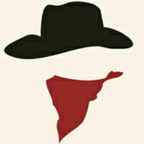
**[译] 使用强大的 Mockito 来测试你的代码**

[](http://www.jianshu.com/u/b7942bef8847)

作者 [edvardhua](http://www.jianshu.com/u/b7942bef8847) 关注

2016.07.22 10:43 字数 3921 阅读 506评论 0喜欢 10

* 原文链接 : [Unit tests with Mockito - Tutorial](http://www.vogella.com/tutorials/Mockito/article.html)
* 原文作者 : [vogella](http://www.vogella.com/)
* 译者 : [edvardhua](https://github.com/edvardHua/)
* 校对者: [hackerkevin](https://github.com/hackerkevin), [futureshine](https://github.com/futureshine)

这篇教程介绍了如何使用 Mockito 框架来给软件写测试用例

**1. 预备知识**

如果需要往下学习，你需要先理解 Junit 框架中的单元测试。

如果你不熟悉 JUnit，请查看下面的教程：  
<http://www.vogella.com/tutorials/JUnit/article.html>

**2. 使用mock对象来进行测试**

**2.1. 单元测试的目标和挑战**

单元测试的思路是在不涉及依赖关系的情况下测试代码（隔离性），所以测试代码与其他类或者系统的关系应该尽量被消除。一个可行的消除方法是替换掉依赖类（测试替换），也就是说我们可以使用替身来替换掉真正的依赖对象。

**2.2. 测试类的分类**

*dummy object* 做为参数传递给方法但是绝对不会被使用。譬如说，这种测试类内部的方法不会被调用，或者是用来填充某个方法的参数。

*Fake* 是真正接口或抽象类的实现体，但给对象内部实现很简单。譬如说，它存在内存中而不是真正的数据库中。（译者注：*Fake* 实现了真正的逻辑，但它的存在只是为了测试，而不适合于用在产品中。）

*stub* 类是依赖类的部分方法实现，而这些方法在你测试类和接口的时候会被用到，也就是说 *stub* 类在测试中会被实例化。*stub* 类会回应任何外部测试的调用。*stub* 类有时候还会记录调用的一些信息。

*mock object* 是指类或者接口的模拟实现，你可以自定义这个对象中某个方法的输出结果。

测试替代技术能够在测试中模拟测试类以外对象。因此你可以验证测试类是否响应正常。譬如说，你可以验证在 Mock 对象的某一个方法是否被调用。这可以确保隔离了外部依赖的干扰只测试测试类。

我们选择 Mock 对象的原因是因为 Mock 对象只需要少量代码的配置。

**2.3. Mock 对象的产生**

你可以手动创建一个 Mock 对象或者使用 Mock 框架来模拟这些类，Mock 框架允许你在运行时创建 Mock 对象并且定义它的行为。

一个典型的例子是把 Mock 对象模拟成数据的提供者。在正式的生产环境中它会被实现用来连接数据源。但是我们在测试的时候 Mock 对象将会模拟成数据提供者来确保我们的测试环境始终是相同的。

Mock 对象可以被提供来进行测试。因此，我们测试的类应该避免任何外部数据的强依赖。

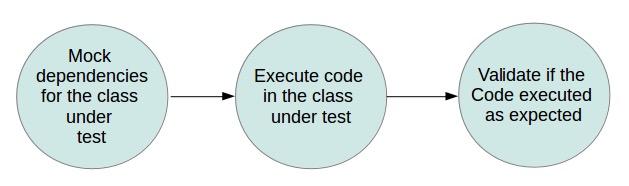
通过 Mock 对象或者 Mock 框架，我们可以测试代码中期望的行为。譬如说，验证只有某个存在 Mock 对象的方法是否被调用了。

**2.4. 使用 Mockito 生成 Mock 对象**

*Mockito* 是一个流行 mock 框架，可以和JUnit结合起来使用。Mockito 允许你创建和配置 mock 对象。使用Mockito可以明显的简化对外部依赖的测试类的开发。

一般使用 Mockito 需要执行下面三步

* 模拟并替换测试代码中外部依赖。
* 执行测试代码
* 验证测试代码是否被正确的执行



mockitousagevisualization

**3. 为自己的项目添加 Mockito 依赖**

**3.1. 在 Gradle 添加 Mockito 依赖**

如果你的项目使用 Gradle 构建，将下面代码加入 Gradle 的构建文件中为自己项目添加 Mockito 依赖

repositories { jcenter() }

dependencies { testCompile "org.mockito:mockito-core:2.0.57-beta" }

**3.2. 在 Maven 添加 Mockito 依赖**

需要在 Maven 声明依赖，您可以在 [http://search.maven.org](http://search.maven.org/) 网站中搜索 g:"org.mockito", a:"mockito-core" 来得到具体的声明方式。

