链滴

Java 单元测试利器 JUnit

作者：leekeggs

原文链接：https://ld246.com/article/1559134842422

来源网站：链滴

单元测试是软件开发中非常重要的一个环节，对我们编程人员来说，要对自己的代码负责，单元测试校验代码行为是否符合期望的有效手段。JUnit是用Java语言编写的一个单元测试工具，它具有非常大的功能，使用起来也非常方便。

1. JUnit快速入门

如果你是使用maven管理项目，那么只需在pom文件添加下面的依赖即可引入junit包。

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

下面我们开始介绍如何使用JUnit进行测试吧。下面是一段代码：这是一个非常简单的工具类，用来理字符串的。

```
public class StringUtils {
    /**
     * 将大写字符串全部转换为小写
     * @param str 要转换的字符串
     * @return 转换后的字符串
     */
    public static String toLowerCase(String str){
        if(str==null){
            throw new NullPointerException();
        }
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < str.length(); i++) {
            char ch=str.charAt(i);
            if(ch>='a' && ch <= 'z'){
                sb.append(ch);
            } else if(ch>='A' && ch<='Z'){
                sb.append((char)(ch+32));
            }else {
                throw new RuntimeException("字母必须在[a-z]和[A-Z]中");
            }
        }
        return sb.toString();
    }
}
```

现在我们开始编写单元测试代码，对上面的**toLowerCase**方法进行校验。

```
public class TestStringUtils {

    /**
     * 测试大写字符组成的字符串
     */
    @Test
    public void testUpperCase(){
        Assert.assertEquals("username",StringUtils.toLowerCase("USERNAME"));
```

```
    }

    /**
     * 测试小写字符组成的字符串
     */
    @Test
    public void testLowerCase(){
        Assert.assertEquals("username",StringUtils.toLowerCase("username"));
    }

    /**
     * 测试大写和小写字符组成的字符串
     */
    @Test
    public void testLowerAndUpperCase(){
        Assert.assertEquals("username",StringUtils.toLowerCase("userNAME"));
    }

    /**
     * 测试大写以外字符组成的字符串，是否抛出RuntimeException异常
     */
    @Test(expected = RuntimeException.class)
    public void testNonLetter(){
        StringUtils.toLowerCase("user123NAME");
    }
}
```

测试类建议以Test开头，这样可以很明显识别出这是一个测试类。上面我们创建了一个测试类estStringUtils，包含四个使用**@Test**注解的方法，用来测试toLowerCase方法。

分别运行上述三个测试方法，就能看到测试通过了，说明我们的代码没有错误。

请牢记一条 JUnit 最佳实践：测试任何可能的错误。单元测试不是用来证明您是对的，而是为了证明没有错。

测试方法的书写规范：

- (1). 测试方法必须使用org.junit.Test注解
- (2). 测试方法必须使用public void修饰，而且不能有任何参数

细心的同学可能会发现，测试方法testNonLetter的注解@Test多了一个expected参数，下面我们深研究一下junit的高级用法。

2. JUnit高级用法

2.1 异常和时间测试

注解org.junit.Test有两个很有用的参数：expected和timeout。

参数expected代表测试方法期望抛出特定的异常，如果运行测试没有抛出该异常，那么JUnit就会告我们测试失败，这为我们验证异常情况是否抛出指定异常提供了便利。比如在上面的例子中，我们测了在传入大小写以外的字符时，是否抛出RuntimeException异常。

参数timeout用于指定测试方法能运行的最长时间是多少（单位：毫秒），如果超过指定时间测试方仍未结束，那么JUnit就会告诉我们测试失败，该参数对于性能测试会有一定的帮助。

2.2 @Before和@After注解

我们经常会碰到这样一种情况，多个测试方法需要一些公共的资源，而我们又不想重复编写代码来处这些公共资源，这会造成大量重复代码。那JUnit有没有为我们提供什么方式来处理这种情况呢，答是yes，@Before和@After就是用来做这事的，设置公共资源。

