目 录

[1 Junit简介 2](#_Toc325124623)

[2 工具包准备 2](#_Toc325124624)

[2.1 JDK 工具包 2](#_Toc325124625)

[2.2 Junit工具包 2](#_Toc325124626)

[2.3 选择第三方IDE 2](#_Toc325124627)

[3 配置环境 2](#_Toc325124628)

[4 创建JUnit测试用例并执行 5](#_Toc325124629)

[4.1 自动生成JUnit测试用例 5](#_Toc325124634)

[4.2 执行测试用例 7](#_Toc325124635)

[4.3 查看测试结果 7](#_Toc325124636)

[5 Junit4的Annotation使用 8](#_Toc325124637)

[9.1 基本元数据 8](#_Toc325124643)

[9.2 参数化 12](#_Toc325124644)

[9.3 异常 14](#_Toc325124645)

[9.4 限时 15](#_Toc325124646)

[9.5 套件 16](#_Toc325124647)

[6 Junit4的断言 17](#_Toc325124648)

# Junit简介

JUnit由Kent Beck和ErichGamma开发，是迄今所开发中用的最重要的第三方Java库，它也成为了Java语言事实上的标准单元测试库。对于程序员来说，Junit是白盒测试，因为程序知道自己需要测试什么功能（如接口和方法等）及如何去测试，所以JUnit能引导并促进了测试先行的编程和测试驱动的开发。

JUnit是一个开放源代码的Java测试框架，用于编写和运行可重复的测试。他是用于单元测试框架体系xUnit的一个实例（用于java语言）。它包括以下特性：

* 用于测试期望结果的断言（Assertion）
* 用于共享共同测试数据的测试工具
* 用于方便的组织和运行测试的测试套件
* 图形和文本的测试运行器

# 工具包准备

要想使用junit来测试，首先要了解都需要使用哪些工具包。

## JDK 工具包

JUnit使用的语言平台是JAVA，故要使用JUnit自然就需要安装JAVA了，JAVA是在JAVA虚拟机上运行的（JVM），而通过JDK的编译可以使你编写的程序在JVM上运行。

JDK工具包也是可以通过网上去下载的，这里就不提供网址了，网上有很多，JDK的JAR包名为：jdk-6u10-rc2-bin-b32-windows-i586-p-12\_sep\_2008.exe，我用的是1.6版本的。

## Junit工具包

可以使用某IDE自带的工具包，也可以自己去网上下载一个Junit工具包（下载地址：https://github.com/KentBeck/junit/downloads），最好是比较新版本的，Junit4.4之上的，下载解压后的文件名为Junit-4.x.jar。

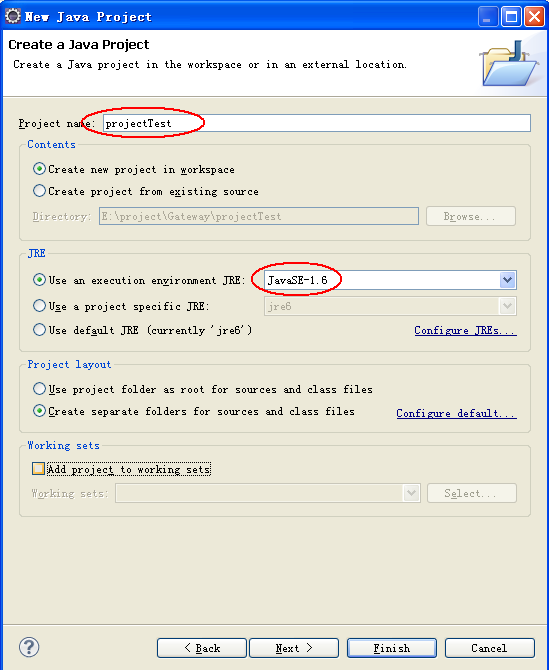
## 选择第三方IDE

通常我们会借用第三方的IDE，（ Eclipse, NetBean, Idea等），基本上它们都全面支持Junit了，但这里我们选择其中一个为Eclipse IDE，它集成了JUnit 组件，无须另行下载和安装。Eclipse 3.5版本是绿色版的，解压后便可使用。

