Android单元测试（四）：Mock以及Mockito的使用

chriszou.com/2016/04/28/android-unit-testing-mock-and-mockito

April 28, 2016

> 几点说明：
>
> 1. 代码中的 //<== 表示跟上面的相比，这是新增的，或者是修改的代码，不知道怎么样在代码块里面再强调几行代码 T_T。。。
> 2. 很多时候，为了避免中文歧义，我会用英文表述

在第一篇文章里面我们提到，返回类型为 void 方法的单元测试方式，往往是验证里面的某个对象的某个方法是否得到了调用。在那篇文章里面，我举的例子是 activity 里面的一个 login 方法：

```
public void login() {
    String username = ...//get username from username EditText
    String password = ...//get password from password EditText
    //do other operation like validation, etc
    ...
    mUserManager.performLogin(username, password);
}
```

对于这个 login 方法的单元测试，应该是调用 Activity 里面的这个 login 方法，然后验证 `mUserManager` 的 `performLogin` 方法得到了调用。但是如果使用 Activity，我们就需要用到Robolectric框架，然而我们到目前为止还没有讲到 Robolectric 的使用。所以在这篇文章中，我们假设这段代码是放在一个 Presenter（LoginPresenter）里面的，这个是MVP 模式里面的概念，这个 `LoginPresenter` 是一个纯 java 类，而用户名和密码是外面传进来的：

```
public class LoginPresenter {
    private UserManager mUserManager = new UserManager();
    public void login(String username, String password) {
        if (username == null || username.length() == 0) return;
        if (password == null || password.length() < 6) return;
        mUserManager.performLogin(username, password);
    }
}
```

根据前面一篇关于 JUnit 的文章的讲解，我们很容易的写出针对 `login()` 方法的单元测试：

```
public class LoginPresenterTest {
    @Test
    public void testLogin() throws Exception {
        LoginPresenter loginPresenter = new LoginPresenter();
        loginPresenter.login("xiaochuang", "xiaochuang password");
        //验证LoginPresenter里面的mUserManager的performLogin()方法得到了调用，同时参数分别
是“xiaochuang”、“xiaochuang password”
        ...
    }
}
```

现在，关键的问题来了，怎么验证 `LoginPresenter` 里面的 `mUserManager` 的 `performLogin()` 方法得到了调用，以及它的参数是正确性呢？如果大家看了该系列的第一篇文章就知道，这里需要用到mock，那么接下来，我们就介绍 mock 这个东西。

Mock 的概念：两种误解

Mock 的概念，其实很简单，我们前面也介绍过：所谓的 mock 就是创建一个类的虚假的对象，在测试环境中，用来替换掉真实的对象，以达到两大目的：

1. 验证这个对象的某些方法的调用情况，调用了多少次，参数是什么等等
2. 指定这个对象的某些方法的行为，返回特定的值，或者是执行特定的动作

要使用 Mock，一般需要用到 mock 框架，这篇文章我们使用Mockito这个框架，这个是 Java 界使用最广泛的一个 mock 框架。
对于上面的例子，我们要验证 `mUserManager` 的一些行为，首先要 mock UserManager 这个类，mock 这个类的方式是：
`Mockito.mock(UserManager.class);`
mock 了 `UserManager` 类之后，我们就可以开始测试了：

```
public class LoginPresenterTest {
    @Test
    public void testLogin() {
        Mockito.mock(UserManager.class);    //<==
        LoginPresenter loginPresenter = new LoginPresenter();
        loginPresenter.login("xiaochuang", "xiaochuang password");
        //验证LoginPresenter里面的mUserManager的performLogin()方法得到了调用，参数分别
是“xiaochuang”、“xiaochuang password”
        ...
    }
}
```

然而我们要验证的是 `LoginPresenter` 里面的 `mUserManager` 这个对象，但是现在我们没有办法获得这个对象，因为 `mUserManager` 是 private 的，怎么办？先不想太多，我们简单粗暴点，给 `LoginPresenter` 加一个 getter，稍后你会明白我现在为什么做这样的决定。

```
public class LoginPresenter {
    private UserManager mUserManager = new UserManager();
    public void login(String username, String password) {
        if (username == null || username.length() == 0) return;
        if (password == null || password.length() < 6) return;
        mUserManager.performLogin(username, password);
    }
    public UserManager getUserManager() {   //<==
        return mUserManager;
    }
}
```

