# Junit

Junit是Java的单元测试框架。

单元测试不是为了证明你是对的，而是为了证明你没有错误。

单元测试主要用来判断程序的执行结果和你所期望的结果是否一致。

## Junit3.8版本

测试方法必须需要满足一下条件：

1. 方法的修饰符为public。
2. 方法的返回值为void。
3. 无方法参数。
4. 方法名必须以test开头。

测试类必须继承TestCase父类。

TestCase：（1）实现一个TestCase的一个子类

（2）定义一个实例变量用来存储测试类的状态。

（3）重写setUp()方法用来初始化

（4）重写tearDown()方法用来清理工作。

注意：TestCase之间一定要保持完全的独立，不允许出现任何的依赖关系，不能依赖测试方法的执行顺序。

测试方法执行循序：setUp()---->testMethod()---->tearDown()，即每个测试方法执行前都会执行setUp()方法，在测试方法执行完毕后执行tearDown()方法，所以

setUp()方法一般 用来初始化工作，而tearDown()方法用来关闭

|  |
| --- |
| **public** **class** Calulate  {  **public** **int** add(**int** num1 ,**int** num2)  {  **return** num1+num2 ;  }  **private** **int** descreate(**int** num1 , **int** num2)  {  **return** num1-num2 ;  }  **private** **int** mulipty(**int** num1 , **int** num2)  {  **return** num1\*num2 ;  }    **public** **int** divide(**int** num1 , **int** num2) **throws** Exception  {  **return** num1/num2 ;  }  **public** **int** getMax(**int**[] nums) **throws** Exception  {  **int** max = nums[0] ;  **if**(nums == **null** || nums.length==0)  {  **throw** **new** Exception("数组不能为空");  }  **else**  {  **for**(**int** i = 0;i<nums.length;i++)  {  **if**(max<nums[i])  max = nums[i];  }  }  **return** max ;  }  } |

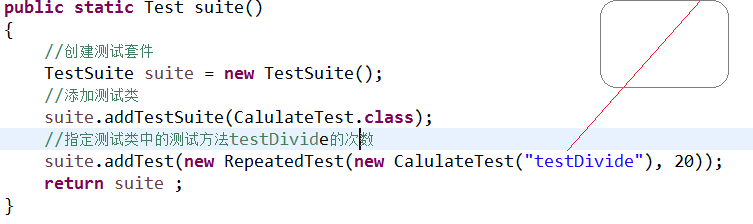
测试用例：

|  |
| --- |
| **import** junit.framework.Assert;  **import** junit.framework.TestCase;  **public** **class** CalulateTest **extends** TestCase  {  Calulate c = **null** ;  @Override  **public** **void** setUp() **throws** Exception  {  c = **new** Calulate();  }  **public** **void** testAdd()  {  **int** result = c.add(10, 29);  Assert.*assertEquals*(39, result);  }  **public** **void** testDivide()  {  **try**  {  **int** result = c.divide(10, 0);  } **catch** (Exception e) {  //终止测试用例的执行  Assert.*fail*("Test fail");  }  }  **public** **void** testGetMax()  {  **int** result = 0 ;  **int**[] nums1 = {10,-23,34,100,56,-78};  **try**  {  result = c.getMax(nums1);  }  **catch**(Exception e)  {  Assert.*fail*("测试失败");  }  Assert.*assertEquals*(100, result);  }  **public** **void** testGetMax2()  {  **int** result = 0 ;  **int**[] nums2= {} ;  Throwable t = **null** ;  **try**  {  result = c.getMax(nums2);  Assert.*fail*("Test Fails");  }  **catch**(Exception e)  {  t = e ;  }  Assert.*assertNotNull*(t);  Assert.*assertEquals*(Exception.**class**, t.getCause());  Assert.*assertEquals*("数组不能为空", t.getMessage());  }  } |

TestSuite：作为测试套件，可以执行多个测试类。

|  |
| --- |
| **import** junit.framework.Test;  **import** junit.framework.TestCase;  **import** junit.framework.TestSuite;  **public** **class** TestAll **extends** TestCase  {  **public** **static** Test suite()  {  //创建测试套件  TestSuite suite = **new** TestSuite();  //添加测试类  suite.addTestSuite(CalulateTest.**class**);  **return** suite ;  }  } |

