PowerMock 实战手册

作者	汪文君
创建时间	2014-10-1
版本	1.0.0
QQ	532500648

更改履历

操作人	类型	时间	内容
汪文君	新建	2014-10-1	创建 PowerMock 实战的大纲
汪文君	新增	2014-10-2	编写第三到第五章的内容
汪文君	新增	2014-10-3	编写第五章到第十二章的内容

目录

前言

以前很讨厌看别人写的书中有《前言》这种东西，但是随着自己编写的资料越来越多，也越来越觉得写这种东西的必要性，因为在这个章节中可以和之前的读者解释一些东西，或者表达一些自己的想法，当然你也可以理解为发牢骚。

1、为什么要写电子书

回想从开始总结的第一天开始已经有过去了五个年头，我虽然产量不高，但是已经有 11 本电子书在互联网上传播，很多时候周围的人问我你为什么写这些东西，你的那破玩意又不能赚钱，这要从 08 年说起，我记得当时做一个关于 web services 的项目，当时在网上的资料不多，官方文档由于本人当时英语能力很差，阅读起来相当费劲，幸好找到了一个 web service 群，群主和其中的一些成员写了两本关于 Axis 和 xfire 的电子书，很完整，总结的很用心，通过阅读这两本电子书我得以顺利完成项目，记得当时也是一个职场新人，第一次感受到来自互联网非常系统的帮助，作者文笔幽默，废话很少，写的很细致，看得出来是一位非常用心的人，后来我们也成了互联网上的好朋友，每次百度或者谷歌一些中文资料，很多人都在相互抄袭，或者原创的一些文档总是站在自己的角度去思考和撰写文档，因此我在思考，自己能否在闲暇之余也通过较为具体，能让所有的人都能够明白的写作方式，让他们懂得某种框架，某种技术，虽然力量绵薄到可以忽略不计，但是长年累月从我收到的各种反馈来看似乎也帮到了一些人，不管是他们对你的赞美还是他们对你写作过程中问题的指责，相互的成长已经让我不能停下总结的脚步。

关于另外一个原因是，口头表达有时候不能够说明一些问题，比如我发音 IP 和 RP 在别人听来一直是同一个东西，虽然我自己分的很清楚，但是项目组的同事们都觉得我是在说

同一个东西，这让我很苦恼，还有诸如此类的 Security，我将重音放在了第一个音节，其实他是第二个音节，所以写出来和大家交流会显得更加具体。

2、为什么要总结 PowerMock

Chad 一直在和我说一个很重要的观点，我们项目组发布出去的产品不可能存在问题，要有问题也是操作不当引起的，Chad 为什么会一直强调这样的一个观点，原因就是我们一直在做持续集成，一个小时一次的全量单元测试，而且单元测试还在不断的完善，不断的增加，几个月以来我们的持续集成环境单元测试运行次数已经超过 5000 多次，这样的产品质量想出问题真的不是一件容易的事情，我们之间还探讨了关于 Free bug 的话题（零 bug），一致认为软件的质量原因有如下几个：

1、架构设计的问题

2、软件编码的问题

3、产品设计和需求分析的题

4、运行环境的问题

其中第二个问题真的可以在单元测试的过程中，以及不断的持续集成过程中让他逐渐的趋于零，当然这是需要一个很漫长的过程，在单元测试中不断的修复，不断的重构直到他不断的趋于完美是完全有可能做到的，这也是为什么要总结一下使用 PowerMock 的原因，或许有些读者会认为 PowerMock 已经是一个快要进博物馆的技术了，为什么还要专门来编写书籍去教大家怎么用，再说了互联网上零零碎碎的知识点足够让所有人掌握这一个测试框架，的确是这样的，但是一群人觉得毫无帮助，不代表所有人觉得毫无帮助，据我了解到的国内还真的有很少比例的公司在做持续集成的开发，如果您是一个 PowerMock 的老手，

看到我的电子书希望能够指正其中的问题，如果您是一个新手希望使用这个框架或者想更加的系统学习这个框架，也许能够带给你一些帮助，不信，走着瞧！

3、分享是一种美德

任何时候都不要觉得自己的分享是毫无价值的，最起码你做到了对某个知识点的系统梳理，在梳理的过程中你会有更多的思考，将您的思考过程以及系统知识整理出来可以让您巩固该知识点并且当作一个复习的资料，如果更进一步的话，那么就分享出来吧，很多人都很想看您是如何思考一个问题，您的思考过程是怎样的，这真的是一件非常有趣的事情，如果您觉得您的文笔不足以描述您所掌握的知识点，那么教会我吧，我个人认为文笔还算不错，交代某些事情还是能够做到条理清晰的，除了我自认为在程序员中我文笔不错之外，我还认为自己是程序员中最帅的一个，当然很多人对此视而不见甚至不承认罢了。

4、感谢

如同我之前讨厌所谓的前言一样，我之前也是不怎么待见《感谢》，但是当自己在一路成长的过程中得到了很多人的支持和帮助，就想找到一个比较书面的机会，默默的表达自己心中的感激，如果有一天他们不小心看到了这样的文字，说不定会流出两行滚烫的泪水，哈哈，感谢我经历的四家公司，在所在的团队从来没有出现过别人抱怨的事情，我总觉得我们之间的协作是如此的融洽，每一个人都是那样的无私，都是那样的乐于分享，让我在成长的过程中收到了很多技术上面的知识，态度上面的纠正，如此这些不仅在影响着我的工作，也影响着我对待生活的方式，感谢和所有团队同事相处的每一次机会，另外要感谢我的家庭，他们包容着我一直宅在家里，还有有时候的坏脾气。

好了废话不多说了，开始我们的 PowerMock 之旅吧！

如何阅读

本文中的文字排版主要有如下几种，主要是为了方便您的阅读而专门设定：

正常字体，主要是本书中的主要字体

粗体字体主要是为了表达一个很重要的观点

斜体加粗主要是为了抛出某个问题

代码片段是灰色背景

```
输出代码是黑色背景绿色字体
```

参考资料

1、PowerMock 的官方资料，本文中几乎所有的知识点都是来自官网的文档，官方地址为：http://code.google.com/p/powermock/

2、《Instant Mock Testing with PowerMock》本书较为系统的介绍了 PowerMock，但是当您看我这本书的时候，完全可以不用理会该书，因为我表达的比它要清楚很多，但是它的确是一本很详细系统的介绍 PowerMock 的书。

适合人群

为了能更好的吸收本文中的知识点，并且能够理解其中的代码片段，希望您具备如下几个知识体系，否则会有些困难。

1、熟悉 Java

2、熟悉 Junit

3、熟悉 EasyMock

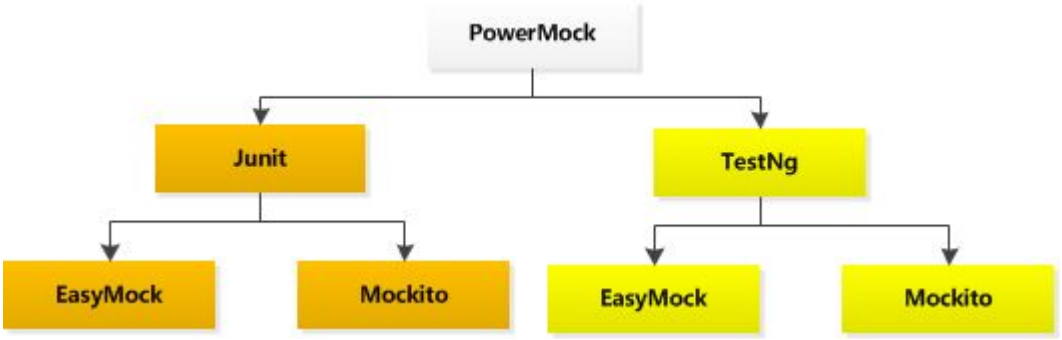
4、熟悉 Mockito

一、PowerMock 介绍

1.1、PowerMock 不是重复发明轮子

在 Java 的 TDD 领域已经有如此多的 Mock 框架，比如 EasyMock，JMock，Mockito 为什么还要有 PowerMock 的存在，上述三个已经有重复发明轮子的嫌疑，为什么还要大家去使用 PowerMock 呢？

