


# Android单元测试(三)：JUnit单元测试框架的使用

 [chriszou.com/2016/04/18/android-unit-testing-junit](http://chriszou.com/2016/04/18/android-unit-testing-junit)

April 18, 2016

我们写单元测试，一般都会用到一个或多个单元测试框架，在这里，我们介绍一下 JUnit4 这个测试框架。这是 Java 界用的最广泛，也是最基础的一个框架，其他的很多框架，包括我们后面会看到的 Robolectric，都是基于或兼容 JUnit4 的。

然而首先要解决的问题是。。。

## 为什么要使用单元测试框架

或者换句话说，单元测试框架能够为我们做什么呢？

从最基本的开始说起，假如我们有这样一个类：

```
public class Calculator {  
    public int add(int one, int another) {  
        // 为了简单起见，暂不考虑溢出等情况。  
        return one + another;  
    }  
    public int multiply(int one, int another) {  
        // 为了简单起见，暂不考虑溢出等情况。  
        return one * another;  
    }  
}
```

如果不用单元测试框架的话，我们要怎么写测试代码呢？我们恐怕得写出下面这样的代码：

```
public class CalculatorTest {  
    public static void main(String[] args) {  
        Calculator calculator = new Calculator();  
        int sum = calculator.add(1, 2);  
        if(sum == 3) {  
            System.out.println("add() works!")  
        } else {  
            System.out.println("add() does not works!")  
        }  
        int product = calculator.multiply(2, 4);  
        if (product == 8) {  
            System.out.println("multiply() works!")  
        } else {  
            System.out.println("multiply() does not works!")  
        }  
    }  
}
```

然后我们再通过某种方式，比如命令行或 IDE，运行这个 `CalculatorTest` 的 `main` 方法，在看着 terminal 的输出，才知道测试是通过还是失败。想想一下，如果我们有很多的类，每个类都有很多方法，那么就要写一堆这样的代码，每个类对于一个含有 `main` 方法的 test 类，同时 `main` 方法里面会有一堆代码。这样既写起来痛苦，跑起来更痛苦，比如说，你怎么样一次性跑所有的测试类呢？所以，一个测试框架为我们做的最基本的事情，就是允许我们按照某种更简单的方式写测试代码，把每一个测试单元写在一个测试方法里面，然后它会自动找出所有的测试方法，并且根据你的需要，运行所有的测试方法，或者是运行单个测试方法，或者是运行部分测试方法等等。

对于上面的 `Calculator` 例子，如果使用 Junit 的话，我们可以按照如下的方式写测试代码：

```
public class CalculatorTest {
    @Test
    public void testAdd() throws Exception {
        Calculator calculator = new Calculator();
        int sum = calculator.add(1, 2);
        Assert.assertEquals(3, sum);
    }
    @Test
    public void testMultiply() throws Exception {
        Calculator calculator = new Calculator();
        int product = calculator.multiply(2, 4);
        Assert.assertEquals(8, product);
    }
}
```

每一个被测试的方法( `add()`, `multiply()` )，写一个对应的测试方法( `testAdd()`, `testMultiply()` )。那 JUnit 怎么知道那些是测试方法，哪些不是呢？这个是通过前面的 `@Test` 注解来标志的，只要有这个注解，JUnit4 就会当做是一个测试方法，方法名其实是可以随意起的。当然，名字还是应该起的更有可读性一点，让人一看就知道，这个测试方法是测试了被测的类的那个方法，或者是测试了那个功能点等等。

除了帮我们找出所有的测试方法，并且方便运行意外，单元测试框架还帮我们做了其他事情。在这个系列的第一篇文章中我们提到，一个测试方法主要包括三个部分：

1. setup
2. 执行操作
3. 验证结果

而一个单元测试框架，可以让我们更方便的写上面的每一步的代码，尤其是第一步和第三步。比如说，在上面的 `CalculatorTest` 中，`testAdd()` 和 `testMultiply()` 都有相同的 setup：`Calculator calculator = new Calculator();`，如果 `Calculator` 还有其他的方法的话，这行代码就得重复更多次，这种 duplication 是没必要的。绝大多数单元测试框架考虑到了这一点，它们知道一个测试类的很多测试方法可能需要相同的 setup，所以为我们提供了便捷方法。对于 JUnit4，是通过 `@Before` 来实现的：

```

public class CalculatorTest {
    Calculator mCalculator;
    @Before
    public void setup() {
        mCalculator = new Calculator();
    }
    @Test
    public void testAdd() throws Exception {
        int sum = mCalculator.add(1, 2);
        assertEquals(3, sum); //为了简洁，往往会static import Assert里面的所有方法。
    }
    @Test
    public void testMultiply() throws Exception {
        int product = mCalculator.multiply(2, 4);
        assertEquals(8, product);
    }
}