RepeatTest:指定测试类中的测试方法重复测试的次数。在生成RepeatTest对象时，该对象的构造方法的第一个参数必须为TestCase的子类，并且该子类的构造方法必须有参数，在RepeatTest个构造方法中生成TestCase子类对象，生成的子类对象指明了要循环测试方法的方法名。



## Junit4

Junit4及以上版本全面引入了注解的方式来执行我们的测试类

Junit4并不要求测试类继承TestCase父类。

在一个测试类中，所有被@Test注解所修饰的方法都是测试类，都可以被Junit所执行。

[在Junit4中注解@Before与Junit3.8](mailto:在Junit4中注解@Before与Junit3.8)中的setUp()方法有同样的功能，@After和tearDown()有同样的功能。

@Before：测试方法执行之前执行，有几个测试方法就执行几次。

@After：测试方法执行之后执行，有几个测试方法就执行几次。

在多个测试方法共享昂贵的初始化工作时（比如：数据库日志），并且独立于测试的，我们可以使用@BeforClass和@AfterClass

@BeforeClass:修饰public static void 方法，当任意测试方法执行之前执行一次。

@AfterClass:修饰public static void 方法，在所有的测试方法执行之后执行。

执行顺序：

@BeforeClass修饰的方法

@Befor修饰的方法

测试方法

@After修饰的方法

@AfterClass修饰的方法

@Ignore:可以修饰类和方法，当修饰类时，表示该类中的所有测试方法都将被忽略。当修饰方法时，表示忽略该测试方法。

参数化测试（Parameters）：当一个类使用参数化运行器时，[必须在该类生命出加上@RunWith(Parameterized.class)](mailto:必须在该类生命出加上@RunWith(Parameterized.class))注解，表示该类将不会使用Junit内置的运行器运行，而使用参数化运行器运行，在参数化运行类中提供参数的方法上要使用@Paramters注解来修饰，同时在测试类的构造方法中为各个参数赋值（构造方法有Junit调用），最后编写测试类，它会根据参数组数来运行测试的次数。

|  |
| --- |
| **import** java.util.Arrays;  **import** java.util.Collection;  **import** **static** org.junit.Assert.*assertEquals*;  **import** org.junit.Before;  **import** org.junit.Test;  **import** org.junit.runner.RunWith;  **import** org.junit.runners.Parameterized;  **import** org.junit.runners.Parameterized.Parameters;  @RunWith(Parameterized.**class**)  **public** **class** ParameterTest  {  Calulate c ;  **private** **int** expect ;  **private** **int** num1 ;  **private** **int** num2 ;  **public** ParameterTest(**int** expect , **int** num1 , **int** num2)  {  **this**.expect = expect ;  **this**.num1 = num1 ;  **this**.num2 = num2 ;  }  @Parameters  **public** **static** Collection getParamert()  {  Object[][] para = {{10,20,-10},{5,1,4},{20,10,10}};  **return** Arrays.*asList*(para) ;  }  @Before  **public** **void** setUp()  {  c = **new** Calulate();  }  @Test  **public** **void** testAdd()  {  *assertEquals*(**this**.expect, c.add(**this**.num1, **this**.num2));  }  } |

测试套件：用于测试多个测试类，[使用@Suite.SuiteClassess](mailto:使用@Suite.SuiteClassess)({测试类.class})注解和@RunWith(Suite.class)注解。

|  |
| --- |
| **import** org.junit.runner.RunWith;  **import** org.junit.runners.Suite;  @RunWith(Suite.**class**)  @Suite.SuiteClasses({Calulate.**class**,FileDemoTest.**class**})  **public** **class** TestAll  {} |

## 源码分析

### 设计模式

#### 模板方法模式（Template Method）

定义一个操作中的算法骨架，而将一些步骤延伸到子类中去，使得子类可以不改变一个算法的结构，即可重新定义该算法的某些特定步骤。这里需要复用的是算法的结构，也就是步骤，而步骤的实现可以在子类中完成。

使用场合：

（1）一次性实现一个算法的不变部分，并且将可变的行为留该给子类来完成。

（2）各子类公共的行为应该被提取出来并集中到一个公共父类中以避免代码的重复，首先识别现有代码的不同之处，并且把不同部分分离为新的操作，最后，用一个调用这些新的操作的模板方法来代替这些不同的代码。

