

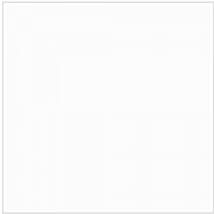


java 是个什么玩意

JavaEE JavaSE JavaME Server DB

目录视图 摘要视图 RSS 订阅

个人资料



kswy521

访问：8120次
积分：96
等级：BLOG 1
排名：千里之外

原创：1篇 转载：0篇
译文：0篇 评论：3条

文章搜索

文章分类

- C++ (0)
- DB2 (0)
- java (1)
- javaEE (0)
- javaME (0)
- javaScript (0)
- javaSE (0)
- Linux (0)
- Oracle (0)
- SQL (0)
- Tomcat (0)
- WebLogic (0)
- Websphere (0)

文章存档

2009年03月 (1)

阅读排行

junit.framework.TestCase (7797)

评论排行

junit.framework.TestCase (3)

推荐文章

junit.framework.TestCase

标签：junit object string null exception 测试

2009-03-10 14:41 7801人阅读 评论(2) 收藏 举报

分类： java

版权声明：本文为博主原创文章，未经博主允许不得转载。

目录(?) [+]

- * CSDN日报20170824——《你为什么跳槽？真正原因找到了吗？》
- * Linux的任督二脉：进程调度和内存管理
- * 秒杀系统的一点思考
- * TCP网络通讯如何解决分包粘包问题
- * 技术与技术人员的价值
- * GitChat: 人工智能 | 除了深度学习，机器翻译还需要啥？

最新评论

junit.framework.TestCase
xiaohu_zeng: very goooooo!

junit.framework.TestCase
BenW1988: 棒

junit.framework.TestCase
liubo9418:

8.1 JUnit介绍

JUnit是一个开源的Java单元测试框架，由 Erich Gamma 和 Kent Beck 开发完成。

8.1.1 JUnit简介

JUnit主要用来帮助开发人员进行Java的单元测试，其设计非常小巧，但功能却非常强大。

下面是JUnit一些特性的总结：

- 提供的API可以让开发人员写出测试结果明确的可重用单元测试用例。
- 提供了多种方式来显示测试结果，而且可以扩展。
- 提供了单元测试批量运行的功能，而且可以和Ant很容易地整合。
- 对不同性质的被测对象，如Class,JSP,Servlet等，JUnit有不同的测试方法。

8.1.2 为什么要使用JUnit

以前，开发人员写一个方法，如下代码所示：

```
//***** AddAndSub.java*****

public Class AddAndSub {

    public static int add(int m, int n) {

        int num = m + n;

        return num;

    }

    public static int sub(int m, int n) {

        int num = m - n;

        return num;

    }

}
```

如果要对AddAndSub类的add和sub方法进行测试，通常要在main里编写相应的测试方法，如下代码所示：

```
//***** MathComputer.java*****

public Class AddAndSub {

    public static int add(int m, int n) {

        int num = m + n;

        return num;

    }

    public static int sub(int m, int n) {

        int num = m - n;

        return num;

    }

}
```

```
public static void main(String args[]) {  
  
    if (add (4, 6) == 10)) {  
  
        System.out.println("Test Ok");  
  
    } else {  
  
        System.out.println("Test Fail");  
  
    }  
  
    if (sub (6, 4) ==2)) {  
  
        System.out.println("Test Ok");  
  
    } else {  
  
        System.out.println("Test Fail");  
  
    }  
  
    }  
}
```

从上面的测试可以看出，业务代码和测试代码放在一起，对于复杂的业务逻辑，一方面代码量会非常庞大，另一方面测试代码会显得比较凌乱，而JUnit就能改变这样的状况，它提供了更好的方法来进行单元测试。使用JUnit来测试前面代码的示例如下：

```
//***** TestAddAndSub.java*****  
  
import junit.framework.TestCase;  
  
public Class TestAddAndSub extends TestCase {  
  
    public void testadd() {  
  
        //断言计算结果与10是否相等  
  
        assertEquals(10, AddAndSub.add(4, 6));  
  
    }  
  
    public void testsub() {  
  
        //断言计算结果与2是否相等  
  
        assertEquals(2, AddAndSub.sub(6, 4));  
  
    }  
  
    public static void main(String args[]){  
  
        junit.textui.TestRunner.run(TestAddAndSub .class);    }  
}
```