其实 PowerMock 并不是重发发明轮子，他的出现只是为了解决上述三种框架根本没有办法完成的工作，比如 Mock 一个 Static 方法等等，更多的将 PowerMock 理解为对现有 Mock 框架的扩展和进一步封装是比较贴切的，PowerMock 现在目前提供了两套 UT（Unit Test）框架的封装，请看下图。



PowerMock 主要围绕着 Junit 测试框架和 TestNg 测试框架进行，其中在每个框架下面对所涉及的 Mock 框架仅仅支撑 EasyMock 和 Mockito，为什么要画这个图呢，是因为 PowerMock 对所依赖的 Jar 包非常的苛刻，如果出现某个依赖包的冲突或者不一致都会出现不能使用的情况，因此根据您的喜好和擅长，请选择官网上提供的 PowerMock 套件，**本文中采用的是 Junit+PowerMock+Mockito 这样的组合来进行讲述的。**

1.2、PowerMock 解决了什么问题

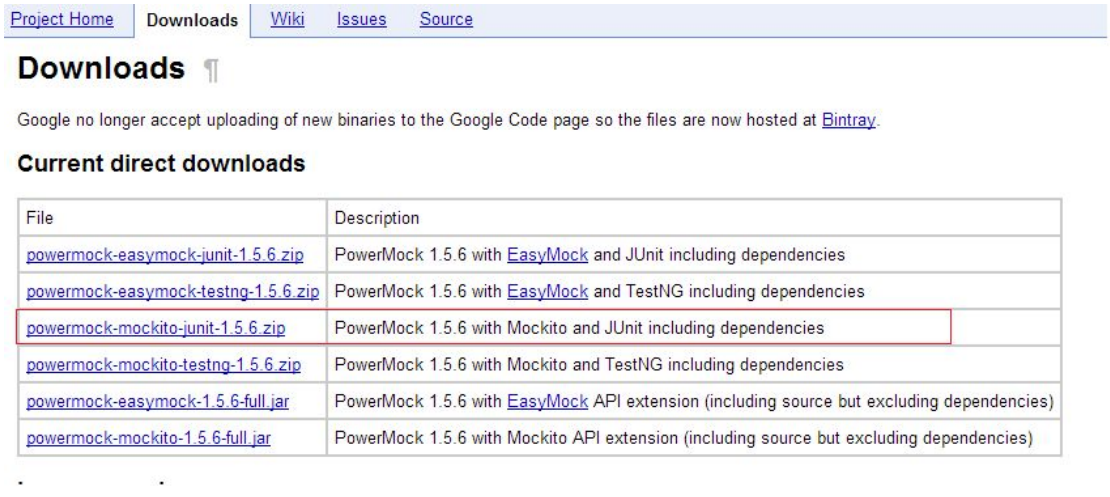
那么到底 PowerMock 解决了哪些问题，PowerMock 的官方文档说的非常的清楚 ,他们解决了如下的问题.

> PowerMock is a framework that extend other mock libraries such as EasyMock with more powerful capabilities. PowerMock uses a custom classloader and bytecode manipulation to enable mocking of static methods, constructors, final classes and methods, private methods, removal of static initializers and more.

使用过 EasyMock 或者 Mockito 的人应该非常清楚，他们两个无法完成对 final 类型的 class 和 method 的 mock 操作，不能完成对类方法（static）的 mock，不能完成对局部变量的 mock 等等，PowerMock 的出现就是为了解决诸如此类的问题，简言之就是**专治各种不服。**

1.3、如何获得 PowerMock

您可以在 PowerMock 的官方网站免费获得测试组件，我之前说过 PowerMock 对所依赖的 library 有些苛刻，因此最好还是老老实实用它提供的套件包，如下所示，您可以下载我红色标注出来的测试套件。



如果您是 Maven 的忠实用户，您可以用将如下的配置信息放到您的 POM 文件中，这样依赖您也会获得 PowerMock 和他所依赖的第三方包。

```
<properties>
      <powermock.version>1.5.6</powermock.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.powermock</groupId>
        <artifactId>powermock-module-junit4</artifactId>
        <version>${powermock.version}</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.powermock</groupId>
        <artifactId>powermock-api-mockito</artifactId>
        <version>${powermock.version}</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

1.4、如何安装 PowerMock

在 1.3 中已经很清楚的讲述了如何获得 PowerMock 的安装包，相信如何安装是非常简单的事情，加载到对应的 classpath 中即可，不再赘述了。

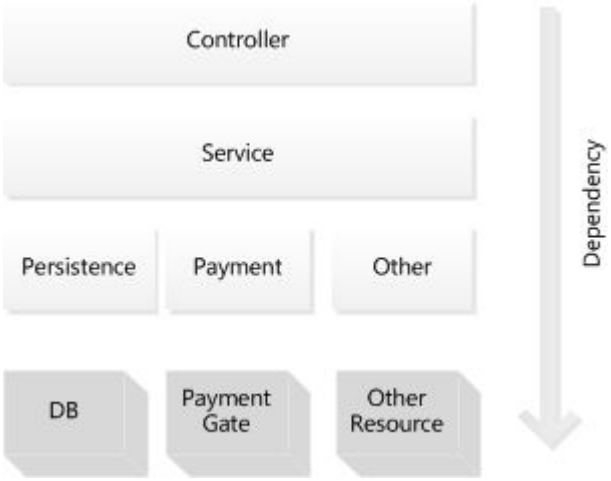
二、PowerMock 入门

2.1、使用场景

在现实的软件开发过程中，我们经常需要协同其他同事一起来完成某个模块的功能开发，或者需要第三方资源，比如您需要一个短信网关，您需要一个数据库，您需要一个消息中间件，当您进行测试依赖于这些资源的代码时候，不可能在每次都具备相应的环境，这将

是一个很不现实的问题，如果当您依赖于其他模块而无法进行单元测试的时候，此时该模块的质量风险就有两个，第一是您所负责的代码，第二是您所依赖的代码，您所依赖您没有办法在很快的时间协调到资源，那么您所负责的代码由于不具备单元测试环境没有办法进行测试，很可能存在极大的风险，因此如何测试您的代码，让他的质量达到百分之百的可用，这就是 Mock 存在的必要，如果您还不能清楚这个场景，我们举一个大家都耳熟能详的例子来探讨一下。

采用三层架构的方式实现某个 WEB 的请求到业务受理，已经变得非常成熟，甚至现在在更多的分层，更多的细化每一个层次的职责，尽量使某一个层次可以最大化的重用，减少耦合，下图您应该非常熟悉了吧。



那么请大家思考如下几个问题：

1、如何保证各个层次模块的代码完全可用，在正常和临界情况下？

2、您使用集成测试做单元测试么？

很多人所谓的测试恐怕更多的是一种集成测试，也就是点击页面某个按钮看看是否能够顺利执行，所依赖的各种资源都必须正常运行，如果出现问题就很难让整个流程执行下去，如上图所示，我们如何在没有数据库的时候能够测试我们的 Service，没有 Payment Gate 的时候进行支付的测试等等，这才是 Mock 要解决的问题，请注意是 Mock 而不是

PowerMock（PowerMock 也是一种 Mock，但是他主要是解决其他 Mock 不能解决的问题）

2.2、PowerMock 之 HelloWorld

如果您已经理解了 2.1 中所表达的观点，那么我们就开始一个简单的例子，让大家先对 PowerMock 有一个比较具象的认识，然后我们在接下来的文章将一一演示如何使用强大的 PowerMock，在这个世界上程序员是最最热爱世界的一类人群，因为他们在学习任何一门语言，学习任何一个框架，都要问候世界好多次。

2.2.1、获取所有员工的个数

我们现在有一个 Service 类，就是 EmployeeService，其中有一个方法需要获取数据库中雇员的数量，Service 代码如下所示：

```
package com.wangwenjun.powermock.helloworld.service;

import com.wangwenjun.powermock.helloworld.dao.EmployeeDao;

public class EmployeeService {

    private EmployeeDao employeeDao;

    public EmployeeService(EmployeeDao employeeDao)
    {
        this.employeeDao = employeeDao;
    }

