Mockito

作者：Alan/阿风

该文档参考了大量的网络资源，如果有引用到您的文章，请告知，为其署名

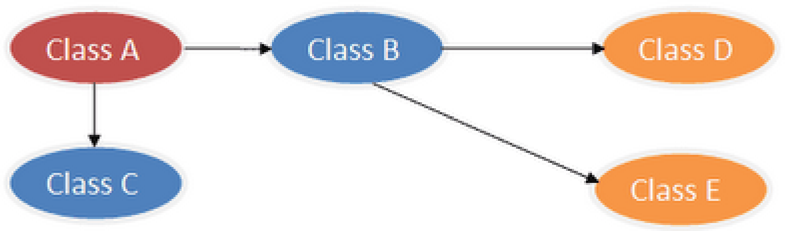
该文档仅用于技术分享

文档尚有不足之处，请见谅

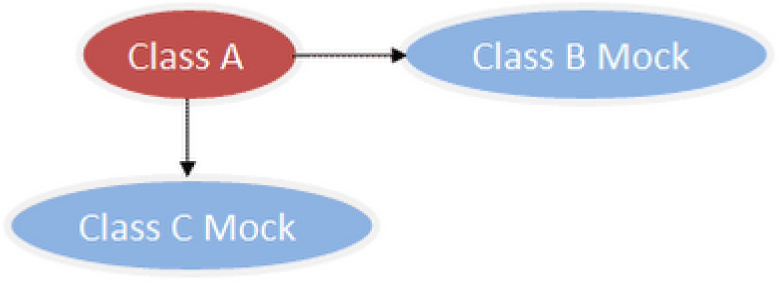
前置了解

mock;为什么使用mocking

测试驱动的开发(TDD)要求我们先写单元测试，再写实现代码。在写单元测试的过程中，我们往往会遇到要测试的类有很多依赖，这些依赖的类/对象/资源又有别的依赖，从而形成一个大的依赖树，要在单元测试的环境中完整地构建这样的依赖，是一件很困难的事情。如下图所示：



为了测试类A，我们需要Mock B类和C类（用虚拟对象来代替）如下图所示：



Stub

Stub: For replacing a method with code that returns a specified result

Stub和Mock异同

- 相同：Stub和Mock都是模拟外部依赖
- 不同：Stub是完全模拟一个外部依赖， 而Mock还可以用来判断测试通过还是失败

> mock除了保证stub的功能之外，还可深入的模拟对象之间的交互方式
>
> stub存在的意图是为了让测试对象可以正常的执行

为什么选用Mockito?

> (っ•̀ω•́)っ✎⁾⁾ 使用方便

常见的Mocking框架有EasyMock、Mockito、PowerMock和JMockit。

- EasyMock最早出现，设计最严谨，但是使用也最不方便。
- Mockito去掉了EasyMock的部分概念，使用方便。
- EasyMock和Mockito的功能都有局限，要支持对构造函数, static方法, final方法, private方法的的Mock，还必须借助于PowerMock。当然，PowerMock也离不开EasyMock和Mockito。所以，最常用的是Mockito和PowerMock的组合。
- JMockit能够不借助于容器对JavaEE项目进行测试。

使用简介

Mockito资源

> 官网： http://mockito.org
>
> 项目源码：https://github.com/mockito/mockito
>
> javadoc:http://www.javadoc.io/doc/org.mockito/mockito-core
>
> maven-central:http://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.mockito%22%20AND%20a%3A%22mockito-core%22

使用场景

- 提前创建测试; TDD（测试驱动开发）
- 团队可以并行工作
- 你可以创建一个验证或者演示程序
- 为无法访问的资源编写测试
- Mock 可以交给用户
- 隔离系统

注意事项:

Mockito 2.x specific limitations

- Requires Java 6+
- Cannot mock static methods
- Cannot mock constructors
- Cannot mock `equals()`, `hashCode()`.
- Mocking is only possible on VMs that are supported by Objenesis.
- Spying on real methods where real implementation references outer `Class` via `OuterClass.this` is impossible.

Mockito 1.x Specific limitations

- Needs Java 5+
- Cannot mock final classes
- Cannot mock final methods - their real behavior is executed without any exception. Mockito cannot warn you about mocking final methods so be vigilant.
- Cannot mock static methods
- Cannot mock constructors
- Cannot mock `equals()`, `hashCode()`.
- Mocking is only possible on VMs that are supported by Objenesis (Note Objenesis is in version 2.1). Don't worry, most VMs should work just fine.
- Spying on real methods where real implementation references outer `Class` via `OuterClass.this` is impossible.