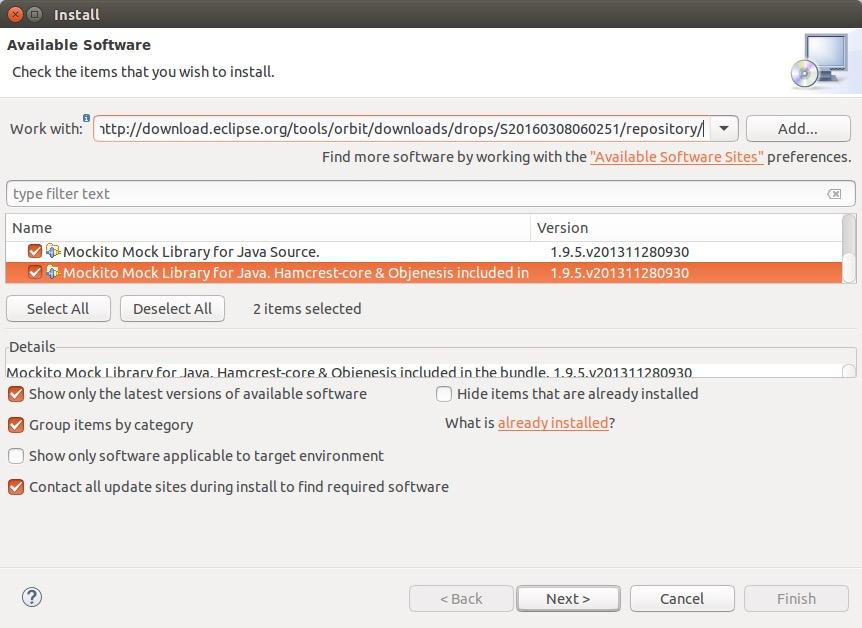
**3.3. 在 Eclipse IDE 使用 Mockito**

Eclipse IDE 支持 Gradle 和 Maven 两种构建工具，所以在 Eclipse IDE 添加依赖取决你使用的是哪一个构建工具。

**3.4. 以 OSGi 或者 Eclipse 插件形式添加 Mockito 依赖**

在 Eclipse RCP 应用依赖通常可以在 p2 update 上得到。Orbit 是一个很好的第三方仓库，我们可以在里面寻找能在 Eclipse 上使用的应用和插件。

Orbit 仓库地址 <http://download.eclipse.org/tools/orbit/downloads>



orbit p2 mockito

**4. 使用Mockito API**

**4.1. 静态引用**

如果在代码中静态引用了org.mockito.Mockito.\*;，那你你就可以直接调用静态方法和静态变量而不用创建对象，譬如直接调用 mock() 方法。

**4.2. 使用 Mockito 创建和配置 mock 对象**

除了上面所说的使用 mock() 静态方法外，Mockito 还支持通过 @Mock 注解的方式来创建 mock 对象。

如果你使用注解，那么必须要实例化 mock 对象。Mockito 在遇到使用注解的字段的时候，会调用MockitoAnnotations.initMocks(this) 来初始化该 mock 对象。另外也可以通过使用@RunWith(MockitoJUnitRunner.class)来达到相同的效果。

通过下面的例子我们可以了解到使用@Mock 的方法和MockitoRule规则。

import static org.mockito.Mockito.\*;

public class MockitoTest {

@Mock

MyDatabase databaseMock; (1)

@Rule public MockitoRule mockitoRule = MockitoJUnit.rule(); (2)

@Test

public void testQuery() {

ClassToTest t = new ClassToTest(databaseMock); (3)

boolean check = t.query("\* from t"); (4)

assertTrue(check); (5)

verify(databaseMock).query("\* from t"); (6)

}

}

1. 告诉 Mockito 模拟 databaseMock 实例
2. Mockito 通过 @mock 注解创建 mock 对象
3. 使用已经创建的mock初始化这个类
4. 在测试环境下，执行测试类中的代码
5. 使用断言确保调用的方法返回值为 true
6. 验证 query 方法是否被 MyDatabase 的 mock 对象调用