    /**
     * 获取所有员工的数量.
     * @return
     */
    public int getTotalEmployee()
    {
        return employeeDao.getTotal();
    }
```

```
}
```

可以看到 ,创建 Service 的时候需要传递一个 EmployeeDao 这个类 ,也就是说 Service 依赖于 Persistence，如果想要测试 Service 就需要完全看 Persistence 的脸色，我们再来看看 Persistence 代码，如下所示

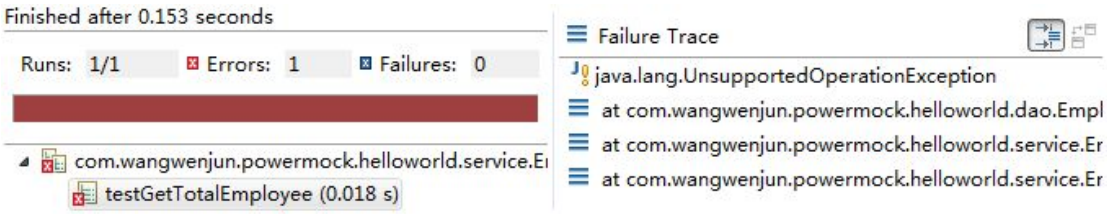
```
package com.wangwenjun.powermock.helloworld.dao;

public class EmployeeDao {

    public int getTotal()
    {
        throw new UnsupportedOperationException();
    }
}
```

哇！你死定了，你肯定调用不了 Dao，无法正常完成 Service 的测试，我为什么要在 Persistence 的方法抛出 UnsupportedOperationException 呢？目的就是告诉大家该方法可能由于某种原因（没有完成，或者资源不存在等）无法为 Service 服务，难道你不需要测试 EmployeeService 么？肯定要测试，那么我们就硬着头皮来写测试用例吧。

测试代码如下所示

```
package com.wangwenjun.powermock.helloworld.service;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

import com.wangwenjun.powermock.helloworld.dao.EmployeeDao;

public class EmployeeServiceTest {

    @Test
    public void testGetTotalEmployee() {
        final EmployeeDao employeeDao = new EmployeeDao();
        final EmployeeService service = new EmployeeService(employeeDao);
        int total = service.getTotalEmployee();
        assertEquals(10, total);
    }
}
```

上面的测试肯定会是以失败收场，大家都已经心知肚明了，但是我还是要无情的运行一次，真的勇士敢于面对惨淡的人生，当我们执行该 unit test 的时候会有如下的惨痛结局。



请大家忘记此时此刻我抛出来的异常，幻想成此时此刻数据库连接不上，问题现在很明显，数据库链接不通，我们无法测试 Service，难道真的就无计可施了么？好吧，有请我们的主角 PowerMock 闪亮登场，请看下面的测试用例

```
package com.wangwenjun.powermock.helloworld.service;

import static org.junit.Assert.assertEquals;

import org.junit.Test;
import org.powermock.api.mockito.PowerMockito;

import com.wangwenjun.powermock.helloworld.dao.EmployeeDao;

public class EmployeeServiceTest {

    @Test
    public void testGetTotalEmployeeWithMock() {
        EmployeeDao employeeDao = PowerMockito.mock(EmployeeDao.class);
        PowerMockito.when(employeeDao.getTotal()).thenReturn(10);
        EmployeeService service = new EmployeeService(employeeDao);
        int total = service.getTotalEmployee();
        assertEquals(10, total);
    }
}
```

当你再次运行时你会发现此时此刻运行通过，编写一下上述的代码我们先来有个简单的认识，所谓 Mock 就是创建一个假的，Mock 那个对象就会创建一个假的该对象，此时该对象是一单纯的白纸，需要你对其进行涂鸦，第二句话 when...then 语法就是您的涂鸦，您期望调用某个方法的时候返回某个您指定的值，最后的代码已经非常熟悉了，此时此刻您已

经完全让 EmployeeDao 根据你的意愿来运行，所以想怎样测试 EmployeeService 就怎样测试。

2.2.2、创建员工

我们再来增加另外一个需求，就是创建一个 Employee，这就意味着我们需要分别在 Service 和 Dao 中增加相应的两个接口，请看代码如下所示

Service 中的 CreateEmployee 方法

```
public void createEmployee(Employee employee)
{
    employeeDao.addEmployee(employee);
}
```

再来看看 Dao 中的方法 addEmployee

```
public void addEmployee(Employee employee)
{
    throw new UnsupportedOperationException();
}
```

至于addEmployee为什么抛出异常，在不多做解释了，同样如果针对createEmployee写单元测试运行结果肯定死得很惨，因为此时“数据库资源不存在”，相信大家一定很清楚这一点，但是这不是本小节中所要讲述的重点，重点在于 addEmployee 方法是一个 void 类型的，也就是我们没有办法断言想要的结果是否一致，***而 mock 厚的 addEmployee 方法事实上是什么都不会做的，此时我们该如何进行测试呢？***

简单思考一下我们其实只是想要知道 addEmployee 方法是否被调用过即可，当然我们可以假设他 add 成功或者失败，这就根据您的 test case 来设定了，好了，有了这个概念之后我们来看看如何测试 void 方法，其实就是 mock 中一个很重要的概念 Verifying 了。

```
@Test
    public void testCreateEmployee() {
        EmployeeDao employeeDao = PowerMockito.mock(EmployeeDao.class);
    Employee employee = new Employee();
```

```
    PowerMockito.doNothing().when(employeeDao).addEmployee(employee);
    EmployeeService service = new EmployeeService(employeeDao);
    service.createEmployee(employee);

        // verify the method invocation.
    Mockito.verify(employeeDao).addEmployee(employee);
}
```

然后用 junit 运行肯定能够通过 ,其中 Mockito.verify 主要用来校验被 mock 出来的对象中的某个方法是否被调用，我们的 PowerMock helloworld 也到此结束了。

2.3、重点 API 解释

其中有几个 API 大家可能看着有点陌生，不用担心，在接下来的章节中会大量的使用到，慢慢您会熟悉起来的，我们简单说说在上面的代码中所涉及到的几个 PowerMockAPI

2.3.1、Mock

Powermockito.mock() 方法主要是根据 class 创建一个对应的 mock 对象，powermock 的创建方式可不像 easymock 等使用 proxy 的方式创建，他是会在你运行的过程中动态的修改 class 字节码文件的形式来创建的。

2.3.2、Do..when..then

我们可以看到，每次当我们想给一个 mock 的对象进行某种行为的预期时，都会使用 do...when...then...这样的语法，其实理解起来非常简单：做什么、在什么时候、然后返回什么。

2.3.3、Verify

当我们测试一个 void 方法的时候，根本没有办法去验证一个 mock 对象所执行后的结果，因此唯一的方法就是检查方法是否被调用，在后文中将还会专门来讲解。

2.4、总结

其实 Hello world 有关 PowerMock 所表现出来的功能，在 EasyMock 中照样是可以实现的，并不能说明 powermock 扩展了什么，其实这不是我所表达的重点，我只是想让读者对如何使用 PowerMock 有一个简单的认识。

既然说道本节中所设计的示例并没有体现出来 PowerMock 的优势，那么请耐着性子往下再去阅读，另外在本章中有很多东西描述的很罗嗦，目的是为了让大家能够了解的清楚一些.

三、Mock Local Variable

3.1、有返回值

在本章中我们设计一个其他 Mock 玩不转的东东，但是在编码的时候会经常遇到，我们先来看看 Service 的代码

```
package com.wangwenjun.powermock.local.service;

import com.wangwenjun.powermock.local.dao.EmployeeDao;

public class EmployeeService {

    public int getTotalEmployee()
    {
        EmployeeDao employeeDao = new EmployeeDao();
        return employeeDao.getTotal();
```

```
    }
}
```

我们并没有采用 hello world 中通过构造方法注入 dao 的方式，而是在方法内部 new 出一个 EmployeeDao，我们通常都会写这样的代码，根据之前的经验，调用肯定是失败的，而要测试这样的代码 EasyMock，Mockito 等显然力不从心，这个时候优势就很明显的体现出来了，请看下面的测试代码

