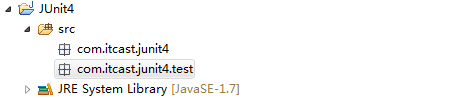
**单元测试**

**第一部分**

先创建一个Java Project，名字就叫做JUnit4,然后创建两个包com.itcast.junit4和com.itcast.junit.test,如下图所示



 其中com.itcast.junit4用于自己写的类和方法的存放，com.itcast.junit4.test用于完成单元测试在com.itcast.junit4包中创建一个类叫做T(名字可以是任意的,这里是为了方便)，在类中添加两个方法add和divide实现两个数的加法和除法运算

1 package com.itcast.junit4;

2

3 public class T {

4

5 public int add(int x,int y){

6 return x+y;

7 }

8 public int divide(int x,int y){

9 return x/y;

10 }

11 public static void main(String[] args) {

12 int z=new T().add(3, 5);

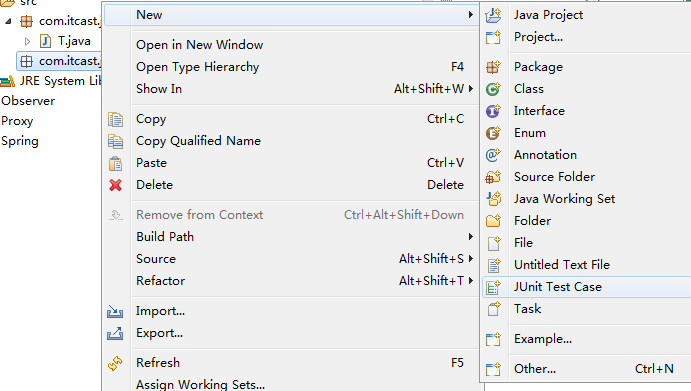
13 System.out.println(z);

14 }

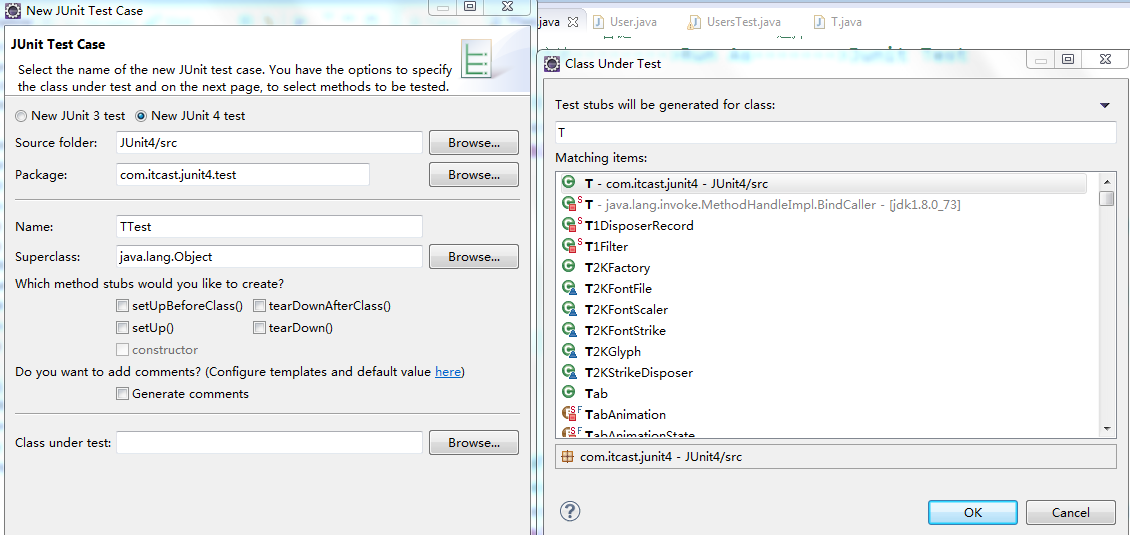
15

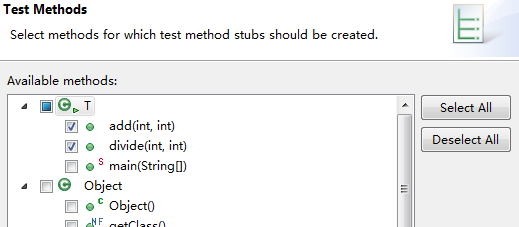
16 }

传统测试的方法通常都是按上面的方式去看add函数是否可以实现我们想要的功能，把方法的调用放在main函数中。下面是单元测试：  
在com.itcast.junit4.test包中创建一个类，叫做TTest(单元测试命名规范：**a)** 类放在test包中;**b)** 类名用XXXTest结尾;**c)** 方法用testMethod命名;)

