Mockito使用介绍

|  |  |  |  |
| --- | --- | --- | --- |
| 版本 | 修订者 | 修订内容 | 时间 |
| 1.0 | 陈航 | 文档初始化 | 2016.06.21 |
| 1.0 | 陈航 | 添加Mockito使用介绍 | 2016.06.21 |

目录

[1. mock和Mockito的关系 3](#_Toc454350201)

[2. Mockito使用举例 3](#_Toc454350202)

[3. Android Studio工程配置Mockito 5](#_Toc454350203)

[4. Mockito使用方法 6](#_Toc454350204)

[4.1. 验证行为 6](#_Toc454350205)

[4.2. Stubbing 7](#_Toc454350206)

[4.3. 参数匹配器 7](#_Toc454350207)

[4.4. 验证准确的调用次数：最多/最少/从未等 8](#_Toc454350208)

[4.5. 为void方法抛异常 8](#_Toc454350209)

[4.6. 验证调用顺序 9](#_Toc454350210)

[4.7. 验证mock对象没有产生过交互 9](#_Toc454350211)

[4.8. 查找是否有未验证的交互 9](#_Toc454350212)

[4.9. @Mock注解 10](#_Toc454350213)

[4.10. 更多的注解：@Captor, @Spy, @InjectMocks 10](#_Toc454350214)

[4.11. 根据调用顺序设置不同的stubbing 11](#_Toc454350215)

[4.12. doReturn()|doThrow()| doAnswer()|doNothing()|doCallRealMethod()等用法 11](#_Toc454350216)

[4.13. spy监视真正的对象 12](#_Toc454350217)

[4.14. 为未stubbing的方法设置默认返回值 12](#_Toc454350218)

[4.15. 参数捕捉器 12](#_Toc454350219)

[4.16. 重置mocks 13](#_Toc454350220)

[4.17. 延时验证 13](#_Toc454350221)

[4.18. 查看对象是否mock或者spy 14](#_Toc454350222)

# mock和Mockito的关系

在软件开发中提及"mock"，通常理解为模拟对象。

为什么需要模拟? 在我们一开始学编程时,我们所写的对象通常都是独立的，并不依赖其他的类，也不会操作别的类。但实际上，软件中是充满依赖关系的，比如我们会基于service类写操作类,而service类又是基于数据访问类(DAO)的，依次下去，形成复杂的依赖关系。

单元测试的思路就是我们想在不涉及依赖关系的情况下测试代码。这种测试可以让你无视代码的依赖关系去测试代码的有效性。核心思想就是如果代码按设计正常工作，并且依赖关系也正常，那么他们应该会同时工作正常。

有些时候，我们代码所需要的依赖可能尚未开发完成，甚至还不存在，那如何让我们的开发进行下去呢？使用mock可以让开发进行下去，**mock技术的目的和作用就是模拟一些在应用中不容易构造或者比较复杂的对象，从而把测试与测试边界以外的对象隔离开**。

我们可以自己编写自定义的Mock对象实现mock技术，但是编写自定义的Mock对象需要额外的编码工作，同时也可能引入错误。现在实现mock技术的优秀开源框架有很多，**Mockito就是一个优秀的用于单元测试的mock框架**。Mockito已经在github上开源，详细请点击：<https://github.com/mockito/mockito>

除了Mockito以外，还有一些类似的框架，比如：

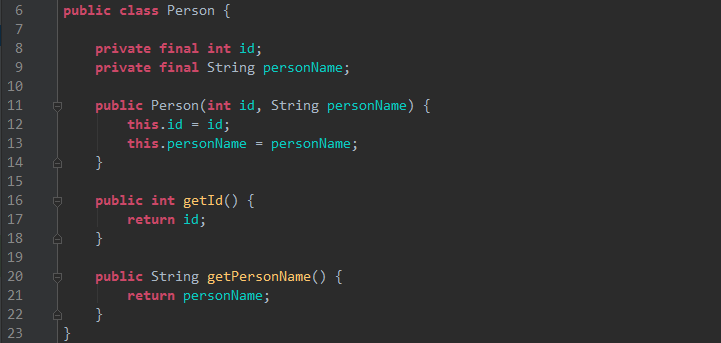
* **EasyMock**：早期比较流行的MocK测试框架。它提供对接口的模拟，能够通过录制、回放、检查三步来完成大体的测试过程，可以验证方法的调用种类、次数、顺序，可以令 Mock 对象返回指定的值或抛出指定异常
* **PowerMock**：这个工具是在EasyMock和Mockito上扩展出来的，目的是为了解决EasyMock和Mockito不能解决的问题，比如对static, final, private方法均不能mock。其实测试架构设计良好的代码，一般并不需要这些功能，但如果是在已有项目上增加单元测试，老代码有问题且不能改时，就不得不使用这些功能了
* **JMockit**：JMockit 是一个轻量级的mock框架是用以帮助开发人员编写测试程序的一组工具和API，该项目完全基于 Java 5 SE 的 java.lang.instrument 包开发，内部使用 ASM 库来修改Java的Bytecode

