Kotlin 写 Android 单元测试（二），JUnit 4 测试框架和 kotlin.test 库的使用


johnnyshieh.me/posts/unit-test-junit4-and-kotlin-test

Kotlin 写 Android 单元测试系列文章：

Kotlin 写 Android 单元测试（一），单元测试是什么以及为什么需要

Kotlin 写 Android 单元测试（二），JUnit 4 测试框架和 kotlin.test 库的使用

Kotlin 写 Android 单元测试（三），Mockito mocking 框架的使用

Kotlin 写 Android 单元测试（四），Robolectric 在 JVM 上测试安卓相关代码

未完待续...

上篇文章介绍了单元测试的概念和为什么要使用单元测试，提到当方法的输出结果是**直接的返回结果**时，可以用 Junit 4 测试框架来测试，这也是 Java 中最基础的测试框架。下面来介绍 JUnit 4 测试框架中的基本概念，以及 Kotlin 中提供的 kotlin.test 库的使用。

JUnit 4 框架

gradle 引入

目前 JUnit 4 的最稳定的版本为 4.12，JUnit 5 还没有正式发布。

```
dependencies {
  testCompile 'junit:junit:4.12'
}
android {
    sourceSets {
        test.java.srcDirs += 'src/test/kotlin'  // 如果不是默认（src/test/java）的话，加上测试代码的路径
    }
}
```

一般运行 Android 项目的测试命令为：

```
./gradle
testDebugUnitTest
```

验证方法结果

JUnit 4 可以很方便地测试方法直接的返回结果，先看下面这段代码：

```
class Calculator {
    fun divide(a: Int, b: Int): Double {
        if (b == 0) throw IllegalArgumentException("division by
zero!")
        return (a.toDouble() / b)
    }
}
```

其对应的单元测试代码如下：

```
class CalculatorTest {
    @Test
    fun testDivide() {
        val calculator = Calculator()
        val result = calculator.divide(2, 1)
        Assert.assertEquals(2.0, result,
0.0001)
    }
}
```

上面单元测试的代码中，首先用 `@Test` 注解标记 `testDivide` 方法，表明该方法是测试方法，测试框架在运行单元测试时会调用该测试方法，测试方法不能有参数。接下来第 6 行代码 `Assert.assertEquals(2.0, result, 0.0001)`，判断返回结果是否等于 2.0，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 0.0001，只要两者在这个偏差值范围内则测试通过。

JUnit 4 中验证返回结果的方法还有：

`assertEquals(expected, actual)`，验证 expected 的值是否和 actual 一样，如果传入是对象的话，那么会使用 equals 方法判断。

`assertArrayEquals(expected, actual)`，验证 actual 数组是否和 expected 一样。

`assertFalse(actual)`，验证 actual 的值为 false。

`assertNotNull(actual)`，验证 actual 的值不是 null。

`assertNull(actual)`，验证 actual 的值是 null。

`assertTrue(actual)`，验证 actual 的值是 true。

`assertThat(actual, matcher)`，验证 actual 是否符合 matcher 中的条件。

注意，上面的 assertXX 方法都有一个重载方法，额外加一个 String 类型的参数在前面，表示测试不通过时，将这个字符串作为失败显示的信息。

比如：Assert.assertEquals("failure - divide method has wrong result", 2.0, result, 0.0001)，这样当 result 不为 2.0 时，测试失败后会显示 `failure - divide method has wrong result` ，增加测试代码的可读性，方便定位测试失败的原因。

更多关于验证结果的用法，可以看 JUnit 4 的官方文档 Assertions。

前面提到都是验证方法的直接返回结果，但是假如方法在某种情况下会抛出异常，应该怎么测试呢？

验证 Exception

例如 Calculator 的 divide 方法中第二个参数为 0，会抛出 IllegalArgumentException 异常，所以更健壮的测试代码应该如下：

```
class CalculatorTest {
    @Test(expected =
IllegalArgumentException::class)
    fun testDivide() {
        val calculator = Calculator()
        val result = calculator.divide(2, 1)
        Assert.assertEquals(2.0, result, 0.0001)
        // test divide zero
        calculator.divide(2, 0)
    }
}
```

