JMockit
官方教程

fartpig

Published
with GitBook

目錄

目录	0
简介	1
模拟	2
伪装	3
代码覆盖	4
企业应用	5

JMockit 测试工具入门

简介

1. 开发人员测试和测试独立、
2. 使用伪装对象进行测试
3. 使用伪装对象进行测试的工具
4. 传统伪装对象的问题
5. 一个例子 6.使用JMockit运行测试

模拟

6. 模拟类型和实例
7. 期望
8. 录制和验证的模式
9. 普通和严格的期望
10. 记录异常的结果
11. 匹配特定实例的调用
12. 对参数值得灵活匹配
13. 定义调用的次数约束
14. 显式验证
15. 捕捉验证时的调用参数
16. 代理：定义自己的结果
17. 级联模拟
18. 部分模拟
19. 捕捉实现类和实例
20. 实例化和注入测试对象
21. 重用期望和验证代码块
22. 其他主题

伪装

23. 伪装方法和伪装类
24. 为测试设置伪装
25. 伪装接口
26. 伪装未被实现的类

27. 调用次数的约束
28. 伪装方法的初始化
29. 使用调用上下文
30. 执行真正的实现部分
31. 在测试时间重用伪装
32. 在测试类和测试套件级别使用伪装

代码覆盖的度量

33. 行覆盖
34. 路径覆盖
35. 数据覆盖
36. 覆盖输出的类型
37. 配置覆盖工具
38. 聚集多个测试结果报告
39. 检测最小的覆盖
40. 在maven项目中激活覆盖
41. 关闭覆盖输出
42. 独立模式

测试企业应用

43. 基于场景的测试
44. 一个例子

简介

1. 自动化开发人员测试和测试独立
2. 使用模拟对象进行测试
3. 使用模拟对象进行测试的工具
4. 常见的使用模拟对象的问题
5. 一个测试用例
    i. 使用 Expectations API
    ii. 使用 Mockups API
6. 使用 JMockit 运行测试用例
    i. 使用 JUnit Ant 任务运行测试用例
    ii. 使用 Maven 运行测试用例

在这个教程中我们将会通过使用测试用例例子进行 JMockit 可以使用的 API 学习。主要的用来进行模拟注入的 API是Expectations API。其次是 Mockups API,主要是用来进行解决伪装使用的，为了避免处理复杂的外部组件。

尽管这个教程比较完整，它仍然不能具体的覆盖所有的已经发布的APIs。 一个完成和详细的定义关于所有的公开类，方法等 是通过API 文档进行提供的。 每一个版本的工具文档都能够在 "jmockit.github.io/api1x" 文件夹下发现, 里面包括了完整的分发zip 文件。 这些主要的库文件"jmockit.jar" (和同等的maven文件中), 包含了 Java 源文件 (包含Javadoc 组件)对于每个 Java IDE都能够轻松的从中获取到 API 源码 和文档。

一个独立的章节包含了代码覆盖测试的工具。

自动化开发人员测试和测试独立

软件测试被软件开发人员编写，用来测试他们自己的代码对于与一个成功的软件开发来说是非常重要的。这些测试总是被通过使用测试框架来编写的，例如 JUnit 或者 TestNG; JMockit 拥有同时支持这两种测试框架。

自动化的开发人员测试可以被分为两种类别：

1. 单元测试，被用来测试相对于独立其他系统的一个类或者是一个组件。
2. 集成测试，被用来测试系统操作用由一个单元和它的依赖完成(其他的类/组件将会与这些测试进行交互)。

即使集成测试还包括了与多个单元的交互，特殊的测试可能不关心测试所有的组件，层或者是其他子系统。这种独立测试代码排除不相关部分的能力是非常有用的。

使用模拟对象进行测试

一个通用的和强大的技术用来测试代码在一个独立的环境的被称为模拟。传统上来说一个模拟对象是一个单独的定义和实现针对单个测试或者相关测试集合的类实例。这个实例在测试的时候被注入到依赖它的代码中。每一个模拟对象的行为应该在测试代码或者是测试代码使用的保持一致，因此所有的测试用例都能够通过。然后这不仅仅是模拟对象的唯一功能。在每一个测试结束断言中，模拟对象还可以添加附加的断言。

在传统的模拟对象工具例如EasyMock, jMock, and Mockito (在下一个章节将会详细介绍)以上的描述总是成立的。JMockit不仅仅包括这些传统的用法，还包括允许模拟对象的方法和构造函数，在"实际" (非模拟) 的类中, 用来减少实例化模拟对象在测试和测试环境下传递他们。此外，通过在测试环境下新建的对象将会被执行按照测试定义的模拟的行为，当方法和构造函数被在实际类中进行调用的时候。使用 JMockit, 原始的已经存在方法或构造函数的实现将会被临时的模拟进行替换，通常用来作为单个测试用例的时候。这种模拟方式相当的通用，因此不仅仅是公开的实例方法，而且包括 final 和 static 方法, 包括构造函数都可以被他们的实现进行替换，因此这些被称为可以模拟。

模拟对于单元测试来说是相当的有用的，但是可以被用来做集成测试。例如你可能需要测试一个展现层对象，在和他与其他同层的类进行交互的时候，不需要实际依赖于调用到其他系统层的代码，例如业务逻辑或者是基础架构层。

使用模拟对象进行测试的工具

可以用来作为模拟对象的测试工具包括 EasyMock 和 jMock, 他们都是基于 java.lang.reflect.Proxy, 通过在运行时给定一个对象接口生成一个实现。它还可以通过针对具体的类通过 基于CGLIB的子类生成。 每一个工具都包含许多 API 来表达在被调用的方法或者死结尾的预期和校验。很少看到在 JUnit tests中使用像EasyMock/jMock的预期那样的校验代码，往往是使用 JUnit自己的断言方法。

JMockit 有自己定义的期望 API, 和其他的 APIs类似，不过远甚于它们提供了模拟所有类型的方法，构造函数和类型的能力(接口，抽象类，最终的或非最终类，从JRE中的类，枚举等)。

有其他组的伪装工具工具，依赖于显示的期望验证而不是隐式的验证: Mockito 和 Unitils Mock。 这些伪装apis的通用特征是他们直接使用对伪装对象的调用来定义期望。 在 EasyMock 和 jMock的案例中, 这些调用只能在测试单元下进行调用, 在所谓的测试录制阶段。在 Mockito 和 Unitils Mock的案例中, 在其他方面，调用甚至能够爱测试单元之后进行。 (录制和再重放阶段之间T，在测试单元眞实执行他们所模拟的调用时。) JMockit提供验证API, 一个对期望 API的自然扩展, 能够允许显示的验证期望在验证阶段。

使用传统的模拟对象的问题

传统的使用模拟对象为了到达独立的解决方案需要强加明确的在测试代码级别的约束。JMockit被创建作为规避这种约束的替代，通过使用 java.lang.instrument Java 6+ 的包进行字节码技术 (还使用了 - 较少的程度上 - 反射 , 动态代理, 和自定义类的载入)。

针对伪装对象的主要约束是需要模拟的可选类不是要实现一个独立的接口，就是所有的需要被模拟的方法必须为可以被重载。此外，依赖的实例化必须依赖于外部的依赖单元，这样才能有一个代理对象（模拟对象）被传入代替实际上每个依赖的实现。此外代理类不能被简单的初始化使用new操作符在客户端代码中，因为构造方法的调用不能通过传统的技术进行拦截。

总结来看，有些一个传统的模拟方法在设计使用中的一些限制:

1. 应用类必须实现一个独立接口 (为了使用 java.lang.reflect.Proxy) 或者是不被什么 final (为了能够动态生成一个包含重写方法的子类的)。在第二个情况下，需要被模拟的实例方法不能为最终的。 显然，建立java接口只用来模拟的实现是不需要的。分离的接口（或者是更普通的抽象）只有当在生成代码中有多个实现的时候才被建立。 在 Java, 标记类和方法为final是可选的，即使大多数的类不是被设计为扩展子类用的。申明他们为最终类是社区或者是通常的java编码实践中所推荐的(像书中 "Effective Java, 2nd edition", 和 "Practical API Design")。 此外，它允许静态的分析工具 (例如Checkstyle, PMD, FindBugs, 或者 你的 Java IDE) 提供了有用的对代码的警告 (例如，关于一个最终方法申明了将抛出一个特殊的需要检查的异常，但是实际上并没有抛出; 这种警告不会出现在一个给定为非最终的方法中，由于它的子类可能重写它，可能抛出异常)。
2. 非静态方法的模拟行为有时需要被调用。 在实际中，许多的 APIs 包括静态方法最为入口或者是一个工厂方法。为了能够偶尔模拟他们是一个很实际难以避免的开销，比如建立包装类而这些类是不存在的。
3. 需要被测试的类需要提供一些方法注入依赖的模拟实例。这通常意味着需要附加的 setter 方法 或者是 构造方法被建立在依赖的类中。 通常一个依赖的类不能够简单的通过使用new操作获取到他们的依赖实例，甚至在自然条件上做这些。依赖注入是一个技术名称意味着从使用中分离出配置, 为了使用对承认的多个实现进行依赖，有其中的一个被选中通过一些配置的代码。 很不幸的是，一些开发人员选了这种技术，建立了多个独立的接口包含各自独立的实现，或者是包含了一个较多数量的不必要的配置代码。过分使用依赖注入框架或者是容器是相当的危险的，当使用无状态的单例对象代替了适当的有状态的短生命周期的对象。 使用 JMockit, 任何设计都将被在独立的环境下测试，不会限制开发人员的自由。 对于可测试使使用传统的伪装的消极影响的设计是无关紧要的，当使用了新的方式。 结果，可测试变成无关紧要的在应用涉及中，允许开发人员避免这种分隔接口，工厂，依赖注入等的复杂性，尽管他们对系统的需求是多么无理。

一个例子

在我们自己讨论每一个模拟 API的方法在如下章节的时候，让我们来看一个快速的例子。 考虑到一个业务服务类提供了包括如下步骤的业务操作:

1. 找到需要被操作所需要的持久化的实体
2. 持久化一个新的实体状态、
3. 发送一个提醒消息给一个有兴趣的部门

第一二步需要使用应用数据库，其通过使用简单的 API 来使用持久化系统。 第三步骤可以通过使用一个第三方的API来发送邮件，例子中是使用 Apache的 通用邮件库。

因此，这个服务类包含了两个独立的依赖，一个是持久化，另外一个是邮件。为了进行业务操作的单元测试来验证恰当的和这些依赖进行合作，我们使用了模拟API。对于整个工作解决方案的完整源码 - 包含所有的测试 - 可以在线查看。

```
package jmockit.tutorial.domain;

import java.math.*;
import java.util.*;
import org.apache.commons.mail.*;
import static jmockit.tutorial.persistence.Database.*;

public final class MyBusinessService
{
   private final EntityX data;

   public MyBusinessService(EntityX data) { this.data = data; }

   public void doBusinessOperationXyz() throws EmailException
   {
      List<EntityX> items =
(1)      find("select item from EntityX item where item.someProperty = ?1", data.getSomeP

      // Compute or obtain from another service a total value for the new persistent enti
      BigDecimal total = ...
      data.setTotal(total);

(2)   persist(data);

      sendNotificationEmail(items);
   }

   private void sendNotificationEmail(List<EntityX> items) throws EmailException
   {
      Email email = new SimpleEmail();
      email.setSubject("Notification about processing of ...");
(3)   email.addTo(data.getCustomerEmail());

      // Other e-mail parameters, such as the host name of the mail server, have defaults
      // through external configuration.

      String message = buildNotificationMessage(items);
      email.setMsg(message);

(4)   email.send();
   }

   private String buildNotificationMessage(List<EntityX> items) { ... }
}
```

数据库类包括了一些静态方法和一个私有的构造函数；查找和持久化的方法是必须的，因此我们这里就没有罗列出来 (假设他们是通过 ORM API来实现的, 比如 JPA)。