Mockito已经被广泛应用，所以这里重点介绍Mockito。

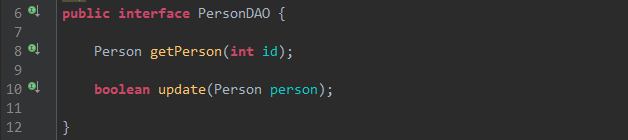
# Mockito使用举例

这里我们直接通过一个代码来说明mockito对单元测试的帮助，代码有三个类，分别截图如下：

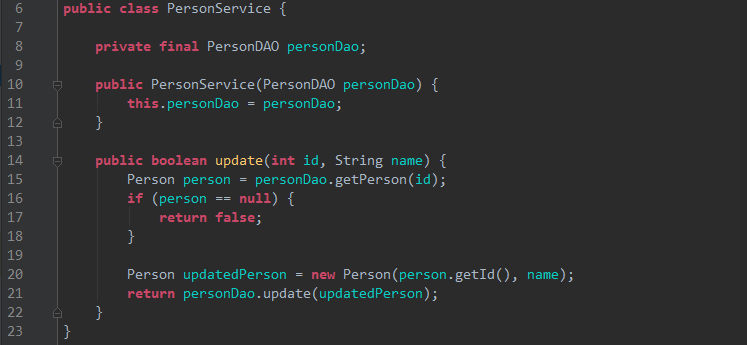
Person类：



PersonDAO：

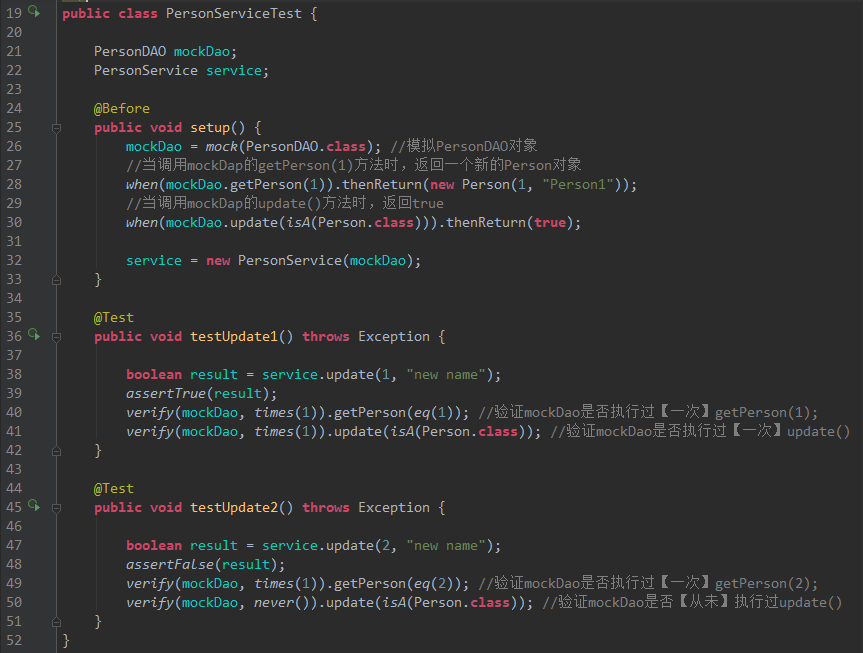


PersonService：



在这里，我们要**进行测试的是PersonService类的update方法**，我们发现，update方法依赖PersonDAO，在开发过程中，**PersonDAO很可能尚未开发完成**，所以我们测试PersonService的时候，所以该怎么测试update方法呢？连接口都还没实现，怎么知道返回的是true还是false？在这里，我们可以这样认为，单元测试的思路就是我们想在不涉及依赖关系的情况下测试代码。这种测试可以让你无视代码的依赖关系去测试代码的有效性。核心思想就是如果代码按设计正常工作，并且依赖关系也正常，那么他们应该会同时工作正常。所以我们的做法是mock一个PersonDAO对象，至于实际环境中，PersonDAO行为是否能按照预期执行，比如update是否能成功，查询是否返回正确的数据，就跟PersonService没关系了。PersonService的单元测试只测试自己的逻辑是否有问题。

下面编写测试代码：



我们对PersonDAO进行mock，并且设置stubbing，stubbing设置如下：

* 当getPerson方法传入1的时候，返回一个Person对象，否则默认返回空
* 当调update方法的时候，返回true

我们验证了两种情况：

* 更新id为1的Person的名字，预期：能在DAO中找到Person并更新成功