这里先不对JUnit的使用方法进行讲解，从上可以看到，测试代码和业务代码分离开，使得代码比较清晰，如果将JUnit放在Eclipse中，测试起来将会更加方便。

8.2 建立JUnit的开发环境

为了不使读者在环节配置上浪费太多时间，这里将一步一步地讲解如何下载和配置JUnit。具体步骤如下：

8.2.1 下载JUnit

从www.junit.org可以进入到JUnit的首页，JUnit的首页画面如图8.1所示。

本书使用的版本是4.3版本，单击“JUnit4.3.zip”即可进入下载JUnit的画面，如图8.2所示。

下载JUnit4.3.zip，下载后解压缩即可。

8.2.2 配置JUnit

下载JUnit4.3.zip完毕，并解压缩到D盘根目录下后，即可开始配置环境变量。用前面介绍的设定系统变量的方法，设定ClassPath，ClassPath=***;D:"junit"junit.jar，如图8.3所示。

图8.1 JUnit的首页画面

图8.2 下载JUnit的画面

查看是否配置好JUnit，在类里添加如下语句：

```
import junit.framework.TestCase;
```

图8.3 设定系统变量ClassPath

如果编译没有错误，则说明配置成功。

8.3 JUnit的使用方法

JUnit的使用非常简单，共有3步：第一步、编写测试类，使其继承TestCase；第二步、编写测试方法，使用test+xxx的方式来命名测试方法；第三步、编写断言。如果测试方法有公用的变量等需要初始化和销毁，则可以使用setUp,tearDown方法。

8.3.1 继承TestCase

如果要使用JUnit，则测试类都必须继承TestCase。当然目前的最新版JUnit是不需要继承它的，但是TestCase类就没有用了，它仍然是JUnit工作的基础。这里先讲述继承TestCase类的方式，稍后再讲。

下面是前面使用JUnit进行测试AddAndSub类的代码，这里进行详细的分析：

```
//***** TestAddAndSub.java*****

import junit.framework.TestCase;

public Class TestAddAndSub extends TestCase {

    public void testadd() {

        //断言计算结果与10是否相等

        assertEquals(10, AddAndSub.add(4, 6));

    }

    public void testsub() {

        //断言计算结果与2是否相等

        assertEquals(2, AddAndSub.sub(6, 4));

    }

    public static void main(String args[]){

        junit.textui.TestRunner.run(TestAddAndSub .class);  }

}
```

代码说明：

— 这里继承TestCase，表示该类是一个测试类。

— 然后使用junit.textui.TestRunner.run方法来执行这个测试类。

这里给出TestCase的源代码：

```
//***** TestCase.java*****

package junit.framework;

import java.lang.reflect.InvocationTargetException;

import java.lang.reflect.Method;

import java.lang.reflect.Modifier;

public abstract class TestCase extends Assert implements Test {

    /**测试案例的名称*/

    private String fName;
```

```
/**构造函数
 *
 public TestCase() {
     fName= null;
 }

/**带参数的构造函数
 *
 public TestCase(String name) {
     fName= name;
 }

/**获取被run执行的测试案例的数量
 *
 public int countTestCases() {
     return 1;
 }

/**创建一个TestResult
 * @see TestResult
 *
 protected TestResult createResult() {
     return new TestResult();
 }

/**执行run方法，返回TestResult
 * @see TestResult
 *
 public TestResult run() {
     TestResult result= createResult();

    圆角矩形: 下面一段代码描述了JUnit如何实现在执行具体的测试方法前，先执行初始化方法，在执行完具体的测试方法后，再执行销毁方法。
     run(result);

     return result;
 }

/**执行run方法，参数为TestResult
 *
 public void run(TestResult result) {
     result.run(this);
 }
```

```
/**执行测试方法，包括初始化和销毁方法

 * @throws Throwable if any exception is thrown

 */

public void runBare() throws Throwable {

    Throwable exception= null;

    setUp();

    try {

        runTest();

    } catch (Throwable running) {

        exception= running;

    }

    finally {

        try {

            tearDown();

        } catch (Throwable tearingDown) {

            if (exception == null) exception= tearingDown;

        }

    }

    if (exception != null) throw exception;

}

/**执行测试方法

 * @throws Throwable if any exception is thrown

 */

protected void runTest() throws Throwable {

    assertNotNull("TestCase.fName cannot be null", fName); // Some VMs crash when calling
    getMethod(null,null);

    Method runMethod= null;

    try {

        //利用反射机制

        runMethod= getClass().getMethod(fName, (Class[])null);

    } catch (NoSuchMethodException e) {

        fail("Method ""+fName+"" not found");

    }

    if (!Modifier.isPublic(runMethod.getModifiers())) {

        fail("Method ""+fName+"" should be public");

    }

}
```



```
//利用反射机制
```

```
try {  
  
    runMethod.invoke(this);  
  
}  
  
catch (InvocationTargetException e) {  
  
    e.fillInStackTrace();  
  
    throw e.getTargetException();  
  
}
```