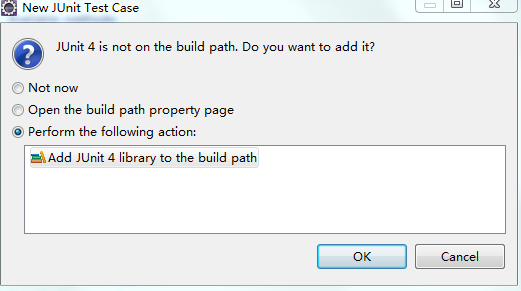


【步骤】com.itcast.junit4.test包-->右键-->New--Junit Test Case，然后选择New Junit 4 test。下面的一行Class under test,单击右侧的Browser,在弹出的输入框中输入我们想要测试的类名T，选中单击next出现了一个界面要选择需要测试的方法，选择add方法和divide;

~~~~



 接着出现了下面的界面：



这一步提示我们是否将需要的JUnit 4的相关包加入到项目的ClassPath路径下，点击OK就行，因为Eclipse中包含JUnit的jar包，先用Eclipse自带的Junit4去测试，后面后将如何用自己下载的JUnit4 JAR包。

做完上面的步骤会创建一个这样的测试类

1 import static org.junit.Assert.\*;

2

3 import org.junit.Test;

4

5 public class TTest {

6

7 @Test

8 public void testAdd() {

9 fail("Not yet implemented");

10 }

11

12 @Test

13 public void testDivide() {

14 fail("Not yet implemented");

15 }

16

17 }

上面的org.junit.Assert.\*;就是静态导入的我们实现单元测试要用到的一些方法；【注意】这是静态引入，可以把方法直接引入,org.junit.Assert是一个类，不是一个包，当然这些方法肯定都是静态方法了。出现的代码都是Assert类中的一些方法，"@Test”表明下面这个方法是一个测试方法，先删除自动生成的fail()函数的代码。添加以下代码：

1 public class TTest {

2

3 @Test

4 public void testAdd() {

5 int z=new T().add(2, 4);

6 //判断z==6,以往的assert

7 assertEquals(6, z);

8 }

9

10 @Test

11 public void testDivide() {

12 //测试T类中的divide方法

13 int z=new T().divide(8, 2);

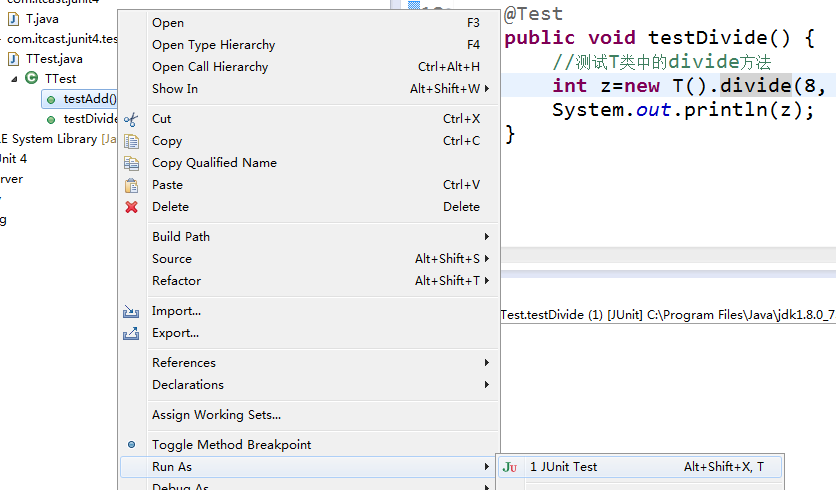
14 System.out.println(z);

15 }

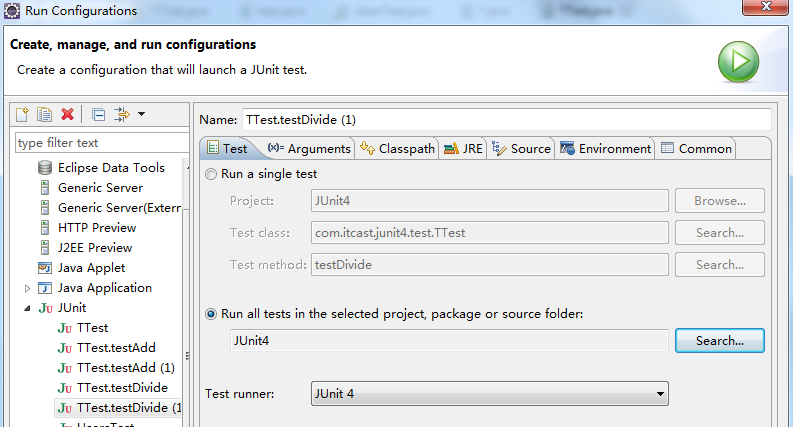
16

17 }

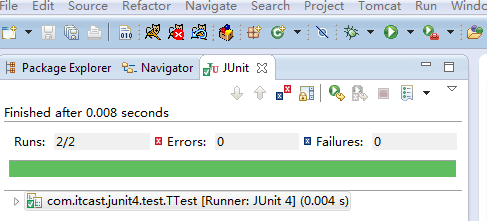
现在开始进行测试：要测试的方法-->右键-->Run As-->JUnit Test



 如果想两个方法一块测试，则Run As-->选择Run Configurations:按图中选择相应的选项，然后点击Run