**4.3. 配置 mock**

当我们需要配置某个方法的返回值的时候，Mockito 提供了链式的 API 供我们方便的调用

when(…​.).thenReturn(…​.)可以被用来定义当条件满足时函数的返回值，如果你需要定义多个返回值，可以多次定义。当你多次调用函数的时候，Mockito 会根据你定义的先后顺序来返回返回值。Mocks 还可以根据传入参数的不同来定义不同的返回值。譬如说你的函数可以将anyString 或者 anyInt作为输入参数，然后定义其特定的放回值。

import static org.mockito.Mockito.\*;

import static org.junit.Assert.\*;

@Test

public void test1() {

// 创建 mock

MyClass test = Mockito.mock(MyClass.class);

// 自定义 getUniqueId() 的返回值

when(test.getUniqueId()).thenReturn(43);

// 在测试中使用mock对象

assertEquals(test.getUniqueId(), 43);

}

// 返回多个值

@Test

public void testMoreThanOneReturnValue() {

Iterator i= mock(Iterator.class);

when(i.next()).thenReturn("Mockito").thenReturn("rocks");

String result=i.next()+" "+i.next();

// 断言

assertEquals("Mockito rocks", result);

}

// 如何根据输入来返回值

@Test

public void testReturnValueDependentOnMethodParameter() {

Comparable c= mock(Comparable.class);

when(c.compareTo("Mockito")).thenReturn(1);

when(c.compareTo("Eclipse")).thenReturn(2);

// 断言

assertEquals(1,c.compareTo("Mockito"));

}

// 如何让返回值不依赖于输入

@Test

public void testReturnValueInDependentOnMethodParameter() {

Comparable c= mock(Comparable.class);

when(c.compareTo(anyInt())).thenReturn(-1);

// 断言

assertEquals(-1 ,c.compareTo(9));

}

// 根据参数类型来返回值

@Test

public void testReturnValueInDependentOnMethodParameter() {

Comparable c= mock(Comparable.class);

when(c.compareTo(isA(Todo.class))).thenReturn(0);

// 断言

Todo todo = new Todo(5);

assertEquals(todo ,c.compareTo(new Todo(1)));

}

对于无返回值的函数，我们可以使用doReturn(…​).when(…​).methodCall来获得类似的效果。例如我们想在调用某些无返回值函数的时候抛出异常，那么可以使用doThrow 方法。如下面代码片段所示

import static org.mockito.Mockito.\*;

import static org.junit.Assert.\*;

// 下面测试用例描述了如何使用doThrow()方法

@Test(expected=IOException.class)

public void testForIOException() {

// 创建并配置 mock 对象

OutputStream mockStream = mock(OutputStream.class);

doThrow(new IOException()).when(mockStream).close();

// 使用 mock

OutputStreamWriter streamWriter= new OutputStreamWriter(mockStream);

streamWriter.close();

}

**4.4. 验证 mock 对象方法是否被调用**

Mockito 会跟踪 mock 对象里面所有的方法和变量。所以我们可以用来验证函数在传入特定参数的时候是否被调用。这种方式的测试称行为测试，行为测试并不会检查函数的返回值，而是检查在传入正确参数时候函数是否被调用。

import static org.mockito.Mockito.\*;

@Test

public void testVerify() {

// 创建并配置 mock 对象

MyClass test = Mockito.mock(MyClass.class);

when(test.getUniqueId()).thenReturn(43);

// 调用mock对象里面的方法并传入参数为12

test.testing(12);

test.getUniqueId();

test.getUniqueId();

// 查看在传入参数为12的时候方法是否被调用

verify(test).testing(Matchers.eq(12));

// 方法是否被调用两次

verify(test, times(2)).getUniqueId();

// 其他用来验证函数是否被调用的方法

verify(mock, never()).someMethod("never called");

verify(mock, atLeastOnce()).someMethod("called at least once");

verify(mock, atLeast(2)).someMethod("called at least twice");

verify(mock, times(5)).someMethod("called five times");

verify(mock, atMost(3)).someMethod("called at most 3 times");

}

**4.5. 使用 Spy 封装 java 对象**

@Spy或者spy()方法可以被用来封装 java 对象。被封装后，除非特殊声明（打桩 *stub*），否则都会真正的调用对象里面的每一个方法

import static org.mockito.Mockito.\*;

// Lets mock a LinkedList

List list = new LinkedList();

List spy = spy(list);

// 可用 doReturn() 来打桩

doReturn("foo").when(spy).get(0);

// 下面代码不生效

// 真正的方法会被调用

// 将会抛出 IndexOutOfBoundsException 的异常，因为 List 为空

when(spy.get(0)).thenReturn("foo");