好了，现在我们可以验证 `mUserManager` 被调用的情况了：

```
public class LoginPresenterTest {
    @Test
    public void testLogin() throws Exception {
        Mockito.mock(UserManager.class);
        LoginPresenter loginPresenter = new LoginPresenter();
        loginPresenter.login("xiaochuang", "xiaochuang password");
        UserManager userManager = loginPresenter.getUserManager();  //<==
        //验证userManager的performLogin()方法得到了调用，参数分别是“xiaochuang”、“xiaochuang
password”
        ...
    }
}
```

终于到了解释如何验证一个对象的某个方法的调用情况了。使用 Mockito，验证一个对象的方法调用情况的姿势是：
`Mockito.verify(objectToVerify).methodToVerify(arguments);`
其中，`objectToVerify` 和 `methodToVerify` 分别是你想要验证的对象和方法。对应上面的例子，那就是：
`Mockito.verify(userManager).performLogin("xiaochuang", "xiaochuang password");`
好，现在我们把这行代码放到测试里面：

```
public class LoginPresenterTest {
    @Test
    public void testLogin() throws Exception {
        Mockito.mock(UserManager.class);
        LoginPresenter loginPresenter = new LoginPresenter();
        loginPresenter.login("xiaochuang", "xiaochuang password");
        UserManager userManager = loginPresenter.getUserManager();
        Mockito.verify(userManager).performLogin("xiaochuang", "xiaochuang password");  //<==
    }
}
```

接着我们跑一下这个测试方法，结果发现，额。。。出错了：

```
org.mockito.exceptions.misusing.NotAMockException:
Argument passed to verify() is of type UserManager and is not a mock!
Make sure you place the parenthesis correctly!
See the examples of correct verifications:
    verify(mock).someMethod();
    verify(mock, times(10)).someMethod();
    verify(mock, atLeastOnce()).someMethod();

    at com.chriszou.auttutorial.test.mockito.LoginPresenterTest.testLogin(LoginPresenterTest.java:22) <26 internal calls>
```

具体出错的是最后这一行代码：`Mockito.verify(userManager).performLogin("xiaochuang", "xiaochuang password");`。这个错误的大概意思是，传给 `Mockito.verify()` 的参数必须是一个 mock 对象，而我们传进去的不是一个 mock 对象，所以出错了。
这就是我想解释的，关于 mock 的第一个误解：**`Mockito.mock()` 并不是 mock 一整个类，而是根据传进去的一个类，mock 出属于这个类的一个对象，并且返回这个 mock 对象；而传进去的这个类本身并没有改变，用这个类 new 出来的对象也没有受到任何改变！**
结合上面的例子，`Mockito.mock(UserManager.class);` 只是返回了一个属于 `UserManager` 这个类的一个 mock 对象。`UserManager` 这个类本身没有受到任何影响，而 `LoginPresenter` 里面直接 `new UserManager()` 得到的 `mUserManager` 也是正常的一个对象，不是一个 mock 对象。`Mockito.verify()` 的参数必须是 mock 对象，也就是说，Mockito 只能验证 mock 对象的方法调用情况。因此，上面那种写法就出错了。
好的，知道了，既然这样，看来我们需要使用 `Mockito.mock(UserManager.class);` 返回的对象来验证，代码如下：

```
public class LoginPresenterTest {
    @Test
    public void testLogin() throws Exception {
        UserManager mockUserManager = Mockito.mock(UserManager.class);  //<==
        LoginPresenter loginPresenter = new LoginPresenter();
        loginPresenter.login("xiaochuang", "xiaochuang password");
        Mockito.verify(mockUserManager).performLogin("xiaochuang", "xiaochuang password");
//<==
    }
}
```

在运行一下，发现，额。。。又出错了：

```
Wanted but not invoked:
userManager.performLogin(
    "xiaochuang",
    "xiaochuang password"
);
-> at com.chriszou.auttutorial.test.mockito.LoginPresenterTest.testLogin(LoginPresenterTest.java:20)
Actually, there were zero interactions with this mock.
```

错误信息的大意是，我们想验证 `mockUserManager` 的 `performLogin()` 方法得到了调用，然而其实并没有。
这就是我想解释的，关于 mock 的第二个误解：**mock 出来的对象并不会自动替换掉正式代码里面的对象，你必须要有某种方式把 mock 对象应用到正式代码里面**