```
package com.wangwenjun.powermock.local.service;


import static org.junit.Assert.*;


import org.junit.Test;

import org.junit.runner.RunWith;

import org.powermock.api.mockito.PowerMockito;

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

import org.powermock.modules.junit4.PowerMockRunner;


import com.wangwenjun.powermock.local.dao.EmployeeDao;


@RunWith(PowerMockRunner.class)

@PrepareForTest(EmployeeService.class)

public class EmployeeServiceTest {
```

```
/**
 * 用传统的方式肯定测试失败。
 */
@Test
public void testGetTotalEmployee() {
    EmployeeService service = new EmployeeService();
    int total = service.getTotalEmployee();
    assertEquals(10, total);
}

/**
 * 采用 PowerMock 进行测试
 */
@Test
public void testGetTotalEmployeeWithMock() {
    EmployeeDao employeeDao = PowerMockito.mock(EmployeeDao.class);
    try {
        PowerMockito.whenNew(EmployeeDao.class).withNoArguments()
                .thenReturn(employeeDao);
        PowerMockito.when(employeeDao.getTotal()).thenReturn(10);

        EmployeeService service = new EmployeeService();
```

```
            int total = service.getTotalEmployee();

            assertEquals(10, total);

        } catch (Exception e) {

            fail("测试失败.");

        }

    }

}
```

运行上面的代码可以很明显的看到其中传统的方式是失败的，而通过 Mock 方式测试的是成功的，当然您可能会非常惊讶 PowerMock 尽然有如此强大的能力，以至于可以 Mock 出局部变量，那么同样我们再来看看局部变量的 void 方法该如何进行测试。

3.2、局部变量的 void 方法

同样在 Service 中增加一个添加 Employee 的方法，如下所示

```
    public void createEmployee(Employee employee)
    {
        EmployeeDao employeeDao = new EmployeeDao();
        employeeDao.addEmployee(employee);
    }
```

我们该如何测试这个方法呢？同样的如果采用之前的传统调用方式肯定会出现错误，我们只能采用 PowerMock 的方式来进行测试，测试代码如下所示：

```
    @Test
    public void testCreateEmployee() {
        EmployeeService service = new EmployeeService();
        Employee employee = new Employee();
        service.createEmployee(employee);
    }

    @Test
    public void testCreateEmployeeWithMock() {
        EmployeeDao employeeDao = PowerMockito.mock(EmployeeDao.class);
```

```
        try {
            PowerMockito.whenNew(EmployeeDao.class).withNoArguments()
                    .thenReturn(employeeDao);
            Employee employee = new Employee();
            EmployeeService service = new EmployeeService();
            service.createEmployee(employee);
            Mockito.verify(employeeDao).addEmployee(employee);
        } catch (Exception e) {
            fail();
        }
    }
```

运行上面的两个测试用例，你会发现其中第一个失败，第二个运行成功，对于测试 void 返回类型的方法，同样我们做的只能是判断他是否被调用，关于 verify 的话题在后文中有专门一个章节来进行讨论。

3.3、@RunWith 和@PrepareForTest 介绍

细心的人可能会发现，我们在编写测试代码的时候增加了两个 Annotation，分别是@RunWith 和@PrepareForTest，那他们的作用是什么呢？

其中@RunWith 就是显式的告诉 Junit 使用某个指定的 Runner 来运行 Test Case，我们使用了 PowerMockRunner 来运行我们的测试用例，如果不指定的话我们就默认使用的是 Junit 提供的 Runner，先来看看下面这张图，了解一下 Runner 家族的成员

由于我们在测试EmployeeService的时候需要改变EmployeeDao的创建行为，通过这个的语句PowerMockito.*whenNew*(EmployeeDao.**class**).withNoArguments().thenReturn(employeeDao);就可以看得出来，因此就需要了解另外一个注解@PrepareForTest，这个注解是告诉PowerMock为我提前准备一个xxx的class，根据我测试预期的行为去准备，根据这段粗糙的文字描述，我们再来详细的整理一下。

在测试EmployeeService的时候，因为涉及到了局部变量的创建，我们必须让他创建成为我们想要的Mock对象，因此需要有一个录制的过程PowerMockito.*whenNew*(EmployeeDao.**class**).withNoArguments().thenReturn(employeeDao); 这句话就是告诉Power Mock录制一个这样的构造行为 ,等我接下来用的时候你就把之前mock的东西给我，而@PrepareForTest是为PowerMock的Runner提前准备一个已经根据某种预期改变过的class，在这个例子中就是EmployeeService，如果大家使用过Jacoco统计过Java测试代码的覆盖率就会发现一个很恼火的问题，那就是不管你写了多少个EmployeeService的测试用例，只要你对其进行了PrepareForTest之后，代码覆盖率仍然是0，为什么会这样呢，因为经过了PrepareForTest的处理之后，PowerMock框架会通过类加载器的方式改变EmployeeService的行为，如何改变呢？当然是产生了一个新的类在运行的过程中，新的类里面但凡碰到要创建EmployeeDao的时候都会将提前录制好的行为重播出来，给出一个Mock对象。

3.4、总结

如果说之前 PowerMock 能做的事情很简单，那么本章中所涉及的 Mock 局部变量的方式是其他 Mock 框架所不能完成的 ,在本章中我们还学习了两个比较重要的 Annotation，其中一个是 RunWith，另外一个是 PrepareForTest，在后面我们会大量的使用到，不用担

心一时半会接受不了，我们还会有专门的章节讲解 PowerMock 的类加载器，到时候您就会明白其中更多的细节。

四、Mock Static

有时候我们写代码的时候喜欢编写已经写工具类，工具类一般都提供了大量的便捷的类方法（static）供调用者使用，在本章中我们来看看如何进行静态方法的测试。

4.1、问题场景

同样 Service 还是两个接口，第一个是查询雇员数量，第二个是创建一个新的雇员，代码如下所示：

```
package com.wangwenjun.powermock.mockstatic.service;

import com.wangwenjun.powermock.Employee;
import com.wangwenjun.powermock.mockstatic.EmployeeUtils;

public class EmployeeService {

    public int getEmployeeCount() {
        return EmployeeUtils.getEmployeeCount();
    }

    public void createEmployee(Employee employee) {
        EmployeeUtils.persistenceEmployee(employee);
    }
}
```

相比这样的代码在前文中已经看得非常多了，重点是其中的一个工具类 EmployeeUtils，同样来看看它的代码

```
package com.wangwenjun.powermock.mockstatic;

import com.wangwenjun.powermock.Employee;
```

```
public class EmployeeUtils {

    public static int getEmployeeCount()
    {
        throw new UnsupportedOperationException();
    }

    public static void persistenceEmployee(Employee employee)
    {
        throw new UnsupportedOperationException();
    }
}
```

4.2、单元测试

上面的两个类代码都非常简单，但是写一个专门针对 EmployeeService 的单元测试类，你会发现运行根本不能通过

```
package com.wangwenjun.powermock.mockstatic.service;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

import org.junit.Test;

import com.wangwenjun.powermock.Employee;

public class EmployeeServiceTest {

    @Test
    public void testGetEmployeeCount() {
        final EmployeeService employeeService = new EmployeeService();
        int count = employeeService.getEmployeeCount();
        assertEquals(10, count);
    }

    @Test
    public void testCreateEmployee() {
        final EmployeeService employeeService = new EmployeeService();
        Employee employee = new Employee();
        employeeService.createEmployee(employee);
        assertTrue(true);
    }
```

```
}
```

上面的测试用例肯定是无法运行通过，想必一看就很清楚，所有就不再赘述，根据目前的 PowerMock 知识，还是不足以完成对 EmployeeUtils 的 Mock 操作，那到底应该如何对其进行测试呢，马上揭晓。

4.3、使用 Mock

使用传统的方式很难对 EmployeeService 完成测试，因为没有办法模拟 EmployeeUtils 的行为，在本节中我们继续使用 PowerMock 完成对 EmployeeService 的测试，代码如下