方法verifyNoMoreInteractions()允许你检查没有其他的方法被调用了。

**4.6. 使用 @InjectMocks 在 Mockito 中进行依赖注入**

我们也可以使用@InjectMocks 注解来创建对象，它会根据类型来注入对象里面的成员方法和变量。假定我们有 ArticleManager 类

public class ArticleManager {

private User user;

private ArticleDatabase database;

ArticleManager(User user) {

this.user = user;

}

void setDatabase(ArticleDatabase database) { }

}

这个类会被 Mockito 构造，而类的成员方法和变量都会被 mock 对象所代替，正如下面的代码片段所示：

@RunWith(MockitoJUnitRunner.class)

public class ArticleManagerTest {

@Mock ArticleCalculator calculator;

@Mock ArticleDatabase database;

@Most User user;

@Spy private UserProvider userProvider = new ConsumerUserProvider();

@InjectMocks private ArticleManager manager; (1)

@Test public void shouldDoSomething() {

// 假定 ArticleManager 有一个叫 initialize() 的方法被调用了

// 使用 ArticleListener 来调用 addListener 方法

manager.initialize();

// 验证 addListener 方法被调用

verify(database).addListener(any(ArticleListener.class));

}

}

1. 创建ArticleManager实例并注入Mock对象

更多的详情可以查看  
<http://docs.mockito.googlecode.com/hg/1.9.5/org/mockito/InjectMocks.html>.

**4.7. 捕捉参数**

ArgumentCaptor类允许我们在verification期间访问方法的参数。得到方法的参数后我们可以使用它进行测试。

import static org.hamcrest.Matchers.hasItem;

import static org.junit.Assert.assertThat;

import static org.mockito.Mockito.mock;

import static org.mockito.Mockito.verify;

import java.util.Arrays;

import java.util.List;

import org.junit.Rule;

import org.junit.Test;

import org.mockito.ArgumentCaptor;

import org.mockito.Captor;

import org.mockito.junit.MockitoJUnit;

import org.mockito.junit.MockitoRule;

public class MockitoTests {

@Rule public MockitoRule rule = MockitoJUnit.rule();

@Captor

private ArgumentCaptor> captor;

@Test

public final void shouldContainCertainListItem() {

List asList = Arrays.asList("someElement\_test", "someElement");

final List mockedList = mock(List.class);

mockedList.addAll(asList);

verify(mockedList).addAll(captor.capture());

final List capturedArgument = captor.>getValue();

assertThat(capturedArgument, hasItem("someElement"));

}

}

**4.8. Mockito的限制**

Mockito当然也有一定的限制。而下面三种数据类型则不能够被测试

* final classes
* anonymous classes
* primitive types

**5. 在Android中使用Mockito**

在 Android 中的 Gradle 构建文件中加入 Mockito 依赖后就可以直接使用 Mockito 了。若想使用 Android Instrumented tests 的话，还需要添加 dexmaker 和 dexmaker-mockito 依赖到 Gradle 的构建文件中。（需要 Mockito 1.9.5版本以上）

dependencies {

testCompile 'junit:junit:4.12'

// Mockito unit test 的依赖

testCompile 'org.mockito:mockito-core:1.+'

// Mockito Android instrumentation tests 的依赖

androidTestCompile 'org.mockito:mockito-core:1.+'

androidTestCompile "com.google.dexmaker:dexmaker:1.2"

androidTestCompile "com.google.dexmaker:dexmaker-mockito:1.2"

}

**6. 实例：使用Mockito写一个Instrumented Unit Test**

**6.1. 创建一个测试的Android 应用**

创建一个包名为com.vogella.android.testing.mockito.contextmock的Android应用，添加一个静态方法  
，方法里面创建一个包含参数的Intent，如下代码所示：

public static Intent createQuery(Context context, String query, String value) {

// 简单起见，重用MainActivity

Intent i = new Intent(context, MainActivity.class);

i.putExtra("QUERY", query);

i.putExtra("VALUE", value);

return i;

}

**6.2. 在app/build.gradle文件中添加Mockito依赖**

dependencies {

// Mockito 和 JUnit 的依赖

// instrumentation unit tests on the JVM

androidTestCompile 'junit:junit:4.12'

androidTestCompile 'org.mockito:mockito-core:2.0.57-beta'

androidTestCompile 'com.android.support.test:runner:0.3'

androidTestCompile "com.google.dexmaker:dexmaker:1.2"

androidTestCompile "com.google.dexmaker:dexmaker-mockito:1.2"