结合上面的例子，`UserManager mockUserManager = Mockito.mock(UserManager.class);` 的确给我们创建了一个 mock 对象，保存在 `mockUserManager` 里面。然而，当我们调用 `loginPresenter.login("xiaochuang", "xiaochuang password");` 的时候，用到的 mUserManager 依然是使用 `new UserManager()` 创建的正常的对象。而 `mockUserManager` 并没有得到任何的调用，因此，当我们验证它的 `performLogin()` 方法得到了调用时，就失败了。
对于这个问题，很明显，我们必须在调用 `loginPresenter.login()` 之前，把 `mUserManager` 引用换成 `mockUserManager` 所引用的 mock 对象。最简单的办法，就是加一个 setter：

```
public class LoginPresenter {
    private UserManager mUserManager = new UserManager();
    public void login(String username, String password) {
        if (username == null || username.length() == 0) return;
        if (password == null || password.length() < 6) return;
        mUserManager.performLogin(username, password);
    }
    public void setUserManager(UserManager userManager) {  //<==
        this.mUserManager = userManager;
    }
}
```

同时，getter 我们用不到了，于是这里就直接删了。那么按照上面的思路，写出来的测试代码如下：

```
@Test
public void testLogin() throws Exception {
    UserManager mockUserManager = Mockito.mock(UserManager.class);
    LoginPresenter loginPresenter = new LoginPresenter();
    loginPresenter.setUserManager(mockUserManager);  //<==
    loginPresenter.login("xiaochuang", "xiaochuang password");
    Mockito.verify(mockUserManager).performLogin("xiaochuang", "xiaochuang password");
}
```

最后运行一次，hu。。。终于通过了！
当然，如果你的正式代码里面没有任何地方用到了那个 setter 的话，那么专门为了测试而增加了一个方法，毕竟不是很优雅的解决办法，更好的解决办法是使用依赖注入，简单解释就是把 `UserManager` 作为 `LoginPresenter` 的构造函数的参数，传进去。具体操作请期待下一篇文章，这里我们专门讲 mock 的概念和 Mockito 的使用。
然而还是忍不住想多嘴一句：
优雅归优雅，有没有必要，值不值得，却又是另外一回事。总体来说，我认为是值得的，因为这可以让这个类变得可测，也就意味着我们可以验证这个类的正确性，更给以后重构这个类有了保障，防止误改错这个类等等。因此，很多时候，如果你为了做单元测试，不得已要给一些类加一些额外的代码。那就加吧！毕竟优雅不能当饭吃，而解决问题、修复 bug 可

以，做出优秀的、少有 bug 的产品更可以，所以，Just Do It!
好了，现在我想大家对 mock 的概念应该有了正确的认识，对怎么样使用 mock 也有了认识，接下来我们就可以全心全意介绍 Mockito 的功能和使用了。

Mockito 的使用

1. 验证方法调用

前面我们讲了验证一个对象的某个 method 得到调用的方法：
`Mockito.verify(mockUserManager).performLogin("xiaochuang", "xiaochuang password");`
这句话的作用是，验证 `mockUserManager` 的 `performLogin()` 得到了调用，同时参数是“xiaochuang”和"xiaochuang password"。其实更准确的说法是，这行代码验证的是， `mockUserManager` 的 `performLogin()` 方法得到了**一次**调用。因为这行代码其实是：
`Mockito.verify(mockUserManager, Mockito.times(1)).performLogin("xiaochuang", "xiaochuang password");`
的简写，或者说重载方法，注意其中的 `Mockito.times(1)` 。
因此，如果你想验证一个对象的某个方法得到了多次调用，只需要将次数传给 `Mockito.times()` 就好了。
`Mockito.verify(mockUserManager, Mockito.times(3)).performLogin(...); //验证mockUserManager的performLogin得到了三次调用。`
对于调用次数的验证，除了可以验证固定的多少次，还可以验证最多，最少从来没有等等，方法分别是： `atMost(count), atLeast(count), never()` 等等，都是 Mockito 的静态方法，其实大部分时候我们会 static import Mockito 这个类的所有静态方法，这样就不用每次加上 `Mockito.` 前缀了。本文下面我也按照这个规则。（其实我早就想说这句话啦，只是一直没找到好的时机[喜极而泣]）
很多时候你并不关心被调用方法的参数具体是什么，或者是你也不知道，你只关心这个方法得到调用了就行。这种情况下，Mockito 提供了一系列的 `any` 方法，来表示任何的参数都行：
`Mockito.verify(mockUserManager).performLogin(Mockito.anyString(), Mockito.anyString());`
`anyString()` 表示任何一个字符串都可以。null？也可以的！
类似 `anyString` ，还有 `anyInt, anyLong, anyDouble` 等等。 `anyObject` 表示任何对象， `any(clazz)` 表示任何属于 clazz 的对象。在写这篇文章的时候，我刚刚发现，还有非常有意思也非常人性化的 `anyCollection，anyCollectionOf(clazz), anyList(Map, set), anyListOf(clazz)` 等等。看来我之前写了不少冤枉代码啊 T_T。。。

