SPECL.COM

学习 Mockito

赵彬

2011-2-15

目录

使用 Mockito 的前期准备

简介

Mockito 是一个流行的 Mocking 框架。它使用起来简单，学习成本很低，而且具有非常简洁的 API，测试代码的可读性很高。因此它十分受欢迎，用户群越来越多，很多的开源的软件也选择了 Mockito。

要想了解更多有关 Mockito 的信息，请访问它的官方网站：http://mockito.org/

Stub 和 Mock

在开始使用 Mockito 之前，先简单的了解一下 Stub 和 Mock 的区别。

Stub 对象用来提供测试时所需要的测试数据，可以对各种交互设置相应的回应。例如我们可以设置方法调用的返回值等等。Mockito 中 `when(…).thenReturn(…)` 这样的语法便是设置方法调用的返回值。另外也可以设置方法在何时调用会抛异常等。

Mock 对象用来验证测试中所依赖对象间的交互是否能够达到预期。Mockito 中用 `verify(…).methodXxx(…)` 语法来验证 methodXxx 方法是否按照预期进行了调用。

有关 stub 和 mock 的详细论述见，Martin Fowler 文章《Mocks Aren't Stub》
http://martinfowler.com/articles/mocksArentStubs.html

在 Mocking 框架中所谓的 mock 对象实际上是作为上述的 stub 和 mock 对象同时使用的。因为它既可以设置方法调用返回值，又可以验证方法的调用。

Mockito 的获取

Jar 包的获取

可以访问下面的链接来下载最新的 Jar 包，笔者使用的当前最新版为：1.8.5
http://code.google.com/p/mockito/downloads/list

Maven

如果项目是通过 Maven 管理的，需要在项目的 Pom.xml 中增加如下的依赖：

```
<dependencies>
    <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-all</artifactId>
        <version>1.8.5</version>
        <scope>test</scope>
```

```
    </dependency>
</dependencies>
```

从一个实例开始

Mocktio 包的引入

在程序中可以`import org.mockito.Mockito;`然后调用它的`static`方法，或者`import static org.mockito.Mockito.*;`个人倾向于后者，因为这样可以更方便些。

一个简单的例子

```
import static org.junit.Assert.*;
import static org.mockito.Mockito.*;
import java.util.Iterator;
import org.junit.Test;

/**
 *
 * @author Brian Zhao
 */
public class SimpleTest {

    @Test
    public void simpleTest(){
        //arrange
        Iterator i=mock(Iterator.class);
        when(i.next()).thenReturn("Hello").thenReturn("World");
        //act
        String result=i.next()+" "+i.next();
        //verify
        verify(i, times(2)).next();
        //assert
        assertEquals("Hello World", result);
    }
}
```

在上面的例子中包含了 Mockito 的基本功能：

创建 Mock 对象

创建 Mock 对象的语法为，`mock(class or interface)`。例子中创建了 Iterator

接口的 mock 对象。

设置方法调用的预期返回

通过 when(mock.someMethod()).thenReturn(value) 来设定 mock 对象某个方法调用时的返回值。例子中我们对 Iterator 接口的 next()方法调用进行了预期设定，当调用 next()方法时会返回"Hello"，由于连续设定了返回值，因此当第二次调用时将返回"World"。

验证方法调用

接下来对 mock 对象的 next()方法进行了一系列实际的调用。mock 对象一旦建立便会自动记录自己的交互行为，所以我们可以有选择的对它的交互行为进行验证。在 Mockito 中验证 mock 对象交互行为的方法是 verify(mock).someMethod(…)。于是用此方法验证了 next()方法调用，因为调用了两次，所以在 verify 中我们指定了 times 参数（times 的具体应用在后面会继续介绍）。最后 assert 返回值是否和预期一样。

Mock 对象的创建和 Stubbing

Mock 对象的创建

```
mock(Class<T> classToMock)
mock(Class<T> classToMock, String name)
```

