目录

[总评 1](#_Toc300924667)

[功能和特点 1](#_Toc300924668)

[背景介绍 1](#_Toc300924669)

[参考资料 2](#_Toc300924670)

[18.2 JMock （2） 2](#_Toc300924671)

[快速上手教程 2](#_Toc300924672)

[Expectations用法简介 5](#_Toc300924673)

[18.2 JMock （3） 6](#_Toc300924674)

[18.2 JMock（4） 8](#_Toc300924675)

[版本信息 8](#_Toc300924676)

[社区视角 9](#_Toc300924677)

# ****总评****

Mock 测试是一种常见的测试方法。通常在执行测试的时候，测试代码往往需要与一些真实对象进行交互，又或者被测代码的执行需要依赖真实对象的功能。此时，我们可以使用一个轻量级的、可控制的Mock 对象来取代真实对象，模拟真实对象的行为和功能，从而方便我们测试。jMock 便是这种方法的一种实现。

jMock 是一个利用Mock 对象来测试Java 代码的轻量级测试工具。毫不例外，它也是xUnit家族的一员，因为它从JUnit 发展而来，是JUnit的一个增强库。jMock 用法简单，易于掌握。利用它，我们可以很容易地快速构造出所需的Mock对象，从而得以方便快捷地编写单元测试代码，很适合测试驱动开发的流畅进行。

# ****功能和特点****

使用jMock，我们就不必像以往那样，停下测试代码的编写工作，转而去写专门的Mock 对象。而且，jMock 允许你以一种十分灵活的方式来精确定义对象之间彼此交互的约束，从而更好地模拟和刻画对象间的调用关系。jMock 的这种对象间调用关系的约束表达十分简洁和紧凑，这  
使得测试代码的编写变得十分简洁，同时又能很好地利用Mock 对象来达成测试意图。此外，jMock 也很容易扩展，我们可以很方便地添加自定义需求。jMock 可以和既有的其他测试框架，如JUnit，很好地整合在一起，共同使用。

# ****背景介绍****

在jMock 的官方网站上有关于该项目开发团队的人员介绍。目前具有提交权限的开发人员有5 位，他们分别是：Steve Freeman、TimMackinnon、Nat Pryce、Mauro Talevi 和JoeWalnes。值得一提的是，这几位开发者几乎都是来自以敏捷实践见长的ThoughtWorks。其中，Steve Freeman 和Nat Pryce 共同参加了2006 年4 月在英国牛津举行的ACCU Conference，并做了主题演讲，内容是关于jMock APIs 的演化，以及在Java 和C# 领域，内嵌式DSL（领域特定语言）的编写技术。

作为jMock 项目主要开发者之一的SteveFreeman，是敏捷软件开发方面的独立咨询师，他还是英国地区极限编程实践的早期推广者。他与除Mauro Talevi 外的另3 位jMock 作者共同撰写了一篇名为“Mock Roles, not Objects”的论文，探讨了有关mock 测试技术方面的经验。这篇论文被收录于2004 年的OOSPA 论文集中，在jMock 的官方主页可以找到该论文的电子版。除了jMock 之外，几位开发者还开发了jMock 的C# 实现版本——nMock。

除了开发人员以外，还有一些jMock 项目的贡献者，他们为项目提供建议、补丁及文档。不过，jMock的在线文档资源并不是很丰富，好在jMock的代码简单而又精巧，因此有兴趣的读者不妨深入代码来一探究竟。

# ****参考资料****

**网站类**

<http://www.jmock.org/>

jMock的官方网站。在这里你可以找到jMock的最新下载版本，了解有关jMock的最新消息，还有相关的文档资源，告诉你如何用jMock来编写测试代码，如何掌握约束，以及与同类型Mock测试工具的对比。

<http://www.easymock.org/>

EasyMock的官方网站。这是一个与jMock有着类似功能的Mock测试框架。目前已经更新到了2.3版本，从2.2.2版本开始，EasyMock除了支持对接口的模拟外，还支持对类的模拟。

# 18.2 JMock （2）

## 快速上手教程

**一个消息发布与订阅系统的例子**

此处，我们通过一个简单的示例来为读者示范jMock 的使用方法。这是一个简化了的消息发布与订阅系统的例子，是典型的Observer 模式，熟悉设计模式的读者对此一定不会陌生。我们用Publisher来代表消息发送方，用Subscriber 来代表消息订阅方（即接收方）。以下是Subscriber 接口的定义：

|  |
| --- |
| interface Subscriber { void receive(String message); } |

一个Publisher 可以将消息（此处以String 类型的字符串对象来表达）发送给0 个或多个Subscriber 的具体实现类。而Subscriber 的具体实现类则通过Publisher 提供的接口向其注册。在本例中，我们旨在测试Publisher 的执行逻辑，而不关心具体Subscriber 的实现逻辑。为此，我们需要构造一个Mock 对象用以模拟Subscriber 的行为。然后将其注册到Publisher 里。

