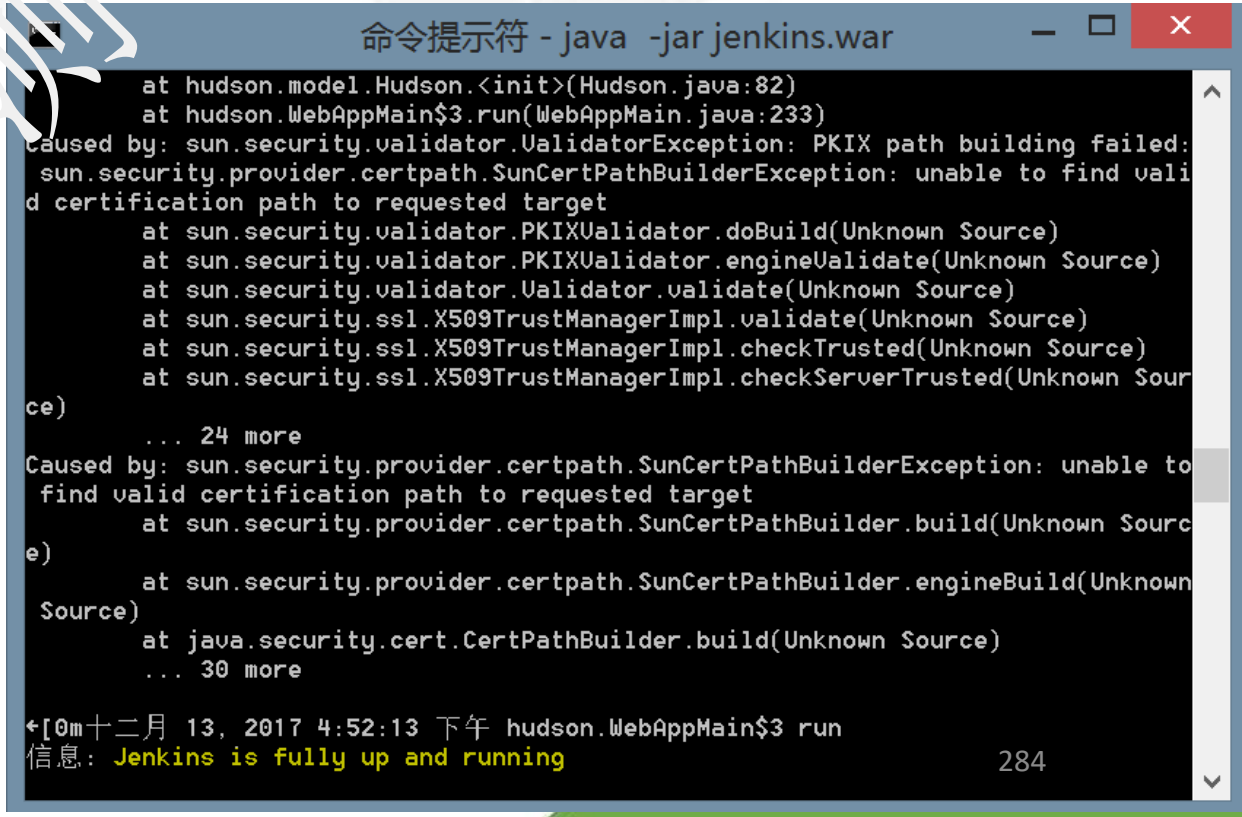
http://jenkins-ci.org/

下载.war文件

4、建立Jenkins路径，将jenkins.war考入其中