可以对类和接口进行 mock 对象的创建，创建的时候可以为 mock 对象命名，也可以忽略命名参数。为 mock 对象命名的好处就是调试的时候会很方便，比如，我们 mock 多个对象，在测试失败的信息中会把有问题的 mock 对象打印出来，有了名字我们可以很容易定位和辨认出是哪个 mock 对象出现的问题。另外它也有限制，对于 final 类、匿名类和 Java 的基本类型是无法进行 mock 的。

Mock 对象的期望行为及返回值设定

我们已经了解到可以通过 when(mock.someMethod()).thenReturn(value) 来设定 mock 对象的某个方法调用时的返回值，但它也同样有限制对于 static 和 final 修饰的方法是无法进行设定的。下面来详细的介绍一下有关方法及返回值的设定：

首先假设我们创建Iterator接口的mock对象

```
Iterator<String> i = mock(Iterator.class);
```

对方法设定返回值

```
when(i.next()).thenReturn("Hello")
```

对方法设定返回异常

```
when(i.next()).thenThrow(new RuntimeException())
```

Mockito 支持迭代风格的返回值设定

第一种方式

```
when(i.next()).thenReturn("Hello").thenReturn("World")
```

第二种方式

```
when(i.next()).thenReturn("Hello", "World")
```

上面的设定相当于：

```
when(i.next()).thenReturn("Hello")
when(i.next()).thenReturn("World")
```

第一次调用 i.next()将返回"Hello"，第二次的调用会返回"World"。

Stubbing 的另一种语法

doReturn(Object) 设置返回值

```
doReturn("Hello").when(i).next();
```

迭代风格

```
doReturn("Hello").doReturn("World").when(i).next();
```

返回值的次序为从左至右，第一次调用返回"Hello"，第二次返回"World"。

doThrow(Throwable) 设置返回异常

```
doThrow(new RuntimeException()).when(i).next();
```

因为这种语法的可读性不如前者，所以能使用前者的情况下尽量使用前者，当然在后面要介绍的 Spy 除外。

对 void 方法进行方法预期设定

void 方法的模拟不支持 when(mock.someMethod()).thenReturn(value)这样的语法，只支持下面的方式：

doNothing() 模拟不做任何返回（mock 对象 void 方法的默认返回）

```
doNothing().when(i).remove();
```

doThrow(Throwable) 模拟返回异常

```
doThrow(new RuntimeException()).when(i).remove();
```

迭代风格

```
doNothing().doThrow(new RuntimeException()).when(i).remove();
```

第一次调用 remove 方法什么都不做，第二次调用抛出 RuntimeException 异常。

Argument Matcher（参数匹配器）

Mockito 通过 equals()方法，来对方法参数进行验证。但有时我们需要更加灵活的参数需求，比如，匹配任何的 String 类型的参数等等。参数匹配器就是一个能够满足这些需求的工具。

Mockito 框架中的 Matchers 类内建了很多参数匹配器，而我们常用的 Mockito 对象便是继承自 Matchers。这些内建的参数匹配器如，anyInt()匹配任何 int 类型参数，anyString()匹配任何字符串，anySet()匹配任何 Set 等。下面通过例子来说明如何使用内建的参数匹配器：

```
@Test
public void argumentMatchersTest(){
    List<String> mock = mock(List.class);
    when(mock.get(anyInt())).thenReturn("Hello").thenReturn("World
    ");
    String result=mock.get(100)+" "+mock.get(200);
    verify(mock,times(2)).get(anyInt());
    assertEquals("Hello World",result);
}
```

Stubbing 时使用内建参数匹配器

例子中，首先 mock 了 List 接口，然后用迭代的方式模拟了 get 方法的返回值，这里用了 anyInt()参数匹配器来匹配任何的 int 类型的参数。所以当第一次调用 get 方法时输入任意参数为 100 方法返回”Hello”，第二次调用时输入任意参数 200 返回值”World”。