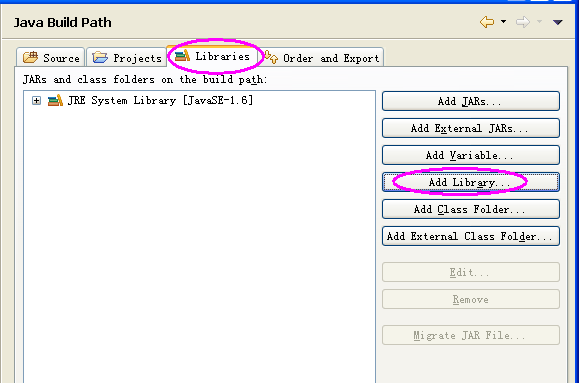
# 配置环境

* 首先将Junit的工具包放到某个目录下，如d:\JunitTest\lib 目录下，此包是用来被IDE引用的。(这步是介绍未使用IDE自身带的Junit包，如果使用IDE自带的则此步要省略)。
* 打开eclipse，创建project并使用Junit包，操作步骤如下：

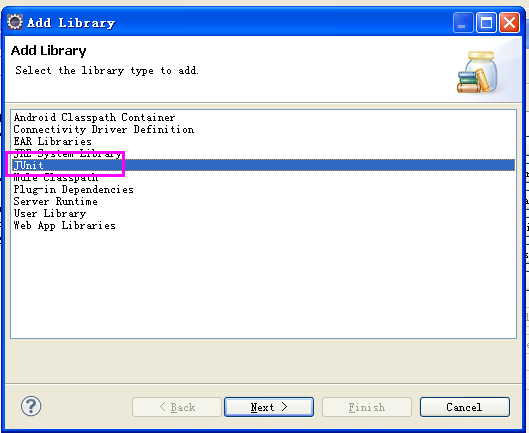
在eclipse的工具栏中选择file ->new ->project ->Java project，

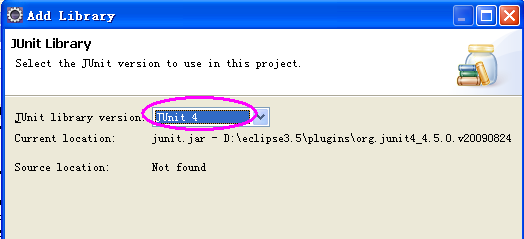


这里输入project名称，并选择的JDK1.6版本，点击【next】，进入到

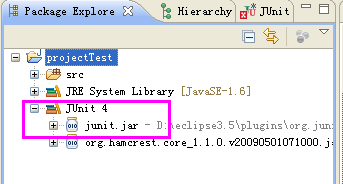


如果要选择不是eclipse自身带的JAR包的话，那么要选择‘Add Exernal JARs…’