点击run会出现下面的结果：



绿条显示两个方法的功能没有错误，有这样的调试准则:keep the bar green,to keep the code clean，绿色代表测试成功，其中Error:程序出错;Failures:测试失败

Error:是程序有问题，比如在testAdd方法中加上这一句：int a=8/0;再次测试这个方法则会出现一个Error



1 public void testAdd() {

2 int z=new T().add(2, 4);

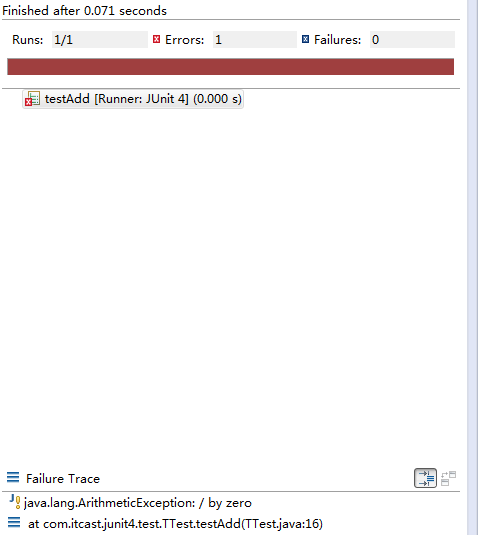
3 //判断z==6,以往的assert

4 assertEquals(6, z);

5 int a=8/0;

6 }

View Code



可以看到最下方提示，我写的方法中出现了除数为0的情况

Failures:测试失败，比如我们在方法改成下面的形式：



1 public void testAdd() {

2 int z=new T().add(2, 4);

3 //判断z==6,以往的assert

4 assertEquals(6, z);

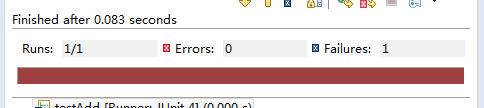
5 assertTrue(z<3);

6 // int a=8/0;

7 }

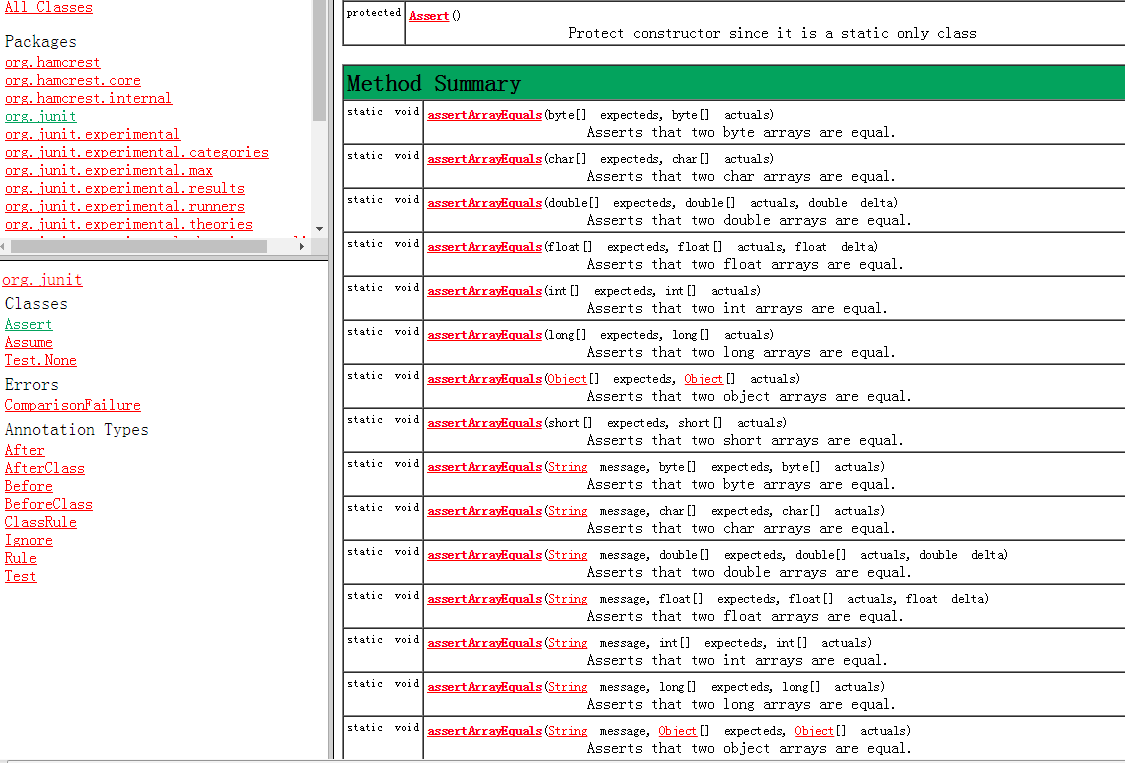
View Code

再次测试一下我的方法：则会出现调试失败的情况



**第二部分**

查看一下JUnit API可以看到org.junit.Assert类有很多类似于assertEquals(6, z);assertTrue(z<3);之类的方法的使用;



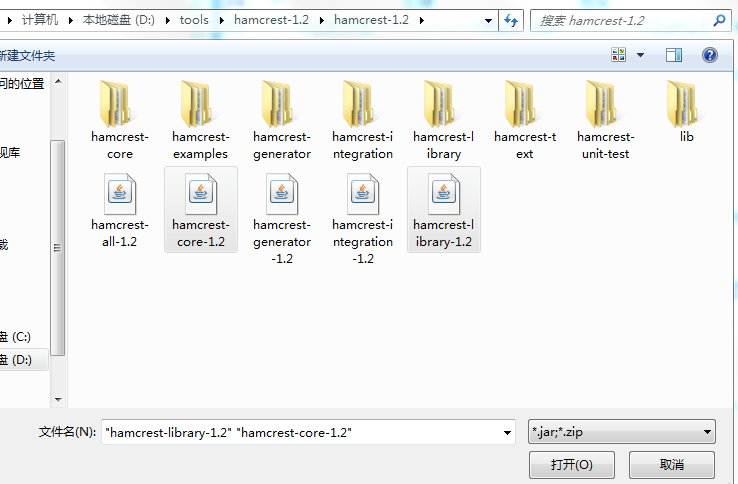
可以看到有很多方法都是以重载的形式出现的，比如我前面的例子，在testAdd()方法中添加assertTrue("z too small",z>10);前面的字符串用于在测试失败的情况下给提示：因为8<10，所以会在测试失败的情况下提示"z too small";

assertThat

assertThat(来自hamcrest包，所以我们需要下载hamcrest这个包,里面有很多平时都可以用到的JAR包和文件，assertThat(actual, matcher);的出现可以替代其他所有的assert。放弃旧的断言,使用hamcrest断言。其中actual参数是实际的值，matcher可以是一个匹配器。在以后的项目开发中就可以使用assertThat代替前面出现的类如assertEquals(6, z); assertTrue(z<3);方法。

首先第一步，想使用assertThat,需要添加两个jar包hamcrest-core-1.2和hamcrest-library-1.2，把这两个包添加进的Java Project中