Verfiy 时使用参数匹配器

最后进行 verfiy 验证的时候也可将参数指定为 anyInt()匹配器，那么它将不关心调用时输入的参数的具体参数值。

注意事项

如果使用了参数匹配器，那么所有的参数需要由匹配器来提供，否则将会报错。假如我们使用参数匹配器 stubbing 了 mock 对象的方法，那么在 verify 的时候也需要使用它。如：

```
@Test
public void argumentMatchersTest(){
    Map mapMock = mock(Map.class);
    when(mapMock.put(anyInt(), anyString())).thenReturn("world");
    mapMock.put(1, "hello");
    verify(mapMock).put(anyInt(), eq("hello"));
}
```

在最后的验证时如果只输入字符串"hello"是会报错的，必须使用 Matchers 类内建的 eq 方法。如果将 anyInt()换成 1 进行验证也需要用 eq(1)。

详细的内建参数匹配器请参考：
http://docs.mockito.googlecode.com/hg/org/mockito/Matchers.html

Mock 对象的行为验证

之前介绍了如何设置 mock 对象预期调用的方法及返回值。下面介绍方法调用的验证，而它关注点则在 mock 对象的交互行为上，比如验证 mock 对象的某个方法调用参数，调用次数，顺序等等。下面来看例子：

```
@Test
public void verifyTestTest() {
    List<String> mock = mock(List.class);
    List<String> mock2 = mock(List.class);

    when(mock.get(0)).thenReturn("hello");

    mock.get(0);
    mock.get(1);
    mock.get(2);

    mock2.get(0);

    verify(mock).get(2);
    verify(mock, never()).get(3);
    verifyNoMoreInteractions(mock);
    verifyZeroInteractions(mock2);
}
```

验证的基本方法
我们已经熟悉了使用`verify(mock).someMethod(…)`来验证方法的调用。例子中，我们 mock 了 List 接口，然后调用了 mock 对象的一些方法。验证是否调用了`mock.get(2)`方法可以通过`verify(mock).get(2)`来进行。verify 方法的调用不关心是否模拟了 get(2)方法的返回值，只关心 mock 对象后，是否执行了 mock.get(2)，如果没有执行，测试方法将不会通过。

验证未曾执行的方法
在 verify 方法中可以传入 never()方法参数来确认`mock.get(3)`方法不曾被执行过。另外还有很多调用次数相关的参数将会在下面提到。

查询多余的方法调用

verifyNoMoreInteractions()方法可以传入多个 mock 对象作为参数，用来验证传入的这些 mock 对象是否存在没有验证过的调用方法。本例中传入参数 mock，测试将不会通过，因为我们只 verify 了 mock 对象的 get(2)方法，没有对 get(0)和 get(1)进行验证。为了增加测试的可维护性，官方不推荐我们过于频繁的在每个测试方法中都使用它，因为它只是测试的一个工具，只在你认为有必要的时候才用。

查询没有交互的 mock 对象

verifyZeroInteractions()也是一个测试工具，源码和 verifyNoMoreInteractions()的实现是一样的，为了提高逻辑的可读性，所以只不过名字不同。在例子中，它的目的是用来确认 mock2 对象没有进行任何交互，但 mock2 执行了 get(0)方法，所以这里测试会报错。由于它和 verifyNoMoreInteractions()方法实现的源码都一样，因此如果在 `verifyZeroInteractions(mock2)`执行之前对 `mock.get(0)`进行了验证那么测试将会通过。

对 Mock 对象方法的调用次数、顺序和超时进行验证

验证方法调用的次数

如果要验证 Mock 对象的某个方法调用次数，则需给 verify 方法传入相关的验证参数，它的调用接口是 verify(T mock, VerificationMode mode)。如：`verify(mock,times(3)).someMethod(argument)` 验证 mock 对象 `someMethod(argument)`方法是否调用了三次。times(N)参数便是验证调用次数的参数，N 代表方法调用次数。其实 verify 方法中如果不传调用次数的验证参数，它默认传入的便是 times(1)，即验证 mock 对象的方法是否只被调用一次，如果有多次调用测试方法将会失败。