教程

使用概观

Mockito is a mocking framework for unit tests in Java. It has been designed to be intuitive to use when the test needs mocks.

- Simple usage : stub, use, verify
- Programatic creation of mocks via `mock()` or `spy()`
- Programmatic stubbing via
  - `Mockito.when(mock.action()).thenReturn(true)`
  - `BDDMockito.given(mock.action()).willReturn(true)`
- Customize mock answer or provide your own
- Programmatic verification via
  - `Mockito.verify(mock).action()`
  - `BDDMockito.then(mock).should().action()`
- Annotation sugar via `@Mock`, `@Spy`, `@Captor` or `@InjectMocks`
- JUnit first class support via the runner `MockitoJUnitRunner` and the now favored rule `MockitoJUnit.rule()`

添加依赖

```
<!--mockito-->
<dependency>
```

```xml
     <groupId>org.mockito</groupId>
     <artifactId>mockito-core</artifactId>
     <version>2.18.3</version>
     <scope>test</scope>
</dependency>
<!--TestNG-->
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>6.1.1</version>
    <scope>provided</scope>
    <!--因为我不想用TestNG自带的Junit所以排除掉-->
    <exclusions>
        <exclusion>
            <artifactId>junit</artifactId>
            <groupId>junit</groupId>
        </exclusion>
    </exclusions>
</dependency>
<!--Junit-->
<!--因为上面排除了Junit所以需要单独导入Junit的依赖-->
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
```

入门案例

Mockito的方式

```java
// You can mock concrete classes and interfaces
TrainSeats seats = mock(TrainSeats.class);

// stubbing appears before the actual execution
when(seats.book(Seat.near(WINDOW).in(FIRST_CLASS))).thenReturn(BOOKED);

// the following prints "BOOKED"
System.out.println(seats.book(Seat.near(WINDOW).in(FIRST_CLASS)));

// the following prints "null" because
// .book(Seat.near(AISLE).in(FIRST_CLASS))) was not stubbed
System.out.println(seats.book(Seat.near(AISLE).in(FIRST_CLASS)));

// the following verification passes because
// .book(Seat.near(WINDOW).in(FIRST_CLASS)) has been invoked
verify(seats).book(Seat.near(WINDOW).in(FIRST_CLASS));

// the following verification fails because
// .book(Seat.in(SECOND_CLASS)) has not been invoked
verify(seats).book(Seat.in(SECOND_CLASS));
```

BDDMockito的方式

```java
// You can mock concrete classes and interfaces
TrainSeats seats = mock(TrainSeats.class);

// stubbing appears before the actual execution
given(seats.book(Seat.near(WINDOW).in(FIRST_CLASS))).willReturn(BOOKED);

// the following prints "BOOKED"
System.out.println(seats.book(Seat.near(WINDOW).in(FIRST_CLASS)));

// the following prints "null" because
// .book(Seat.near(AISLE).in(FIRST_CLASS))) was not stubbed
System.out.println(seats.book(Seat.near(AISLE).in(FIRST_CLASS)));

// the following verification passes because
// .book(Seat.near(WINDOW).in(FIRST_CLASS)) has been invoked
then(seats).should().book(Seat.near(WINDOW).in(FIRST_CLASS));

// the following verification fails because
// .book(Seat.in(SECOND_CLASS)) has not been invoked
then(seats).should().book(Seat.in(SECOND_CLASS));
```

1. 校验对象

```java
@Test
public void verify_behaviour(){
    //模拟创建一个List对象
    List mock = Mockito.mock(List.class);
    //使用mock的对象
    mock.add(1);
    mock.clear();
    //验证add(1)和clear()行为是否发生
    Mockito.verify(mock).add(1);
    Mockito.verify(mock).clear();
}
```

2..模拟我们所期望的结果

```java
/**
 * 迭代结果
 */
@Test
public void when_thenReturn(){
    //mock一个Iterator类
    Iterator iterator = Mockito.mock(Iterator.class);
    //预设当iterator调用next()时第一次返回hello，第n次都返回world
    Mockito.when(iterator.next()).thenReturn("hello").thenReturn("world");
    //使用mock的对象
    String result = iterator.next() + " " + iterator.next() + " " + iterator.next();
    //验证结果
    Assert.assertEquals("hello world world",result);
}