首先，我们必须引入jMock 的相关包，并构造一个Mockery 对象。该对象是jMock 提供Mock 能力的统一入口，后面我们将利用它来模拟Subscriber 的行为，并用它来检验Publisher 对Subscriber的模拟对象调用过程的正确性。

|  |
| --- |
| import org.jmock.Expectations; class PublisherTest extends TestCase { Mockery context = new Mockery(); ... } |

现在，我们来编写测试方法，该测试方法的测试场景是：由Publisher 向注册其中的一个Subscriber 实例发送一条消息。

|  |
| --- |
| public void testOneSubscriberReceivesAMessage() { ... } |

我们先利用Mockery 实例来构造一个模拟的Subscriber 对象，再构造一个Publisher 对象，并将Subscriber 注册其中，然后，我们再定义一则待发布的消息。

|  |
| --- |
| final Subscriber subscriber = context.mock(Subscriber. class); Publisher publisher = new Publisher(); publisher.add(subscriber); final String message = "message"; |

紧接着，我们利用Mockery 来为模拟的Subscriber 对象定义“Expectations”——指定Publisher与Subscriber 的交互规则。此处的Expectations 是jMock 框架的一个概念。简言之，Expectations 是一组约束规则，它用于定义在测试运行期间，我们期望Mock 对象接受调用的方式。例如在本例中，我们期望Publisher 会调用Subscriber 的receive 方法一次，并且receive 方法会接收到一个String 类型的message 对象。有关Expectations 的详细说明请见后文。

|  |
| --- |
| context.checking(new Expectations() {{ one (subscriber).receive(message); }}); |

Expectations 是jMock 的一大特色，是它有别于其他Mock 测试工具的主要特征。jMock 提供了一整套丰富而灵活、简洁而紧凑的Expectations，其表达形式也很接近自然语言。整个Mock 对象的构造过程，即是利用一两行Expectations 的定义来完成的。这是内嵌式DSL 的一个典型应用，按jMock作者的说法，jMock 的Expectations 是Mock 测试这一特殊领域的DSL。接下来，我们开始调用Publisher 的执行逻辑，并验证调用后的结果：

|  |
| --- |
| publisher.publish（message）; context.assertIsSatisfied(); |

此处，我们再次利用了Mockery 实例，用以验证Publisher 对Subscriber 的调用是否如期执行。假如调用并非如预期的那样，则测试会失败。

以下是完整的示例代码：

|  |
| --- |
| import org.jmock.Mockery; import org.jmock.Expectations; class PublisherTest extends TestCase { Mockery context = new Mockery(); public void testOneSubscriberReceivesAMessage() { // set up final Subscriber subscriber = context. mock(Subscriber.class); Publisher publisher = new Publisher(); publisher.add(subscriber); final String message = "message"; // expectations context.checking(new Expectations() {{ one (subscriber).receive(message); }}); // execute publisher.publish(message); // verify context.assertIsSatisfied(); } } |

通过上面的示例，我们可以归纳出利用jMock 进行Mock 测试的一般过程，用伪代码整理如下：

|  |
| --- |
| ... 创建Mockery对象 ... public void testSomeAction() { ... 一些set up的工作 ... context.checking(new Expectations() {{ ... 此处定义Expectations ... }}); ... 调用被测逻辑 ... context.assertIsSatisfied(); ... 执行其他断言 ... } |

# Expectations用法简介

jMock 的Expectations 具有如下结构：

|  |
| --- |
| invocation-count (mock-object).method(argumentconstraints); inSequence(sequence-name); when(state-machine.is(state-name)); will(action); then(state-machine.is(new-state-name)); |

其中，mock-object 是事先构造好的Mock 对象，如前例的Subscriber；而method 则是即将接受调用的Mock 对象的方法名称，如前例Subscriber 接口的receive 方法。除去invocation-count 和mock-object 外，后续内容都是可选的。同时，你也可以根据实际需要，为某个Expectation 追加多个inSequence、when、will 和then 子句。

# 18.2 JMock （3）

**invocation-count**

代表期望的方法调用次数，jMock 提供了表达方法调用次数的多种手段，如表18-1 所示：

表18-1 jMock提供的方法调用次数的表达形式

|  |
| --- |
| <http://book.51cto.com/files/uploadimg/20080626/0916030.jpg> |
|  |

argument-constraints

代表方法调用传入参数的约束条件，可以是精确匹配的条件，如下例所示，calculator 的add 方

法只期望接受两个整数1 作为参数：

|  |
| --- |
| one (calculator).add(1, 1); |

也可以利用with 子句定义模糊匹配条件，同样是calculator 的add 方法，在下例中则期望接受任意int 类型的参数：

|  |
| --- |
| allowing (calculator).add(with(any(int.class)), with(any(int.class))); |

除any 外，jMock 还提供了各种其他形式的参数约束子句，如表18-2 所示：

表18-2 jMock提供的参数约束子句

|  |
| --- |
| <http://book.51cto.com/files/uploadimg/20080626/0916031.jpg> |
|  |

**will**

代表方法调用返回情况的约束条件，jMock 提供的返回约束如表18-3 所示：

表18-3 jMock提供的返回约束

|  |
| --- |
| <http://book.51cto.com/files/uploadimg/20080626/0916032.jpg> |
|  |