Mockito 除了提供 times(N)方法供我们调用外，还提供了很多可选的方法：
never() 没有被调用，相当于 times(0)
atLeast(N) 至少被调用 N 次
atLeastOnce() 相当于 atLeast(1)
atMost(N) 最多被调用 N 次

超时验证

Mockito 提供对超时的验证，但是目前不支持在下面提到的顺序验证中使用。进行超时验证和上述的次数验证一样，也要在 verify 中进行参数的传入，参数为 timeout(int millis)，timeout 方法中输入的是毫秒值。下面看例子：

验证 `someMethod()`是否能在指定的 `100` 毫秒中执行完毕

```
verify(mock, timeout(100)).someMethod();
```

结果和上面的例子一样，在超时验证的同时可进行调用次数验证，默认次数为 1
```
verify(mock, timeout(100).times(1)).someMethod();
```

在给定的时间内完成执行次数
```
verify(mock, timeout(100).times(2)).someMethod();
```

给定的时间内至少执行两次
```
verify(mock, timeout(100).atLeast(2)).someMethod();
```

另外 timeout 也支持自定义的验证模式，

```
verify(mock, new Timeout(100,
yourOwnVerificationMode)).someMethod();
```

验证方法调用的顺序

Mockito 同样支持对不同 Mock 对象不同方法的调用次序进行验证。进行次序验证是，我们需要创建 InOrder 对象来进行支持。例：

创建 mock 对象
```
List<String> firstMock = mock(List.class);
List<String> secondMock = mock(List.class);
```

调用 mock 对象方法
```
firstMock.add("was called first");
firstMock.add("was called first");
secondMock.add("was called second");
secondMock.add("was called third");
```

创建 InOrder 对象
inOrder 方法可以传入多个 mock 对象作为参数，这样便可对这些 mock 对象的方法进行调用顺序的验证 `InOrder inOrder = inOrder( secondMock, firstMock );`

验证方法调用

接下来我们要调用 InOrder 对象的 verify 方法对 mock 方法的调用顺序进行验证。注意，这里必须是你对调用顺序的预期。

InOrder 对象的 verify 方法也支持调用次数验证，上例中，我们期望`firstMock.add("was called first")`方法先执行并执行两次，所以进行了下面的验证`inOrder.verify(firstMock,times(2)).add("was called first")`。其次执行了`secondMock.add("was called second")`方法，继续验证此方法的执行`inOrder.verify(secondMock).add("was called second")`。如果 mock

方法的调用顺序和InOrder中verify的顺序不同，那么测试将执行失败。

InOrder 的 verifyNoMoreInteractions()方法

它用于确认上一个顺序验证方法之后，mock 对象是否还有多余的交互。它和 Mockito 提供的静态方法 verifyNoMoreInteractions 不同，InOrder 的验证是基于顺序的，另外它只验证创建它时所提供的 mock 对象，在本例中只对 firstMock 和 secondMock 有效。例如：

```
inOrder.verify(secondMock).add("was called second");
inOrder.verifyNoMoreInteractions();
```

在验证`secondMock.add("was called second")`方法之后，加上InOrder的`verifyNoMoreInteractions`方法，表示此方法调用后再没有多余的交互。例子中会报错，因为在此方法之后还执行了`secondMock.add("was called third")`。现在将上例改成：

```
inOrder.verify(secondMock).add("was called third");
inOrder.verifyNoMoreInteractions();
```

测试会恢复为正常，因为在`secondMock.add("was called third")`之后已经没有多余的方法调用了。如果这里换成Mockito类的verifyNoMoreInteractions方法测试还是会报错，它查找的是mock对象中是否存在没有验证的调用方法，和顺序是无关的。

Mock 对象的重置

Mockito 提供了 reset(mock1,mock2.....)方法，用来重置 mock 对象。当 mock 对象被重置后，它将回到刚创建完的状态，没有任何 stubbing 和方法调用。这个特性平时是很少用到的，因为我们大都为每个 test 方法创建 mock，所以没有必要对它进行重置。官方提供这个特性的唯一目的是使得我们能在有容器注入的 mock 对象中工作更为方便。所以，当决定要使用这个方法的时候，首先应该考虑一下我们的测试代码是否简洁和专注，测试方法是否已经超长了。