@Before注解的方法会在@Test注解的方法之前运行，因此我们可以使用该注解做一些初始化操作

@After注解的方法会在@Test注解的方法之后运行，因此我们可以用该注解做一些清理工作。

注意：@Before注解的方法会在每个@Test注解的方法之前运行，@After注解的方法会在每个@Tes注解的方法之后运行

@Before和@After注解的方法必须使用public void修饰，而且不能带有任何参数。

2.3 @BeforeClass和@AfterClass注解

和@Before和@After类似，@BeforeClass和@AfterClass注解也是用来设置公共资源的。不同的是@BeforeClass注解的方法在当前测试类中所有方法之前运行，@AfterClass注解的方法在当前测试中所有方法之后运行，并且都只运行一次。这在设置一些耗时的公共资源时非常有用（比如说数据库接），提升测试效率。

2.4 测试运行器

JUnit中所有的测试方法都是有测试运行器来负责的，JUnit提供了默认的测试运行器，但是我们也可通过继承org.junit.runner.Runner自定义测试运行器。

在运行一个测试方法时，如果没有显示指定测试运行器，那么JUnit就使用默认的测试运行器。我们以使用@RunWith注解显示指定测试运行器，比如下面一段代码

```
@RunWith(CustomRunner.class)
public class TestStringUtils {
}
```

2.5 测试套件

在实际项目中，我们编写的测试类肯定不只一个，当测试类的数量很多时，一个一个测试肯定时不现的。JUnit提供了一种批量运行测试类的方法：测试套件。我们可以在一个测试套件中添加多个测试，一次运行多个测试类。定义测试套件规则如下：

- (1). 创建一个空类，使用@RunWith和@Suite.SuiteClasses注解修饰，作为测试套件的入口
- (2). 使用org.junit.runners.Suite作为测试运行器
- (3). 将测试类组成的数组作为@Suite.SuiteClasses的参数
- (4). 必须使用public修改该空类，且不能存在带有参数的构造函数

```
@RunWith(Suite.class)
@Suite.SuiteClasses({FirstTest.class,SecondTest.class})
public class MyTestSuit {
```

```
}
```

上面我们定义了一个测试套件类MyTestSuit，包含两个测试类：FristTest和SecondTest。直接运行类，就会看到FristTest和SecondTest方法中的测试类都被运行了。

2.6 参数化测试

回顾一下我们的测试类TestStringUtils中的前三个方法，他们代码结构基本相同，除了期望值和目标不同以外。那么我们是否有一种更好的方法，能把代码中相同的结构抽离出来，减少代码的冗余量呢从JUnit4.0开始引入了参数化测试概念，很好地解决了我们刚才提到的问题。

定义参数化测试类的过程如下：

- (1). 定义一个测试类，使用@RunWith注解修饰，其参数值为 **Parameterized.class**
- (2). 在该测试类中声明一个public static修饰的方法，其返回值是Collection，并在该方法中初始化要测试的参数对
- (3). 在该测试类中声明几个变量，用于保存期望数据和实际的测试数据
- (4). 为该类声明一个public修饰的有参的构造参数，并在其中为(3)中声明的变量赋值
- (5). 编写测试方法，使用(3)中定义的变量进行测试

```
@RunWith(Parameterized.class)
public class TestStringUtilsParameter {
    private String expected;
    private String actual;

    @Parameterized.Parameters
    public static Collection param(){
        return Arrays.asList(new String[][]{{"username","USERNAME"},{"username","username"},{"
sername","userName"}});
    }

    public TestStringUtilsParameter(String expected, String actual) {
        this.expected = expected;
        this.actual = actual;
    }

    @Test
    public void testStringUtils(){
        Assert.assertEquals(expected,StringUtils.toLowerCase(actual));
    }
}
```

看一下上面这个测试类和TestStringUtils相比，是不是简洁多了，没有冗余的代码结构，而且当我们要增加测试情况时，只需要添加相应的数组即可。