这里添加是eclipse本身自带的JUnit 的JAR包，然后一路保存后，project就创建成功了，这时可以看到刚刚选择的JUnit包已经添加上去了，



现在就可以向这个project 里创建测试用例了。

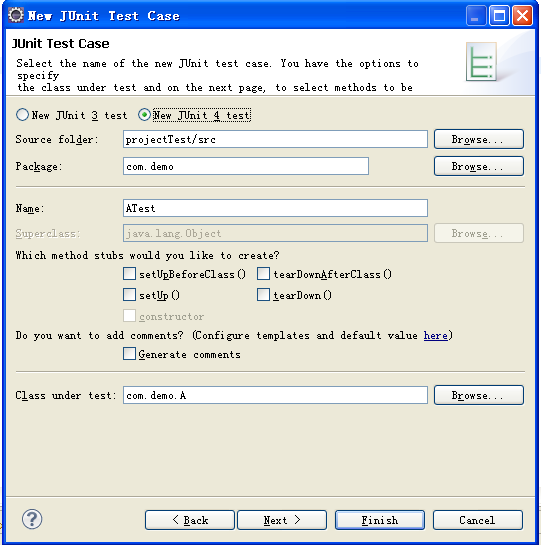
# 创建JUnit测试用例并执行



## 自动生成JUnit测试用例

假设现在需要测试一个类中的所有方法，方法分别为a，b，c，那么可以通过Junit自动生成，操作步骤如下：

右击需要测试的类文件，选择new->junit Testcase->

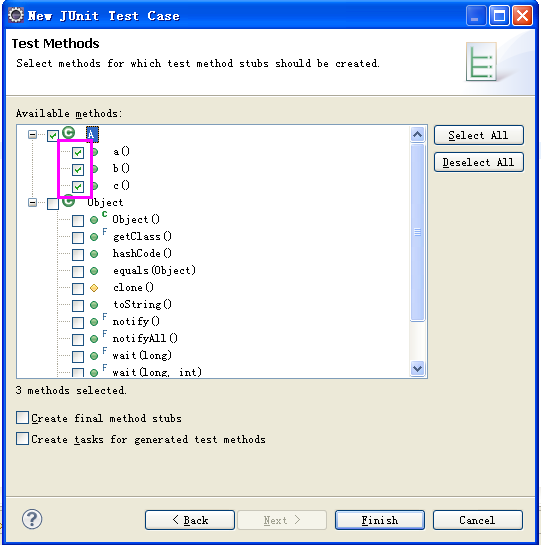


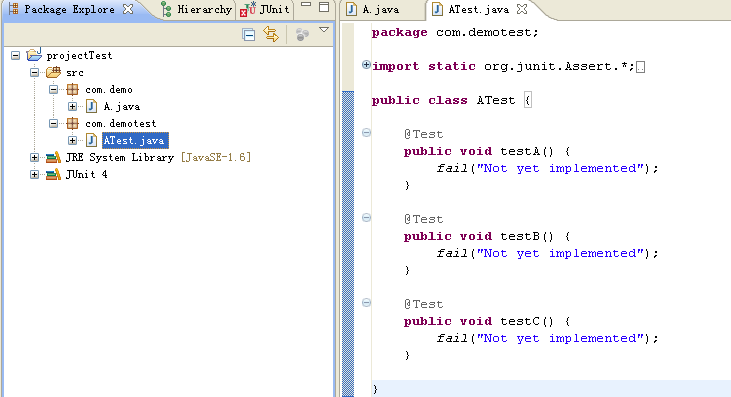
New Junit 4 test ：代表你使用的是Junit 4 版本的jar包。

Source folder + package ：代表将要生成测试文件的路径。

Name ：代表将要生成测试的文件名称。

选择 [next]，

 点击完成，自动创建测试用例成功，大家可以看一个生成的类文件内容。

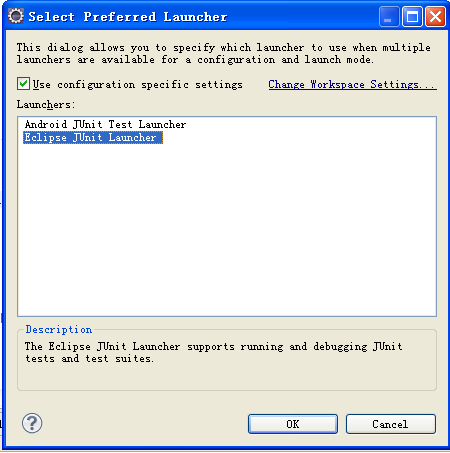


补充说明：这时生成的测试类只是一个框架来的，具体需要如何测试以及得到什么样的测试结果，还需要自己修改测试方法，以达到测试需要的最终结果。

## 执行测试用例

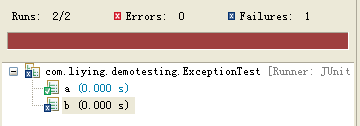
当测试类编写完成后，需要对其执行才能看到预期结果，执行步骤如下：

右面需要测试的类，选择Run As ->JUnit Test，对于已经执行过且没有做过修改的测试类，再次执行时没有任何提示，直接显示测试结果，但如果是第一次执行一个测试类或这个测试类被修改后以再次被执行，那么会有下面这个提示，