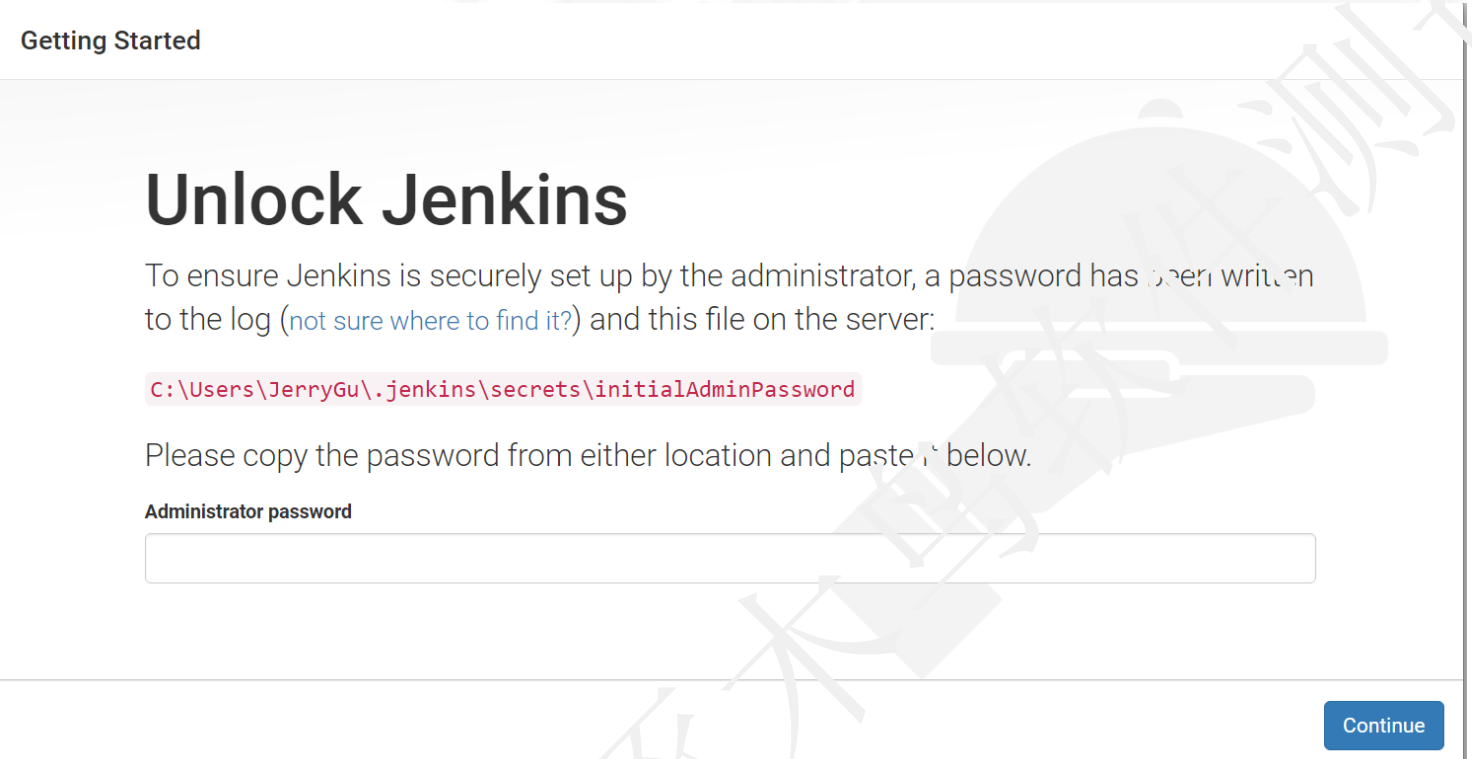
5、打开命令框，输入java -jar jenkins.war --ajp13Port=-1 --httpPort=8081，出现如下提示，则Jenkins就运行了