```
package com.wangwenjun.powermock.mockstatic.service;

import static org.junit.Assert.assertEquals;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.powermock.api.mockito.PowerMockito;
import org.powermock.core.classloader.annotations.PrepareForTest;
import org.powermock.modules.junit4.PowerMockRunner;

import com.wangwenjun.powermock.Employee;
import com.wangwenjun.powermock.mockstatic.EmployeeUtils;

@RunWith(PowerMockRunner.class)
@PrepareForTest(EmployeeUtils.class)
public class EmployeeServiceTest {

    @Test
    public void testGetEmployeeCountWithMock() {
        PowerMockito.mockStatic(EmployeeUtils.class);

    PowerMockito.when(EmployeeUtils.getEmployeeCount()).thenReturn(10);
        final EmployeeService employeeService = new EmployeeService();
        int count = employeeService.getEmployeeCount();
        assertEquals(10, count);
    }

    @Test
    public void testCreateEmployeeWithMock() {
        PowerMockito.mockStatic(EmployeeUtils.class);
```

```
        Employee employee = new Employee();
        PowerMockito.doNothing().when(EmployeeUtils.class);
        final EmployeeService employeeService = new EmployeeService();
        employeeService.createEmployee(employee);
        PowerMockito.verifyStatic();
    }
}
```

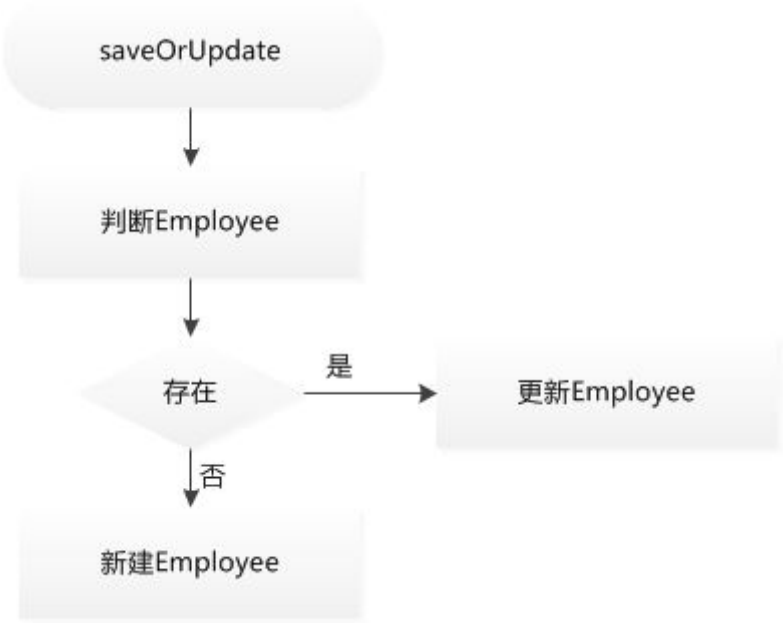
运行上述的单元测试，你就会得到正确的测试结果和预期，首先我们让 PowerMock 在运行之前准备 `EmployeeUtils` 类，并且我们采用了 mockstatic 的方法，其中这个有别于 mock 非静态的类，然后其他的操作就会完全一样了。

五、Verifying

Verifying 是一个非常强大的测试工具，不仅在 powermock 中使用，就连 easymock，mockito 等框架都会使用，他的目的是为了检查某个被测试的方法是否顺利的被调用了，可能三言两语各位读者还是不能理解，我们举一个简单的例子，然后我们针对这样的代码进行测试。

5.1、使用场景

假设我们有一个这样的业务接口，传递一个 Employee 实例参数，然后通过 DAO 查询该 Employee 是否在对应的数据库中存在，如果不存在则创建一个 Employee 数据，否则就更新原来的 Employee，其实如此简单的逻辑我根本不用画图演示，但是为了能照顾到初学者，我还是画一个简单的图示吧。



其中大名鼎鼎的 hibernate 就有一个这样的接口，通过上图我们不难看出，其中 saveOrUpdate 里面有两个分支的存在，而且针对每个分支可能会调用不同的 DAO 接口，我们来看代码吧。

5.2、业务代码

EmployeeDao.java 代码如下所示：

```
package com.wangwenjun.powermock.verify.dao;

import com.wangwenjun.powermock.Employee;

/**
 * @author wangwenjun
 */
public class EmployeeDao {

    /**
     * @param employee
     */
    public void saveEmployee(Employee employee) {
        throw new UnsupportedOperationException();
    }

    /**
```

```
     * @param employee
     */
    public void updateEmploye(Employee employee) {
        throw new UnsupportedOperationException();
    }

    /**
     * @param employee
     * @return
     */
    public long getCount(Employee employee) {
        throw new UnsupportedOperationException();
    }
}
```

同样为了模拟 DAO 比较难以测试，我们依旧抛出了运行时异常，好了，再来看一看 Service 中是如何调用的吧。

```
package com.wangwenjun.powermock.verify.service;

import com.wangwenjun.powermock.Employee;
import com.wangwenjun.powermock.verify.dao.EmployeeDao;

public class EmployeeService {

    public void saveOrUpdate(Employee employee) {
        final EmployeeDao employeeDao = new EmployeeDao();
        long count = employeeDao.getCount(employee);
        if (count > 0)
            employeeDao.updateEmployee(employee);
        else
            employeeDao.saveEmployee(employee);
    }
}
```

在开始编写测试代码之前，我们先来思考一下，其中 Service 中的 saveOrUpdate 方法是用了 EmployeeDao ,并且作为一个局部变量被实例化 ,因此我们必须采用 PowerMock 的方法进行测试，但是最重要的是由于其中多了一层逻辑判断，这给我们的断言带来了一定的难度，由于不能真正的连接数据库，所以没有办法产生真实的数据，所以我们断言的依据是什么？是新增了一条数据还是更改了一条记录呢？

显然采用传统的方式我们更本没有办法做到这一点，但是 Verifying 却可以让我们很容易的应对这样一个场景，首先我们不难看出，当 count 大于 0 的时候我们就需要执行更新操作，当 count 小于等于 0 的时候我们就需要执行新增的操作，那么我们就可以通过验证 DAO 中所涉及的方法是否被调用来进行验证了，如果这么说您还是不能理解，接着看下面的两个测试用例。

5.3、测试代码

```
package com.wangwenjun.powermock.verify.service;

import static org.junit.Assert.fail;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mockito;
import org.powermock.api.mockito.PowerMockito;
import org.powermock.core.classloader.annotations.PrepareForTest;
import org.powermock.modules.junit4.PowerMockRunner;

import com.wangwenjun.powermock.Employee;
import com.wangwenjun.powermock.verify.dao.EmployeeDao;

@RunWith(PowerMockRunner.class)
@PrepareForTest(EmployeeService.class)
public class EmployServiceTest {

    @Test
    public void testSaveOrUpdateCountLessZero() {
        try {
            EmployeeDao                    employeeDao                    =
PowerMockito.mock(EmployeeDao.class);
            PowerMockito.whenNew(EmployeeDao.class).withNoArguments()
                    .thenReturn(employeeDao);
            Employee employee = new Employee();

    PowerMockito.when(employeeDao.getCount(employee)).thenReturn(0L);
            EmployeeService employeeService = new EmployeeService();
            employeeService.saveOrUpdate(employee);
```

```
            Mockito.verify(employeeDao).saveEmployee(employee);

            Mockito.verify(employeeDao, Mockito.never())
                    .updateEmploye(employee);
        } catch (Exception e) {
            e.printStackTrace();
            fail();
        }
    }

    @Test
    public void testSaveOrUpdateCountMoreThanZero() {
        EmployeeDao employeeDao = PowerMockito.mock(EmployeeDao.class);
        try {
            PowerMockito.whenNew(EmployeeDao.class).withNoArguments()
                    .thenReturn(employeeDao);
            Employee employee = new Employee();

    PowerMockito.when(employeeDao.getCount(employee)).thenReturn(1L);

            EmployeeService employeeService = new EmployeeService();
            employeeService.saveOrUpdate(employee);