**inSequence**

用于定义多个方法调用的执行顺序，inSequence 子句可以定义多个，其在测试代码中出现的次序，便是方法调用的执行顺序。为了定义一个新的顺序，首先需要定义一个Sequence 对象，如下所示：

|  |
| --- |
| final Sequence sequence-name = context.sequence("sequencename"); |

而后，为了定义方法调用的执行顺序，可以在依序写好的每个Expectation 后面添加一个inSequence 子句。如下所示：

|  |
| --- |
| one (turtle).forward(10); inSequence(drawing); one (turtle).turn(45); inSequence(drawing); one (turtle).forward(10); inSequence(drawing); |

when和then

用于定义方法仅当某些条件为true 的时候才调用执行，从而进一步对Mock 对象的调用情况进行约束。在jMock 中，这种约束是通过状态机的形式来达成的。首先，我们需要定义一个状态机实例，其中的初始状态（initial-state）是可选的：

|  |
| --- |
| final States state-machine-name = context.states("state-machine-name").startsAs("initialstate"); |

然后，我们可以利用when 子句来定义当处于某状态时方法被调用执行，利用then 来定义当某方法被调用执行后，状态的迁移情况。举例如下：

|  |
| --- |
| final States pen = context.states("pen").startsAs("up"); one (turtle).penDown(); then(pen.is("down")); one (turtle).forward(10); when(pen.is("down")); one (turtle).turn(90); when(pen.is("down")); one (turtle).forward(10); when(pen.is("down")); one (turtle).penUp(); then(pen.is("up")); |

# ****18.2 JMock（4）****

## ****版本信息****

从jMock 的官方网站可以下载到当前的最新版本。目前，它的最新版本是2007 年8 月发布的2.4.0 版。该版本引入了对JUnit 4.4 的支持，并做了许多小的改进。

jMock 自2006 年发布1.2.0 版以后，其API组成有了较大的变化，进而对jMock 的使用方法也产生了影响。目前，在jMock 的官方网站上分别有jMock 1 和jMock 2 两个系列版本的下载，而相应的文档也各自有两份。可以认为，jMock 1 和jMock 2 是两个并行独立的分支。可能这一点对于以往习惯了使用jMock 1 系列版本进行Mock 测试代码编写的开发者而言会有些困惑。不过，既然是两个并行独立的分支，并且目前也都已进入了“Stable”阶段，对于jMock 的老用户而言，就无需担心因jMock 版本升级而造成的代码不兼容问题了，因为我们仍然可以使用jMock 1，而不必刻意升级到jMock 2。当然，如果是新启动的项目，那么使用jMock 2 会是更好的选择。

## ****社区视角****

jMock 的使用可以给基于Mock 技术的单元测试编写提供很大的便利性，利用它我们可以快速编写出Mock 对象，进而对被测对象进行隔离测试。

其实，在Mock 测试领域里，还有不少优秀的Mock 测试框架，比如时常被人们与jMock 相提并论的EasyMock，优秀的开源软件Spring 中便使用了EasyMock，而Spring 自身也提供了一组方便的Mock 对象，这些Mock 对象与Spring的发布包一起发布。jMock 并非绝对优于其他同类软件，每个Mock 测试框架都有其自身的特点，也各有优缺点。

此处，笔者摘选jMock 作为介绍对象的主要原因，是其在Mock 对象构造方面的独特方式。简洁而接近自然语言的表达形式，是DSL 技术的一个有趣应用，同时也使得Mock 对象的构造过程既简单又精确。此外，以往人们在使用jMock 1的时候，往往会抱怨jMock 要求TestCase 必须继承自MockObjectTestCase。而这一点在以单根继承为特征的Java 语言里是很忌讳的，因为这样将阻碍测试用例继承其他的父类。不过，从上面的例子中大家已经看到，jMock 2 在这方面做了改进，不再对TestCase 有特殊父类的限制，这应该说是一个很大的改善。

此外，在Mock 技术的运用方面也需要有一个准确的把握，过多的使用Mock 对象也可能会导致问题出现。当一个对象在测试之前需要构造过多的Mock 对象时，当测试代码中构造Mock对象的代码逻辑占据了绝大多数篇幅时，往往意味着被测代码本身存在着设计上的问题。而像jMock 这样的Mock 测试工具的引入，使得Mock对象的构造变得十分容易，这往往也会助长代码中“Bad Smell”的蔓延：即便代码中存在过度的耦合也没有关系，因为似乎一切可以很方便地依赖于Mock。没有银弹，设计的缺陷是单纯的Mock技术所无法解决的，也不是它本该解决的，这属于技术的误用。

最后需要指出的是，虽然jMock 提供了很多对Mock 对象方法调用进行约束的表达手段，但是多数时候我们只需要用到其中的很小一部分即可。过多的对Mock 对象方法调用的约束，也就意味着你需要对被测代码所依赖的其他对象有更多的依赖和认识，而一旦被依赖的对象面临重构，则往往会导致相关测试用例的失败。有时候，修复这些失败的测试用例往往是一件很繁琐的事情，这对TDD 和重构的流畅实践是一大障碍。