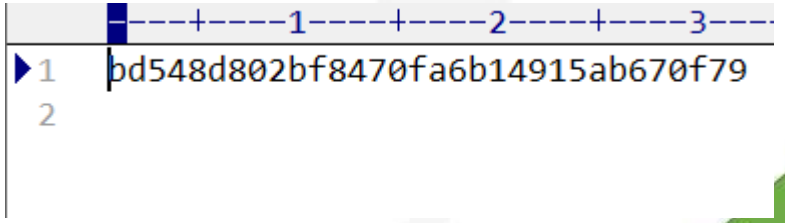
持续集成自动化回归测试

安装

6、打开浏览器，输入localhost:8081



7、到C:\Users\JerryGu\.jenkins\secrets\initialAdminPassword找到密码，输入



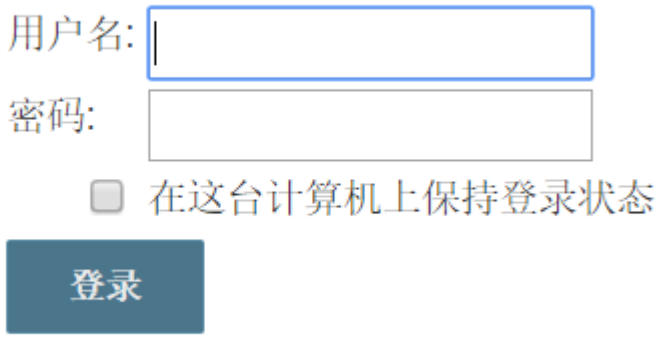
8、点【Continue】

9、重新启动服务，java -jar jenkins.war

持续集成自动化回归测试

安装

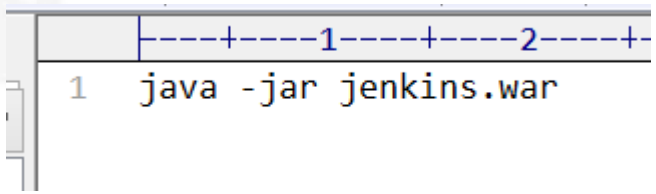
10、打开浏览器，输入localhost:8081



用户名为admin，密码为第7步的密码

11、修改密码

点“用户”>“admin”->“设置”，在密码处修改密码



12、在C:\jenkins处建立一个jenkins.bat文件，为了以后方便，可以在桌面建立快捷键

持续集成自动化回归测试

创建新任务

持续集成自动化回归测试

创建新任务

General | 源码管理 | 构建触发器 | 构建 | 构建后操作

项目名称 WebDriver

描述 这是一个WebDriver的自动化测试项目

[Plain text] 预览

丢弃旧的构建

参数化构建过程

关闭构建

在必要的时候并发构建

高级...

主要用来配置构建历史保存的几个版本，每次执行构建都会产生日志，日志的生成会占用一定的磁盘空间，通过勾选此选项就可以设置保持构建天数和构建次数。

持续集成自动化回归测试

创建新任务

高级选项

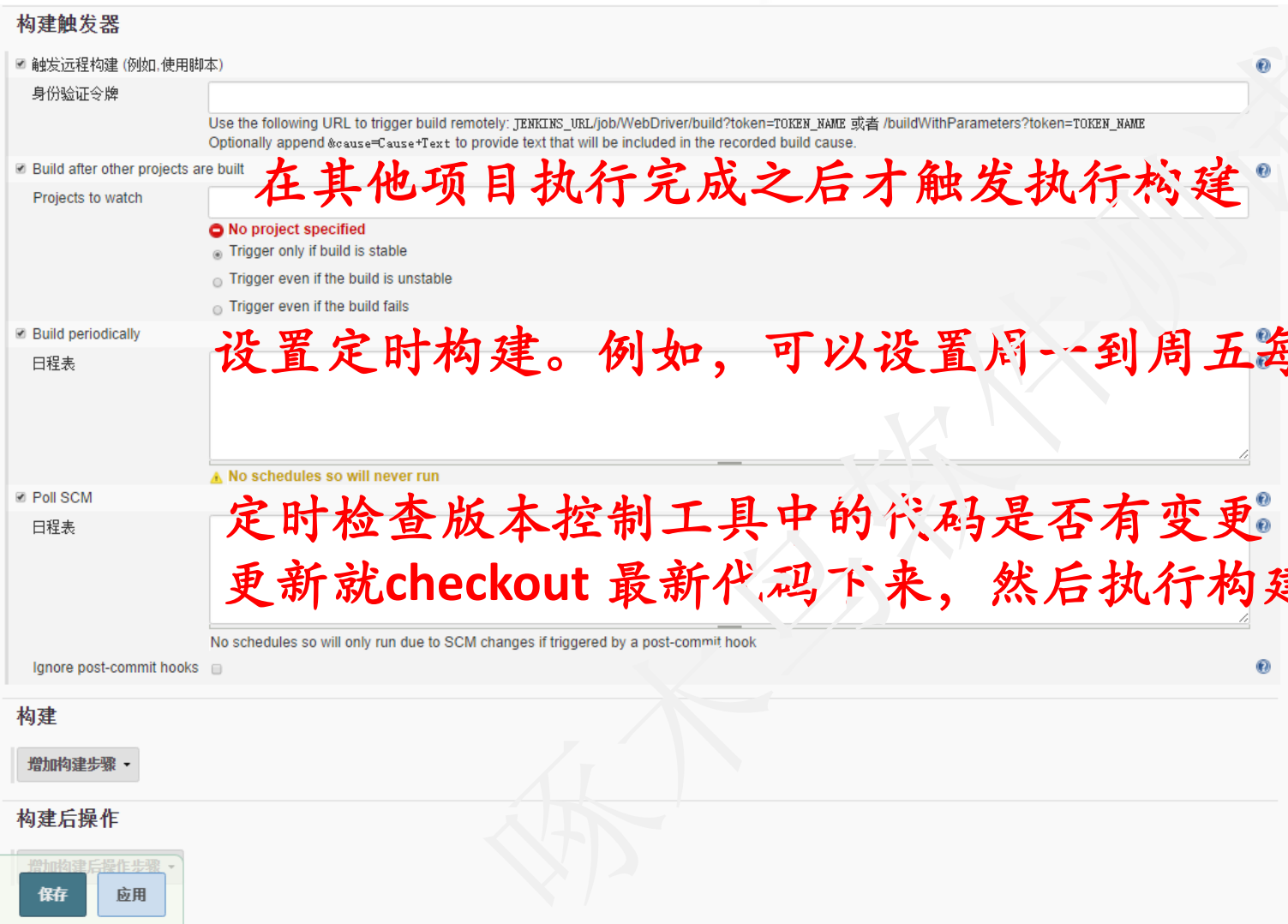
- [ ] 安静期 如果设置此选项，则一个计划中的构建在开始之前需要等待选项中设置的秒数
- [ ] 重试次数 如果从版本库签出代码失败，则Jenkins会按照这个指定的次数进行重试之后再放弃。
- [ ] 该项目的上游项目正在构建时阻止该项目构建
- [ ] 该项目的下游项目正在构建时阻止该项目构建
- [ ] 使用自定义的工作空间