            Mockito.verify(employeeDao).updateEmploye(employee);

            Mockito.verify(employeeDao,
Mockito.never()).saveEmployee(employee);
        } catch (Exception e) {
            fail();
        }
    }
}
```

读者可以运行一下上面的两个测试用例，你会发现运行通过，上面的代码也比较简单，但是我还是要主要强调一下其中的重点，就是 verify 方法的使用，我们根据 getCount 的返回的结果去验证对应的方法是否被调用了，从而来测试我们的 service 是否逻辑运行正确

当然 Verify 的所涉及的方法还不至这些，其中有很多的 verification modes，读者可以一一验证，我们在下一节中也将罗列出来。

5.4、Verifying 其他 API

```
Mockito.verify(employeeDao, Mockito.never()).saveEmployee(employee);
          Mockito.verify(employeeDao,
Mockito.atLeastOnce()).saveEmployee(employee);
          Mockito.verify(employeeDao,
Mockito.times(1)).saveEmployee(employee);
          Mockito.verify(employeeDao,
Mockito.atMost(1)).saveEmployee(employee);
          Mockito.verify(employeeDao,
Mockito.atLeast(1)).saveEmployee(employee);
```

六、Mock final

本章的内容也是比较简单，就是使用 PowerMock 测试一下某个 final 修饰的 class 或者 method，但是为了更加有说服力，本章会引入 easymock 作为对比（其中 easymock 的内容不在本书范围之内，请翻阅相关资料）看看 easymock 是否能够 mock 一个 final 修饰的 class 或者 method

6.1、业务代码

其中业务代码和之前的业务代码没有什么不一样，都是一个 service 调用其中的一个 dao，但是这个 dao class 是被 final 修饰的，也就是说他是不能被继承的，我们看一下代码吧。

Service 代码

```
package com.wangwenjun.powermock.mockfinal.service;

import com.wangwenjun.powermock.Employee;
import com.wangwenjun.powermock.mockfinal.dao.EmployeeDao;

/**
 * 和之前的业务代码没有什么太大的区别，所以只写一个方法即可
 *
```

```
 * @author wangwenjun
 *
 */
public class EmployeeService {

    private EmployeeDao employeeDao;

    public EmployeeService(EmployeeDao employeeDao) {
        this.employeeDao = employeeDao;
    }

    public void createEmployee(Employee employee) {
        employeeDao.insertEmployee(employee);
    }
}
```

Dao 的代码如下所示

```
package com.wangwenjun.powermock.mockfinal.dao;

import com.wangwenjun.powermock.Employee;

final public class EmployeeDao {

    public boolean insertEmployee(Employee employee) {
        throw new UnsupportedOperationException();
    }
}
```

Service 代码没有什么，重点看一下 DAO 的代码，修饰符多了一个 final 类型，同时为了能够方便 EasyMock 进行测试，我将 EmployeeDao 在 EmployeeService 中通过了构造方法传入（因为 EasyMock 不能在 mock 方法内部的变量）

6.2、EasyMock 测试

```
package com.wangwenjun.powermock.mockfinal.service;

import org.easymock.EasyMock;
import org.junit.Test;

import com.wangwenjun.powermock.Employee;
import com.wangwenjun.powermock.mockfinal.dao.EmployeeDao;
```

```
public class EmployeeServiceEasyMockTest {

    @Test
    public void test() {
        EmployeeDao employeeDao = EasyMock.createMock(EmployeeDao.class);

        Employee employee = new Employee();

    EasyMock.expect(employeeDao.insertEmployee(employee)).andReturn(true
);

        EasyMock.replay(employeeDao);

        EmployeeService          employeeService          =          new
EmployeeService(employeeDao);
        employeeService.createEmployee(employee);

        EasyMock.verify(employeeDao);
    }
}
```

执行上面的单元测试，你会发现运行出现了错误，错误信息如下所示：

```
java.lang.IllegalArgumentException: Cannot subclass final class class
com.wangwenjun.powermock.mockfinal.dao.EmployeeDao
    at net.sf.cglib.proxy.Enhancer.generateClass(Enhancer.java:446)
    at
net.sf.cglib.core.DefaultGeneratorStrategy.generate(DefaultGeneratorStrategy.java:25)
    at net.sf.cglib.core.AbstractClassGenerator.create(AbstractClassGenerator.java:216)
    at net.sf.cglib.proxy.Enhancer.createHelper(Enhancer.java:377)
    at net.sf.cglib.proxy.Enhancer.createClass(Enhancer.java:317)
    at
org.easymock.internal.ClassProxyFactory.createProxy(ClassProxyFactory.java:166)
    at org.easymock.internal.MocksControl.createMock(MocksControl.java:59)
    at org.easymock.EasyMock.createMock(EasyMock.java:103)
    at
com.wangwenjun.powermock.mockfinal.service.EmployeeServiceEasyMockTest.test(Em
ployeeServiceEasyMockTest.java:13)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
    at java.lang.reflect.Method.invoke(Unknown Source)
    at
org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:4
7)
```

```
    at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)
    at
org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:44)
    at
org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17)
    at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:271)
    at
org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:70)
    at
org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:50)
    at org.junit.runners.ParentRunner$3.run(ParentRunner.java:238)
    at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:63)
    at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:236)
    at org.junit.runners.ParentRunner.access$000(ParentRunner.java:53)
    at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:229)
    at org.junit.runners.ParentRunner.run(ParentRunner.java:309)
    at
org.eclipse.jdt.internal.junit4.runner.JUnit4TestReference.run(JUnit4TestReference.java:5
0)
    at org.eclipse.jdt.internal.junit.runner.TestExecution.run(TestExecution.java:38)
    at
org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.runTests(RemoteTestRunner.java:
459)
    at
org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.runTests(RemoteTestRunner.java:
675)
    at
org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.run(RemoteTestRunner.java:382)
    at
org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.main(RemoteTestRunner.java:192)
```

但是当您将 EmployeeDao 的 final 修饰符去掉之后就会运行成功，由此可见 EasyMock 是通过代理的方式实现（继承代理）产生一个新的 Mock 对象的，同样 final 修饰的方法也会不适合 EasyMock 的场景，读者自己进行测试吧。

6.3、PowerMock 测试

好了，我们来看看 PowerMock 的方式吧，其实 PowerMock 的方式的章节中所涉及的代码没有什么区别，出现在这里显得有点罗嗦，但是为了更加能够说明问题，我还是将代码贴出来吧，如果您觉得我多此一举，就直接将该部分的代码跳过即可。

```
package com.wangwenjun.powermock.mockfinal.service;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mockito;
import org.powermock.api.mockito.PowerMockito;
import org.powermock.core.classloader.annotations.PrepareForTest;
import org.powermock.modules.junit4.PowerMockRunner;

import com.wangwenjun.powermock.Employee;
import com.wangwenjun.powermock.mockfinal.dao.EmployeeDao;

@RunWith(PowerMockRunner.class)
@PrepareForTest(EmployeeDao.class)
public class EmployeeServicePowerMockTest {

    @Test
    public void test() {
        EmployeeDao employeeDao = PowerMockito.mock(EmployeeDao.class);
        Employee employee = new Employee();
        PowerMockito.when(employeeDao.insertEmployee(employee))
                .thenReturn(true);

        EmployeeService employeeService = new
EmployeeService(employeeDao);
        employeeService.createEmployee(employee);

        Mockito.verify(employeeDao).insertEmployee(employee);
    }
}
```

当您运行上面的单元测试时，您会发现出现了可爱的小绿条，恭喜你，成功了！

七、Mock constructors

有些时候我们需要在类的构造函数中传递某些参数进去 ,这也是非常常见的一种编码习惯，这个时候我们应该如何去 Mock 这样的类呢？其实在前面的文章中已经有了相关的代码，在本章中我们继续深入探讨一下。

7.1、使用场景

举个例子,DAO 的构造需要传递两个参数，其中一个是 boolean 类型标识是否懒加载，另外一个是 enumeration 类型，标识采用何种数据库方言，当然有可能构造该 DAO 需要很重的资源，并且在我们的单元测试环境下不一定能够很容易的获得，因此需要 mock。

7.2、业务代码

我们先来看看 DAO 的代码