在这里选择即可，系统会自动执行测试类文件。

## 查看测试结果



如上图结果说明：

* Runs: 2/2 代表此测试类共有两个方法（后面的数字2），共执行了两个方法（前面的数字2）； Errors: 0 代表没有发生错误；Failures:1 代表此测试类中有一个方法未通过。
* 当这个类文件中存在任一个方法未测试通过则会以红色显示如：
* 查看方法是否通过，通过的方法用绿色的对勾显示如，如果未通过则以蓝色的叉显示如。

# Junit4的Annotation使用

JUnit4是JUnit框架有史以来的最大改进，其主要目标便是利用Java5的Annotation特性简化测试用例的编写。

先简单解释一下什么是Annotation，这个单词一般是翻译成元数据。元数据就是描述数据的数据。也就是说，这个东西在Java里面可以用来和public、static等关键字一样来修饰类名、方法名、变量名。修饰的作用描述这个数据是做什么用的，差不多和public描述这个数据是公有的一样。想具体了解可以看Core Java2。

Junit4的Annotation有很多，最常用的有下面几个：

org.junit.After;

org.junit.AfterClass;

org.junit.Before;

org.junit.BeforeClass;

org.junit.Test;

org.junit.Ignore;

org.junit.runner.RunWith;

org.junit.runners.Parameterized;

org.junit.runners.Suite

接下来给大家一个个讲述这些元据是如何使用的。



## 基本元数据

@before:初始化方法

@BeforeClass：针对所有测试，只执行一次，且必须为static void

@Test：测试方法，在这里可以测试期望异常和超时时间

@Ignore：忽略的测试方法

@After：释放资源

@AfterClass：针对所有测试，只执行一次，且必须为static void

这里对After，AfterClass，Before，BeforeClass，Test，Ignore统一举例说明。

* 首先要了解一下这些元数据的执行顺序

创建一个测试类文件 AnnotationQueryTest.java ，代码如下：

|  |
| --- |
| **import** org.junit.After;  **import** org.junit.AfterClass;  **import** org.junit.Before;  **import** org.junit.BeforeClass;  **import** org.junit.Test;  **import** org.junit.Ignore;  **public** **class** AnnotationQueryTest {  @BeforeClass  **public** **static** **void** setUpBeforeClass() **throws** Exception {  System.*out*.println("@BeforeClass......setUpBeforeClass");  }  @AfterClass  **public** **static** **void** tearDownAfterClass() **throws** Exception {  System.*out*.println("@AfterClass......tearDownAfterClass");  }  @Before  **public** **void** a() {  System.*out*.println("@Before......a");  }  @Before  **public** **void** setUp() **throws** Exception {  System.*out*.println("@Before......setUp");  }  @After  **public** **void** tearDown() **throws** Exception {  System.*out*.println("@After......tearDown");  }  @After  **public** **void** c() **throws** Exception {  System.*out*.println("@After......c");  System.*out*.println("");  }  @Ignore  @Test  **public** **void** testFoo() {  System.*out*.println("@Test......FOO");  }  @Test  **public** **void** testEquals() {  System.*out*.println("@Test......testEquals");  }  } |

对其执行，显示结果为如下：

|  |
| --- |
| @BeforeClass......setUpBeforeClass  @Before......a  @Before......setUp  @Test......testEquals  @After......tearDown  @After......c  @AfterClass......tearDownAfterClass |