```

如果一个方法被 `@Before` 修饰过了，那么在每个测试方法调用之前，这个方法都会得到调用。所以上面的例子中，`testAdd()` 被运行之前，`setup()` 会被调用一次，把 `mCalculator` 实例化，接着运行 `testAdd()`；`testMultiply()` 被运行之前，`setup()` 又会被调用一次，把 `mCalculator` 再次实例化，接着运行 `testMultiply()`。如果还有其他的测试方法，则以此类推。

对应于 `@Before` 的，有一个 `@After`，作用估计你也猜得到，那就是每个测试方法运行结束之后，会得到运行的方法。比如一个测试文件操作的类，那么在它的测试类中，可能 `@Before` 里面需要去打开一个文件，而每个测试方法运行结束之后，都需要去 `close` 这个文件。这个时候就可以把文件 `close` 的操作放在 `@After` 里面，让它自动去执行。

类似的，还有 `@BeforeClass` 和 `@AfterClass`。`@BeforeClass` 的作用是，在跑一个测试类的所有测试方法之前，会执行一次被 `@BeforeClass` 修饰的方法，执行完所有测试方法之后，会执行一遍被 `@AfterClass` 修饰的方法。这两个方法可以用来 `setup` 和 `release` 一些公共资源，需要注意的是，被这两个 annotation 修饰的方法必须是静态的。

前面讲的是单元测试框架对于一个测试方法的第一步“`setup`”，为我们做的事情。而对于第三步“验证结果”，则一般是通过一些 `assert` 方法来完成的。JUnit 为我们提供的 `assert` 方法，多数都在 `Assert` 这个类里面。最常用的那些如下：

`assertEquals(expected, actual)`

验证 `expected` 的值跟 `actual` 是一样的，如果是一样的话，测试通过，不然的话，测试失败。如果传入的是 `object`，那么这里的对比用的是 `equals()`

`assertEquals(expected, actual, tolerance)`

这里传入的 `expected` 和 `actual` 是 `float` 或 `double` 类型的，大家知道计算机表示浮点型数据都有一定的偏差，所以哪怕理论上他们是相等的，但是用计算机表示出来则可能不是，所以这里运行传入一个偏差值。如果两个数的差异在这个偏差值之内，则测试通过，否则测试失败。

`assertTrue(boolean condition)`

验证 `condition` 的值是 `true`

`assertFalse(boolean condition)`

验证 condition 的值是 false

```
assertNull(Object obj)
```

验证 obj 的值是 null

```
assertNotNull(Object obj)
```

验证 obj 的值不是 null

```
assertSame(expected, actual)
```

验证 expected 和 actual 是同一个对象，即指向同一个对象

```
assertNotSame(expected, actual)
```

验证 expected 和 actual 不是同一个对象，即指向不同的对象

```
fail()
```

让测试方法失败

注意：上面的每一个方法，都有一个重载的方法，可以在前面加一个 String 类型的参数，表示如果验证失败的话，将用这个字符串作为失败的结果报告。

比如：

```
assertEquals("Current user Id should be 1", 1, currentUser.id());
```

当 `currentUser.id()` 的值不是 1 的时候，在结果报道里面将显示 "Current user Id should be 1"，这样可以让测试结果更具有可读性，更清楚错误的原因是什么。

比较有意思的是最后一个方法，`fail()`，你或许会好奇，这个有什么用呢？其实这个在很多情况下还是有用的，比如最明显的一个作用就是，你可以验证你的测试代码真的是跑了的。此外，它还有另外一个重要作用，那就是验证某个被测试的方法会正确的抛出异常，不过这点可以通过下面讲到的方法，更方便的做到，所以就不讲了。