源码管理 源码管理

- (•) None
- ( ) CVS
- ( ) CVS Projectset
- ( ) Subversion

持续集成自动化回归测试

创建新任务　构建触发器



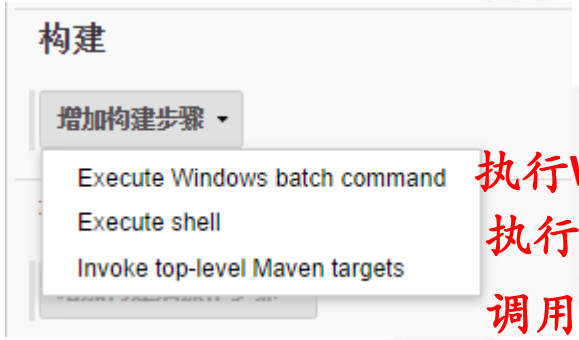
在其他项目执行完成之后才触发执行构建

设置定时构建。例如，可以设置周一到周五每天晚上12 点处理构建

定时检查版本控制工具中的代码是否有变更（根据SCM 软件的版本号），如果有更新就checkout 最新代码下来，然后执行构建动作

持续集成自动化回归测试

创建新任务　构建



执行Windows 批处理命令

执行shell 脚本

调用Maven对象

持续集成自动化回归测试

运行任务

持续集成自动化回归测试

查看结果

Jenkins ▸ WebDriver ▸ #3

- 返回到工程
- 状态集
- 变更记录
- Console Output
- 编辑编译信息
- 删除本次生成
- 前一次构建

控制台输出

```
Started by user admin
Building in workspace C:\Users\JerryGu\.jenkins\workspace\WebDriver
[WebDriver] $ cmd /c call C:\Users\JerryGu\AppData\Local\Temp\jenkins1049566955008831463.bat

C:\Users\JerryGu\.jenkins\workspace\WebDriver>python C:\jenkins\test.py
Hello Jenkins

C:\Users\JerryGu\.jenkins\workspace\WebDriver>exit 0
Finished: SUCCESS
```