Answer 接口（方法预期回调接口）的应用

Answer 接口说明

对 mock 对象的方法进行调用预期的设定，可以通过 thenReturn()来指定返回值，thenThrow()指定返回时所抛异常，通常来说这两个方法足以应对一般的需求。但有时我们需要自定义方法执行的返回结果，Answer 接口就是满足这样的需求而存在的。另外，创建 mock 对象的时候所调用的方法也可以传入 Answer 的实例 mock(java.lang.Class<T> classToMock, Answer defaultAnswer)，它可以用来处理那

些 mock 对象没有 stubbing 的方法的返回值。

InvocationOnMock 对象的方法

Answer 接口定义了参数为 InvocationOnMock 对象的 answer 方法，利用 InvocationOnMock 提供的方法可以获取 mock 方法的调用信息。下面是它提供的方法：

getArguments() 调用后会以 Object 数组的方式返回 mock 方法调用的参数。
getMethod() 返回 java.lang.reflect.Method 对象
getMock() 返回 mock 对象
callRealMethod() 真实方法调用，如果 mock 的是接口它将会抛出异常

通过一个例子来看一下 Answer 的使用。我们自定义 CustomAnswer 类，它实现了 Answer 接口，返回值为 String 类型。

```
public class CustomAnswer implements Answer<String> {
    public String answer(InvocationOnMock invocation) throws
Throwable {
        Object[] args = invocation.getArguments();
        Integer num = (Integer)args[0];
        if( num>3 ){
            return "yes";
        } else {
            throw new RuntimeException();
        }
    }
}
```

这个返回值是这样的逻辑，如果调用 mock 某个方法输入的参数大于 3 返回”yes”，否则抛出异常。

Answer 接口的使用

应用方式如下：

首先对List接口进行mock

```
List<String> mock = mock(List.class);
```

指定方法的返回处理类CustomAnswer，因为参数为4大于3所以返回字符串”yes”

```
when(mock.get(4)).thenAnswer(new CustomAnswer());
```

另外一种方式

```
doAnswer(new CustomAnswer()).when(mock.get(4));
```

对 void 方法也可以指定 Answer 来进行返回处理，如：

```
doAnswer(new xxxAnswer()).when(mock).clear();
```

当设置了Answer后，指定方法的调用结果就由我们定义的Answer接口来处理了。

另外我们也可以使用匿名内部类来进行应用：

```
@Test
public void customAnswerTest(){
    List<String> mock = mock(List.class);
    when(mock.get(4)).thenAnswer(new Answer(){
    public String answer(InvocationOnMock invocation) throws
Throwable {
          Object[] args = invocation.getArguments();
          Integer num = (Integer)args[0];
         if( num>3 ){
             return "yes";
          } else {
             throw new RuntimeException();
          }
        }
    });
    System.out.println(mock.get(4));
}
```

自定义参数匹配器

Mockito 参数匹配器的实现使用了 Hamcrest 框架（一个书写匹配器对象时允许直接定义匹配规则的框架，网址：http://code.google.com/p/hamcrest/）。它已经提供了许多规则供我们使用， Mockito 在此基础上也内建了很规则。但有时我们还是需要更灵活的匹配，所以需要自定义参数匹配器。

ArgumentMatcher 抽象类

自定义参数匹配器的时候需要继承 ArgumentMatcher 抽象类，它实现了 Hamcrest 框架的 Matcher 接口，定义了 describeTo 方法，所以我们只需要实现 matches 方法在其中定义规则即可。