* 更新id为2的Person的名字，预期：不能在DAO中找到Person，更新失败

这样，根据PersonService的update方法的逻辑，通过这两个test case之后，我们认为代码是没有问题的。mockito在这里扮演了一个为我们模拟DAO对象，并且帮助我们验证行为（比如验证是否调用了getPerson方法及update方法）的角色。

# Android Studio工程配置Mockito

Android Studio中使用Mockito非常简单，只需要在build.gradle文件中加入依赖即可。如图：



添加依赖之后，点击sync,即可自动下载：

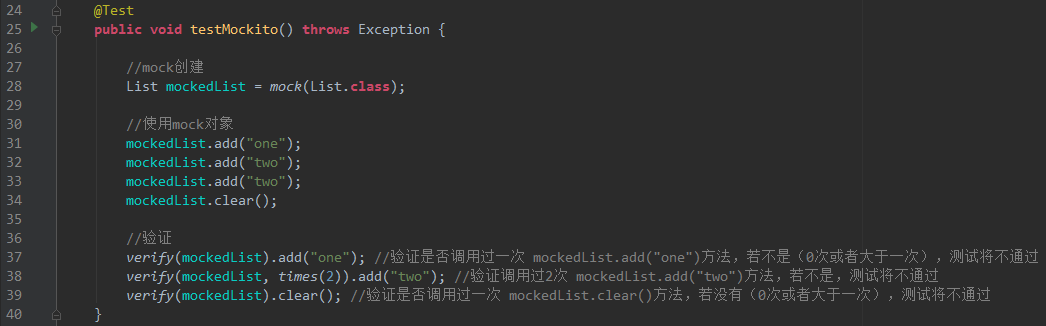


# Mockito使用方法

Mockito的使用，有详细的api文档，具体可以查看：<http://site.mockito.org/mockito/docs/current/org/mockito/Mockito.html>，下面是整理的一些常用的使用方式。

## 验证行为

一旦创建，mock会记录所有交互，你可以验证所有你想要验证的东西



## Stubbing



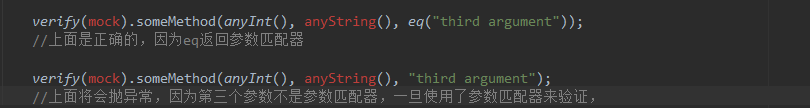
对于stubbing，有以下几点需要注意：

* 对于有返回值的方法，mock会默认返回null、空集合、默认值。比如，为int/Integer返回0，为boolean/Boolean返回false
* stubbing可以被覆盖，但是请注意覆盖已有的stubbing有可能不是很好
* 一旦stubbing，不管调用多少次，方法都会永远返回stubbing的值
* 当你对同一个方法进行多次stubbing，最后一次stubbing是最重要的

## 参数匹配器



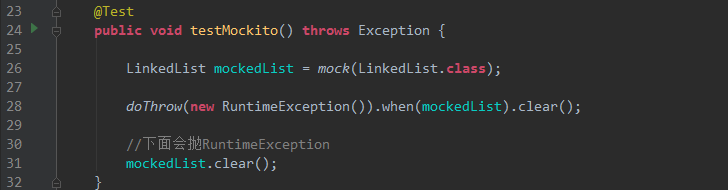
需要注意的是，**如果你使用了参数匹配器，那么所有参数都应该使用参数匹配器。**