圆角矩形: 下面一段代码定义了要想实现初始化和销毁方法，需继承这两个方法。 }

```
catch (IllegalAccessException e) {  
  
    e.fillInStackTrace();  
  
    throw e;  
  
}  
  
}
```

```
/**测试前的初始化方法
```

```
*/
```

```
protected void setUp() throws Exception {  
  
}
```

```
/**测试后的销毁方法
```

```
*/
```

```
protected void tearDown() throws Exception {  
  
}
```

```
/**返回测试案例的名称
```

```
* @return the name of the TestCase
```

```
*/
```

```
public String getName() {  
  
    return fName;  
  
}
```

```
/**设定测试案例的名称
```

```
* @param name the name to set
```

```
*/
```

```
public void setName(String name) {  
  
    fName= name;  
  
}
```

```
}
```

代码说明：

— 该类继承了Assert 类，实现了Test接口。

— 可以看出，TestCase类正是通过runBare实现了在测试方法前初始化相关变量和环境，在测试方法后销毁相关变量和环境。

8.3.2 编写测试方法

测试方法名要以test+方法名来命名，当然最新版的JUnit支持直接以方法名来命名测试方法。这是通过TestCase类里的runTest方法来实现的，主要利用了Java的反射机制，runTest方法的代码如下：

```
protected void runTest() throws Throwable {

    assertNotNull("TestCase fName cannot be null", fName); // Some VMs crash when calling
    getMethod(null,null);

    Method runMethod= null;

    try {

        // 获取要测试的方法

        runMethod= getClass().getMethod(fName, (Class[])null);

    } catch (NoSuchMethodException e) {

        fail("Method '"+fName+"' not found");

    }

    //判断要测试的方法是否为公用方法

    if (!Modifier.isPublic(runMethod.getModifiers())) {

        fail("Method '"+fName+"' should be public");

    }

    //Java的反射机制

    try {

        runMethod.invoke(this);

    }

    //抛出调用异常

    catch (InvocationTargetException e) {

        e.fillInStackTrace();

        throw e.getTargetException();

    }

    catch (IllegalAccessException e) {

        e.fillInStackTrace();

        throw e;

    }

}
```

8.3.3 编写断言

JUnit主要有以下断言：

- assertEquals（期望值，实际值），检查两个值是否相等。
- assertEquals（期望对象，实际对象），检查两个对象是否相等，利用对象的equals()方法进行判断。
- assertEquals（期望对象，实际对象），检查具有相同内存地址的两个对象是否相等，利用内存地址进行判断，注意和上面assertEquals方法的区别。
- assertEquals（期望对象，实际对象），检查两个对象是否不相等。
- assertEquals（对象1，对象2），检查一个对象是否为空。
- assertEquals（对象1，对象2），检查一个对象是否不为空。
- assertEquals(布尔条件)，检查布尔条件是否为真。
- assertEquals(布尔条件)，检查布尔条件是否为假。