因此，我们改如何不需要任何修改已经存在应用代码的情况下进行"doBusinessOperationXyz" 方法的单元测试呢？JMockit 提供了两种不同的模拟APIs, 每一种都能满足你的需求。我们将看到每种的写法的不同例子。 在每一个情况下， 一个 JUnit测试例子将会验证单元测试对外部依赖所兴趣的调用。这些调用都是通过点 (1)-(4) 如上进行标注。

使用期望 **API**

首先来看看期望 API。

```
package jmockit.tutorial.domain;

import org.apache.commons.mail.*;
import jmockit.tutorial.persistence.*;

import org.junit.*;
import mockit.*;

public final class MyBusinessService_ExpectationsAPI_Test
{
    @Mocked(stubOutClassInitialization = true) final Database unused = null;
    @Mocked SimpleEmail anyEmail;

    @Test
    public void doBusinessOperationXyz() throws Exception
    {
      final EntityX data = new EntityX(5, "abc", "abc@xpta.net");
      final EntityX existingItem = new EntityX(1, "AX5", "someone@somewhere.com");

      new Expectations() {{
           Database.find(withSubstring("select"), any); (1)
         result = existingItem; // automatically wrapped in a list of one item
      }};

      new MyBusinessService(data).doBusinessOperationXyz();

        new Verifications() {{ Database.persist(data); }};(2)
        new Verifications() {{ anyEmail.send(); times = 1; }};(4)
    }

    @Test(expected = EmailException.class)
    public void doBusinessOperationXyzWithInvalidEmailAddress() throws Exception
    {
      new Expectations() {{
            anyEmail.addTo((String) withNotNull()); result = new EmailException(); (3)
      }};

      EntityX data = new EntityX(5, "abc", "someone@somewhere.com");
      new MyBusinessService(data).doBusinessOperationXyz();
    }
}
```

在行为导向的模拟 API像 JMockit 期望那样, 每一个测试都可以被分为三个连续的步骤: 录制，重放，验证。 在期望的录制块中定义录制, 在期望验证块中进行验证; 重放指的是在测试的内部代码中进行调用。注意，只有模拟方法的调用次数; 隶属于类或者实例中的方法 (或 构造方法)和相关的模拟属性或者模拟参数将不被模拟，因此不能被录制，更不能进行验证或者是重放了。

正如上面的实例测试，录制和验证期望可以通过调用所需要方法在一个录制或者是验证的块中来实现 (包括构造方法，及时没有在这里显示)。 这些方法的参数匹配可以通过 API的属性例如 "any" 和 "anyString", 或者是通过 API 方法 比如 "withNotNull()"。 在重放中所匹配的调用返回值 (或 抛出的异常)可以在录制阶段通过result属性进行定义。调用次数的约束可以被定义,不仅仅可以在录制阶段也可以在验证阶段, 通过 API 属性赋值比如 "times = 1"。

使用 Mockups API

接下来，让我们看使用 Mockups API 进行编写的例子。

```
package jmockit.tutorial.domain;

import java.util.*;
import org.apache.commons.mail.*;
import jmockit.tutorial.persistence.*;

import static org.junit.Assert.*;
import org.junit.*;
import mockit.*;

public final class MyBusinessService_MockupsAPI_Test
{
    public static final class MockDatabase extends MockUp<Database>
    {
      @Mock
      public void $clinit() { /* do nothing */ }

      @Mock(invocations = 1)
        public List<EntityX> find(String ql, Object... args)(1)
      {
         assertNotNull(ql);
         assertTrue(args.length > 0);
         return Arrays.asList(new EntityX(1, "AX5", "someone@somewhere.com"));
      }

      @Mock(maxInvocations = 1)
      public void persist(Object o) { assertNotNull(o); }(2)
    }

    @BeforeClass
    public static void mockUpPersistenceFacade()
    {
      // Applies the mock class by invoking its constructor:
      new MockDatabase();
    }

    final EntityX data = new EntityX(5, "abc", "5453-1");

    @Test
```

```
    public void doBusinessOperationXyz() throws Exception
    {
      // Defines and applies a mock class in one operation:
      new MockUp<Email>() {
         @Mock(invocations = 1)
         Email addTo(Invocation inv, String email)
         {
            assertEquals(data.getCustomerEmail(), email);
            return inv.getInvokedInstance();
         }

         @Mock(invocations = 1)
         String send() { return ""; }(4)
      };

      new MyBusinessService(data).doBusinessOperationXyz();
    }

    @Test(expected = EmailException.class)
    public void doBusinessOperationXyzWithInvalidEmailAddress() throws Exception
    {
      new MockUp<Email>() {
         @Mock
         Email addTo(String email) throws EmailException (3)
         {
            assertNotNull(email);
            throw new EmailException();
         }
      };

      new MyBusinessService(data).doBusinessOperationXyz();
    }
}
```

在这里，我们不采用对模拟类或实例的调用进行录制或者是调用，我们直接伪装了所感兴趣的方法和构造方法。

通常来说，大多数的测试都可以通过使用期望 API进行编写。 然而也有一些情况下，伪装API对于完成这些工作也是有用的。

使用 **JMockit** 运行测试用例

为了运行使用 JMockit APIs的测试, 可以使用你的 Java IDE, Ant/Maven 脚步, 等等，或者是你通常使用的方式。 原则上，任何 JDK 1.6 或者是更新的, 在 Windows, Mac OS X, 或者 Linux, 都能被使用。 JMockit 支持 (需要) 使用 JUnit 或 TestNG; 每一个测试框架的详细定义如下所示:

- 针对 JUnit 4.5+ 测试套件, 确保 在classpath上 jmockit.jar 出现在 JUnit jar之前。此外，

使用 @RunWith(JMockit.class)所注释的测试用例。 (关于 Eclipse 用户请注意: 当定义 jars的顺序在 classpath的时候, 确保使用的是 "Order and Export" 标签 在 "Java Build Path" 窗口中。 同样的, 确保 Eclipse 项目使用的JRE是从一个JDK中安装的而不是一个纯的 JRE, 因为后者缺少附加本地库的功能。)

- 针对 TestNG 6.2+ 测试套件, 简单的添加 jmockit.jar 到 classpath下 (在任何位置)。 在其他的情况下 (像运行其他 JDK 的实现而不是 Oracle JDK), 你可能需要添加 "-javaagent:/jmockit.jar" 作为JVM初始化的参数。 这些可以通过在"Run/Debug Configuration"菜单 在 Eclipse 和 IntelliJ IDEA中, 或者 使用构建工具例如Ant 和 Maven 进行。

使用 **JUnit Ant** 任务运行测试用例

当在build.xml脚步中使用 元素时, 使用一个独立的JVM实例是重要的。 例如，如下的一些内容:

```
<junit fork="yes" forkmode="once" dir="directoryContainingJars">
   <classpath path="jmockit.jar"/>

   <!-- To generate (if desired) a code coverage HTML report: -->
   <classpath path="jmockit-coverage.jar"/>

   <!-- Additional classpath entries, including the appropriate junit.jar -->

   <batchtest>
      <!-- filesets specifying the desired test classes -->
   </batchtest>
</junit>
```

使用 **Maven** 运行测试用例

JMockit安装包已经部署到了 Maven的中心仓库中。 为了在测试套件中使用它们, 添加如下的内容到你的 pom.xml 文件中:

```
<properties>
   <jmockit.version>desired version</jmockit.version>
</properties>

<dependencies>
   <dependency>
      <groupId>org.jmockit</groupId>
      <artifactId>jmockit</artifactId>
      <version>${jmockit.version}</version>
      <scope>test</scope>
   </dependency>
</dependencies>
```

确保所定义的版本是你所需要的。(当然这个所用的属性是可选的。)当使用JUnit, 这些依赖不要在"junit" 依赖之前。

针对更多的 JMockit Coverage在maven，可以看相关在那章中的内容。

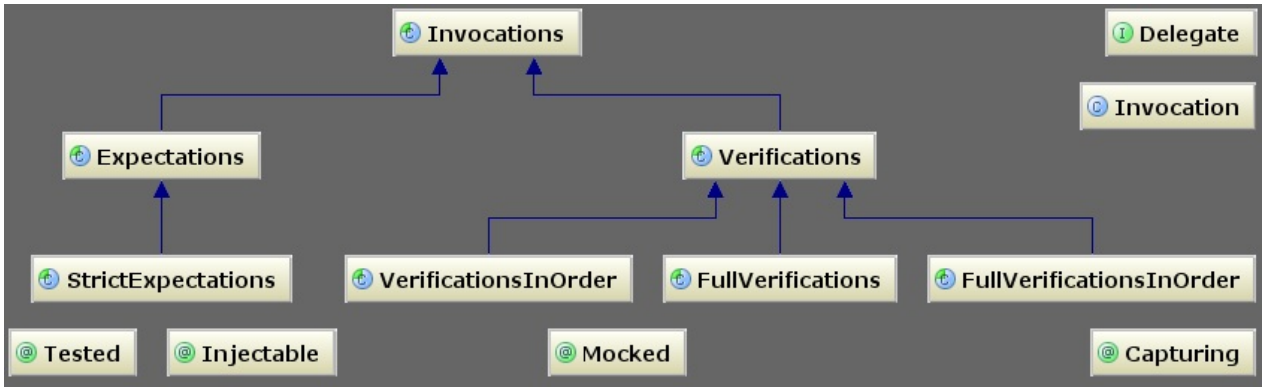
模拟

1. 模拟类型和实例
2. 期望
3. 录制和验证的模式
4. 普通和严格的期望
    i. 严格和非严格的模拟
5. 记录所期望的结果
6. 匹配特定实例的调用
    i. 注入模拟实例
    ii. onInstance(m) 约束
    iii. 使用给定构造方法新建实例
7. 对参数值得灵活匹配
    i. 使用any 属性匹配参数
    ii. 使用with 方法进行参数匹配
    iii. 使用 null 值匹配所有任何其他对象的引用、
    iv. 通过 varargs 参数匹配参数的传入
8. 定义调用的次数约束
9. 显式验证
    i. 验证调用从来没有发生
    ii. 验证顺序
    iii. 部分顺序的验证
    iv. 全验证
    v. 按顺序进行全验证
    vi. 限制模拟类型集合进行全验证
    vii. 验证没有调用发生
    viii. 验证未定义的调用不能发生
10. 捕捉验证时的调用参数
    i. 从单个调用中捕捉参数
    ii. 从多个调用中捕捉参数
    iii. 捕捉新实例
11. 代理：定义自己的结果
12. 级联模拟
    i. 级联静态工厂方法
    ii. 级联自返回的方法
13. 部分模拟
14. 捕捉实现类和实例
    i. 模拟未明确实现的类

    ii. future 实例的定义行为
15. 实例化和注入测试对象
16. 重用期望和验证代码块
17. 其他主题
    i. 同时模拟多个接口
    ii. 迭代期望
    iii. 验证迭代

在 JMockit 工具中, 期望 API 对使用模拟在自动化测试开发中提供了丰富的支持。当模拟被使用的时候，一个测试关注于测试代码的结果是否与它交互的其他类型一致。模拟通常被适用于独立的单元测试的构造中，既一个测试单元被相对于独立其他的所依赖的单元实现。通常来说，一个单元的行为被嵌入到单个类中，但是它被适用于一个具有很强关联的类的集合中，作为一个独立的测试单元 (通常来说这种情况下，我们包含一个中心的类包含一个或者是多个帮助类，可能是包内私有的); 通常来说，独立的方法不应该被当做一个他们分开的单元。

然而严格的单元测试不是所推荐的方式；不需要尝试模拟所有独立的依赖。模拟最好被适度的应用；任何可能的，更好的情况下集成测试远胜于单独的单元测试。这也就是说，模拟只在偶尔建立集成测试的时候比较有用，当有一个特殊的依赖不能被很容易的建立实现或者是当尝试建立对边缘测试的时候能够起到很好的作用。



在两个类之间的交互通常采用一个方法或者是构造方法的调用。从一个测试类到它依赖的一些列调用，包括参数和返回值在他们之间，定义了对特殊类在测试中的有兴趣的行为。此外，一个给定的测试可能需要验证在多个调用中的相对顺序。