## 验证准确的调用次数：最多/最少/从未等



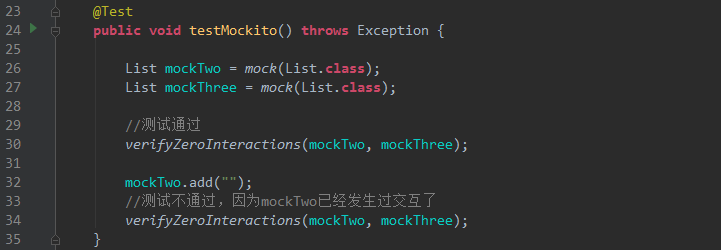
## 为void方法抛异常



## 验证调用顺序



## 验证mock对象没有产生过交互



## 查找是否有未验证的交互

注意：**不建议过多使用**，原文：A word of warning: Some users who did a lot of classic, expect-run-verify mocking tend to use verifyNoMoreInteractions() very often, even in every test method. verifyNoMoreInteractions() is not recommended to use in every test method. verifyNoMoreInteractions() is a handy assertion from the interaction testing toolkit. Use it only when it's relevant. Abusing it leads to overspecified, less maintainable tests.



## @Mock注解



使用注解有以下优点：

* 减少代码
* 增强可读性
* 让verify出错信息更易读，因为变量名可用来描述标记mock对象

## 更多的注解：@Captor, @Spy, @InjectMocks

* **@Capto**r 创建ArgumentCaptor
* **@Spy** 可以代替spy(Object).
* **@InjectMocks** 如果此注解声明的变量需要用到mock对象，mockito会自动注入mock或spy成员

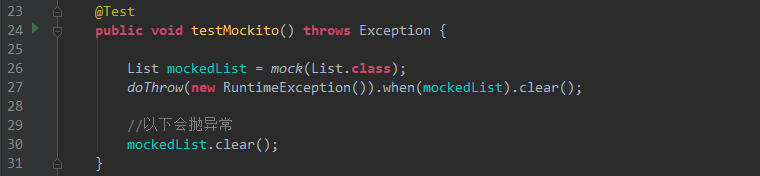
## 根据调用顺序设置不同的stubbing



这样，设置之后，就会根据调用次数返回不同的结果

## doReturn()|doThrow()| doAnswer()|doNothing()|doCallRealMethod()等用法

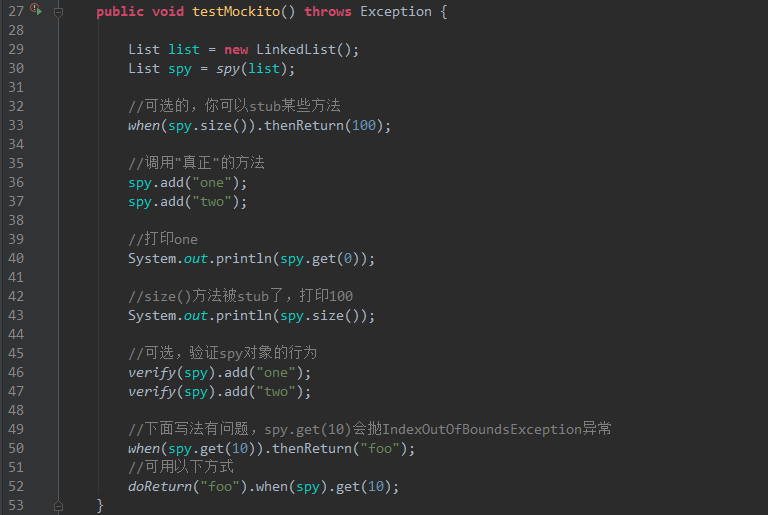
有时候我们需要在调用的时候，模拟抛出异常的情况，Mockito也为我们提供了相应的实现