JUnit4 Test-->右键-->Build Path-->Add External Archives,将这两个Jar加进去



添加以后：就可以使用assertThat



把testAdd()方法改成下面的形式：



1 @Test

2 public void testAdd() {

3 int z=new T().add(2, 4);

4 assertThat(z, is(8));

5 //判断z==6,以往的assert

6 // assertEquals(6, z);

7 // assertTrue(z<3);

8 // int a=8/0;

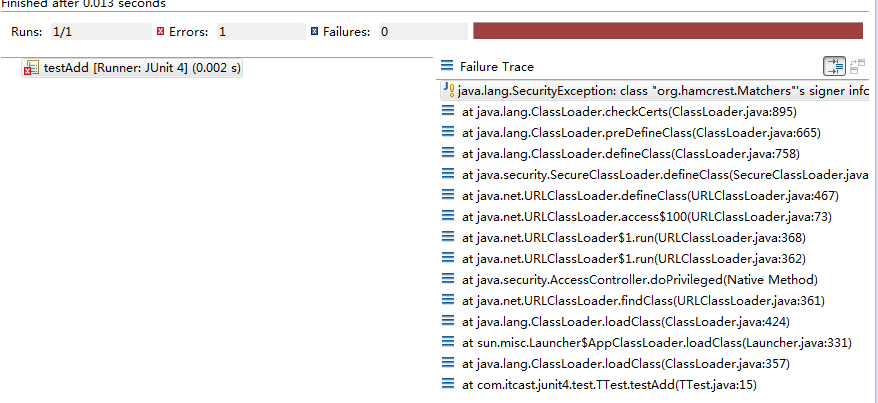
9 }

View Code

代码中的is()方法是在import static org.hamcrest.Matchers(这个类在我们加进来的hamcrest-core-1.2.jar内)类的一个方法需要将其静态引入**，**所以在最上面要加上下面这一句，应该就可以了;

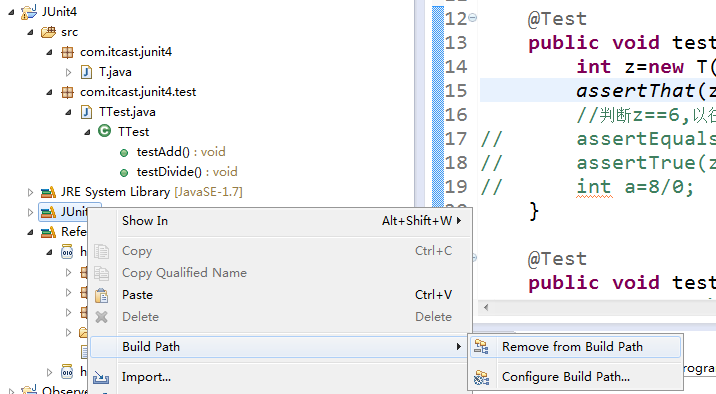
import static org.hamcrest.Matchers.\*;

但是，测试又出现了这样的错误：



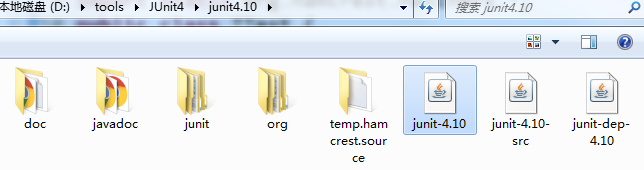
可以看到Failure Trace第一行什么ClassLoader的错误，这是因为在这里用了两种包，一个是hamcrest包，一个是JUnit4的包，这两个包它们的ClassLoader用的不是一个。

解决方法：在JUnit4项目中-->右键JUnit->Build Path-->Remove from Build Path，如图所示：



然后将JUnit包引入进来

JUnit4 Test-->右键-->Build Path-->Add External Archives,选择JUnit包中的junit-4.10，如下图所示：



ok！测试成功！



1 示例

2 a)assertThat( n, allOf( greaterThan(1), lessThan(15) ) );

3 assertThat( n, anyOf( greaterThan(16), lessThan(8) ) );

4 assertThat( n, anything() );

5 assertThat( str, is( "bjsxt" ) );

6 assertThat( str, not( "bjxxt" ) );

7

8 b)assertThat( str, containsString( "bjsxt" ) );

9 assertThat( str, endsWith("bjsxt" ) );

10 assertThat( str, startsWith( "bjsxt" ) );

11 assertThat( n, equalTo( nExpected ) );