这些断言主要定义在JUnit的Assert类里，Assert类的示例代码如下：

```
//***** Assert.java*****

package junit.framework;

/**一系列的断言方法
 */

public class Assert {

    /**构造函数
     */

    protected Assert() {

    }

    /**断言是否为真，带消息
     */

    static public void assertTrue(String message, boolean condition) {

        if (!condition)

            fail(message);

    }

    /**断言是否为真
     */

    static public void assertTrue(boolean condition) {

        assertTrue(null, condition);

    }

    /**断言是否为假，带消息
     */

    static public void assertFalse(String message, boolean condition) {

        assertTrue(message, !condition);

    }

    /**断言是否为假
```

```
*/

static public void assertFalse(boolean condition) {

    assertFalse(null, condition);

}
```

圆角矩形: 下面一段代码描述了如何在JUnit中实现判断是否相等的方法，这些方法要实现的内容相同，只是参数不同，从而实现了可以针对不同类型的数据来判断是否相等的功能。

/**断言是否为失败

```
*/

static public void fail(String message) {

    throw new AssertionError(message);

}
```

/**断言是否为失败

```
*/

static public void fail() {

    fail(null);

}
```

/**是否相等的断言，带消息Object

```
*/

static public void assertEquals(String message, Object expected, Object actual) {

    if (expected == null && actual == null)

        return;

    if (expected != null && expected.equals(actual))

        return;

    failNotEquals(message, expected, actual);

}
```

/**是否相等的断言，Object

```
*/

static public void assertEquals(Object expected, Object actual) {

    assertEquals(null, expected, actual);

}
```

/**是否相等的断言，带消息String

```
*/

static public void assertEquals(String message, String expected, String actual) {

    if (expected == null && actual == null)

        return;

    if (expected != null && expected.equals(actual))
```

```
        return;

        throw new ComparisonFailure(message, expected, actual);
    }

    /**是否相等的断言，String

    */

    static public void assertEquals(String expected, String actual) {

        assertEquals(null, expected, actual);
    }

    /**是否相等的断言，带消息double

    */

    static public void assertEquals(String message, double expected, double actual, double delta) {

        if (Double.compare(expected, actual) == 0)

            return;

        if (!(Math.abs(expected-actual) <= delta))

            failNotEquals(message, new Double(expected), new Double(actual));
    }

    /**是否相等的断言，double

    */

    static public void assertEquals(double expected, double actual, double delta) {

        assertEquals(null, expected, actual, delta);
    }

    /**是否相等的断言，带消息float

    */

    static public void assertEquals(String message, float expected, float actual, float delta) {

        if (Float.compare(expected, actual) == 0)

            return;

        if (!(Math.abs(expected - actual) <= delta))

            failNotEquals(message, new Float(expected), new Float(actual));
    }

    /**是否相等的断言，float

    */

    static public void assertEquals(float expected, float actual, float delta) {

        assertEquals(null, expected, actual, delta);
    }

    /**是否相等的断言，带消息long

    */
```

```
static public void assertEquals(String message, long expected, long actual) {

    assertEquals(message, new Long(expected), new Long(actual));

}

/**是否相等的断言 , long

*/

static public void assertEquals(long expected, long actual) {

    assertEquals(null, expected, actual);

}

/**是否相等的断言 , 带消息boolean

*/

static public void assertEquals(String message, boolean expected, boolean actual) {

    assertEquals(message, Boolean.valueOf(expected), Boolean.valueOf(actual));

}

/**是否相等的断言 , boolean

*/

static public void assertEquals(boolean expected, boolean actual) {

    assertEquals(null, expected, actual);

}

/**是否相等的断言 , 带消息byte

*/

static public void assertEquals(String message, byte expected, byte actual) {

    assertEquals(message, new Byte(expected), new Byte(actual));

}

/**是否相等的断言 , byte

*/

static public void assertEquals(byte expected, byte actual) {

    assertEquals(null, expected, actual);

}

/**是否相等的断言 , 带消息char

*/

static public void assertEquals(String message, char expected, char actual) {

    assertEquals(message, new Character(expected), new Character(actual));

}

/**是否相等的断言 , char

*/

static public void assertEquals(char expected, char actual) {
```

```

        assertEquals(null, expected, actual);
    }

    /**是否相等的断言，带消息short
     */

    static public void assertEquals(String message, short expected, short actual) {
        assertEquals(message, new Short(expected), new Short(actual));
    }

    /**是否相等的断言，short
     */

    static public void assertEquals(short expected, short actual) {
        assertEquals(null, expected, actual);
    }

    /**是否相等的断言，带消息int
     */

    static public void assertEquals(String message, int expected, int actual) {
        assertEquals(message, new Integer(expected), new Integer(actual));
    }

    /**是否相等的断言，int
     */

    static public void assertEquals(int expected, int actual) {
        assertEquals(null, expected, actual);
    }
}