模拟类型和实例

在测试中对一个依赖的方法和构造方法的调用是模拟的对象。模拟提供了我们需要独立测试代码从他们的依赖中的机制。我们指定为一个给定测试的特殊的依赖被模拟通过申明合适的模拟属性和或者是模拟参数; 模拟属性被使用注释在测试类属性上进行申明, 然而模拟参数要在一个测试方法的参数上使用注释申明。 需要被模拟的依赖的类型可以使模拟属性或是参数。这个类型可以是任何类型的引用: 一个接口，一个类 (包含抽象和最后的类型), 一个注释，一个枚举。

缺省的，所有的模拟类中的非私有方法 (包括任何的静态，最终或者是本地) 可以在测试运行时被模拟。 如果被申明的模拟类型是一个类，所有的它父类不包括 java.lang.Object 都将被递归的模拟。因此，继承的方法都将被自动的模拟。同样的在这个类中的所有非私有构造方法都将被模拟。

当一个方法或者是构造方法被模拟的时候，它原来的实现代码就不会在测试调用的时候执行。而是调用被重定向到 JMockit中，因此它能够被按照隐式或显式的定义的方法进行测试。

如下的例子测试骨骼服务作为一个基础插图正如他们被测试代码所使用的那样，使用了模拟属性和模拟参数的申明。在这个教程中，我们使用很多代码片段，黑体部分是我们主要关注的说明。

```
// "Dependency" is mocked for all tests in this test class.
// The "mockInstance" field holds a mocked instance automatically created for use in each
@Mocked Dependency mockInstance;

@Test
public void doBusinessOperationXyz(@Mocked final AnotherDependency anotherMock)
{
   ...
   new Expectations() {{ // an "expectation block"
      ...
      // Record an expectation, with a given value to be returned:
      mockInstance.mockedMethod(...); result = 123;
      ...
   }};
   ...
   // Call the code under test.
   ...
   new Verifications() {{ // a "verification block"
      // Verifies an expected invocation:
      anotherMock.save(any); times = 1;
   }};
   ...
}
```

作为一个被申明的模拟参数在测试方法中，申明类型的实例将会被自动被JMockit建立和通过JUnit/TestNG运行期自动传入，当执行到这些测试方法的时候; 因此，参数值不可能为 null。对于一个模拟属性，被申明类型的实例将被JMockit 自动建立和赋值给属性，确保其不是不可变的。

对于模拟属性和参数有一些注释申明上的不同，缺省的模拟行为可以被修改来满足特殊的测试。这个章节的其他部分将详细讨论这些，但是基础是: @Mocked 是中心模拟注释，包含一些可选的属性在特殊情况下比较有用; @Injectable 是另外一个模拟注释，其约定只模拟单个

模拟实例中的实例方法; 而 @Capturing 是另外一个模拟注释, 其扩展到实现模拟接口的类, 或者是继承自模拟类的子类。 当 @Injectable 或 @Capturing 被应用到一个模拟属性或者是模拟参数的时候，, @Mocked 就被隐式的使用了，不需要再使用 (但是也可以) 。

JMockit 建立的模拟实例能够正常的在测试代码中使用(为了录制和验证期望), 和/或 传递到测试代码中。或者他们可能简单的不被使用。于其他的 mocking APIs不同的是, 这些模拟对象不需要被测试代码使用，当调用了依赖中的实例方法的时候。缺省的 (例如，当 @Injectable 不被使用的时候), JMockit 不关系哪些模拟实例方法被调用。这允许透明的模拟实例能够在测试代码中直接建立，当说到代码调用构造方法构建新的实例使用new操作符；这些实例化的类不需要被所申明的模拟类型所覆盖, 就这么多了。

期望

期望是与给定测试相关的需要被模拟的方法/构造方法的调用集合。一个期望可能覆盖多个对相同方法或构造方法的不同调用，但是它不需要覆盖在测试执行是偶的所有可能调用。是否一个特殊的调用匹配给定的期望或是不仅仅依赖于方法/构造方法的签名，还需要包括方法调用的实例，参数值，和或者是已经被匹配的调用数目。因此，有不同类型的匹配约束可选的别定义在给定的期望中。

当我们有一个或者是多个调用参数需要使用的时候，一个确定的参数值可能需要针对每个参数进行定义。例如，字符串 "test string" 可能被定义为一个字符串参数, 导致期望只匹配当真实的参数值是这个的时候。正如我们后来看到的，不仅仅可以定义准确的值，我们还可以定义更多灵活的约束可能匹配所有的不同参数值集合。

下面将会显示一个针对 Dependency#someMethod(int, String)的期望, 只会匹配确切的参数值的方法调用。注意这个期望只定义通过一个独立对模拟方法的调用。没有特殊的 API 方法 methods 涉及, 正如其他的模拟 APIs一样。 但是这个调用却没有计算一个我们所感兴趣的测试中的真是调用次数。只是简单的定义了期望。

```
@Test
public void doBusinessOperationXyz(@Mocked final Dependency mockInstance)
{
   ...
   new Expectations() {{
      ...
      // An expectation for an instance method:
      mockInstance.someMethod(1, "test"); result = "mocked";
      ...
   }};

   // A call to code under test occurs here, leading to mock invocations
   // that may or may not match specified expectations.
}
```

在我们深入了解在录制，重放和调用验证不同之后，我们将看到更多关于期望的使用。

录制和验证的模式

任何一个开发测试都可以被分成至少三个不同的独立执行阶段。阶段顺序被执行，一次只执行一个，如下所示。

```
@Test
public void someTestMethod()
{
   // 1. Preparation: whatever is required before the code under test can be exercised.
   ...
   // 2. The code under test is exercised, usually by calling a public method.
   ...
   // 3. Verification: whatever needs to be checked to make sure the code exercised by
   //    the test did its job.
   ...
}
```

首先，我们有一个准备阶段，这里需要的对象和数据项目被创建或者是从一些地方获取。然后，测试代码被执行，从测试代码中执行之后的结果和预期的值进行比较。

这个三阶段模式也被称为安排，行动，断言语法，或者是简称为AAA.不同的单词，但是意思是共同的。

在基于模拟类型的行为测试的上下文中 (和他们的模拟实例), 我们可能定义如下的其他阶段, 相对于之前描述的三个常用测试阶段:

1. 录制阶段，这里调用被录制。这个发生在测试准备阶段，在我们需要的测试被执行之前。
2. 重放阶段，这时所感兴趣的模拟调用有机会被执行，正是测试代码被执行时。之前被模拟的方法/构造方法调用将被重新再现。虽然，通常没有从调用录制到重放的一一对应。
3. 验证阶段，在这里调用被验证是否和期望的一样。这通常发生在测试验证，在测试调用之后被执行。 基于行为的测试在 JMockit 中写起来，满足如下的模板:

```
import mockit.*;
... other imports ...

public class SomeTest
{
   // Zero or more "mock fields" common to all test methods in the class:
   @Mocked Collaborator mockCollaborator;
   @Mocked AnotherDependency anotherDependency;
   ...
```

```
    @Test
    public void testWithRecordAndReplayOnly(mock parameters)
    {
       // Preparation code not specific to JMockit, if any.

       new Expectations() {{ // an "expectation block"
          // One or more invocations to mocked types, causing expectations to be recorded.
          // Invocations to non-mocked types are also allowed anywhere inside this block
          // (though not recommended).
       }};

       // Unit under test is exercised.

       // Verification code (JUnit/TestNG assertions), if any.
    }

    @Test
    public void testWithReplayAndVerifyOnly(mock parameters)
    {
       // Preparation code not specific to JMockit, if any.

       // Unit under test is exercised.

       new Verifications() {{ // a "verification block"
          // One or more invocations to mocked types, causing expectations to be verified.
          // Invocations to non-mocked types are also allowed anywhere inside this block
          // (though not recommended).
       }};

       // Additional verification code, if any, either here or before the verification blo
    }

    @Test
    public void testWithBothRecordAndVerify(mock parameters)
    {
       // Preparation code not specific to JMockit, if any.

       new Expectations() {{
          // One or more invocations to mocked types, causing expectations to be recorded.
       }};

       // Unit under test is exercised.

       new VerificationsInOrder() {{ // an ordered verification block
          // One or more invocations to mocked types, causing expectations to be verified
          // in the specified order.
       }};

       // Additional verification code, if any, either here or before the verification blo
    }
 }
```

还包括一些与以上模板不同之处，但是实质上就是期望属于录制阶段在测试代码被执行之前，验证代码块属于验证阶段。一个测试可以包含任意多个期望代码块，也可以没有。针对验证代码块也是如此。

实际上匿名内部类被用来是允许代码块可以被分割，充分使用当代 Java IDEs的代码折叠功能。如下的图片显示了在 IntelliJ IDEA的样子。

```
@Test
public void createOrder() throws Exception
{
   // Test data:
   List<OrderItem> expectedItems = asList(
      new OrderItem("393439493", "Core Java 5 6ed", 2, new BigDecimal("45.00")),
      new OrderItem("04940458", "JUnit Recipes", 1, new BigDecimal("49.95")));
   final List<OrderItem> actualItems = new ArrayList<>();

   new Expectations() {...};
   new Expectations() {{
      order.getItems(); result = actualItems;
   }};
   final Order created = new OrderFactory().createOrder(customerId, expectedItems);

   // Verify that expected invocations (excluding the ones inside a previous Expectations block)
   // actually occurred:
   new Verifications() {...};

   // Conventional JUnit state-based verifications:
   assertNotNull(created);
   assertEquals(expectedItems, actualItems);
}
```

普通和严格的期望

期望被录制在 "new Expectations() {...}" 块中是通常的方式。 这意味着我们所期待的调用至少在重放阶段发生一次; 他们可能发生不只一次, 虽然可以和其他的录用期望的相对顺序不同; 此外, 不匹配任何录制期望是可以以任何个数和任意顺序。如果没有任何调用匹配上给定的期望，一个调用缺失的错误将会在测试结束之后抛出, 导致其失败 (这是仅有的缺省的行为，虽然这可以被重写)。

API 也支持严格期望的概念: 指的是当我们录制的时候只允许在重放的调用必须准确的匹配录制(当需要的时候，可以通过显式的定义允许), 无论是匹配的调用次数(缺省是只有一次)还是他们的发生顺序。在重放阶段的调用如果没有按照严格录制阶段中所期待的那样将会失败，迅速导致一个 "unexpected invocation" 错误, 结果测试失败。 这是通过使用 StrictExpectations 子类来实现的。

注意在严格期望的情况下，所有在重放阶段的调用匹配录制的期望都是隐式验证的。任何余下的调用不匹配一个期望都将被认为是不可预期的，导致测试失败。如果任何录制的严格期望确实了，测试也是失败的，比如，在重放节点没有匹配任何调用。

我们可以混合不同级别的严格条件在相同的测试中通过使用多个期望块，一些是普通的 (使用 Expectations), 其他是严格的 (使用 StrictExpectations)。 尽管通常来说，一个给定的模拟属性或模拟参数将会在一种类型的期望出现。

大多数的测试都会简单的使用普通的期望。使用严格的期望更可能是一个个人喜好问题。

严格和非严格的模拟

注意我们不需要定义给定的模拟类型/实例是严格或不是。相反的，一个给定的模拟属性/参数是否严格是通过它在测试中如何使用来决定的。一旦第一个严格期望被录制以 "new StrictExpectations() {...}" 代码块的形式, 相关的模拟类型/实例将被认为是在整个测试中严格的;否则就不是严格的。

记录所期望的结果

对于一个包含返回类型的方法，一个返回值可以通过给result属性赋值的方式录制。当在重放阶段中方法被调用的，所定义的返回值将会被返回给调用者。对 result的赋值应该正确的出现在期望代码块中的给定录制调用之后。

如果测试需要一个异常或错误被抛出在方法执行之后，result的属性还是可以被使用: 简单的赋值所需要的抛出的异常实例给它。注意录制的要被抛出的异常/错误需要符合模拟方法（任何返回类型），也包括模拟的构造方法。