// Mockito 和 JUnit 的依赖

// tests on the JVM

testCompile 'junit:junit:4.12'

testCompile 'org.mockito:mockito-core:1.+'

}

**6.3. 创建测试**

使用 Mockito 创建一个单元测试来验证在传递正确 extra data 的情况下，intent 是否被触发。

因此我们需要使用 Mockito 来 mock 一个Context对象，如下代码所示：

package com.vogella.android.testing.mockitocontextmock;

import android.content.Context;

import android.content.Intent;

import android.os.Bundle;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.mockito.Mockito;

import static org.junit.Assert.assertEquals;

import static org.junit.Assert.assertNotNull;

public class TextIntentCreation {

@Test

public void testIntentShouldBeCreated() {

Context context = Mockito.mock(Context.class);

Intent intent = MainActivity.createQuery(context, "query", "value");

assertNotNull(intent);

Bundle extras = intent.getExtras();

assertNotNull(extras);

assertEquals("query", extras.getString("QUERY"));

assertEquals("value", extras.getString("VALUE"));

}

}

**7. 实例：使用 Mockito 创建一个 mock 对象**

**7.1. 目标**

创建一个 Api，它可以被 Mockito 来模拟并做一些工作

**7.2. 创建一个Twitter API 的例子**

实现 TwitterClient类，它内部使用到了 ITweet 的实现。但是ITweet实例很难得到，譬如说他需要启动一个很复杂的服务来得到。

public interface ITweet {

String getMessage();

}

public class TwitterClient {

public void sendTweet(ITweet tweet) {

String message = tweet.getMessage();

// send the message to Twitter

}

}

**7.3. 模拟 ITweet 的实例**

为了能够不启动复杂的服务来得到 ITweet，我们可以使用 Mockito 来模拟得到该实例。

@Test

public void testSendingTweet() {

TwitterClient twitterClient = new TwitterClient();

ITweet iTweet = mock(ITweet.class);

when(iTweet.getMessage()).thenReturn("Using mockito is great");

twitterClient.sendTweet(iTweet);

}

现在 TwitterClient 可以使用 ITweet 接口的实现，当调用 getMessage() 方法的时候将会打印 "Using Mockito is great" 信息。

**7.4. 验证方法调用**

确保 getMessage() 方法至少调用一次。

@Test

public void testSendingTweet() {

TwitterClient twitterClient = new TwitterClient();

ITweet iTweet = mock(ITweet.class);

when(iTweet.getMessage()).thenReturn("Using mockito is great");

twitterClient.sendTweet(iTweet);

verify(iTweet, atLeastOnce()).getMessage();

}

**7.5. 验证**

运行测试，查看代码是否测试通过。

**8. 模拟静态方法**

**8.1. 使用 Powermock 来模拟静态方法**

因为 Mockito 不能够 mock 静态方法，因此我们可以使用 Powermock。

import java.net.InetAddress;

import java.net.UnknownHostException;

public final class NetworkReader {

public static String getLocalHostname() {

String hostname = "";

try {

InetAddress addr = InetAddress.getLocalHost();

// Get hostname

hostname = addr.getHostName();

} catch ( UnknownHostException e ) {

}

return hostname;

}

}

我们模拟了 NetworkReader 的依赖，如下代码所示：

import org.junit.runner.RunWith;

import org.powermock.core.classloader.annotations.PrepareForTest;

@RunWith( PowerMockRunner.class )

@PrepareForTest( NetworkReader.class )

public class MyTest {

// 测试代码

@Test

public void testSomething() {

mockStatic( NetworkUtil.class );

when( NetworkReader.getLocalHostname() ).andReturn( "localhost" );

// 与 NetworkReader 协作的测试

}

**8.2.用封装的方法代替Powermock**

有时候我们可以在静态方法周围包含非静态的方法来达到和 Powermock 同样的效果。

class FooWraper {

void someMethod() {

Foo.someStaticMethod()

}

}

**9. Mockito 参考资料**

[http://site.mockito.org](http://site.mockito.org/) - Mockito 官网

<https://github.com/mockito/mockito-> Mockito Github

<https://github.com/mockito/mockito/blob/master/doc/release-notes/official.md> - Mockito 发行说明

<http://martinfowler.com/articles/mocksArentStubs.html> 与Mocks，Stub有关的文章

<http://chiuki.github.io/advanced-android-espresso/> 高级android教程（竟然是个妹子）