2. 指定 mock 对象的某些方法的行为

到目前为止，我们介绍了 mock 的一大作用：验证方法调用。我们说 mock 主要有两大作用，第二个大作用是：指定某个方法的返回值，或者是执行特定的动作。
那么接下来，我们就来介绍 mock 的第二大作用，先介绍其中的第一点：指定 mock 对象的某个方法返回特定的值。
现在假设我们上面的 `LoginPresenter` 的 `login` 方法是如下实现的：

```
public void login(String username, String password) {
    if (username == null || username.length() == 0) return;
    //假设我们对密码强度有一定要求，使用一个专门的validator来验证密码的有效性
    if (mPasswordValidator.verifyPassword(password)) return;  //<==
    mUserManager.performLogin(null, password);
}
```

这里，我们有个 `PasswordValidator` 来验证密码的有效性，但是这个类的 `verifyPassword()` 方法运行需要很久，比如说需要联网。这个时候在测试的环境下我们想简单处理，指定让它直接返回 true 或 false。你可能会想，这样做可以吗？真的好吗？回答是肯定的，因为这里我们要测的是 `login()` 这个方法，这其实跟 `PasswordValidator` 内部的逻辑没有太大关系，这才是单元测试真正该有的粒度。
话说回来，这种指定 mock 对象的某个方法，让它返回特定值的写法如下：
`Mockito.when(mockObject.targetMethod(args)).thenReturn(desiredReturnValue);`
应该很好理解，结合上面 PasswordValidator 的例子：

```
//先创建一个mock对象
PasswordValidator mockValidator = Mockito.mock(PasswordValidator.class);
//当调用mockValidator的verifyPassword方法，同时传入"xiaochuang_is_handsome"时，返回true
Mockito.when(mockValidator.verifyPassword("xiaochuang_is_handsome")).thenReturn(true);
//当调用mockValidator的verifyPassword方法，同时传入"xiaochuang_is_not_handsome"时，返回false
Mockito.when(validator.verifyPassword("xiaochuang_is_not_handsome")).thenReturn(false);
```

同样的，你可以用 `any` 系列方法来指定"无论传入任何参数值，都返回 xxx"：

```
//当调用mockValidator的verifyPassword方法时，返回true，无论参数是什么
Mockito.when(validator.verifyPassword(anyString())).thenReturn(true);
```

指定方法返回特定值就介绍到这，更详细更高级的用法大家可以自己 google。接下来介绍，怎么样指定一个方法执行特定的动作，这个功能一般是用在目标的方法是 void 类型的时候。
现在假设我们的 `LoginPresenter` 的 `login()` 方法是这样的：

```
public void loginCallbackVersion(String username, String password) {
    if (username == null || username.length() == 0) return;
    //假设我们对密码强度有一定要求，使用一个专门的validator来验证密码的有效性
    if (mPasswordValidator.verifyPassword(password)) return;
    //login的结果将通过callback传递回来。
    mUserManager.performLogin(username, password, new NetworkCallback() {  //<==
        @Override
        public void onSuccess(Object data) {
            //update view with data
        }
        @Override
        public void onFailure(int code, String msg) {
            //show error msg
        }
    });
}
```