多个连续的结果（返回值或抛出的异常）可以在同一个期望中被录制，通过简单的多次设置值在一行中。多个返回值和或者异常和错误的录制可以在同一个期望中自由的混合。在一个简单的对返回函数的调用中，可以给定期望中录制多个连续返回结果的例子。同样的，一个简单的对结果属性的赋值也能达到这个效果，如果赋值的是一个列表或者是数组包含连续的值。

以下的示例测试录制针对一个模拟依赖 DependencyAbc 类的方法的多个类型结果，当他们在 UnitUnderTest 类的测试调用中将会被使用。让我们看看在测试下的这个类的实现:

```
public class UnitUnderTest
{
(1)private final DependencyAbc abc = new DependencyAbc();

   public void doSomething()
   {
(2)   int n = abc.intReturningMethod();

      for (int i = 0; i < n; i++) {
         String s;

         try {
(3)         s = abc.stringReturningMethod();
         }
         catch (SomeCheckedException e) {
            // somehow handle the exception
         }

         // do some other stuff
      }
   }
}
```

一个针对 doSomething() 方法的可能测试将抛出 SomeCheckedException，在一个连续的随意值的迭代中。假设我们想要（无论什么原因）录制一个完整的期望集合来与其他两个类交互，我们可能编写一下的测试。(通常，定义所有需要模拟方法的调用和特殊的模拟构造方法在给定的测试中是不需要或重要的。我们将会之后举个例子。)

```
@Test
public void doSomethingHandlesSomeCheckedException(@Mocked final DependencyAbc abc) throw
{
   new Expectations() {{
(1)   new DependencyAbc();

(2)   abc.intReturningMethod(); result = 3;

(3)   abc.stringReturningMethod();
      returns("str1", "str2");
      result = new SomeCheckedException();
   }};

   new UnitUnderTest().doSomething();
}
```

这个测试录制了三个不同的期望。第一个通过调用 DependencyAbc() 构造方法来使用的, 仅仅为了表示在测试代码下如何通过无参数的构造方法进行初始化; 这个调用不需要定义结果, 除了偶尔需要抛出错误或异常 (构造方法不包含返回类型，因此录制从它们中返回值是没有意

义的)。第二个期望定义了 调用 intReturningMethod() 方法将会返回3。 第三个定义了一些列三个连续的结果针对 stringReturningMethod() 方法, 这里最后一个结果是所需异常的实例, 允许测试来实现这个目标 (注意这个测试只会在异常不被抛出才通过)。

匹配特定实例的调用

之前，我们已经解释了如何在模拟实例上录制期望，例如 "abc.someMethod();" 将会准备的匹配 对 DependencyAbc#someMethod()方法的调用在任何依赖 DependencyAbc 类的实例上。在大多数的例子中，测试代码使用了一个独立的依赖实例，因此无论是否模拟实例可以在测试下被传递或创建是无关紧要且安全忽略的。 但是我们需要验证对指定实例的调用，在测试代码下被使用的一些实例中？同样的, 是否一个或者是一些模拟对象的实例将被真的被模拟, 相同类的其他实例没有被模拟?(第二个例子将标准java类库或者是从第三方的库中类进行模拟是很普遍的。) JMockit 提供了模拟注释 , @Injectable, 只会模拟被模拟类型的一个实例, 而不会影响其他的实例。 此外，它还提供了一些方法来限制期望的匹配针对特定的 @Mocked 实例, 而不是模拟所有被模拟类的实例。

注入模拟实例

假设我们需要测试和多个给定类实例一起工作的代码，其中有我们需要模拟的部分。如果一个实例需要被模拟进行传递或者是注入到测试代码中，我们需要申明一个 @Injectable 模拟属性或者是模拟参数。 由JMockit建立的@Injectable 实例将是一个排他的模拟实例；任何其他相同模拟类型实例除非从一个分隔开的模拟属性/参数中获取，将仍然是一个正常的，非模拟实例。

同样需要注意，由于注入的模拟实例只能影响到对应那个实例的行为，因此静态方法和构造方法同样没有被模拟。毕竟一个静态方法是和改类的任何实例无关的，而构造方法只与新建的(和因此不同的) 实例相关.

例如，我们有如下的类需要被测试。

```
public final class ConcatenatingInputStream extends InputStream
{
   private final Queue<InputStream> sequentialInputs;
   private InputStream currentInput;

   public ConcatenatingInputStream(InputStream... sequentialInputs)
   {
      this.sequentialInputs = new LinkedList<InputStream>(Arrays.asList(sequentialInputs)
      currentInput = this.sequentialInputs.poll();
   }

   @Override
   public int read() throws IOException
   {
      if (currentInput == null) return -1;

      int nextByte = currentInput.read();

      if (nextByte >= 0) {
         return nextByte;
      }

      currentInput = sequentialInputs.poll();
      return read();
   }
}
```

这个类可以很容易的被测试，不需要通过使用 ByteArrayInputStream 对象进行模拟输入，但是我们需要保证在构造方法中 InputStream#read()方法在每一个输入流中被调用。如下的测试代码可以实现。

```
@Test
public void concatenateInputStreams(
   @Injectable final InputStream input1, @Injectable final InputStream input2)
   throws Exception
{
   new Expectations() {{
      input1.read(); returns(1, 2, -1);
      input2.read(); returns(3, -1);
   }};

   InputStream concatenatedInput = new ConcatenatingInputStream(input1, input2);
   byte[] buf = new byte[3];
   concatenatedInput.read(buf);

   assertArrayEquals(new byte[] {1, 2, 3}, buf);
}
```

注意这里明确的使用 @Injectable 是很有必要的，因为在测试中的类继承了模拟类，对 ConcatenatingInputStream类的方法调用确实定义在 InputStream 基类中。 如果 InputStream 被普通的模拟， read(byte[]) 方法将会被模拟, 无论在任何调用它的实例中。

onInstance(m) 约束

当使用 @Mocked 或 @Capturing (和不是 @Injectable 在相同的模拟属性/参数中), 我们可以匹配重放对期望中录制的调用在特定的模拟实例上。比如，我们使用 onInstance(mockObject) 方法当录制期望的时候, 正如下面的例子所示。

```
@Test
public void matchOnMockInstance(@Mocked final Collaborator mock)
{
   new Expectations() {{
      onInstance(mock).getValue(); result = 12;
   }};

   // Exercise code under test with mocked instance passed from the test:
   int result = mock.getValue();
   assertEquals(12, result);

   // If another instance is created inside code under test...
   Collaborator another = new Collaborator();

   // ...we won't get the recorded result, but the default one:
   assertEquals(0, another.getValue());
}
```

以上的测试只会在测试代码(这里简短的嵌入到测试方法中) 调用 getValue() 在和录制调用时候相同的实例上才会通过。 这对于测试代码中多两个或者是更多的相同类型下的实例进行调用和测试想要验证每一个调用都发生在恰当的实例上的时候是很有用的。

为了避免在测试中使用多种不同的方法在多个相同类型的实例的每一个期望中使用 onInstance(m)方法, JMockit 自动引用 "onInstance" 的匹配基于模拟类型集合的范围内。 特别是，当有两个或者是更多的模拟属性/参数在一个给定测试范围中是相同类型情况下，对他们实例调用实例方法，将总是匹配到期望录制中相同的实例中。 因此，在这种普通的情况下，是不需要显示的使用 onInstance(m) 方法。

使用给定构造方法新建实例

特殊的 future 实例将会之后被测试代码创建 , JMockit 提供了一系列的机制来匹配在它们上面的调用。所有的机制都需要在特定的模拟类上的构造方法调用中录制期望(一个新的表达式)。

第一个机制仅仅使用从录制的构造期望中获取新实例，当在实例方法进行期望的时候。让我来看例子。

```
@Test
public void newCollaboratorsWithDifferentBehaviors(@Mocked Collaborator anyCollaborator)
{
   // Record different behaviors for each set of instances:
   new Expectations() {{
      // One set, instances created with "a value":
      Collaborator col1 = new Collaborator("a value");
      col1.doSomething(anyInt); result = 123;

      // Another set, instances created with "another value":
      Collaborator col2 = new Collaborator("another value");
      col2.doSomething(anyInt); result = new InvalidStateException();
   }};

   // Code under test:
   new Collaborator("a value").doSomething(5); // will return 123
   ...
   new Collaborator("another value").doSomething(0); // will throw the exception
   ...
}
```

在上面的测试中，我们申明了一个独立的模拟属性或模拟参数在所需的类上，使用 @Mocked 注释。 然而模拟的属性/参数却没有在录制期望的时候使用; 而是使用初始化的实例录制进一步在实例方法上的期望。 匹配上的构造方法调用建立future 实例将映射到对应的录制实例。同样的，不需要一对一映射，而是可以多对一的映射，从潜在在多个 future 实例对应一个录制期望中使用的单个实例。

第二个机制中让我们能够录制一个替代的实例针对那些匹配了录制中的构造方法的实例。通过这种机制，我们可以重写以上的例子。

```
@Test
public void newCollaboratorsWithDifferentBehaviors(
   @Mocked final Collaborator col1, @Mocked final Collaborator col2)
{
   new Expectations() {{
      // Map separate sets of future instances to separate mock parameters:
      new Collaborator("a value"); result = col1;
      new Collaborator("another value"); result = col2;

      // Record different behaviors for each set of instances:
      col1.doSomething(anyInt); result = 123;
      col2.doSomething(anyInt); result = new InvalidStateException();
   }};

   // Code under test:
   new Collaborator("a value").doSomething(5); // will return 123
   ...
   new Collaborator("another value").doSomething(0); // will throw the exception
   ...
}
```

两个版本的测试是等价的。第二个对实际的（非模拟的）用来替换的实例也是允许的，当和部分模拟组合使用的时候。

对参数值得灵活匹配

在录制和验证阶段，一个对模拟方法或构造方法的调用被定义在期望中。如果方法/构造方法包含一个或者是多个参数，然后录制/验证期望像某些东西(1, "s", true); 只会在重放阶段匹配相等的参数值得调用。对于那些普通的对象的参数 (非原始或是数组), equals(Object) 方法将被用来相等检测。 针对数组类型的参数, 相等检测扩充到每一个元素中; 因此，两个不同的数组实例在相同的维度中包含相同的长度和对应的元素也是相等的，则被认为相等。

在给定的测试中，我们可能不知道其中的参数值到底是什么，或者他们不是需要测试所关注的。因此，可以允许一个录制或验证调用可以匹配一系列重放的调用使用不同的参数值，我们可以灵活的定义参数匹配而不是实际参数值。这可以通过使用 anyXyz 属性或者是 withXyz(...) 方法来进行。 "any" 属性和"with" 方法都在 mockit.Invocations中定义, 该类是所有在测试中被使用的 expectation/verification 类的基类; 因此，他们既可以在期望中也可以在验证代码块中使用。

使用any 属性匹配参数

大多数的普通参数值约束匹配也可以为最不严格的: 匹配给定参数为任何职的调用(当然了需要有恰当的参数类型)。 针对这种例子，我们需要包含整个特殊值参数来匹配属性, 针对每一个原始类型只 (和对应的包装类型), 一个针对字符串, 和一个所有的对象类型。下面的测试展示

了一些例子。

```
@Test
public void someTestMethod(@Mocked final DependencyAbc abc)
{
   final DataItem item = new DataItem(...);

   new Expectations() {{
      // Will match "voidMethod(String, List)" invocations where the first argument is
      // any string and the second any list.
      abc.voidMethod(anyString, (List<?>) any);
   }};

   new UnitUnderTest().doSomething(item);

   new Verifications() {{
      // Matches invocations to the specified method with any value of type long or Long.
      abc.anotherVoidMethod(anyLong);
   }};
}
```