下面自定义的参数匹配器是匹配 size 大小为 2 的 List：

```
class IsListOfTwoElements extends ArgumentMatcher<List> {
    public boolean matches(Object list) {
        return ((List) list).size() == 2;
    }
}

@Test
public void argumentMatchersTest(){
```

```
    List mock = mock(List.class);
    when(mock.addAll(argThat(new
    IsListOfTwoElements()))).thenReturn(true);

    mock.addAll(Arrays.asList("one", "two", "three"));
    verify(mock).addAll(argThat(new IsListOfTwoElements()));
}
```

argThat(Matcher<T> matcher)方法用来应用自定义的规则，可以传入任何实现 Matcher 接口的实现类。上例中在 stubbing 和 verify addAll 方法时通过 argThat(Matcher<T> matcher)，传入了自定义的参数匹配器 IsListOfTwoElements 用来匹配 size 大小为 2 的 List。因为例子中传入 List 的元素为三个，所以测试将失败。

较复杂的参数匹配将会降低测试代码的可读性。有时实现参数对象的 equals()方法是个不错的选择（Mockito 默认使用 equals()方法进行参数匹配），它可以使测试代码更为整洁。另外，有些场景使用参数捕获器（ArgumentCaptor）要比自定义参数匹配器更加合适。

利用 ArgumentCaptor（参数捕获器）捕获方法参数进行验证

在某些场景中，不光要对方法的返回值和调用进行验证，同时需要验证一系列交互后所传入方法的参数。那么我们可以用参数捕获器来捕获传入方法的参数进行验证，看它是否符合我们的要求。

ArgumentCaptor 介绍

通过 ArgumentCaptor 对象的 forClass(Class<T> clazz)方法来构建 ArgumentCaptor 对象。然后便可在验证时对方法的参数进行捕获，最后验证捕获的参数值。如果方法有多个参数都要捕获验证，那就需要创建多个 ArgumentCaptor 对象处理。

ArgumentCaptor 的 Api

argument.capture() 捕获方法参数
argument.getValue() 获取方法参数值，如果方法进行了多次调用，它将返回最后一个参数值
argument.getAllValues() 方法进行多次调用后，返回多个参数值

应用实例

```
@Test
public void argumentCaptorTest() {
    List mock = mock(List.class);
    List mock2 = mock(List.class);
    mock.add("John");
```

```
    mock2.add("Brian");
    mock2.add("Jim");

    ArgumentCaptor argument = ArgumentCaptor.forClass(String.class);

    verify(mock).add(argument.capture());
    assertEquals("John", argument.getValue());

    verify(mock2, times(2)).add(argument.capture());

    assertEquals("Jim", argument.getValue());
    assertArrayEquals(new
    Object[]{"Brian","Jim"},argument.getAllValues().toArray());
}
```

首先构建 ArgumentCaptor 需要传入捕获参数的对象，例子中是 String。接着要在 verify 方法的参数中调用 `argument.capture()`方法来捕获输入的参数，之后 `argument` 变量中就保存了参数值，可以用 `argument.getValue()`获取。当某个对象进行了多次调用后，如 `mock2` 对象，这时调用 `argument.getValue()`获取到的是最后一次调用的参数。如果要获取所有的参数值可以调用 `argument.getAllValues()`，它将返回参数值的 `List`。

在某种程度上参数捕获器和参数匹配器有很大的相关性。它们都用来确保传入 mock 对象参数的正确性。然而，当自定义的参数匹配器的重用性较差时，用参数捕获器会更合适，只需在最后对参数进行验证即可。

Spy-对象的监视

Mock 对象只能调用 stubbed 方法，调用不了它真实的方法。但 Mockito 可以监视一个真实的对象，这时对它进行方法调用时它将调用真实的方法，同时也可以 stubbing 这个对象的方法让它返回我们的期望值。另外不论是否是真实的方法调用都可以进行 verify 验证。和创建 mock 对象一样，对于 final 类、匿名类和 Java 的基本类型是无法进行 spy 的。

监视对象

监视一个对象需要调用 spy(T object)方法，如：`List spy = spy(new LinkedList());`那么 `spy` 变量就在监视 `LinkedList` 实例。

被监视对象的 Stubbing

stubbing 被监视对象的方法时要慎用 when(Object)，如：

```
List spy = spy(new LinkedList());
```

```
//Impossible: real method is called so spy.get(0) throws
IndexOutOfBoundsException (the list is yet empty)
when(spy.get(0)).thenReturn("foo");
//You have to use doReturn() for stubbing
doReturn("foo").when(spy).get(0);
```