12 assertThat( str, equalToIgnoringCase( "bjsxt" ) );

13 assertThat( str, equalToIgnoringWhiteSpace( "bjsxt" ) );

14

15 c)assertThat( d, closeTo( 3.0, 0.3 ) );

16 assertThat( d, greaterThan(3.0) );

17 assertThat( d, lessThan (10.0) );

18 assertThat( d, greaterThanOrEqualTo (5.0) );

19 assertThat( d, lessThanOrEqualTo (16.0) );

20

21 d)assertThat( map, hasEntry( "bjsxt", "bjsxt" ) );

22 assertThat( iterable, hasItem ( "bjsxt" ) );

23 assertThat( map, hasKey ( "bjsxt" ) );

24 assertThat( map, hasValue ( "bjsxt" ) );

View Code

**第三部分**JUnit4 Annotation

几种常见的注释形式：

@Test: 测试方法

a) (expected=XXException.class)

b) (timeout=xxx)

@Ignore: 被忽略的测试方法

@Before: 每一个测试方法之前运行

@After: 每一个测试方法之后运行

@BeforeClass: 所有测试开始之前运行

@AfterClass: 所有测试结束之后运行

分别进行解释

1) @Test注解表明下面的方法是一个测试方法，a),  b)两种形式，比如@Test(expected=java.lang.ArithmeticException.class,timeout=100)a是在测试出现异常的情况下告知出现的异常信息，类似与try-catch中的e.printstacktrace() 方法。b中的timeout=100，运行时间限制在100ms以内

2)@ignore: 被忽略的测试方法(就是测试的时候跳过ignor标记的模块或方法)   
有时候某些方法还不具备测试的条件，暂时还不能测试或者某些方法已经不需要再做测试了，这就可以进行忽略的操作了。   
有时方法的测试条件还没满足，整个项目还差一个模块，则可以采用该方法假定测试条件成立。

3)@after和@before

把代码改成这个样子：为方便起见先把那个divide方法删掉



1 public class TTest {

2 @Before

3 public void before() {

4 System.out.println("befor");

5 }

6

7 @Test

8 public void testAdd() {