```

圆角矩形: 下面一段代码描述了JUnit中如何实现判断是否为null的方法，这些方法的功能相同，只是一个带消息，一个不带消息。

```

    /**是否不为null的断言 Object
     */

    static public void assertNotNull(Object object) {
        assertNotNull(null, object);
    }

    /**是否不为null的断言，带消息Object
     */

    static public void assertNotNull(String message, Object object) {
        assertTrue(message, object != null);
    }

    /**是否为null的断言Object
     */

```

圆角矩形: 下面一段代码描述了JUnit中如何实现判断是否相同的方法，这些方法要实现的内容相同，只是参数不同。

```

    static public void assertNull(Object object) {

        assertNull(null, object);
    }

    /**是否为null的断言，带消息Object
     */

    static public void assertNull(String message, Object object) {

        assertTrue(message, object == null);
    }

    /**是否相同的断言，带消息*/

    static public void assertEquals(String message, Object expected, Object actual) {

        if (expected == actual)

            return;
    }
}

```

```
failNotSame(message, expected, actual);

}

/**是否相同的断言，Object
 */

static public void assertSame(Object expected, Object actual) {

    assertSame(null, expected, actual);

}

/**是否不相同的断言，带消息
 */

static public void assertNotSame(String message, Object expected, Object actual) {

    if (expected == actual)

        failSame(message);

}

/**是否不相同的断言Object
 */

static public void assertNotSame(Object expected, Object actual) {

    assertNotSame(null, expected, actual);

}

/**相同时失败
 */

static public void failSame(String message) {

    String formatted= "";

    if (message != null)

        formatted= message+" ";

    fail(formatted+"expected not same");

}

/**不相同失败
 */

static public void failNotSame(String message, Object expected, Object actual) {

    String formatted= "";

    if (message != null)

        formatted= message+" ";

    fail(formatted+"expected same:<"+expected+"> was not:<"+actual+">");

}

/**不相等失败
 */
```



```
static public void failNotEquals(String message, Object expected, Object actual) {  
  
    fail(format(message, expected, actual));  
  
}  
  
/**格式化消息  
  
*/  
  
public static String format(String message, Object expected, Object actual) {  
  
    String formatted= "";  
  
    if (message != null)  
  
        formatted= message+" ";  
  
    return formatted+"expected:<"+expected+"> but was:<"+actual+">";  
  
}  
}
```

从上述代码中，读者可以研读JUnit中有关断言的实现方式，其实，最终都是使用后面的几个static方法实现的。

8.4 JUnit的新特性

Java 5的发布为JUnit带来了新的特性。自JUnit 4.0之后，JUnit大量使用了annotations特性，使编写单元测试变得更加简单。

8.4.1 改变测试方法的命名方式

前面讲过，使用JUnit 4.0以上版本可以不用遵循以前JUnit约定的测试方法命名方法，以前命名方法的示例代码如下：

```
//***** TestAddAndSub.java*****

import junit.framework.TestCase;

public class TestAddAndSub extends TestCase {

    public void testadd() {

        //断言计算结果与10是否相等

        assertEquals(10, AddAndSub.add(4, 6));

    }

    public void testsub() {

        //断言计算结果与2是否相等

        assertEquals(2, AddAndSub.sub(6, 4));

    }

    public static void main(String args[]){

        junit.textui.TestRunner.run(TestAddAndSub .class);  }

}
```