只需要在 `@Test` 注解加上 `expected` 参数，表明期望出现的异常，如上面代码中，如果去掉 `calculator.divide(2, 0)` 这行，就会因为没有产生期望的 IllegalArgumentException 而导致测试不通过。

不过这种验证异常的方法还是有所限制的，不能验证异常中的 message，也无法验证出现异常中其他属性的值，不过 JUnit 4 提供了另外一种方式可以验证异常，可以解决这些问题，后面的文章会提到。

初始化和收尾工作

前面讲到，首先需要 `@Test` 注解标记测试方法，让 JUnit 4 框架知道这是一个测试方法，然后介绍 JUnit 4 中验证方法直接返回结果的一些方法。但是在上一篇介绍单元测试的文章中，单元测试分为 4 步：初始化 -> 调用要测试方法 -> 验证结果 -> 收尾工作，其中第四步收尾工作可能没有。

在编写单元测试的代码过程中，经常会发现几个测试方法都会有相同的创建实例的初始化工作，假设 Calculator 还有一个 add 方法，那么对应的测试代码：

```
class CalculatorTest {
    @Test(expected =
IllegalArgumentException::class)
    fun testDivide() {
        val calculator = Calculator()
        val result = calculator.divide(2, 1)
        Assert.assertEquals(2.0, result, 0.0001)
        calculator.divide(2, 0)
    }
    @Test
    fun testAdd() {
        val calculator = Calculator()
        val result = calculator.add(1, 5)
        Assert.assertEquals(6, result)
    }
}
```

现在 `testDivide` 和 `testAdd` 都有同样的初始化工作，在我们写测试代码时也经常遇到这种情况，有没有什么方法把相同的初始化工作抽象出来呢，避免重复的代码，JUnit 4 提供了 `@Before` 和 `@After` 注解可以很好地定义初始化和收尾工作：

```
class CalculatorTest {
    lateinit var calculator: Calculator
    @Before
    fun setup() {
        calculator = Calculator()
    }
    @After
    fun cleanup() {
        ...
    }
    @Test(expected =
IllegalArgumentException::class)
    fun testDivide() {
        val result = calculator.divide(2, 1)
        Assert.assertEquals(2.0, result, 0.0001)
        calculator.divide(2, 0)
    }
    @Test
    fun testAdd() {
        val result = calculator.add(1, 5)
        Assert.assertEquals(6, result)
    }
}
```

上面的测试代码中，`testDivide` 和 `testAdd` 测试方法运行之前会先运行 `setup` 方法，测试方法执行后会运行 `cleanup` 方法。其实上面例子中 `setup` 方法不需要每个测试方法之前都运行一次，只需要执行一遍就可以。JUnit 4 针对这种情况提供 `@BeforeClass` 和 `@AfterClass` 注解， `@BeforeClass` 标记的方法会在该类的测试方法运行前运行一遍，只会执行一次，然后在所有测试方法运行完后会运行一次 `@AfterClass` 标记的方法。

不过 `@BeforeClass` 和 `@AfterClass` 注解，标记的方法应该为静态方法。

```
class CalculatorTest {
    companion object {
        lateinit var calculator:
Calculator
        @BeforeClass
        @JvmStatic
        fun setup() {
            calculator = Calculator()
        }
        @AfterClass
        @JvmStatic
        fun cleanup() {
            ...
        }
    }
    ...
}
```

JUnit 的其他一些方法

忽略某些测试方法

有时因为一些原因，例如正式代码还没有实现，想让 JUnit 暂时不允许某些测试方法，这时就可以使用 `@Ignore` 注解，例如：

```
class CalculatorTest {
    lateinit var calculator:
Calculator
    @Test
    @Ignore("not implemented
yet")
    fun testSubtract() {}
    ...
```

fail 方法

有时候可能需要故意让测试方法运行失败，例如在 catch 到某些异常时，这时可以使用 `fail` 方法：

```
Assert.fail()
Assert.fail(mes
sage)
```

TestRule

除了上面的 @Before、@After 让测试方法运行前执行额外代码外，JUnit 4 中的 TestRule 可以达到同样的效果，TestRule 可以很方便地添加额外代码或者重新定义测试行为。Junit 4 中自带的 Rule 有 ErrorCollector、ExpectedException、ExternalResource、TemporaryFolder、TestName、TestWatcher、Timeout、Verifier，其中 ExpectedException 可以验证异常的详细信息，Timeout 可以指定测试方法的最大运行时间。