无论如何，使用 "any" 属性必须出现在调用语句中确切的参数位置。你也可以使用普通的参数值针对其他的参数在相同的调用中。更具体的说明,可以查看 API 文档。

使用with 方法进行参数匹配

当录制或者验证一个期望的时候，withXyz(...) 方法可以在任何调用参数值传递的地方进行使用。 它们可以自由的和普通参数值传递混合使用(使用文本值 , 本地参数, 等等。)。唯一需要的是这些调用必须出现在录制/验证调用语句中出现，而不是在他们之前。比如首次对 withNotEqual(val)调用的参数值给一个本地变量，然后在调用语句中使用参数是不可能的。一个测试例子使用 "with" 方法如下所示。

```
@Test
public void someTestMethod(@Mocked final DependencyAbc abc)
{
   final DataItem item = new DataItem(...);

   new Expectations() {{
      // Will match "voidMethod(String, List)" invocations with the first argument
      // equal to "str" and the second not null.
      abc.voidMethod("str", (List<?>) withNotNull());

      // Will match invocations to DependencyAbc#stringReturningMethod(DataItem, String)
      // with the first argument pointing to "item" and the second one containing "xyz".
      abc.stringReturningMethod(withSameInstance(item), withSubstring("xyz"));
   }};

   new UnitUnderTest().doSomething(item);

   new Verifications() {{
      // Matches invocations to the specified method with any long-valued argument.
      abc.anotherVoidMethod(withAny(1L));
   }};
}
```

好包含更多的 "with"方法不仅仅如上所示。可以通过查看 API 文档了解更多详细信息。

除了包含一些预定义在API中的约束匹配，JMockit 允许用户提供自定义的约束, 通过使用 with(Delegate) 和d withArgThat(Matcher) 方法。

使用 **null** 值匹配所有任何其他对象的引用、

当在一个期望中使用至少一个参数匹配方法或属性，我们可以使用快捷的方式来定义任何对象引用可以被接受的 (针对引用的参数类型)。简单的传递 null 值而不是 withAny(x) 或任何的参数匹配。 特别之处在于这允许不需要将值强制转成对应所申明的参数类型。然而, 牢牢记住这个特征只使用于至少一个明确参数匹配被使用于期望中(无论是 "with" 方法或一个 "any" 属性) 。当传递的调用中不包含任何匹配, null 值将只会匹配 null 引用。在之前的测试中，因此我们可以重写为:

```
@Test
public void someTestMethod(@Mocked final DependencyAbc abc)
{
   ...
   new Expectations() {{
      abc.voidMethod(anyString, null);
   }};
   ...
}
```

当定义对一个给定参数接受null引用的验证时，withNull() 匹配可以被使用。

通过 **varargs** 参数匹配参数的传入

偶尔我们需要处理可变长参数的方法或构造函数的期望。可以通过传递普通的值作为可变参数的参数值, 也可以通过使用"with"/"any" 匹配针对这些值。然而，却不可以在相同的期望中组合两种类型的值传递, 当目标是一个可变参数时。我们不是用普通的值就是用那些参数匹配器中的值。

当我们需要匹配可变长参数接受任意数量的值的调用时(包含 0), 我们可以定义期望使用 "(Object[]) any" 约束 final 的可变长度参数。

可能最好的方式用来明白确切的可变长参数匹配就是查看实际的测试代码 (由于没有涉及确切的 API)。测试类实际上展示了所有可能。

定义调用的次数约束

到目前为止，我们已经看到了除了可以关联方法或构造方法，一个期望可以包含调用结果和参数值匹配。在给定的测试代码中可以调用相同的方法或构造方法多次使用不同的或相同的参数，我们有时需要一个方法对所有的这些分隔的调用进行计数。

对一个允许的匹配上的给定期望中的调用次数可以通过调用次数约束进行定义。模拟 API提供了三种不同类型的属性来定义: 次数, 最少次数, 和最大次数。这些属性不仅仅可以用在录制阶段还可以被用在验证期望中。在任何中的一个例子中, 和期望相关的方法或构造方法将被约束在一个接受次数范围的区间内。 任何调用必须各自的少于或对于给定所期望的最少或上限, 和测试执行将会自动失败。让我们来查看一些例子。

```
@Test
public void someTestMethod(@Mocked final DependencyAbc abc)
{
   new Expectations() {{
      // By default, at least one invocation is expected, i.e. "minTimes = 1":
      new DependencyAbc();

      // At least two invocations are expected:
      abc.voidMethod(); minTimes = 2;

      // 1 to 5 invocations are expected:
      abc.stringReturningMethod(); minTimes = 1; maxTimes = 5;
   }};

   new UnitUnderTest().doSomething();
}

@Test
public void someOtherTestMethod(@Mocked final DependencyAbc abc)
{
   new UnitUnderTest().doSomething();

   new Verifications() {{
      // Verifies that zero or one invocations occurred, with the specified argument valu
      abc.anotherVoidMethod(3); maxTimes = 1;

      // Verifies the occurrence of at least one invocation with the specified arguments:
      DependencyAbc.someStaticMethod("test", false); // "minTimes = 1" is implied
   }};
}
```

不像result属性，每一组三个属性都可以在一个期望中定义最多一次。任何非负的整数值可以使用到调用次数约束中。 如果times = 0 或 maxTimes = 0 被定义, 在重放节点的首次匹配调用中将导致测试失败(如果 任何)。

显式验证

除了可以定义调用次数的约束在录制期望阶段，我们还可以显式的在验证代码块中验证匹配的调用，在测试代码执行之后。针对于普通的期望是可用，但是对于严格的期望是不行的，因为他们总是隐式的验证; 在显式的验证块中进行重新校验是没有意义的。

在 "new Verifications() {...}" 块中我们可以使用相同的 API，和在 "new Expectations() {...}" 块中一样有效的, 使用异常方法或属性用来录制返回值和抛出异常/错误。也就是说, 我们可以自由的使用 anyXyz 属性, withXyz(...) 参数匹配方法, 和次数, 最小次数, 和醉倒调用次数的约束属性。 如果是一个测试例子。

```
@Test
public void verifyInvocationsExplicitlyAtEndOfTest(@Mocked final Dependency mock)
{
   // Nothing recorded here, though it could be.

   // Inside tested code:
   Dependency dependency = new Dependency();
   dependency.doSomething(123, true, "abc-xyz");

   // Verifies that Dependency#doSomething(int, boolean, String) was called at least once
   // with arguments that obey the specified constraints:
   new Verifications() {{ mock.doSomething(anyInt, true, withPrefix("abc")); }};
}
```

注意，缺省的一个验证检测在重放至少包含一次的匹配调用。当我们需要验证确切的调用次数时(包括 1), times = n 约束必须被定义。

验证调用从来没有发生

为了在一个验证块中做这些，添加 "times = 0" 赋值在那些被期望在重放阶段不被使用的调用之后。如果一个或更多的匹配调用发生了，测试将失败。

验证顺序

普通的使用Verifications类建立的验证块是无序的。实际的相对顺序在 aMethod() 和 anotherMethod() 方法在重放中被调用时不被验证, 但是每一个方法必须至少执行一次。如果你想要验证相对的调用顺序, "new VerificationsInOrder() {...}" 块不需要被使用。在这个块中，简单的编写对一个或者更多模拟类型的调用按照他们所期待执行顺序。

```
@Test
public void verifyingExpectationsInOrder(@Mocked final DependencyAbc abc)
{
   // Somewhere inside the tested code:
   abc.aMethod();
   abc.doSomething("blah", 123);
   abc.anotherMethod(5);
   ...

   new VerificationsInOrder() {{
      // The order of these invocations must be the same as the order
      // of occurrence during replay of the matching invocations.
      abc.aMethod();
      abc.anotherMethod(anyInt);
   }};
}
```

注意对 abc.doSomething(...) 调用在测试中没有验证, 因此它可以发送在任何时候(或根本不发送)。

部分顺序的验证

加入你想验证一个特别的方法 (或构造方法) 在其他调用之前/之后, 但是你不关系其他方法调用的顺序。 在顺序验证的块中，这可以通过简单的调用 unverifiedInvocations() 方法在恰当的位置。下面的例子展现了这个。

```
@Mocked DependencyAbc abc;
@Mocked AnotherDependency xyz;

@Test
public void verifyingTheOrderOfSomeExpectationsRelativeToAllOthers()
{
   new UnitUnderTest().doSomething();

   new VerificationsInOrder() {{
      abc.methodThatNeedsToExecuteFirst();
      unverifiedInvocations(); // Invocations not verified must come here...
      xyz.method1();
      abc.method2();
      unverifiedInvocations(); // ... and/or here.
      xyz.methodThatNeedsToExecuteLast();
   }};
}
```

上面的例子确实有些复杂，正如它验证一些事情: a) 一个方法必须在其他之前; b) 一个方法必须在其他之后; 和 c) AnotherDependency#method1() 必须只在 DependencyAbc#method2() 调用之前。 在大多数的测试中，我们将可能只做这些不同种类顺序相关验证中的一个。 但是使得包含所有种类的复杂验证是相当容易的。

其他的上面例子没有包含的是我们需要验证确定的调用以一个给定的相对顺序, 然而需要验证其他的调用 (以任何顺序、)。针对这种情况，我们需要编写两个独立的验证块，如下面所示的 (这里模拟的是测试类中的属性)。

```
@Test
public void verifyFirstAndLastCallsWithOthersInBetweenInAnyOrder()
{
   // Invocations that occur while exercising the code under test:
   mock.prepare();
   mock.setSomethingElse("anotherValue");
   mock.setSomething(123);
   mock.notifyBeforeSave();
   mock.save();

   new VerificationsInOrder() {{
      mock.prepare(); // first expected call
      unverifiedInvocations(); // others at this point
      mock.notifyBeforeSave(); // just before last
      mock.save(); times = 1; // last expected call
   }};

   // Unordered verification of the invocations previously left unverified.
   // Could be ordered, but then it would be simpler to just include these invocations
   // in the previous block, at the place where "unverifiedInvocations()" is called.
   new Verifications() {{
      mock.setSomething(123);
      mock.setSomethingElse(anyString);
   }};
}
```

通常，当一个测试包含多个验证块，它们的相对执行顺序是重要的。在之前的测试中，如果一个无序的块出现在它之前将会不会留下 "unverified invocations" 进行之后的对unverifiedInvocations()的匹配; 这个测试也会通过 (加入它原来是能通过的) 因为它不需要一个非验证的调用发生在调用位置之前, 但是它不会验证在首个和最后一个其他调用之间的未排序的调用组。

全验证

有时可能包含在测试中的所有的参与的模拟类型中的调用被验证。这可以通过使用严格的期望来自动实现，由于任何未被期待的调用都将导致测试失败。然而当在正常的期望中可以显式的验证，通过使用 "new FullVerifications() {...}" 块来确保没有调用没被验证。

```
@Test
public void verifyAllInvocations(@Mocked final Dependency mock)
{
   // Code under test included here for easy reference:
   mock.setSomething(123);
   mock.setSomethingElse("anotherValue");
   mock.setSomething(45);
   mock.save();

   new FullVerifications() {{
      // Verifications here are unordered, so the following invocations could be in any o
      mock.setSomething(anyInt); // verifies two actual invocations
      mock.setSomethingElse(anyString);
      mock.save(); // if this verification (or any other above) is removed the test will
   }};
}
```

注意当一个最低限制 (一个最少的调用次数约束) 被在期望中定义时, 这个约束将会隐式的被在测试结束之后验证。因此显式的验证这个期望在全验证的块中是不需要的。

按顺序进行全验证

我们现在已经知道如何使用Verifications 不按顺序验证，使用 VerificationsInOrder 进行按顺序的验证， 和使用 FullVerifications进行全验证。但是如何进行全顺序验证？也是很容易的:

```
@Test
public void verifyAllInvocationsInOrder(@Mocked final Dependency mock)
{
   // Code under test included here for easy reference:
   mock.setSomething(123);
   mock.setSomethingElse("anotherValue");
   mock.setSomething(45);
   mock.save();

   new FullVerificationsInOrder() {{
      mock.setSomething(anyInt);
      mock.setSomethingElse(anyString);
      mock.setSomething(anyInt);
      mock.save();
   }};
}
```