JUnit 4.0以上版本的命名方式，是在测试方法前使用@Test注释，示例代码如下：

```
//***** TestAddAndSub.java*****

import junit.framework.TestCase;

import org.junit.*;

public class TestAddAndSub extends TestCase {

    @Test public void add() {

        //断言计算结果与10是否相等

        assertEquals(10, AddAndSub.add(4, 6));

    }

    @Test public void sub() {

        //断言计算结果与2是否相等

        assertEquals(2, AddAndSub.sub(6, 4));

    }

}
```

这个时候，测试方法的命名将不再重要，开发人员可以按照自己的命名方式来命名。

8.4.2 不再继承TestCase

新版本的JUnit将不再强制继承TestCase，但需要import org.junit.Assert来实现断言，示例代码如下：

```
//***** TestAddAndSub.java*****

import static org.junit.Assert.assertEquals;

import org.junit.*;

public class TestAddAndSub{

    @Test public void add() {

        //断言计算结果与10是否相等

        assertEquals(10, AddAndSub.add(4, 6));

    }

    @Test public void sub() {

        //断言计算结果与2是否相等

        assertEquals(2, AddAndSub.sub(6, 4));

    }

}
```

8.4.3 改变初始化和销毁方式

以前，JUnit使用SetUp和TearDown方法来进行初始化和销毁动作，JUnit 4.0以上版本将不再强制使用SetUp和TearDown方法来进行初始化和销毁，原来使用SetUp和TearDown方法的示例代码如下：

```
//***** TestAddAndSub.java*****

import junit.framework.TestCase;

public class TestAddAndSub extends TestCase {

    private int m = 0;

    private int n = 0;

    //初始化

    protected void setUp() {

        m = 4;

        n = 6;

    }

    public void testadd() {

        //断言计算结果与10是否相等

        assertEquals(10, AddAndSub.add(m, n));

    }

    public void testsub() {

        //断言计算结果与2是否相等
```

```
        assertEquals(2, AddAndSub.sub(n, m));

    }

    //销毁

    protected void tearDown() {

        m = 0;

        n = 0;

    }

}
```

不使用SetUp和TearDown方法的示例代码如下：

```
/** ***** TestAddAndSub.java ***** */

import static org.junit.Assert.assertEquals;

import org.junit.*;

public class TestAddAndSub {

    protected int m = 0;

    protected int n = 0;

    //初始化

    @Before public void init() {

        m = 4;

        n = 6;

    }

    @Test public void add() {

        //断言计算结果与10是否相等

        assertEquals(10, AddAndSub.add(m, n));

    }

    @Test public void sub() {

        //断言计算结果与2是否相等

        assertEquals(2, AddAndSub.sub(n, m));

    }

    //销毁

    @After public void destory() {

        m = 0;

        n = 0;

    }

}
```

上面示例中的初始化和销毁都是针对一个方法来说的，每个方法执行前都要进行初始化，执行完毕都要进行销毁。而JUnit的最新版本则提供了新的特性，针对类进行初始化和销毁。也就是说，该类中的方法只进行一次初始化和销

关闭

毁，方法就是使用@Before和@After，示例代码如下：

```
//***** TestAddAndSub.java*****

import static org.junit.Assert.assertEquals;

import org.junit.*;

public class TestAddAndSub {

    protected int m = 0;

    protected int n = 0;

    //初始化

    @BeforeClass public void init() {

        m = 4;

        n = 6;

    }

    @Test public void add() {

        //断言计算结果与10是否相等

        assertEquals(10, AddAndSub.add(m, n));

    }

    @Test public void sub() {

        //断言计算结果与2是否相等

        assertEquals(2, AddAndSub.sub(n, m));

    }

    //销毁

    @AfterClass public void destory() {

        m = 0;

        n = 0;

    }

}
```