实际上他的执行顺序是：@BeforeClass –> @Before –> @Test –> @After –> @AfterClass

* 再看一下这几个元数据的继承关系（正常情况下，我们测试的时候是不推荐这些元数据继承的，这种使用比较复杂。）

首先先创建三个测试类文件分别为：ParentTestcase.java，FirstChildTestcase.java，SecondChildTestcase.java， 代码如下：

|  |
| --- |
| import org.junit.AfterClass;  import org.junit.BeforeClass;  import org.junit.After;  import org.junit.Before;  import org.junit.Test;  public class ParentTestcase {  @BeforeClass  public static void beforeClass() {  System.out.println("@BeforeClass");  }  @AfterClass  public static void afterClass() {  System.out.println("@AfterClass");  }  @Before  public void before() {  System.out.println("@Before");  }  @After  public void after() {  System.out.println("@After");  }  @Test  public void test() {  System.out.println("@test");  // assertTrue(true);  }  } |

|  |
| --- |
| import org.junit.After;  import org.junit.AfterClass;  import org.junit.Before;  import org.junit.BeforeClass;  import org.junit.Test;  public class FirstChildTestcase extends ParentTestcase{  @BeforeClass  public static void beforeClass() {  System.out.println("first @BeforeClass");  }  @AfterClass  public static void afterClass() {  System.out.println("first @AfterClass");  }  @Before  public void before() {  System.out.println("first @Before");  }  @After  public void after() {  System.out.println("first @After");  }  @Test  public void test1() {  System.out.println("first @test1");  }  } |

|  |
| --- |
| import org.junit.After;  import org.junit.AfterClass;  import org.junit.Before;  import org.junit.BeforeClass;  import org.junit.Test;  public class SecondChildTestcase extends ParentTestcase{  @BeforeClass  public static void secondBeforeClass() {  System.out.println("second @BeforeClass");  }  @AfterClass  public static void secondAfterClass() {  System.out.println("second @AfterClass");  }  @Before  public void secondBefore() {  System.out.println("second @Before");  }  @After  public void secondAfter() {  System.out.println("second @After");  }  @Test  public void secondTest() {  System.out.println("second @test");  }  } |

当执行ParentTestcase测试类时，结果很简单也容易看明白；

当执行FirstChildTestcase 测试类时，这个类继承了父类的所有方法，即执行时不会再读父类的相同方法了，已被子类覆盖掉。

当执行SecondChildTestcase测试类时，是先执行父类的tearUp方法，再执行子类的tearUp方法，然后再先执行子类的tearDown方法，再执行父类的tearDown方法，其结果

如下：

|  |
| --- |
| @BeforeClass  second @BeforeClass  @Before  second @Before  second @test  second @After  @After  @Before  second @Before  @test  second @After  @After  second @AfterClass  @AfterClass |

## 参数化

当我们测试一个方法时，需要给这个方法传入多种不同情况的参数，在传统的方式下，是需要写多个测试方法，但是有JUnit的Parameterized这个元数据，那么我们只需要写一个方法就可以了，JUnit自己会帮我们调用这些事先定义的参数。

使用形式：@RunWith(Parameterized.**class**)[和@Parameterized.Parameters](mailto:和@Parameterized.Parameters)

代码如下：

首先创建一个类文件ParameterizedDemo.java

|  |
| --- |
| **public** **class** ParameterizedDemo {    **public** **boolean** convertString(String str){  **if** (str.equals("liying")){  **return** **true**;  }**else**  **return** **false**;    }    **public** **void** printString(String str, String value){  System.*out*.println(str + " " + value);  }  } |

然后再创建一个测试类文件ParameterizedTest.java

|  |
| --- |
| **import** java.util.Arrays;  **import** java.util.List;  **import** org.junit.After;  **import** org.junit.Before;  **import** org.junit.Test;  **import** org.junit.runner.RunWith;  **import** org.junit.runners.Parameterized;  **import** com.liying.demo.ParameterizedDemo;  @RunWith(Parameterized.**class**)  **public** **class** ParameterizedTest {    ParameterizedDemo junitAnnotation;    @Parameterized.Parameters  **public** **static** List<Object[]> data2() {  **return** Arrays.*asList*(**new** Object[][] {{ "li","1" }, { "liy","2" },{ "liying","3"}});  }    String key;  String value;  **public** ParameterizedTest(String key,String value){  **this**.key = key;  **this**.value = value;  }    @Before  **public** **void** setUp() **throws** Exception {  junitAnnotation = **new** ParameterizedDemo();  }    /\*\*  @After  **public** **void** tearDown() **throws** Exception {  **try** {  // Assert  System.*out*.println("after...not exception");  } **catch** (Exception e) {  System.*out*.println("after...throw exception: "+ e.getMessage());  }  }  \*/  @Test  **public** **void** printStr(){  junitAnnotation.printString(key,value);  }  } |