注意尽管这里没有明显的语法上的区别。在 以上的verifyAllInvocations 测试中, 我们能够匹配两个独立的 mock.setSomething(...) 使用使用独立的在验证块中的调用。在 verifyAllInvocationsInOrder 测试中, 然而我们需要在块中编写两个独立的调用方法, 按照他们各自独立的调用顺序。

限制模拟类型集合进行全验证

缺省的，在一个给定测试中的所有的对模拟实例/类型的调用必须显式的被验证当使用 "new FullVerifications() {}" 或 "new FullVerificationsInOrder() {}" 块的时候。现在，假如我们有一个测试包含了两个模拟类型 (或更多) 但是我们只想对他们中的一个进行调用全验证 (或者对任何模拟类型的部分不只有两个)? 可以使用 FullVerifications(mockedTypesAndInstancesToVerify) 构造方法, 当只有一个给定的模拟实例和模拟类型需要被使用(例如, 类对象/名称)。如下测试提供了一个例子。

```
@Test
public void verifyAllInvocationsToOnlyOneOfTwoMockedTypes(
   @Mocked final Dependency mock1, @Mocked AnotherDependency mock2)
{
   // Inside code under test:
   mock1.prepare();
   mock1.setSomething(123);
   mock2.doSomething();
   mock1.editABunchMoreStuff();
   mock1.save();

   new FullVerifications(mock1) {{
      mock1.prepare();
      mock1.setSomething(anyInt);
      mock1.editABunchMoreStuff();
      mock1.save(); times = 1;
   }};
}
```

在如上的测试中，mock2.doSomething() 调用从来没有被验证。

为了只是对一个独立的模拟类中的方法/构造方法进行验证, 传递类名到 FullVerifications(...) 或 FullVerificationsInOrder(...) 构造方法中。 例如, 新的 FullVerificationsInOrder(AnotherDependency.class) { ... } 块只会使得所有对模拟的 AnotherDependency 类的调用被验证。

验证没有调用发生

为了验证在测试中没有调用发生在模拟的类型/实例上，可以添加一个空的全验证块到其中。通常的，注意所有的通过一个 times/minTimes约束的录制期望都将被隐式的验证，因此不管验证块中的全验证; 在这个例子中空的验证块会验证没有其他的任何调用发生。此外，在相同的测试中之前的验证块一斤验证了期望，他们也将被全验证块所忽视。

如果测试使用了两个或者更多的模拟类型/实例,你需要验证在他们之中没有调用发生，定义所需的模拟类型和/或实例到构造方法中在空的验证块中。一个测试用例如下。

```
@Test
public void verifyNoInvocationsOnOneOfTwoMockedDependenciesBeyondThoseRecordedAsExpected(
   @Mocked final Dependency mock1, @Mocked final AnotherDependency mock2)
{
   new Expectations() {{
      // These two are recorded as expected:
      mock1.setSomething(anyInt);
      mock2.doSomething(); times = 1;
   }};

   // Inside code under test:
   mock1.prepare();
   mock1.setSomething(1);
   mock1.setSomething(2);
   mock1.save();
   mock2.doSomething();

   // Will verify that no invocations other than to "doSomething()" occurred on mock2:
   new FullVerifications(mock2) {};
}
```

验证未定义的调用不能发生

一个全验证块（排序或者没有）允许我们验证确定的方法和/或构造方法从没有被调用，不需要在录制或验证他们的时候使用对应 times = 0 的赋值。 如下例子所示。

```
@Test
public void readOnlyOperation(@Mocked final Dependency mock)
{
   new Expectations() {{
      mock.getData(); result = "test data";
   }};

   // Code under test:
   String data = mock.getData();
   // mock.save() should not be called here
   ...

   new FullVerifications() {{
      mock.getData(); minTimes = 0; // calls to getData() are allowed, others are not
   }};
}
```

如果对依赖的类的任何方法（或构造方法）的调用在重放阶段，除非这个被显式的在验证块中验证 (Dependency#getData() 在这个例子中),以上的测试将会失败。从另一方面来说, 在这个例子中更好的使用严格的期望, 完全不需要使用任何的验证块。

捕捉验证时的调用参数

调用参数可以在之后的验证中被捕捉通过一个系列的特殊 "withCapture(...)" 方法。有三种的类型, 每一个都包含他们特定的捕捉方法 : 1) 在单个调用中验证的参数是否被传入到模拟方法中: T withCapture(); 2) 验证参数是否在多个调用中被传入: T withCapture(List); 和 3) 验证参数被传入到模拟构造方法中: List withCapture(T).

从单个调用中捕捉参数

为了从单个对模拟方法或构造方法的调用捕捉参数，我们使用 "withCapture()", 正如下面的例子所展示的。

```
@Test
public void capturingArgumentsFromSingleInvocation(@Mocked final Collaborator mock)
{
   // Inside tested code:
   new Collaborator().doSomething(0.5, new int[2], "test");

   new Verifications() {{
      double d;
      String s;
      mock.doSomething(d = withCapture(), null, s = withCapture());

      assertTrue(d > 0.0);
      assertTrue(s.length() > 1);
   }};
}
```

withCapture() 方法可以被使用在验证块中。通常我们使用它在一个单独的调用匹配是否发生; 如果有多个这个调用发生, 最后一个调用将会覆盖之前所捕捉到的值。这在处理复杂的参数类型(比如 JPA @Entity)有用, 可能包含一些项的值需要被验证。

从多个调用中捕捉参数

如果多个对模拟方法或构造方法得调用需要期待，我们需要捕捉他们中的所有值，可以使用 withCapture(List) 方法, 如下的例子所示。

```
@Test
public void capturingArgumentsFromMultipleInvocations(@Mocked final Collaborator mock)
{
   mock.doSomething(dataObject1);
   mock.doSomething(dataObject2);

   new Verifications() {{
      List<DataObject> dataObjects = new ArrayList<>();
      mock.doSomething(withCapture(dataObjects));

      assertEquals(2, dataObjects.size());
      DataObject data1 = dataObjects.get(0);
      DataObject data2 = dataObjects.get(1);
      // Perform arbitrary assertions on data1 and data2.
   }};
}
```

不同于 withCapture(), withCapture(List) 重载可以在期望录制块中。 blocks.

捕捉新实例

最后，我们捕捉一个模拟类型的新创建实例在测试阶段。

```
@Test
public void capturingNewInstances(@Mocked Person mockedPerson)
{
   // From the code under test:
   dao.create(new Person("Paul", 10));
   dao.create(new Person("Mary", 15));
   dao.create(new Person("Joe", 20));

   new Verifications() {{
      // Captures the new instances created with a specific constructor.
      List<Person> personsInstantiated = withCapture(new Person(anyString, anyInt));

      // Now captures the instances of the same type passed to a method.
      List<Person> personsCreated = new ArrayList<>();
      dao.create(withCapture(personsCreated));

      // Finally, verifies both lists are the same.
      assertEquals(personsInstantiated, personsCreated);
   }};
}
```

代理：定义自己的结果

我们已经知道如何通过复值给result或使用 returns(...)方法在录制调用结果。我们也知道如何使用 withXyz(...)的一组方法和不同的 anyXyz 属性来进行灵活的调用参数匹配。但是假如一个测试需要基于它在重放阶段所得到的参数设定录制阶段的值？ 我们可以使用代理实例,如下所示。

```
@Test
public void delegatingInvocationsToACustomDelegate(@Mocked final DependencyAbc anyAbc)
{
   new Expectations() {{
      anyAbc.intReturningMethod(anyInt, null);
      result = new Delegate() {
         int aDelegateMethod(int i, String s)
         {
            return i == 1 ? i : s.length();
         }
      };
   }};

   // Calls to "intReturningMethod(int, String)" will execute the delegate method above.
   new UnitUnderTest().doSomething();
}
```

Delegate 接口是空的, 简单的通过代理 "delegate" 方法在赋值对象上来告诉 JMockit 在重放实际阶段上如何操作。 这个方法可以包含任何名称, 在代理对象上声明它为非私有方法。作为代理方法的参数，他们既可以是匹配录制方法的参数或他们是空的。在任何的例子中，代理方法允许包含附加的类型为Invocation的参数作为它的第一个参数。(在重放阶段所接收的 Invocation 对象将会提供调用实例和真是调用参数的方法，同样包含其他的能力。) 一个代理方法返回的类型不需要和录制方法中的一样，, 尽管它需要兼容避免之后造成 ClassCastException 异常。

构造方法可以通过代理方法进行处理。如下的例子演示了构造方法调用是如何通过一个代理的方法来有条件的情况下抛出异常。

```
@Test
public void delegatingConstructorInvocations(@Mocked Collaborator anyCollaboratorInstance
{
   new Expectations() {{
      new Collaborator(anyInt);
      result = new Delegate() {
         void delegate(int i) { if (i < 1) throw new IllegalArgumentException(); }
      };
   }};

   // The first instantiation using "Collaborator(int)" will execute the delegate above.
   new Collaborator(4);
}
```

级联模拟

当使用复杂的 APIs 来处理多个分布式的不同对象上的功能时，很少看到使用链式调用以 obj1.getObj2(...).getYetAnotherObj().doSomething(...)的格式。 在这种情况下，可能需要模拟所有的对象/类在链中, 从 obj1开始的。

所有的三个模拟注释都提供这个功能。 如下的例子显示了一个级别的例子, 使用 java.net 和 java.nio APIs。

```
@Test
public void recordAndVerifyExpectationsOnCascadedMocks(
   @Mocked Socket anySocket, // will match any new Socket object created during the test
   @Mocked final SocketChannel cascadedChannel // will match cascaded instances
) throws Exception
{
   new Expectations() {{
      // Calls to Socket#getChannel() will automatically return a cascaded SocketChannel;
      // such an instance will be the same as the second mock parameter, allowing us to
      // use it for expectations that will match all cascaded channel instances:
      cascadedChannel.isConnected(); result = false;
   }};

   // Inside production code:
   Socket sk = new Socket(); // mocked as "anySocket"
   SocketChannel ch = sk.getChannel(); // mocked as "cascadedChannel"

   if (!ch.isConnected()) {
      SocketAddress sa = new InetSocketAddress("remoteHost", 123);
      ch.connect(sa);
   }

   InetAddress adr1 = sk.getInetAddress();  // returns a newly created InetAddress instan
   InetAddress adr2 = sk.getLocalAddress(); // returns another new instance
   ...

   // Back in test code:
   new Verifications() {{ cascadedChannel.connect((SocketAddress) withNotNull()); }};
}
```

在如上的测试，对模拟Socket类上的合适方法的调用都将返回一个级联模拟的对象，在测试阶段中。激烈模拟将会允许更进一步的级联，因此一个空引用将不会从返回对象的引用的方法中获取(除了一个不合适的返回类型对象或字符串返回null，或集合类型中返回了一个非模拟的空类型。)。

除了一个可用的在模拟属性/参数（例如上面的 cascadedChannel）中的模拟实例，一个新的级联实例将会在首次的对每一个模拟方法的调用中创建。在上面的例子中，对相同的 InetAddress 返回类型的不同方法将会创建和返回不同的级联实例; 虽然，相同的方法将会返回相同的级联实例。

新的级联实例将会通过使用 @Injectable 语法创建，因此不会影响在测试中相同类型的其他实例。

最后，如果必要的话，值得注意的是级联实例可以被非模拟类型，被不同的模式实例，或一个无返回的替换; 为此，使用所需要返回的实例对result属性赋值的期望录制或使用null如果没有这个实例需要。