其他方法的使用类似，不再一一细说，但是要注意：

* doReturn只能使用在有返回值的方法上
* doNothing只能用在无返回值的方法上

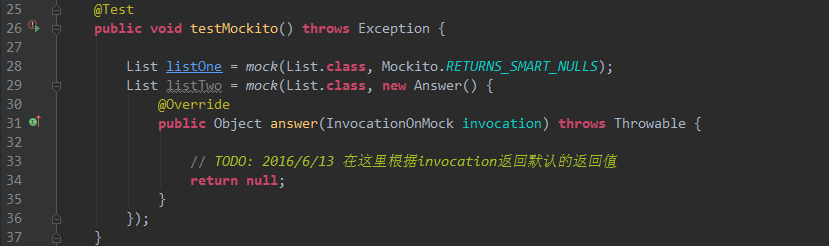
## spy监视真正的对象



使用spy，可以让我们使用对象真正的实现，但又使得对象变得可验证。

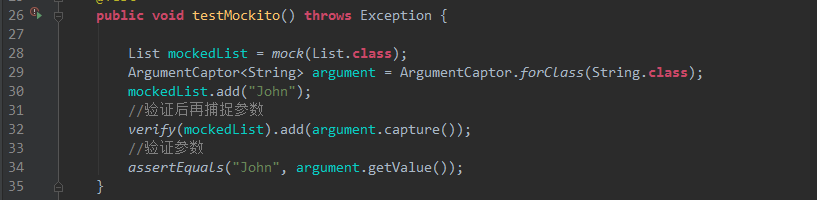
* spy是创建一个拷贝，如果你保留原始的list，并用它来进行操作，那么spy并不能检测到其交互
* spy一个真正的对象+试图stub一个final方法，这样是会有问题的

## 为未stubbing的方法设置默认返回值

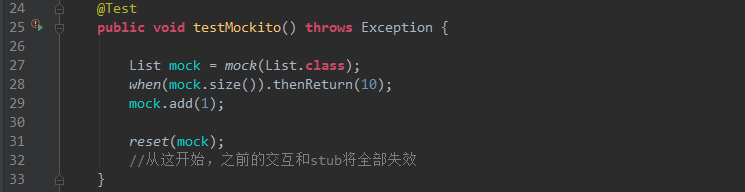


## 参数捕捉器

参数捕捉器在verify之后在assert参数的时候，会很有用



## 重置mocks



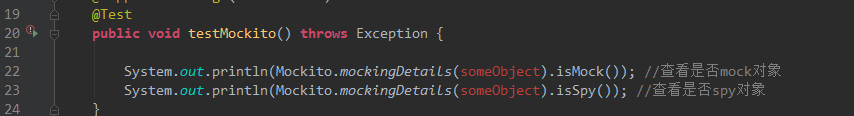
**尽量不要在测试代码中间使用reset**，以下是官方原文：Don't harm yourself. reset() in the middle of the test method is a code smell (you're probably testing too much).

## 延时验证



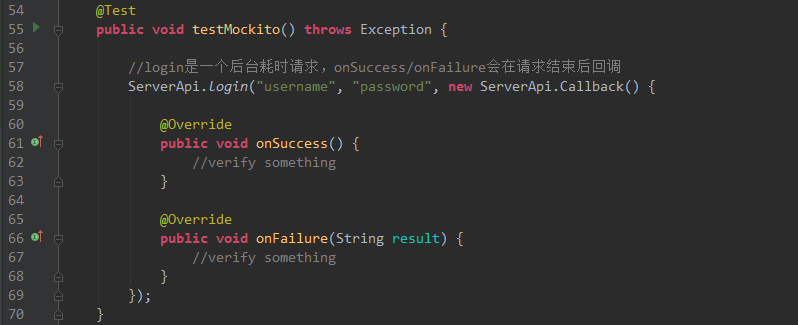
延时验证在并发环境的时候，将会用得上。可以在一段时间过去之后再进行验证。

## 查看对象是否mock或者spy



# 目前发现的不足（或需要讨论）

Mockito在单元测试中可以帮助我们解决很多依赖，也提供了丰富的验证接口，验证我们的程序是否正常运行。但目前发现的一个不足是，在异步调用中将很难测试，比如以下



我们原本的意图是想在请求结束之后，在onSuccess或者onFailure中进行某些验证，但事实上，既然login是一个后台请求，那么就不会阻塞测试线程，测试线程会先于后台线程结束，这样，整个测试也已经结束了，onSuccess和onFailure里的验证将变得不可验证。

以上是我在学习过程中遇到的疑惑，是否这样的测试代码是不好的测试代码？是否有更好的解决办法？