当调用 when(spy.get(0)).thenReturn("foo")时，会调用真实对象的 get(0)，由于 list 是空的所以会抛出 IndexOutOfBoundsException 异常，用 doReturn 可以避免这种情况的发生，因为它不会去调用 get(0)方法。

下面是官方文档给出的例子：

```
@Test
public void spyTest2() {

    List list = new LinkedList();
    List spy = spy(list);

    //optionally, you can stub out some methods:
    when(spy.size()).thenReturn(100);

    //using the spy calls real methods
    spy.add("one");
    spy.add("two");

    //prints "one" - the first element of a list
    System.out.println(spy.get(0));

    //size() method was stubbed - 100 is printed
    System.out.println(spy.size());

    //optionally, you can verify
    verify(spy).add("one");
    verify(spy).add("two");
}
```

RETURNS_SMART_NULLS 和 RETURNS_DEEP_STUBS

RETURNS_SMART_NULLS

RETURNS_SMART_NULLS 是实现了 Answer 接口的对象，它是创建 mock 对象时的一个可选参数，mock(Class, Answer)。在创建 mock 对象时，有的方法我们没有进行 stubbing，所以在调用的时候有时会返回 Null 这样在进行处理时就很可能抛出

NullPointerException。如果通过 RETURNS_SMART_NULLS 参数来创建的 mock 对象在调用没有 stubbed 的方法时他将返回 SmartNull。例如：返回类型是 String 它将返回空字符串""；是 int，它将返回 0；如果是 List，它会返回一个空的 List。另外，在堆栈中可以看到 SmartNull 的友好提示。

```
@Test
public void returnsSmartNullsTest() {
    List mock = mock(List.class, RETURNS_SMART_NULLS);
    System.out.println(mock.get(0));
    System.out.println(mock.toArray().length);
}
```

由于使用了 RETURNS_SMART_NULLS 参数来创建 mock 对象，所以在执行下面的操作时将不会抛出 NullPointerException 异常，另外堆栈也提示了相关的信息“SmartNull returned by unstubbed get() method on mock”。

RETURNS_DEEP_STUBS

同上面的参数一样 RETURNS_DEEP_STUBS 也是一个创建 mock 对象时的备选参数。例如我们有 Account 对象和 RailwayTicket 对象，RailwayTicket 是 Account 的一个属性。

```
public class Account {
    private RailwayTicket railwayTicket;
    public RailwayTicket getRailwayTicket() {
        return railwayTicket;
    }
    public void setRailwayTicket(RailwayTicket railwayTicket) {
        this.railwayTicket = railwayTicket;
    }
}

public class RailwayTicket {
    private String destination;
    public String getDestination() {
        return destination;
    }
    public void setDestination(String destination) {
        this.destination = destination;
    }
}
```

下面通过 RETURNS_DEEP_STUBS 来创建 mock 对象。

```
@Test
```

```
public void deepstubsTest(){
    Account account = mock(Account.class, RETURNS_DEEP_STUBS);
    when(account.getRailwayTicket().getDestination()).thenReturn("
    Beijing");
    account.getRailwayTicket().getDestination();
    verify(account.getRailwayTicket()).getDestination();
    assertEquals("Beijing",
    account.getRailwayTicket().getDestination());
}
```

上例中，我们只创建了 Account 的 mock 对象，没有对 RailwayTicket 创建 mock，因为通过 RETURNS_DEEP_STUBS 参数程序会自动进行 mock 所需要的对象，所以上面的例子等价于：

```
@Test
public void deepstubsTest2(){
    Account account = mock(Account.class);
    RailwayTicket railwayTicket = mock(RailwayTicket.class);

    when(account.getRailwayTicket()).thenReturn(railwayTicket);
    when(railwayTicket.getDestination()).thenReturn("Beijing");

    account.getRailwayTicket().getDestination();

    verify(account.getRailwayTicket()).getDestination();
    assertEquals("Beijing",
    account.getRailwayTicket().getDestination());
}
```