级联静态工厂方法

在一个模拟类中包含了静态工厂方法的时候级联是很有用的。在如下的测试例子中，让我们看看如何模拟 javax.faces.context.FacesContext 类从 JSF (Java EE)中。

```
@Test
public void postErrorMessageToUIForInvalidInputFields(@Mocked final FacesContext jsf)
{
   // Set up invalid inputs, somehow.

   // Code under test which validates input fields from a JSF page, adding
   // error messages to the JSF context in case of validation failures.
   FacesContext ctx = FacesContext.getCurrentInstance();

   if (some input is invalid) {
      ctx.addMessage(null, new FacesMessage("Input xyz is invalid: blah blah..."));
   }
   ...

   // Test code: verify appropriate error message was added to context.
   new Verifications() {{
      FacesMessage msg;
      jsf.addMessage(null, msg = withCapture());
      assertTrue(msg.getSummary().contains("blah blah"));
   }};
}
```

上面测试代码中很有意思的是我们不需要担心 FacesContext.getCurrentInstance()方法, 由于 "jsf" 模糊实例将自动返回。

级联自返回的方法

其他场景如当我们使用流式接口在测试代码中的时候，级联也是有用的，一个建造者对象通过它的大多数方法最后返回它自身。因此，我们结束一个方法调用链来产生一些 final 对象或状态，在上面的测试例子中我们模拟了 java.lang.ProcessBuilder 类。

```
@Test
public void createOSProcessToCopyTempFiles(@Mocked final ProcessBuilder pb) throws Except
{
   // Code under test creates a new process to execute an OS-specific command.
   String cmdLine = "copy /Y *.txt D:\\TEMP";
   File wrkDir = new File("C:\\TEMP");
   Process copy = new ProcessBuilder().command(cmdLine).directory(wrkDir).inheritIO().sta
   int exit = copy.waitFor();
   ...

   // Verify the desired process was created with the correct command.
   new Verifications() {{ pb.command(withSubstring("copy")).start(); }};
}
```

如上，方法 command(...), directory(...), 和 inheritIO() 配置了 process 来建立, 然而 start() 最终建立了它。模拟的流程构建者对象自动返回了它的 ("pb") 从这些调用, 同时也返回了一个新的模拟 Process从调用 start()之后。

部分模拟

缺省的，所有的方法和构造方法被调用在模拟类型中的和它的父类 (除了 java.lang.Object) 都被模拟了。这对于大多数的测试用例是适用的，但是在一些情况下，我们需要选择其中的一些确定的方法或构造方法来进行模拟。方法/构造方法在模拟类中没有被模拟将按照普通的调用 方式执行。

当一个类或对象被部分模拟，JMockit 决定是否执行一个方法或一个构造方法的真是实现在它的测试代码中，基于哪些在期望中被录制，哪些没有。接下来的例子将展示这个用法。

```
public class PartialMockingTest
{
   static class Collaborator
   {
      final int value;

      Collaborator() { value = -1; }
      Collaborator(int value) { this.value = value; }

      int getValue() { return value; }
      final boolean simpleOperation(int a, String b, Date c) { return true; }
      static void doSomething(boolean b, String s) { throw new IllegalStateException(); }
   }

   @Test
   public void partiallyMockingAClassAndItsInstances()
   {
      final Collaborator anyInstance = new Collaborator();
```

```
        new Expectations(Collaborator.class) {{
           anyInstance.getValue(); result = 123;
        }};

        // Not mocked, as no constructor expectations were recorded:
        Collaborator c1 = new Collaborator();
        Collaborator c2 = new Collaborator(150);

        // Mocked, as a matching method expectation was recorded:
        assertEquals(123, c1.getValue());
        assertEquals(123, c2.getValue());

        // Not mocked:
        assertTrue(c1.simpleOperation(1, "b", null));
        assertEquals(45, new Collaborator(45).value);
    }

    @Test
    public void partiallyMockingASingleInstance()
    {
        final Collaborator collaborator = new Collaborator(2);

        new Expectations(collaborator) {{
           collaborator.getValue(); result = 123;
           collaborator.simpleOperation(1, "", null); result = false;

           // Static methods can be dynamically mocked too.
           Collaborator.doSomething(anyBoolean, "test");
        }};

        // Mocked:
        assertEquals(123, collaborator.getValue());
        assertFalse(collaborator.simpleOperation(1, "", null));
        Collaborator.doSomething(true, "test");

        // Not mocked:
        assertEquals(2, collaborator.value);
        assertEquals(45, new Collaborator(45).getValue());
        assertEquals(-1, new Collaborator().getValue());
    }
}
```

如上所示， Expectations(Object...) 构造方法接收一个或者是多个类或对象来进行部分模拟。如果给定一个类对象，在该类中的所有的方法和构造方法包括其父类中的会被模拟; 所定义的类中的所有实例将被模拟。从另外一方面来说，如果给定一个普通实例，只有方法而不是构造方法在类层次中被模拟; 甚至，只有特定的实例将会被模拟。

注意在两个测试例子中没有包含模拟的属性或模拟参数。部分模拟构造方法对于提供另外一种方式来模拟类是有用的。它也允许我们能够转变本地存储的对象实例为模拟实例。这些对象可以在内置的实例属性中建立任意数量的状态；他们将会保持状态被模拟。

也应该注意到，我们需要一个类或实例被部分模拟，也可以包含调用验证在其中，即使验证方法/构造方法没有被录制。例如考虑如下的例子。

```
@Test
public void partiallyMockingAnObjectJustForVerifications()
{
  final Collaborator collaborator = new Collaborator(123);

  new Expectations(collaborator) {};

  // No expectations were recorded, so nothing will be mocked.
  int value = collaborator.getValue(); // value == 123
  collaborator.simpleOperation(45, "testing", new Date());
  ...

  // Unmocked methods can still be verified:
  new Verifications() {{ c1.simpleOperation(anyInt, anyString, (Date) any); }};
}
```

最后，一个简单的方法来实现部分模式在一个测试类上是在测试类中标记一个测试同时为 @Tested (看如下的章节) 和 @Mocked。在这个例子中，测试类是不需要被传递到期望构造方法中的，但是我们还是需要录制任何方法的所需模拟结果在期望中。

捕捉实现类和实例

在讨论这个特征的时候我们基于如下的代码（特意制作的）。

```
public interface Service { int doSomething(); }
final class ServiceImpl implements Service { public int doSomething() { return 1; } }

public final class TestedUnit
{
   private final Service service1 = new ServiceImpl();
   private final Service service2 = new Service() { public int doSomething() { return 2;

   public int businessOperation()
   {
      return service1.doSomething() + service2.doSomething();
   }
}
```

我们想要测试的方法，businessOperation(), 使用了实现一个独立接口的类， Service。一个这些实现的通过一个在匿名的内部类定义，而这些通过客户端代码是不可见的(除了使用反射)。

模拟未明确实现的类

给定一个基类 (可能是一个接口，一个抽象类，或任何类型的基类), 我们可以编写测试只需知道这个需要被模拟基类而不是所有的实现/继承实现类。为了这样做，我们声明了一个 "捕捉的" 模拟类型引用了只知道的基类型。不仅仅已经被 JVM 载入的实现类可以被模拟，也包括一些其他的类在之后测试执行中才被 JVM载入的。 这个能力可以通过 @Capturing 注释激活, 可以用于模拟的属性和模拟参数，如下所示。

```
public final class UnitTest
{
   @Capturing Service anyService;

   @Test
   public void mockingImplementationClassesFromAGivenBaseType()
   {
      new Expectations() {{ anyService.doSomething(); returns(3, 4); }};

      int result = new TestedUnit().businessOperation();

      assertEquals(7, result);
   }
}
```

在如上的测试中，两个返回值被定义对 Service#doSomething() 方法。这个期望将会匹配所有对这个方法的调用, 不管实际实例在什么时候调用，也不管实际的类是如何实现这个方法。

future 实例的定义行为

一个其他能力关于抓捕在 future 实例上是可赋值给模拟类型, 通过使用 "maxInstances" 可选的属性来实现。 这个属性定义一个整数值来定义模拟类型的future实例将被相关的模拟属性/参数所覆盖的最大次数; 当没有明确定义, 所有的可赋值实例，包括之前存在还是在测试中被建立的豆浆被覆盖。

在一个给定捕捉的模拟属性或参数上进行期望的录制或录制将匹配在任何future实例上的通过模拟属性/参数的调用覆盖。这允许我们录制和/或验证不同的行为针对每一个future实例集合; 为了这个，我们需要声明两个或者更多的相同类型的捕捉模拟属性/参数，每一个都包含自己的最大实例数目（可能要除了最后一个模拟属性/参数，可能会覆盖余下的future实例）。

为了展示这个功能，如下的例子使用了 java.nio.Buffer 子类和他们的 future 实例;在真实的例子中最好是使用真实的 buffers rather 而不是模拟的。

```
@Test
public void testWithDifferentBehaviorForFirstNewInstanceAndRemainingNewInstances(
   @Capturing(maxInstances = 1) final Buffer firstNewBuffer,
   @Capturing final Buffer remainingNewBuffers)
{
   new Expectations() {{
      firstNewBuffer.position(); result = 10;
      remainingNewBuffers.position(); result = 20;
   }};

   // Code under test creates several buffers...
   ByteBuffer buffer1 = ByteBuffer.allocate(100);
   IntBuffer  buffer2 = IntBuffer.wrap(new int[] {1, 2, 3});
   CharBuffer buffer3 = CharBuffer.wrap("                ");

   // ... and eventually read their positions, getting 10 for
   // the first buffer created, and 20 for the remaining ones.
   assertEquals(10, buffer1.position());
   assertEquals(20, buffer2.position());
   assertEquals(20, buffer3.position());
}
```

应该注意到当在一个范围内捕捉模拟类型，所有的实现类都将被模拟，不管任何 maxInstances 约束的定义。

实例化和注入测试对象

通常一个测试类将会使用一个独立的测试类。 JMockit 可以帮助自动实例化这个类，和可选的注入相关的模拟依赖。这也是 @Tested 注释使用的地方。

一个非final的实例属性在测试类中被注释将会被认为是一个自动初始化和注入，在一个测试方法被执行之前。如果这个时候这个属性还是为null 引用, 一个实例将会通过使用在测试类中的恰当的构造方法建立, 但是需要确保其内置的依赖能够被切当的注入 (当使用的时候)。如果这个属性已经被初始化 (不是 null), 则不会有任何事情发生。

为了能够注入 ,测试类中必须包含一个或多个模拟属性或模拟参数声明为 @Injectable。 模拟属性/参数只通过 @Mocked 或 @Capturing注释的将不被认为注入。换句话来说就是，不是所有的可以注入的属性/参数都需要包含模拟类型; 他们也可以包含原始或数组类型。如下的测试类将展示。

```
public class SomeTest
{
   @Tested CodeUnderTest tested;
   @Injectable Dependency dep1;
   @Injectable AnotherDependency dep2;
   @Injectable int someIntegralProperty = 123;

   @Test
   public void someTestMethod(@Injectable("true") boolean flag, @Injectable("Mary") Strin
   {
      // Record expectations on mocked types, if needed.

      tested.exerciseCodeUnderTest();

      // Verify expectations on mocked types, if required.
   }
}
```

注意一个非模拟类型的注入属性/参数都必须显式的定义一个值，否则缺省值将被使用。在这个注入属性的例子中，值可以通过简单的赋值给属性。此外，它可以提供一个 "value" 属性在 @Injectable中, 这是唯一的方式在可注入的测试方法参数中定义值。

两种注入类型能够被支持: 构造注入和属性注入。在第一个例子中，测试类中必须包含一个构造方法能够满足注入。 注意在一个给定的测试中，一系列可用的注入包括声明为测试类实例属性的注入属性和在测试方法中定义的可注入参数; 因此，在相同的测试类中的不同测试可以提供不同的注入在相同的测试代码中。

一旦测试类在被选中的构造方法初始化，它的非final实例属性将被认为是注入的。针对每一个注入的属性，相同类型的注入属性将被在测试中类搜索。如果只找到了一个，它的当前值将被读取和存储到注入属性中。如果找到不只一个，注入属性名将被用来作为从相同类型的注入属性中选择。