执行原理：细心的朋友可以看到方法data2中定义一个二维数组变量，分别写入需要测试的不同情况下的参数，再看构造函数ParameterizedTest，这个函数有两个入参数：key和value，这里实际上就是data2方法中设置的二维数组对应的值，如果二维数组的长度大于1时，系统会自动顺序将二维数组中所有数据列依次传给构造函数，传参顺序分别为key和value，直到数组的值全部赋值完为止。每次赋值会执行一次构造函数。

执行测试类文件结果如下：

|  |
| --- |
| li 1  liy 2  liying 3 |

## 异常

设置一个需要测试的函数在需要抛出异常时是否能正确的抛出异常。

使用形式：@Test(expected = \*.**class**)

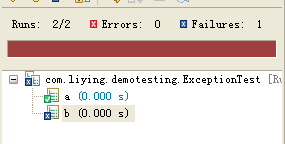
首先创建一个类文件ExceptionDemo.java

|  |
| --- |
| **public** **class** ExceptionDemo {    **public** **void** exitException() **throws** Exception{  **throw** **new** Exception();  }    **public** **void** noException() **throws** Exception{  }  } |

再创建一个测试类

|  |
| --- |
| **import** org.junit.Test;  **import** com.liying.demo.ExceptionDemo;  **public** **class** ExceptionTest {  ExceptionDemo exceptionDemo = **new** ExceptionDemo();    @Test(expected = Exception.**class**)  **public** **void** a() **throws** Exception{  exceptionDemo.exitException();  }    @Test(expected = Exception.**class**)  **public** **void** b() **throws** Exception{  exceptionDemo.noException();  }  } |

执行结果：



很明显， a() 方法是用来测试抛异常的，而预期结果也正好抛出了异常，所以测试通过；而方法b()不是用来测抛异常的，反而却抛出了异常，所以测试失败。大家以后用熟悉了，就能了解到底在什么情况下需要抛异常，什么情况下不需要，以便灵活运用。

## 限时

给这些测试函数设定一个执行时间，超过了这个时间，他们就会被系统强行终止，并且系统还会向你汇报该函数结束的原因是因为超时，这样你就可以发现这些Bug了。要实现这一功能，

使用形式：@Test(timeout = xxx)，代码如下：

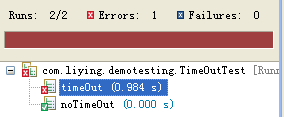
首先创建一个类文件TimeOutDemo.java

|  |
| --- |
| **public** **class** TimeOutDemo {    **public** **void** timeOut(){  **for** (; ;) ;  }    **public** **void** noTimeOut(){  System.*out*.println("no time out");  }  } |

再创建一个测试类TimeOutTest.java

|  |
| --- |
| **import** org.junit.Test;  **import** com.liying.demo.TimeOutDemo;  **public** **class** TimeOutTest {  TimeOutDemo timeOutDemo = **new** TimeOutDemo();    @Test(timeout = 1000)  **public** **void** timeOut(){  timeOutDemo.timeOut();  }    @Test(timeout = 100)  **public** **void** noTimeOut(){  timeOutDemo.noTimeOut();  }  } |

执行结果：



timeout()方法是用法测超时的，恰好被测的方法是一个死循环，一旦被执行测进入死循环状态，所以在1000秒是不可能执行完的，达到超时的目的。而下一个方法noTimeout()方法是用来测不会超时的方法，测试结果通过。

## 套件

在一个项目中，只写一个测试类是不可能的，我们会写出很多很多个测试类。可是这些测试类必须一个一个的执行，也是比较麻烦的事情。鉴于此，JUnit为我们提供了打包测试的功能，将所有需要运行的测试类集中起来，一次性的运行完毕，大大的方便了我们的测试工作。