持续集成自动化回归测试

构建历史

Jenkins ▸ WebDriver ▸

- 返回面板
- 状态
- 修改记录
- 工作空间
- 立即构建
- 删除 Project
- 配置
- Polling Log

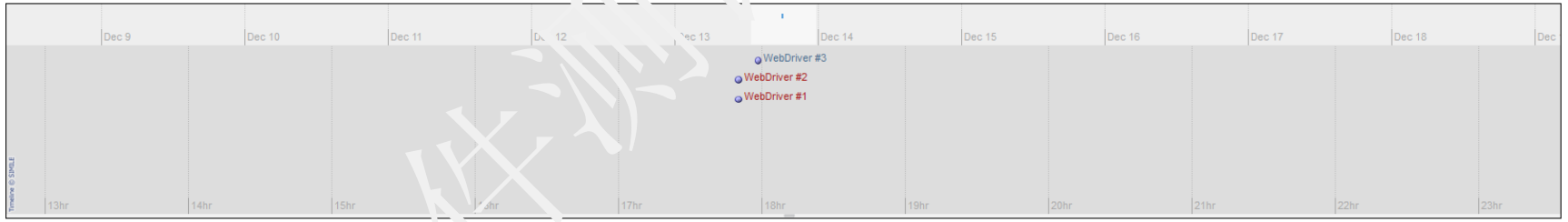
Build History 构建历史

find

#3	2017-12-13 下午5:58
#2	2017-12-13 下午5:50
#1	2017-12-13 下午5:50

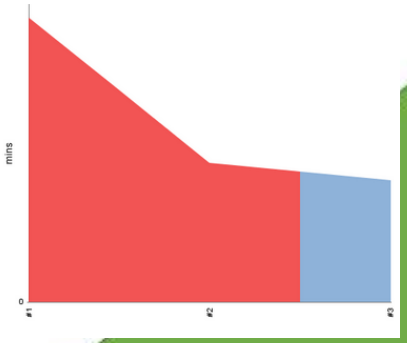
RSS 全部 RSS 失败

时间轴



构建时间趋势

构建	持续时间
#3	0.35 秒
#2	0.4 秒
#1	0.82 秒



持续集成自动化回归测试

定时构建

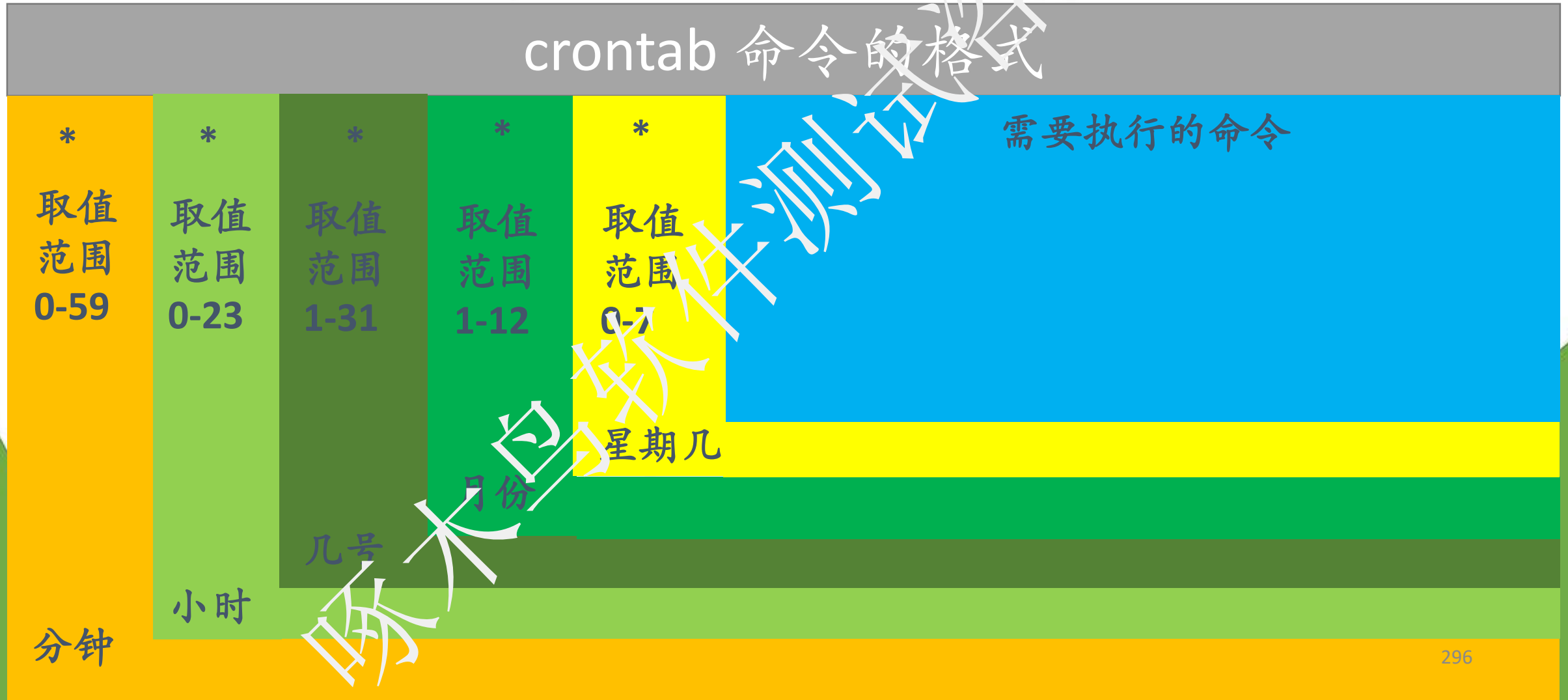
可以使用java 发送邮件，也可以使用其他程序的发邮件功能

遵循cron 的语法，每行包含五个字段，通过Tab 或空格分隔

字 段	说 明
MINUTE	Minutes within the hour (0–59)
HOUR	The hour of the day (0–23)
DOM	The day of the month (1–31)
MONTH	The month (1–12)
DOW	The day of the week (0–7)where 0 and 7 are Sunday

持续集成自动化回归测试

定时构建



持续集成自动化回归测试

定时构建

- 星号（*）：代表所有可能的值。例如，month 字段如果是星号，则表示在满足其他字段的制约条件后每月都执行该命令操作。
- 逗号（,）：可以用逗号隔开的值指定一个列表范围。例如，“1,2,5,7,8,9”。
- 中杠（-）：可以用整数之间的中杠表示一个整数范围。例如，“2-6”表示“2,3,4,5,6”。
- 正斜线（/）：可以用正斜线指定时间的间隔频率。例如，“0-23/2”表示每两小时执行一次。同时正斜线可以和星号一起使用，例如，*/10，如果用在minute 字段，表示每十分钟执行一次。

持续集成自动化回归测试

定时构建-案例

- 10 22 * * 1-5

表示每周一至周五的晚上22:10运行自动化测试用例

- 每周一和周三下午6 点半运行自动化测试用例

30 18 * * 1,3

- 周一至周五每天早上4 点跑自动化测试用例

* 4 * * 1-5

持续集成自动化回归测试

Python的发送邮件功能程序 Sendemail.py

```
#!/usr/bin/env python
#coding:utf-8
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart

#发送邮箱服务器
smtpserver = 'smtp.126.com'
#发送邮箱
sender = 'xianggu625@126.com'
#接受邮箱
receiver = 'xianggu625@126.com'
#发送邮箱用户名、密码
username = 'xianggu625@126.com'
password = 'knyzh@625'
#邮件主题
subject = 'Python send email'
#发送的附件
sendfile = open('C:\\knyjava\\JUnit\\MyProject\\junit.rar','rb').read()

att = MIMEText(sendfile,'base64','utf-8')
att["Content-Type"] = 'application/octest-stream'
att["Content-Disposition"] = 'attachment; filename="junit.rar"'

msgRoot = MIMEMultipart('related')
msgRoot['Subject'] = subject
msgRoot.attach(att)

smtp = smtplib.SMTP()
smtp.connect(smtpserver)
smtp.login(username,password)
smtp.sendmail(sender,receiver,msgRoot.as_string())
smtp.quit()
```

持续集成自动化回归测试

案例

Execute Windows batch command

命令

```
cd C:\myjava\JUnit\MyProject
ant AllTestcases
```

参阅 可用环境变量列表

高级...

Execute Windows batch command

命令

```
cd C:\Program Files (x86)\WinRAR
Rar a C:\myjava\JUnit\MyProject\junit C:\myjava\JUnit\MyProject\junit
python C:\python\WebDriver\DataDriver\SendMail.py
del C:\myjava\JUnit\MyProject\junit.rar
```

参阅 可用环境变量列表

练习

搭建jekins

结束

Thank You

顾翔

啄木鸟软件测试培训网

QQ：2025344

微信号：xianggu0625

Email：xianggu625@126.com

网站：www.3testing.com

微信公众号：见二维码