```
package com.wangwenjun.powermock.construction.dao;

import com.wangwenjun.powermock.Employee;

public class EmployeeDao {

    public enum Dialect {
        MYSQL,
        ORACLE
    }

    public EmployeeDao(boolean lazy,Dialect dialect)
    {
        throw new UnsupportedOperationException();
    }

    public void insertEmploye(Employee employee)
    {
        throw new UnsupportedOperationException();
```

```
    }
}
```

再来看看 Service 的代码，同样代码也是超级简单

```
package com.wangwenjun.powermock.construction.service;

import com.wangwenjun.powermock.Employee;
import com.wangwenjun.powermock.construction.dao.EmployeeDao;
import com.wangwenjun.powermock.construction.dao.EmployeeDao.Dialect;

public class EmployeeService {

    public void createEmployee(final Employee employee)
    {
        EmployeeDao employeeDao = new EmployeeDao(false, Dialect.MYSQL);
        employeeDao.insertEmploye(employee);
    }
}
```

我们测试的重点是看看如何构造出来 EmployeeDao，而不再关注其中方法的调用，现在我们就来写测试用例来 mock 一个 EmployeeDao

7.3、PowerMock 测试

好了，我们一起来测试一下上面的这个 Service 吧，然后重点的部分我用数字标识了出来，顺便做一下探讨。

```
package com.wangwenjun.powermock.construction.service;

import static com.wangwenjun.powermock.construction.dao.EmployeeDao.Dialect.MYSQL;
import static org.junit.Assert.fail;
import static org.powermock.api.mockito.PowerMockito.mock;
import static org.powermock.api.mockito.PowerMockito.whenNew;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mockito;
import org.powermock.core.classloader.annotations.PrepareForTest;
import org.powermock.modules.junit4.PowerMockRunner;

import com.wangwenjun.powermock.Employee;
```

```
import com.wangwenjun.powermock.construction.dao.EmployeeDao;

@RunWith(PowerMockRunner.class)
@PrepareForTest(EmployeeService.class)
public class EmployeeServiceTest {

    @Test
    public void test() {

        EmployeeDao employeeDao = mock(EmployeeDao.class);// (1)
        try {
            whenNew(EmployeeDao.class).withArguments(false,
MYSQL).thenReturn(
                    employeeDao);// (2)
            EmployeeService employeeService = new EmployeeService();
            Employee employee = new Employee();
            employeeService.createEmployee(employee);
            Mockito.verify(employeeDao).insertEmploye(employee);// (3)
        } catch (Exception e) {
            fail();
        }
    }
}
```

（1）我们首先 mock 了一个 EmployeeDao 实例

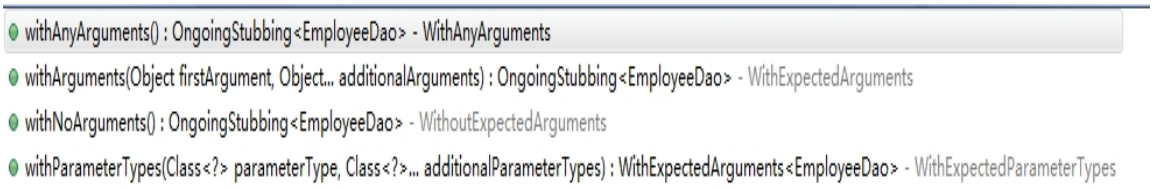
（2）我们用 whenNew 语法，并且委以相关的参数，注意这里的参数必须和 Service 中的参数一致，否则在 Service 中还会继续创建一个新的 EmployeeDao 实例。

（3）我们验证了方法的调用。

7.4、whenNew 语法

我们在上面的 PowerMock 测试用例中看到了一个新的语法那就是 whenNew，其实我们在前面的章节中都有涉猎该语法，这个语法的意思是当碰到 new 这个关键字时，返回某个被 Mock 的对象而不是让真的 new 行为发生，回想我们在 Service 中 new 一个新的 EmployeeDao，如果我们不提前准备这个 new 行为，那么他会在运行期创建一个新的 EmployeeDao 实例，此时我们 Mock 出来的实例将会不起任何作用的，并且对应的参数一

定要一致，如果您不能确保参数的一致性，那就是用 withAnyArguments，当然还有对应的很多 whenNew 语法，我们来一起看一下。



我就不多做赘述，自己看一看上面的截图就可以了。

八、Arguments Matcher

Arguments Matcher 是一个比较强大的 Mock 接口 ,其实这并不是 PowerMock 的原创，其实在 EasyMock，Mockito 中均有相关的支持，同时它也是一个比较灵活的 Mock 手段，在本章中我们来一起讨论一下如何使用 Arguments Matcher。

8.1、使用场景

有些时候我们 Mock 一个对象的时候，需要根据不同的参数返回不同的内容，虽然 Mock 的对象是假的，但是我们还是希望能够这样做，做一些临界值的测试，并且得到相关的预期结果，好了，我们先来看看业务代码，其中业务代码也比较简单，我们从最基本的 Controller 开始，然后通过 Controller 调用相应的 Service。

8.2、业务代码

Controller 的代码如下所示

```
package com.wangwenjun.powermock.matcher.controller;

import com.wangwenjun.powermock.matcher.service.EmployeeService;
```

```
public class EmployeeController {

    public String getEmail(final String userName) {
        EmployeeService employeeService = new EmployeeService();
        String email = employeeService.findEmailByUserName(userName);
        return email;
    }
}
```

Service 的代码如下所示

```
package com.wangwenjun.powermock.matcher.service;

public class EmployeeService {

    public String findEmailByUserName(String userName) {
        throw new UnsupportedOperationException();
    }
}
```

根据以前的知识，我们只能 Mock 一个 Service，并且能够预先设定一个返回结果，但是我们在编写代码的时候，提前会根据输入的参数获得不同的返回值，比如参数为 Null 或者一个不存在的结果等等。

总而言之，我们就像根据参数的不同而获得不同的返回结果。

8.3、PowerMock 测试

```
package com.wangwenjun.powermock.matcher.controller;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.fail;
import static org.powermock.api.mockito.PowerMockito.mock;
import static org.powermock.api.mockito.PowerMockito.when;
import static org.powermock.api.mockito.PowerMockito.whenNew;

import org.junit.Test;
import org.mockito.ArgumentMatcher;
import org.mockito.Mockito;

import com.wangwenjun.powermock.matcher.service.EmployeeService;
```

```
public class EmployeeControllerTest {

    @Test
    public void testGetEmail() {
        EmployeeService employeeService = mock(EmployeeService.class);
        when(employeeService.findEmailByUserName(Mockito.argThat(new
ArgumentMatcher<String>(){
            @Override
            public boolean matches(Object argument) {
                String arg = (String)argument;
                if(arg.startsWith("wangwenjun")||arg.startsWith("wwj"))
                    return true;
                else
                    throw new RuntimeException();
            }
        }))).thenReturn("wangwenjun@gmail.com");

        try {

    whenNew(EmployeeService.class).withAnyArguments().thenReturn(employe
eService);
            EmployeeController controller = new EmployeeController();
            String email = controller.getEmail("wangwnejun");
            assertEquals("wangwenjun@gmail.com",email);

            controller.getEmail("liudehua");
            fail("should not process to here.");
        } catch (Exception e) {
            assertTrue(e instanceof RuntimeException);
        }
    }
}
```

上面的这个 PowerMock 测试代码，我们通过实现一个匿名 ArgumentMatcher 类，然后就实现了根据不同参数获得不同的返回结果预期，这样我们就可以少些很多 when...thenReturn

九、Answer interface

其实 Answer 接口的作用和 Arguments Matcher 比较类似，但是它比 Arguments Matcher 更加强大，我们还是根据上一章节中的那个示例然后来说明 Answer 的使用方法。

9.1、使用场景

使用场景在这里还是罗嗦一下吧，当用户名是 wangwenjun 或者 wwj 的时候能够返回 wangwenjun@gmail.com ，当用户名是 liudehua 或者 ldh 的时候返回的是 andy@gmail.com，如果用户名是其他则抛出无法找到的异常。

9.2、业务代码

其实业务代码和上一张中一模一样，但是为了方便读者能够更好的查看，就将代码罗列到这个地方了，希望不要闲多余哈。

Service 代码