这部分相对来说还是很好理解的，不做过多解释。

## JUnit 的其他功能

---

### Ignore 一些测试方法

---

很多时候，因为某些原因（比如正式代码还没有实现等），我们可能想让 JUnit 忽略某些方法，让它在跑所有测试方法的时候不要跑这个测试方法。要达到这个目的也很简单，只需要在要被忽略的测试方法前面加上 `@Ignore` 就可以了，如下：

```
public class CalculatorTest {
    Calculator mCalculator;
    @Before
    public void setup() {
        mCalculator = new Calculator();
    }
    // Omit testAdd() and testMultiply() for brevity
    @Test
    @Ignore("not implemented yet")
    public void testFactorial() {
    }
}
```

### 验证方法会抛出某些异常

---

有的时候，抛出异常是一个方法正确工作的一部分。比如一个除法函数，当除数是 0 的时候，它应该抛出异常，告诉外界，传入的被除数是 0，示例代码如下：

```
public class Calculator {  
    // Omit testAdd() and testMultiply() for brevity  
    public double divide(double dividend, double divisor) {  
        if (divisor == 0) throw new IllegalArgumentException("Divisor cannot be 0");  
        return dividend / divisor;  
    }  
}
```

那么如何测试当传入的除数是 0 的时候，这个方法应该抛出 `IllegalArgumentException` 异常呢？

在 JUnit 中，可以通过给 `@Test` annotation 传入一个 `expected` 参数来达到这个目的，如下：

```
public class CalculatorTest {  
    Calculator mCalculator;  
    @Before  
    public void setup() {  
        mCalculator = new Calculator();  
    }  
    // Omit testAdd() and testMultiply() for brevity  
    @Test(expected = IllegalArgumentException.class)  
    public void test() {  
        mCalculator.divide(4, 0);  
    }  
}
```

`@Test(expected = IllegalArgumentException.class)` 表示验证这个测试方法将抛出 `IllegalArgumentException` 异常，如果没有抛出的话，则测试失败。

## 在 Android 项目里面使用 JUnit

在 Android 项目里面使用 JUnit 是很简单的，你只需要将 JUnit 这个 library 加到你的 dependencies 里面。

`testCompile 'junit:junit:4.12'`

如果你通过 AndroidStudio 创建一个项目，这个 dependency 默认是加上了的，所以你甚至这步都可以省略。

此外，你需要把测试代码放到 `src/test/java` 目录下面。

接下来关于怎么样运行测试代码，怎么样看结果，请参考[这个系列的第一篇文章](#)的相关部分，因为图比较多，这边就不重复了。

这里让大家看一下运行的结果是什么样子的，其中有一个失败的测试用例是故意的。如果你直接在 AndroidStudio 里面跑上面的测试类 `CalculatorTest` 的所有测试方法的话，会看到如下的结果：

□



左边可以看到所有的测试方法，以及每个方法跑出来的结果，绿色表示测试通过的测试方法，黄色的感叹号或红色的表示测试失败的。第三个那个有条纹的球球表示被忽略的测试方法。

如果是通过 terminal 跑的话，则会看到如下的测试结果：

```
:app:testDebugUnitTest
com.chriszou.auttutorial.test.CalculatorTest > testAdd2 FAILED
    junit.framework.AssertionFailedError at CalculatorTest.java:31

5 tests completed, 1 failed, 1 skipped
:app:testDebugUnitTest FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':app:testDebugUnitTest'.
> There were failing tests. See the report at: file:///Users/xiaochuang/AndroidStudioProjects/learn/androidio

* Try:
Run with --stacktrace option to get the stack trace. Run with --info or --debug option to get more log output.

BUILD FAILED

Total time: 2.373 secs
```

这篇文章的相关代码可以在[github](#) 的这个 [project](#)看到。

## 小结

这篇文章大概简单介绍了 JUnit 的使用，相对来说是比较简单，也是比较容易理解的，希望能帮助到大家。其中 Assert 部分，可以帮我们验证一个方法的返回结果。然而，这些只能帮我们测试有返回值的那些方法。在第一篇文章里面我们讲了，一个类的方法分两种，一是有返回值的方法，这些可以通过我们今天讲的 JUnit 来做测试。而另外一种没有返回值的方法，即 void 方法，则要通过另外一个框架，Mockito，来验证它的正确性。至于怎么样验证 void 方法的正确性，以及 Mockito 的使用，请关注下一篇文章。

最后，如果你对安卓单元测试感兴趣，欢迎加入我们的交流群，因为群成员超过 100 人，没办法扫码加入，请关注下方公众号获取加入方法。

参考：

<http://junit.org/junit4/>

<http://www.vogella.com/tutorials/JUnit/article.html>