上述初始化和销毁动作，只执行一次即可。

8.4.4 改变异常处理的方式

以前，使用JUnit进行单元测试时，如果遇到异常情况，需要使用try...catch的形式来捕捉异常，示例代码如下：

```
//***** TestAddAndSub.java*****

import junit.framework.TestCase;

public class TestAddAndSub extends TestCase {

    private int m = 0;

    private int n = 0;

    //初始化
```

```
protected void setUp() {

    m = 4;

    n = 6;

}

public void testadd() {

    //断言计算结果与10是否相等

    assertEquals(10, AddAndSub.add(m, n));

}

public void testsub() {

    //断言计算结果与2是否相等

    assertEquals(2, AddAndSub.sub(n, m));

}

public void testdiv() {

    //断言除数为0

    try {

        int n = 2 / 0;

        fail("Divided by zero!");

    }

    catch (ArithmeticException success) {

        assertNotNull(success.getMessage());

    }

}

//销毁

protected void tearDown() {

    m = 0;

    n = 0;

}

}

JUnit4.0以后的版本将不再使用try...catch的方式来捕捉异常，示例代码如下：

//***** TestAddAndSub.java*****

import static org.junit.Assert.assertEquals;
import org.junit.*;
public class TestAddAndSub {
    protected int m = 0;
    protected int n = 0;
    //初始化
    @Before public void init() {
        m = 4;
```

```
n = 6;
}
@Test public void add() {
    //断言计算结果与10是否相等
    assertEquals(10, AddAndSub.add(m, n));
}
@Test public void sub() {
    //断言计算结果与2是否相等
    assertEquals(2, AddAndSub.sub(n, m));
}
@Test t(expected=ArithmeticException.class) public void div() {
    //断言除数是否为0
    int n = 2 / 0;
}
//销毁
@After public void destory() {
    m = 0;
    n = 0;
}
}
```

当然，JUnit还有许多新的特性，限于篇幅原因，这里只对比较重要的特性进行讲解，其余将不再多讲，想要了解的读者可以到JUnit的相关网站进行学习。

8.5 小结

本章首先讲述了JUnit的下载和安装，接着又讲解了JUnit的相关知识，最后讲解了JUnit的新特性。

JUnit对开发人员进行大规模的单元测试来说，是非常有用的，但对于大量的代码如何来管理就只有靠CVS了。CVS是一个**版本控制**系统，主要用来管理开发人员代码的历史，下一章主要讲如何使用CVS。

顶 踩
1 0

相关文章推荐

- JUnit写TestCase
- 【直播】机器学习&数据挖掘7周实训-韦玮
- JUnit核心——测试类（ TestCase ）、测试集（ Tes...
- 【直播】如何高速通过软考--任铎
- JUnit 介绍--- junit.framework.TestCase
- 【直播】打通Linux脉络 进程、线程和调度--宋宝华
- JUnit TestCase
- 【套餐】Java高级程序员专业学习路线--肖海鹏

- JUnit3 junit.framework 单元测试，简单实例说明.
- 【课程】C++语言基础-贺利坚
- JUnit测试出现异常：Exception in thread "main" ja...
- 【课程】深度学习基础与TensorFlow实践-AI100
- java.lang.NoSuchMethodError: org.junit.runner.D...
- Android Junit
- maven+spring+junit测试程序时时出现NoSuchMet...
- Eclipse中怎么使用junit测试

查看评论

3楼 [xiaohu_zeng](#) 2012-12-11 11:35发表



very gooooood!

2楼 [BenW1988](#) 2012-03-07 16:44发表



棒

1楼 [liubo9418](#) 2010-04-17 15:17发表




[e03]

发表评论

用户名：[qq_36596145](#)

评论内容：



提交

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

[公司简介](#) | [招贤纳士](#) | [广告服务](#) | [联系方式](#) | [版权声明](#) | [法律顾问](#) | [问题报告](#) | [合作伙伴](#) | [论坛反馈](#)

[网站客服](#) [杂志客服](#) [微博客服](#) [webmaster@csdn.net](#) 400-660-0108 | 北京创新乐知信息技术有限公司 版权所有 | 江苏知之为计算机有限公司 |

江苏乐知网络技术有限公司

京 ICP 证 09002463 号 | Copyright © 1999-2017, CSDN.NET, All Rights Reserved 