```
package com.wangwenjun.powermock.answer.service;

public class EmployeeService {

    public String findEmailByUserName(String userName) {
        throw new UnsupportedOperationException();
    }
}
```

Controller 代码

```
package com.wangwenjun.powermock.answer.controller;

import com.wangwenjun.powermock.matcher.service.EmployeeService;

public class EmployeeController {

    public String getEmail(final String userName) {
```

```
        EmployeeService employeeService = new EmployeeService();
        String email = employeeService.findEmailByUserName(userName);
        return email;
    }
}
```

9.3、PowerMock 测试

一切准备就绪，我们就是用 Answer 来进行一下 PowerMock 测试，测试代码如下

```
package com.wangwenjun.powermock.answer.controller;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.fail;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mockito;
import org.mockito.invocation.InvocationOnMock;
import org.mockito.stubbing.Answer;
import org.powermock.api.mockito.PowerMockito;
import org.powermock.core.classloader.annotations.PrepareForTest;
import org.powermock.modules.junit4.PowerMockRunner;

import com.wangwenjun.powermock.answer.service.EmployeeService;

@RunWith(PowerMockRunner.class)
@PrepareForTest(EmployeeController.class)
public class EmployeeControllerTest {

    @Test
    public void testGetEmail() {
        EmployeeService employeeService =
PowerMockito.mock(EmployeeService.class);

    PowerMockito.when(employeeService.findEmailByUserName(Mockito.anyStr
ing())).then(new Answer<String>(){

            @Override
            public String answer(InvocationOnMock invocation) throws
Throwable {
                String argument = (String) invocation.getArguments()[0];
                if("wangwenjun".equals(argument))
```

```
                return "wangwenjun@gmail.com";
            else if("liudehua".equals(argument))
                return "andy@gmail.com";
            throw new NullPointerException();
        }
    });
    try {

    PowerMockito.whenNew(EmployeeService.class).withNoArguments().thenRe
turn(employeeService);
        EmployeeController controller = new EmployeeController();
        String email = controller.getEmail("wangwenjun");
        assertEquals("wangwenjun@gmail.com",email);
        email = controller.getEmail("liudehua");
        assertEquals("andy@gmail.com",email);
        email = controller.getEmail("JackChen");
        fail("should not process to here.");
    } catch (Exception e) {
        assertTrue(e instanceof NullPointerException);
    }
  }
}
```

9.4、answer 接口中参数 InvocationOnMock

```
invocation.getArguments();（1）
invocation.callRealMethod();（2）
invocation.getMethod();（3）
invocation.getMock();（4）
```

（1）获取 mock 方法中传递的入参

（2）获取是那个真实的方法调用了该 mock 接口

（3）获取是那么 mock 方法被调用了

（4）获取被 mock 之后的对象

十、Mocking with spies

这个 spies 单词比较奇怪，是间谍，密探等意思，但是放到这个应用场合我总觉得别扭，因为它的意思好像和在 PowerMock 中的使用场合不太一样，如果读者还有更好的翻译，发信息给我哈，好了我们来看看如何使用 spies。

10.1、使用场景

经过了前面九个章节的学习，相信读者应该清楚了一点，如果某个对象是 mock 产生的，那么他什么都不会做，除非你对其做了 when...thenReturn 这样的操作，否则所 mock 的对象调用任何方法，什么都不会做的，如果您没有做过类似的测试，我们在这里一起来看看。

```
package com.wangwenjun.powermock.spy.resource;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class FileService {

    public void write(final String text) {
        BufferedWriter bw = null;
        try {
            bw = new BufferedWriter(new FileWriter(
                    System.getProperty("user.dir") + "/wangwenjun.txt"));
            bw.write(text);
            bw.flush();
            System.out.println("content write successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (bw != null)
                try {
                    bw.close();
                } catch (IOException e) {
```

```
                    // be quietly
                }
            }
        }
    }
```

上面的代码其实完全不可以用 mock 的方式来测试，但是为了说明我刚才的理论，我们来写一个测试用例来看一看。

```
package com.wangwenjun.powermock.spy.resource;

import org.junit.Test;
import org.powermock.api.mockito.PowerMockito;

public class FileServiceTest {

    @Test
    public void testWrite() {
        FileService fileService = PowerMockito.mock(FileService.class);
        fileService.write("liudehua");
    }
}
```

运行上面的测试用例，你会发现根本没有生成一个 wangwenjun.txt 文件，也就意味着被 mock 的 class 是个彻头彻尾的假对象，什么都干不了的。

10.2、PowerMock 测试

我们采用 spy 的方式 mock 一个对象，然后再调用其中的某个方法，他就会根据真实 class 的所提供的方法来调用，好了，我们再来写一个 spy 的测试

```
@Test
    public void testWriteSpy()
    {
        FileService fileService = PowerMockito.spy(new FileService());
        fileService.write("liudehua");
        File                file                =                new
File(System.getProperty("user.dir")+"/wangwenjun.txt");
        assertTrue(file.exists());
    }
```

运行上面的测试用例，你会发现生成了 wangwenjun.txt 文件，并且里面有 liudehua 字样。

10.3、何时使用 Spy

Spy 是一个特别有意思的 API，他能让你 mock 一个对象，并且只 mock 个别方法的行为，保留对某些方法原始的业务逻辑，根据具体情况选择使用。

十一、Mocking private methods

单纯就测试 private 修饰的方法而言，这个非常有争议，有人说测试 private 方法采用反射的方式会破坏 class 的封装性，有人说既然是单元测试需要面面俱到，在互联网上争论都是比较激烈的，那方都没有说服另外一方，我个人也是比较赞成不要通过反射的方式去测试一个私有方法，如果私有方法写得好，对调用处的代码写好单元测试就会完全覆盖私有方法的逻辑。

但是本章中所要体现出来的场景还真的需要去 mock 一个 private 方法，好了，来看看最后一个 PowerMock 的功能吧。

11.1、使用场景

假设我们有一个类，其中有一个方法 A 是公有方法，但是他需要依赖一个私有方法 B，但是其中的方法 B 是一个很难在单元测试中真实模拟，所以我们需要 mock 私有方法的行为，好了我们同样看一看业务代码，然后再通过 PowerMock 的方式来进行一下测试

11.2、业务代码

```
package com.wangwenjun.powermock.mockprivate;

public class EmployeeService {

    public boolean exist(String userName) {
        checkExist(userName);
        return true;
    }

    private void checkExist(String userName) {
        throw new UnsupportedOperationException();
    }
}
```

11.3、PowerMock 测试

上面的业务代码中我们发现如果要测试 exist 方法，肯定需要实现 mock 出来 checkExist 的行为，否则真的没有办法对 exist 进行测试。

```
package com.wangwenjun.powermock.mockprivate;

import static org.junit.Assert.assertTrue;
import static org.junit.Assert.fail;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.powermock.api.mockito.PowerMockito;
import org.powermock.core.classloader.annotations.PrepareForTest;
import org.powermock.modules.junit4.PowerMockRunner;


@RunWith(PowerMockRunner.class)
@PrepareForTest(EmployeeService.class)
public class EmployeeServiceTest {

    @Test
    public void testExist() {

        try {
            EmployeeService employeeService = PowerMockito.spy(new
```

```
EmployeeService());

            PowerMockito.doNothing().when(employeeService,  "checkExist",
"wangwenjun");
            boolean result = employeeService.exist("wangwenjun");

            assertTrue(result);

    PowerMockito.verifyPrivate(employeeService).invoke("checkExist","wan
gwenjun");
        } catch (Exception e) {
            e.printStackTrace();
            fail();
        }
    }
}
```

十二、总结

截至目前已经写了很多本电子书了，但是这一本写的非常仓促，而且也没有来得及进行审核，但是可以保证其中的代码都能运行通过，如果其中的一些例子运行有问题，可以发信息给我。