使用形式：Suite.SuiteClasses({ ParameterizedTest.**class**,DemoTest.**class** })

创建一个测试类文件SuiteAnnotationTest.java

|  |
| --- |
| **import** org.junit.runner.RunWith;  **import** org.junit.runners.Suite;  @RunWith(Suite.**class**)  @Suite.SuiteClasses({ ParameterizedTest.**class**,TimeOutTest.**class** })  **public** **class** SuiteAnnotationTest {  } |

执行原理：

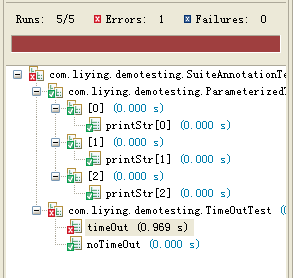
@RunWith(Suite.**class**)

@Suite.SuiteClasses({ ParameterizedTest.**class**,TimeOutTest.**class** })

这个是将ParameterizedTest.**class**,TimeOutTest.**class**一次性打包统一执行。

而下面的类是一个空类，没有内容的，这个只是一个形式，JUnit规定就是这个形式的，所以类名无所谓，随便起就是了。

执行结果：



# Junit4的断言

断言(Assert)是JUnit包中的一个类文件，它是测试中最常用到且用的最多的，这个类中定义了很多个静态方法，类的package 是junit.framework。下面举个实例对Assert中的各种方法如何使用做个简单的介绍。

创建一个类文件AssertDemo.java

|  |
| --- |
| **public** **class** AssertDemo {  String str;  **public** **boolean** trueOrFalse(**boolean** value) {  **return** !value;  }    **public** **int** sum(**int** a, **int** b) {  **return** a \* b;  }  **public** **int**[] array1() {  **int**[] score = { 1, 2, 3, 4 };  **return** score;  }  **public** **int**[] array2() {  **int**[] score = { 1, 2, 3, 5 };  **return** score;  }  **public** String getObject() {  **return** str;  }  **public** String getString() {  str = "abc";  **return** str;  }    **public** **void** exitException(String str) **throws** Exception{  **if** (!str.equals("abc"))  **throw** **new** Exception("throw exception!");  }  } |

再创建一个测试类文件AssertTest.java

|  |
| --- |
| **import** org.junit.Test;  **import** com.liying.demo.AssertDemo;  **import** **static** org.junit.Assert.*assertEquals*;  **import** **static** org.junit.Assert.*assertArrayEquals*;  **import** **static** org.junit.Assert.*assertNotNull*;  **import** **static** org.junit.Assert.*assertSame*;  **import** **static** org.junit.Assert.*assertTrue*;  **import** **static** org.junit.Assert.*assertThat*;  **import** **static** org.junit.Assert.*fail*;  **import** **static** org.hamcrest.CoreMatchers.*equalTo*;  **public** **class** AssertTest {  AssertDemo assertDemo = **new** AssertDemo();  @Test  **public** **void** isFalse() {  *assertTrue*("expected result should be true, but it was <"  + assertDemo.trueOrFalse(**true**) + ">", assertDemo  .trueOrFalse(**true**));  }  @Test  **public** **void** examineSum() {  *assertEquals*(11, assertDemo.sum(1, 10));  }  @Test  **public** **void** matcheArray() {  *assertArrayEquals*(assertDemo.array1(), assertDemo.array2());  }  @Test  **public** **void** notNullObject() {  *assertNotNull*("actual result was null!", assertDemo.getObject());  }  @Test  **public** **void** isSameString() {  *assertSame*("a", assertDemo.getString());  }  @Test  **public** **void** isThat() {  *assertThat*(11, *equalTo*(assertDemo.sum(1, 10)));  }  @Test  **public** **void** isFaile() **throws** Exception {  **try** {  assertDemo.exitException("abc");  *fail*("shoud throw exception, but was not!");  } **catch** (Exception e) {  System.*out*.println(e.getMessage());  }  }  } |

执行结果：