下面自定义一个 CustomRule，说明 TestRule 的用法：

```
class CustomRule : TestRule {
    override fun apply(base: Statement?, description: Description?): Statement = object:
Statement() {
        fun before() {
            println("before test")
        }
        override fun evaluate() {
            before()
            base?.evaluate() // 原测试方法执行
            after()
        }
        fun after() {
            println("after test")
        }
    }
}
```

使用 TestRule 也很简单，可以作为测试类的成员属性或者返回 TestRule 的方法，再用 @Rule 或 @ClassRule 标注即可

```
class TestA {
    @Rule
    @JvmField
    val customRule =
CustomRule()
    // classRule 方式
    companion object {
        @ClassRule
        @JvmField
        val customRule =
CustomRule()
    }
    @Test
    fun testMethod1() { ... }
    @Test
    fun testMethod2() { ... }
}
```

用 @Rule 标注时和 @Before、@After 类似，每个测试方法运行前后 CustomRule 的 before、after 方法都会执行一次；而 @ClassRule 标准时和 @BeforeClass、@AfterClass 类似，测试类的所有测试运行前后才会执行一次 before、after 方法。

更多关于 TestRule 的内容，推荐大家阅读 Junit wiki：Junit 4 Rules

kotlin.test 库

前面介绍的都是 JUnit 4 测试框架的一些基本用法，但是 Kotlin 语言还提供了一个 kotlin.test 库，它定义了一些全局函数，可以在编写测试代码不用导入 `org.junit.Assert` ，还可以使用高阶函数作为验证语句的参数，简单地说就是简化测试代码的使用并加入了 kotlin 语言的特性。

kotlin.test 库提供一些全局函数，如 assertEquals、expect，更多详细内容请看 Package kotlin.test。

gradle 引入

使用了 kotlin.test 库，build.gradle 需要稍微修改下：

```
dependencies {
  testCompile 'junit:junit:4.12'
  testCompile "org.jetbrains.kotlin:kotlin-test-junit:$kotlin_version"
}
android {
    sourceSets {
        test.java.srcDirs += 'src/test/kotlin'  // 如果不是默认（src/test/java）的话，加上测试代码
的路径
    }
}
```

具体使用

下面我只是给出上面的测试代码使用 kotlin.test 库后的修改，具体 kotlin.test 库的使用请看官方文档：

```
import org.junit.AfterClass
import org.junit.BeforeClass
import org.junit.Test
import kotlin.test.assertEquals
class CalculatorTest {
    companion object {
        lateinit var calculator:
Calculator
        @BeforeClass
        @JvmStatic
        fun setup() {
            calculator = Calculator()
        }
    }
    @Test
    fun testAdd() {
        // 使用 assertEquals
        val result = calculator.add(1,
5)
        assertEquals(6, result)
        // 使用 expect
        expect(6, { calculator.add(1,
5) })
    }
}
```

运行单元测试

一般写好测试代码后，AndroidStudio 中在测试方法和测试类的左侧就会出现一个图标，点击会出现三个选项，如下图所示：

运行后就会出现下面的视图：

还可以通过命令行的方式运行所有单元测试代码：

```
gradlew testDebugUnitTest  // debug 版本
gradlew testReleaseUnitTest    // release 版本
```

运行后会生成一个测试报告，在 project_root/app/build/reports/tests/testDebug UnitTest/index.html 路径下可以看到，测试失败时，可以在测试报告中查看详细信息。

小结

上面介绍了 JUnit 4 测试框架的基本用法以及 kotlin.test 库的使用，但是要在 Android 项目使用 Kotlin 语言写测试代码还有其他问题。首先，JUnit 框架只能验证有直接返回结果的方法，当返回结果是其他方法的调用时怎么验证呢，其次，这些测试代码都没有设计到 Android 的代码，如何测试跟 Android 有关的代码呢，又如何测试 app 的 UI 界面呢，这些问题会在后续的文章的一一解决。下一篇文章介绍如何使用 Mockito 框架验证没有直接返回结果时的测试代码，以及在 Kotlin 中 Mockito 框架出现的问题及解决方案。