在这里，我们想进一步测试传给 `mUserManager.performLogin` 的 `NetworkCallback` 里面的代码，验证 view 得到了更新等等。在测试环境下，我们并不想依赖 `mUserManager.performLogin` 的真实逻辑，而是让 `mUserManager` 直接调用传入的 `NetworkCallback` 的 `onSuccess` 或 `onFailure` 方法。这种指定 mock 对象执行特定的动作的写法如下：

`Mockito.doAnswer(desiredAnswer).when(mockObject).targetMethod(args);`

传给 `doAnswer()` 的是一个 `Answer` 对象，我们想要执行什么样的动作，就在这里面实现。结合上面的例子解释：

```
Mockito.doAnswer(new Answer() {
   @Override
   public Object answer(InvocationOnMock invocation) throws Throwable {
      //这里可以获得传给performLogin的参数
      Object[] arguments = invocation.getArguments();
      //callback是第三个参数
      NetworkCallback callback = (NetworkCallback) arguments[2];
      callback.onFailure(500, "Server error");
      return 500;
   }
}).when(mockUserManager).performLogin(anyString(), anyString(), any(NetworkCallback.class));
```

这里，当调用 `mockUserManager` 的 `performLogin` 方法时，会执行 answer 里面的代码，我们上面的例子是直接调用传入的 `callback` 的 `onFailure` 方法，同时传给 `onFailure` 方法 500 和"Server error"。

当然，使用 `Mockito.doAnswer()` 需要创建一个 Answer 对象，这有点麻烦，代码看起来也繁琐，如果想简单的指定目标方法“什么都不做”，那么可以使用 `Mockito.doNothing()` 。如果想指定目标方法“抛出一个异常”，那么可以使用 `Mockito.doThrow(desiredException)` 。如果你想让目标方法调用真实的逻辑，可以使用 `Mockito.doCallRealMethod()` 。（什么??? 默认不是会这样吗？?? No! ）

Spy

最后介绍一个 Spy 的东西。前面我们讲了 mock 对象的两大功能，对于第二大功能: 指定方法的特定行为，不知道你会不会好奇，如果我不指定的话，它会怎么样呢？那么现在补充一下，如果不指定的话，一个 mock 对象的所有非 void 方法都将返回默认值：int、long 类型方法将返回 0，boolean 方法将返回 false，对象方法将返回 null 等等；而 void 方法将什么都不做。

然而很多时候，你希望达到这样的效果：除非指定，否者调用这个对象的默认实现，同时又能拥有验证方法调用的功能。这正好是 spy 对象所能实现的效果。创建一个 spy 对象，以及 spy 对象的用法介绍如下：

```
//假设目标类的实现是这样的
public class PasswordValidator {
   public boolean verifyPassword(String password) {
      return "xiaochuang_is_handsome".equals(password);
   }
}
@Test
public void testSpy() {
   //跟创建mock类似，只不过调用的是spy方法，而不是mock方法。spy的用法
   PasswordValidator spyValidator = Mockito.spy(PasswordValidator.class);
   //在默认情况下，spy对象会调用这个类的真实逻辑，并返回相应的返回值，这可以对照上面的真实逻辑
   spyValidator.verifyPassword("xiaochuang_is_handsome"); //true
   spyValidator.verifyPassword("xiaochuang_is_not_handsome"); //false
   //spy对象的方法也可以指定特定的行为
   Mockito.when(spyValidator.verifyPassword(anyString())).thenReturn(true);
   //同样的，可以验证spy对象的方法调用情况
   spyValidator.verifyPassword("xiaochuang_is_handsome");
   Mockito.verify(spyValidator).verifyPassword("xiaochuang_is_handsome"); //pass
}
```

总之，spy 与 mock 的唯一区别就是默认行为不一样：spy 对象的方法默认调用真实的逻辑，mock 对象的方法默认什么都不做，或直接返回默认值。

小结

这篇文章介绍了 mock 的概念以及 Mockito 的使用，可能 Mockito 的很多的一些其他方法没有介绍，但这只是阅读文档的问题而已，更重要的是理解 mock 的概念。
如果你想了解 Mockito 更详细的用法可以参考这篇文章，写的是相当的好。
下一篇文章我们将介绍依赖注入的概念，以及（或许）使用 dagger2 来更方便的做依赖注入，以及在单元测试里面的应用，这里依然后很多的误区，需要大家注意的，想知道具体是什么吗？那就
Stay tuned！
文中代码在Github
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。