9 int z = new T().add(2, 4);

10 assertThat(z, is(8));

11 // 判断z==6,以往的assert

12 // assertEquals(6, z);

13 // assertTrue(z<3);

14 // int a=8/0;

15 }

16

17 @After

18 public void after() {

19 System.out.println("after");

20 }

21

22 }

View Code

在上面需要添加这两句：

1 import org.junit.After;

2 import org.junit.Before;

测试一下的testAdd()方法，控制台输出：

befor

after

说明@Before在每一个测试方法(@Test方法)之前运行 @After:在每一个测试方法之后运行。它们两个的应用场合： 有些方法需要执行的时候需要一些先决条件，比如打开某文件、获取资源，搭建环境，执行完之后需要关闭文件、释放资源、卸载环境这就需要before和after操作。

4)@BeforeClass;@AfterClass，它们两个都是静态的方法。继续改写代码：



1 public class TTest {

2 @BeforeClass

3 public static void beforeClass(){

4 System.out.println("before class..");

5 }

6 @Before

7 public void before() {

8 System.out.println("befor");

9 }

10

11 @Test

12 public void testAdd() {

13 int z = new T().add(2, 4);

14 assertThat(z, is(6));

15 // 判断z==6,以往的assert

16 // assertEquals(6, z);

17 // assertTrue(z<3);

18 // int a=8/0;

19 }

20

21 @After

22 public void after() {

23 System.out.println("after");

24 }

25 @AfterClass

26 public static void afterClass(){

27 System.out.println("after class..");

28 }

29

30 }

View Code

当然也要加上：

1 import org.junit.BeforeClass;

2 import org.junit.AfterClass;

输出结果：

before class..

befor

after

after class..

这就说明了@BeforeClass 所有测试开始之前运行;@AfterClass: 所有测试结束之后运行