（3）控制子类的扩展。

模板方法模式组成：父类角色：提供模板。

子类角色：为模板提供实现。

|  |
| --- |
| **public** **abstract** **class** AbstraceClass  {  **public** **abstract** **void** method1();  **public** **abstract** **void** method2();  **public** **abstract** **void** method3();  **public** **void** template()  {  method1();  method2();  method3();  } |
| **public** **class** ConcrateClass **extends** AbstraceClass  {  @Override  **public** **void** method1() {  System.***out***.println("method1 invoked");  }  @Override  **public** **void** method2() {  System.***out***.println("method2 invoked");  }  @Override  **public** **void** method3() {  System.***out***.println("method3 invoked");  }  } |
| **public** **class** Client  {  **public** **static** **void** main(String[] args)  {  AbstraceClass ac = **new** ConcrateClass();  ac.template();  }  } |

在Junit3.8中TestCase类规定了测试类的执行顺序，即功能的实现就是基于模板方法所设计的。

#### 适配器模式

在软件系统中，由于应用环境的变化，常常需要将一些现存的对象放在新的环境中应用，但是新环境要求的接口是这些现存对象不满足的，那么如何应对这种“迁移的变化？”如何既能利用现有对象的良好实现，同时又能满足新的应用环境所要求的接口？

将一个类的接口转化成客户希望的另外一个接口，Adapter模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

适配器模式构成：

目标抽象角色（Target）：定义客户要用的特定领域的接口。

适配器（Adapter）：调用另一个接口，作为一个转换器

适配器（Adaptee）：定义一个接口，Adapter需要接入。

客户端（Client）：协同对象符合Adapter适配器。

适配器分类：（1）类适配器（采用继承的方式）

（2）对象适配器（采用对象组合的方式）

类适配器：

|  |
| --- |
| /\*\*  \*定义用户要使用特定接口  \*/  **public** **interface** Target  {  **public** **void** method1();  } |
| /\*\*  \*用户无法直接使用的类  \*/  **public** **class** Adaptee  {  **public** **void** method2()  {  System.***out***.println("method2 invoked");  }  } |
| /\*\*  \* 适配器  \*  \*/  **public** **class** Adapter **extends** Adaptee **implements** Target  {  @Override  **public** **void** method1()  {  **this**.method2();  }  } |
| **public** **class** Client  {  **public** **static** **void** main(String[] args)  {  Target t = **new** Adapter();  t.method1();  }  } |

对象组合方法：

|  |
| --- |
| /\*\*  \*定义用户要使用特定接口  \*/  **public** **interface** Target  {  **public** **void** method1();  } |
| /\*\*  \*用户无法直接使用的类  \*/  **public** **class** Adaptee  {  **public** **void** method2()  {  System.***out***.println("method2 invoked");  }  } |
| /\*\*  \* 适配器  \*  \*/  **public** **class** Adapter **implements** Target  {  **private** Adaptee ad ;    **public** Adapter(Adaptee ad)  {  **this**.ad = ad ;  }  @Override  **public** **void** method1()  {  ad.method2();  }  } |
| **public** **class** Client  {  **public** **static** **void** main(String[] args)  {  Adaptee ad = **new** Adaptee();  Target t = **new** Adapter(ad);  t.method1();  }  } |

缺省的适配器：

Java中AWT和Swing使用的缺省的适配器较多。

|  |
| --- |
| **public** **interface** AbstractService  {  **public** **void** service1();  **public** **void** service2();  **public** **void** service3();  } |
| /\*\*  \* 空实现，有子类实现自己需要的方法。  \*/  **public** **class** ServiceAdapter **implements** AbstractService  {  @Override  **public** **void** service1() {  }  @Override  **public** **void** service2() {  }  @Override  **public** **void** service3() {    }  } |
| /\*\*  \* 完成service1的操作  \*/  **public** **class** ConreteService **extends** ServiceAdapter  {  @Override  **public** **void** service1()  {  System.***out***.println("service1 invoked");  }  } |

#### 命令模式

命令模式将一个请求封装成一个对象，从而使你可用不同的请求对客户进行参数化：对请求进行排队或记录请求日志，以及支持可撤销的操作，命令模式告诉我们可以为一个操作生成一个对象并给出它的一个execute方法。



命令模式构成：

客户角色：创建一个具体命令对象，并确定其接受者

命令角色：声明一个给所有具体命令类的抽象接口，这是一个抽象色，通常由一 个接口或抽象类实现。

具体命令角色：定义一个接受者和行为之间的弱耦合，实现execute方法，负责调用接受者的相应操作。

请求者角色：负责调用命令对象执行请求

接受者角色：负责具体实施和执行一个请求

# TestNG