为了代码整洁和确保它的可读性，我们应该少用这个特性。

Mockito 对 Annotation 的支持

Mockito 支持对变量进行注解，例如将 mock 对象设为测试类的属性，然后通过注解的方式@Mock 来定义它，这样有利于减少重复代码，增强可读性，易于排查错误等。除了支持@Mock，Mockito 支持的注解还有@Spy（监视真实的对象），@Captor（参数捕获器），@InjectMocks（mock 对象自动注入）。

Annotation 的初始化

只有 Annotation 还不够，要让它们工作起来还需要进行初始化工作。初始化的方法为：`MockitoAnnotations.initMocks(testClass)`参数 `testClass` 是你所写的测试类。一般情况下在 `Junit4` 的`@Before` 定义的方法中执行初始化工作，如下：

```
@Before
public void initMocks() {
    MockitoAnnotations.initMocks(this);
}
```

除了上述的初始化的方法外，还可以使用 Mockito 提供的 Junit Runner：MockitoJUnitRunner 这样就省略了上面的步骤。

```
@RunWith(MockitoJUnit44Runner.class)
public class ExampleTest {
    ...
}
```

@Mock 注解

使用@Mock 注解来定义 mock 对象有如下的优点：

1. 方便 mock 对象的创建
2. 减少 mock 对象创建的重复代码
3. 提高测试代码可读性
4. 变量名字作为 mock 对象的标示，所以易于排错

@Mock 注解也支持自定义 name 和 answer 属性。下面是官方给出的@Mock 使用的例子：

```
public class ArticleManagerTest extends SampleBaseTestCase {
    @Mock
    private ArticleCalculator calculator;
    @Mock(name = "dbMock")
    private ArticleDatabase database;
    @Mock(answer = RETURNS_MOCKS)
    private UserProvider userProvider;

    private ArticleManager manager;

    @Before
    public void setup() {
        manager = new ArticleManager(userProvider, database,
calculator);
    }
}

public class SampleBaseTestCase {
    @Before
    public void initMocks() {
        MockitoAnnotations.initMocks(this);
```

```
    }
}
```

@Spy 注解

Spy 的使用方法请参阅前面的章节，在此不再赘述，下面是使用方法：

```
public class Test{
    @Spy
    Foo spyOnFoo = new Foo();

    @Before
    public void init(){
       MockitoAnnotations.initMocks(this);
    }
    ...
}
```

@Captor 注解

@Captor 是参数捕获器的注解，有关用法见前章，通过注解的方式也可以更便捷的对它进行定义。使用例子如下：

```
public class Test {
    @Captor
    ArgumentCaptor<AsyncCallback<Foo>> captor;
    @Before
    public void init() {
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void shouldDoSomethingUseful() {
        // ...
        verify(mock.doStuff(captor.capture()));
        assertEquals("foo", captor.getValue());
    }
}
```

@InjectMocks 注解

通过这个注解，可实现自动注入 mock 对象。当前版本只支持 setter 的方式进行注入，Mockito 首先尝试类型注入，如果有多个类型相同的 mock 对象，那么它会根据名称进行注入。当注入失败的时候 Mockito 不会抛出任何异常，所以你可能需要手动去验证它的安全性。

例：

```
@RunWith(MockitoJUnit44Runner.class)
public class ArticleManagerTest {
    @Mock
    private ArticleCalculator calculator;
    @Mock
    private ArticleDatabase database;
    @Spy
    private UserProvider userProvider = new ConsumerUserProvider();
    @InjectMocks
    private ArticleManager manager = new ArticleManager();

    @Test
    public void shouldDoSomething() {
        manager.initiateArticle();
        verify(database).addListener(any(ArticleListener.class));
    }
}
```

上例中，ArticleDatabase 是 ArticleManager 的一个属性，由于 ArticleManager 是注解@InjectMocks 标注的，所以会根据类型自动调用它的 setter 方法为它设置 ArticleDatabase。