```

3.模拟方法体抛出异常

```
/**
 * 抛出异常
 */
@Test(expected = RuntimeException.class)
    public void doThrow_when(){
    List list = mock(List.class);
    doThrow(new RuntimeException()).when(list).add(1);
    list.add(1);
}
```

4.使用注解 mocking

```
/**
 *方式一
 */
public class MockitoExample {
    @Mock
    private List mockList;

   /* public MockitoExample(){
        MockitoAnnotations.initMocks(this);
    }*/

    //或者使用TestNG的注解@BeforeMethod/@BeforeClass  Junit的使用:@Before/@BeforeClass
    @BeforeMethod
    public void init(){
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void shorthand(){
        mockList.add(1);
        verify(mockList).add(1);
    }
}

/**
 * 方式2:使用junit的Runwith注解.
```

```java
 * 仅适用于Junit
 */
@RunWith(MockitoJUnitRunner.class)
public class MockitoExample {
    @Mock
    private List mockList;
    @Test
    public void shorthand(){
        mockList.add(1);
        verify(mockList).add(1);
    }
}
```

5. 参数匹配

```java
@Test
public void with_arguments(){
    Comparable comparable = mock(Comparable.class);
    //@方式1:
    //预设根据不同的参数返回不同的结果
    when(comparable.compareTo("Test")).thenReturn(1);
    when(comparable.compareTo("Omg")).thenReturn(2);
    assertEquals(1, comparable.compareTo("Test"));
    assertEquals(2, comparable.compareTo("Omg"));
    //对于没有预设的情况会返回默认值
    assertEquals(0, comparable.compareTo("Not stub"));
}


/**
 *  @Todo 匹配任意参数
 */
@Test
public void with_unspecified_arguments(){
    List list = mock(List.class);
    //@方式二:
    //匹配任意参数
    when(list.get(anyInt())).thenReturn(1).thenReturn(2).thenReturn(3);
    //方式三:
    //自定义参数匹配
    when(list.contains(argThat(new IsValid()))).thenReturn(true);
    /*注意:对于thenReturn()的返回值,是方法依次执行所的到的结果,如下:*/
    assertEquals(1,list.get(0),"期望值与实际值不匹配");
    assertEquals(2,list.get(1),"期望值与实际值不匹配");
    assertEquals(3,list.get(2),"期望值与实际值不匹配");
    assertTrue(list.contains(1));
    assertTrue(list.contains(9));
}

private class IsValid implements ArgumentMatcher<Integer> {

    public boolean matches(Integer i) {
        return i<10;
    }
}

```

8.void method

```
/**
 *  @Todo void Method
 */
@Test
public void voidMethod(){
    TempTestClass mock = mock(TempTestClass.class);
    //匹配任意参数
    doNothing().when(mock).say("hello,mockito!");
    mock.say("Hello,mockito!");
}
```

9.实际对象的创建

- Mock不是真实的对象，它只是用类型的class创建了一个虚拟对象，并可以设置对象行为
- Spy是一个真实的对象，但它可以设置对象行为
- InjectMocks创建这个类的对象并自动将标记@Mock、@Spy等注解的属性值注入到这个中

```
@Test(expected = IndexOutOfBoundsException.class)
    public void spy_on_real_objects(){
        List list = new LinkedList();
        List spy = spy(list);
        //下面预设的spy.get(0)会报错，因为会调用真实对象的get(0)，所以会抛出越界异常
        //when(spy.get(0)).thenReturn(3);

        //使用doReturn-when可以避免when-thenReturn调用真实对象api
        doReturn(999).when(spy).get(999);
        //预设size()期望值
        when(spy.size()).thenReturn(100);
        //调用真实对象的api
        spy.add(1);
        spy.add(2);
        assertEquals(100,spy.size());
        assertEquals(1,spy.get(0));
        assertEquals(2,spy.get(1));
        verify(spy).add(1);
        verify(spy).add(2);
        assertEquals(999,spy.get(999));
        spy.get(2);
    }
```

10.调用实际的方法

```java
/**
*  @Todo 调用实际的方法
*/
@Test
public void voidMethod3(){
    TempTestClass mock = spy(TempTestClass.class);
    //匹配任意参数
    doNothing().doCallRealMethod().when(mock).say("");
    mock.say("Hello,mockito!");
}
```

11.重置mock

```java
@Test
public void reset_mock(){
    List list = mock(List.class);
    when(list.size()).thenReturn(10);
    list.add(1);
    assertEquals(10,list.size());
    //重置mock，清除所有的互动和预设
    reset(list);
    assertEquals(0,list.size());
}
```