重用期望和验证代码块

最简单的复用测试代码在 JMockit 中就是在测试类级别中声明模拟属性。正如下面的例子所示，被建立和赋值到这些属性中的对象将被使用在任意数量的测试方法中(通过 JMockit 或显示的测试代码)。

```
public final class LoginServiceTest
{
   @Tested LoginService service;
   @Mocked UserAccount account;

   @Before
   public void init()
   {
      new Expectations() {{ UserAccount.find("john"); result = account; minTimes = 0; }};
   }

   @Test
   public void setAccountToLoggedInWhenPasswordMatches() throws Exception
   {
      willMatchPassword(true);

      service.login("john", "password");

      new Verifications() {{ account.setLoggedIn(true); }};
   }

   void willMatchPassword(final boolean match)
   {
      new Expectations() {{ account.passwordMatches(anyString); result = match; }};
   }

   @Test
   public void notSetAccountLoggedInIfPasswordDoesNotMatch() throws Exception
   {
      willMatchPassword(false);

      service.login("john", "password");

      new Verifications() {{ account.setLoggedIn(true); times = 0; }};
   }

   // other tests that use the "account" mock field
}
```

上面的测试例子中测试类使用了 LoginService#login(String accountId, String password) 方法。这个方法首次尝试查找一个存在用户账号从给定的登录名称中 ("accountId", 作为所有账号中的唯一标识)。 由于一些不同的测试需要完全的使用这个方法，一个对 UserAccount#find(String accountId) 方法的调用被录制在这个测试类中的, 使用一个特殊的登录名 ("john") 和模拟账号作为返回值。 注意， 任何给定的测试都可以使用多个期望和/或验证块。这些块可以各自的写在共享的 "before" 和 "after" 方法中。

另外上面一个重用的例子通过使用 willMatchPassword(boolean) 方法来显示，其中包括了另外一个可以重用的期望块。在这个例子中,一个匹配任何密码值对 UserAccount#passwordMatches(String) 方法的调用被录制，在重用方法中通过参数提供了一个调用返回值。

另外一个重用期望和验证的方法就是建立命名类而不是匿名的。例如，可以使用一个重用的内部类来代替 the willMatchPassword(boolean) 方法的使用:

```
final class PasswordMatchingExpectations extends Expectations
{
   PasswordMatchingExpectations(boolean match)
   {
      account.passwordMatches(anyString); result = match;
   }
}

@Test
public void setAccountToLoggedInWhenPasswordMatches() throws Exception
{
   new PasswordMatchingExpectations(true);

   ...
}
```

注意这些类必须声明为final,除非他们将被作为一个基类针对进一步的扩展。例如非final的基类必须以"Expectations" 或 "Verifications"命名结束;要不然他们将不会被 JMockit识别。

最后，可重用的期望/验证子类也可以是顶级类，允许他们在任意数量的测试类中使用。

其他主题

接下来的章节将会介绍很少发生的情况。

同时模拟多个接口

假如一个测试代码需要模拟一个实现了两个甚至更多的接口的类。如下的例子展示如何实现。

```
public interface Dependency
{
   String doSomething(boolean b);
}

public class MultiMocksTest<MultiMock extends Dependency & Runnable>
{
   @Mocked MultiMock multiMock;

   @Test
   public void mockFieldWithTwoInterfaces()
   {
      new Expectations() {{ multiMock.doSomething(false); result = "test"; }};

      multiMock.run();
      assertEquals("test", multiMock.doSomething(false));

      new Verifications() {{ multiMock.run(); }};
   }

   @Test
   public <M extends Dependency & Serializable> void mockParameterWithTwoInterfaces(
      @Mocked final M mock)
   {
      new Expectations() {{ mock.doSomething(true); result = "abc"; }};

      assertEquals("abc", mock.doSomething(true));
      assertTrue(mock instanceof Serializable);
   }
}
```

在如上的测试中，两个接口在一起被模拟：依赖于 java.lang.Runnable 针对一个模拟属性，和依赖于 java.io.Serializable 针对模拟参数。我们使用类型变量 MultiMock (在整个测试代码中定义的) 和 M (定义了一个测试方法) 因此 JMockit 将指导每一个例子中的组合接口。

迭代期望

当一系列连续的调用被在严格的期望中录制 (否则在调用的相关顺序是无关的), 整个序列期待在重放阶段准备的执行一次。然而考虑到, 这个例子中测试代码将在一个循环中进行调用 (或任何类型的迭代)。假如在测试中的迭代数目是知道的，我们可以录制这些期望通过在一个循环中简单的调用每一个方法/构造方法(也就是，不需要在一个期望块中编写一个循环或重复这个期望)。接下来的测试将会展示这个特征，使用 StrictExpectations(numberOfIterations) 构造方法。

```
@Test
public void recordStrictInvocationsInIteratingBlock(@Mocked final Collaborator mock)
{
   new StrictExpectations(2) {{
      mock.setSomething(anyInt);
      mock.save();
   }};

   // In the tested code:
   mock.setSomething(123);
   mock.save();
   mock.setSomething(45);
   mock.save();
}
```

针对一组的调用定义迭代次数的能力也适用于普通的期望。然而这个例子中所定义的迭代次数只是作为上下限调用次数约束的乘数 (包括隐式和显式的)。

验证迭代

正如我们所看到的录制期望块，验证块也可以处理在外部循环中调用，针对一个特殊数目的循环迭代。

```
@Test
public void verifyAllInvocationsInLoop(@Mocked final Dependency mock)
{
   int numberOfIterations = 3;

   // Code under test included here for easy reference:
   for (int i = 0; i < numberOfIterations; i++) {
      DataItem data = getData(i);
      mock.setData(data);
      mock.save();
   }

   new Verifications(numberOfIterations) {{
      mock.setData((DataItem) withNotNull());
      mock.save();
   }};

   new VerificationsInOrder(numberOfIterations) {{
      mock.setData((DataItem) withNotNull());
      mock.save();
   }};
}
```

以上两个验证块只是为了解释不通的语法在按顺序和无序的迭代验证块。在第一个块中，每一个验证调用将会匹配至少三个调用相同的方法在重复阶段，因为这是构造方法中传递的迭代数目。 针对一个无序的迭代块，特定数目的迭代可以通过最小和上限与调用次数进行相乘; 即使一个显式的约束被定义在块中，例如 minTimes = 1; maxTimes = 4; 成对的赋值，在这个特殊例子中将为 minTimes = 3; maxTimes = 12;在第二个块中，在另外一方面来看, 调用次数是没有作用的。相反的，这是效果等价于循环展开，好像整个在块中的调用验证是在每个迭代中进行重复。

一个迭代 FullVerifications 块的语法和普通的 Verifications块类似。这也同样适用于迭代的 FullVerificationsInOrder 块，正如VerificationsInOrder 块。

伪装

1. 伪装方法和伪装类
2. 为测试设置伪装
    i. 可以被伪装的方法种类
    ii. 内联的伪装类
3. 伪装接口
4. 伪装未被实现的类
5. 调用次数的约束
6. 伪装方法的初始化
7. 使用调用上下文
8. 执行真正的实现部分
9. 在测试时间重用伪装
    i. 使用 before/after 方法
    ii. 重用伪装类
10. 在测试类和测试套件级别使用伪装

在 JMockit toolkit中, 伪装 API 提供了建立假的实现或者是伪装。 伪装区别于模拟的API在与，前者不仅仅只在测试调用的中被调用的时候返回我们所期待的值，我们还可以修改其中的依赖，使得它能满足测试的需要。通常，只是伪装部分的方法或者是构造函数在需要伪装的类中，使其他的方法和构造函数不修改。

伪装的实现在依赖于外部的实体或者是比如网络，文件系统的资源集成测试中特别有用。伪装的实现可以接触外部的实体在相同的集成环境中使用两种模式: 1) 真实的模式，所有的代码(被测试的代码和它的依赖)被正常的执行; 2) 模拟模式, 有问题的依赖将会被伪装的实例进行替换，因此他们可以在没有网络连接，没有文件系统，甚至没有任何他们所需要的外部依赖的情况下成功运行。使用伪装的对象对真实实现的替换对于使用这些依赖的代码来说是完全透明的，而且可以在不同的测试运行的时候进行开启或者是关闭。

对于这章的剩余部分，我们将为一个使用javax.security.auth.login.LoginContext 类型进行用户验证的应用编写测试。 在这个例子中，我们必须要所有的测试真实运行在任何 JAAS 代码, 因为它可能依赖于外部的配置，而且很不容易在开发者的测试环境中部署起来。因此，一个依赖于LoginContext 的应用类将会被测试，然而 LoginContext 类 (所依赖的)将至少有一些方法和构造函数被伪装，为了测试验证逻辑。

伪装方法和伪装类

在 Mockups API的上下文中, mock方法可以使一个伪装类中的任何方法，一个伪装的类通过使用@Mock 注释。简单的来说，在这章我们将简单注释伪装的方法称为"mocks";在其他的上下文中"mock" 可能只的是一个伪装类的实例。 一个伪装的类可以使一个任何继承自 mockit.MockUp 泛型的基类, 其中 指的是需要被伪装的类。在下面的例子中显示了一些伪装在我们例子中的真实类中 javax.security.auth.login.LoginContext。

```
public final class MockLoginContext extends MockUp<LoginContext>
{
   @Mock
   public void $init(String name, CallbackHandler callbackHandler)
   {
      assertEquals("test", name);
      assertNotNull(callbackHandler);
   }

   @Mock
   public void login() {}

   @Mock
   public Subject getSubject() { return null; }
}
```

当伪装类被使用到一个真实类的时候，之后获取对应被临时实现这些方法和构造函数的实现的将会被替换。换句话来说，真实的被伪装的类将在测试运行的时候被按照约定的方式进行调用修改。在运行时，被伪装的方法或者是构造函数将会被拦截和重定向到对应被伪装的函数中，然后执行并返回给原始的调用者(除非有 exception/error 被抛出), 不是当前方法而是一个不同的方法被真实的执行。通常在测试中的调用类的依赖类是伪装类。

伪装类通常在JUnit/TestNG测试类中被定义为嵌套的 (静态), 内部的 (非静态), 或甚至是匿名类。 尽管伪装类也可以是顶端的类。这在多一个测试用例类中复用伪装对象比较有用。正如我们将看到的，通常最方便的方法来实现一个伪装类是通过使他们成为匿名本地的类对于每一个测试方法。

当我们在生产代码中定义的真实类需要在测试中被伪装，一个新的伪装类将会被建立。其中必须至少顶一个伪装的方法，还可以包含任何数量的其他方法和构造方法; 它仍然可以定义任意多个字段。

每一个被 @Mock的方法必须有和在目标真实对象上的真实的方法或构造函数具有相同的签名。对于伪装的方法，签名包括 方法名和参数; 对于伪装的构造函数, 它仅仅是参数以及使用特殊的"$init"作为伪装的方法名。 如果在给定的伪装方法没有找到匹配的方法或构造函数在定义的实际类或者时其超类中 (不包括 java.lang.Object), 将会在尝试使用伪装类的时候抛出 IllegalArgumentException。注意这个异常可能被一个修改过的实际类导致(例如重命名实际的方法名), 因此需要明白它为什么会发生。

最后，注意不必伪装在实际类中所有的方法或者是构造方法。任何没有被伪装的在实际类中的方法将会入它原来一样，不会被伪装。当然是在假定没有其他的伪装类被使用在相同的实际类中，有是合法的(有时这比较有用)。 当有两个或者是多个伪装类使用在同一相同的实际类中，需要确保被伪装的类不能定义相同的伪装，如果有重复的定义，最后一个定义将会被使用。

为测试设置伪装

使用给定的伪装类对应于真实的类需要有一个结果。我们称这个为伪装类的设置。这些事情通常在独立测试方法或者是的在 @BeforeMethod (TestNG) 或 @Before (JUnit 4) 方法中执行。一旦一个伪装被设置，所有对于实际类中伪装方法和构造方法的执行将会被自动重定向到对应的伪装方法中。

为了设置上面的 MockLoginContext伪装类，我们简单的将其实例化:

```
@Test
public void settingUpAMockClass() throws Exception
{
   new MockLoginContext());

   // Inside an application class which creates a suitable CallbackHandler:
   new LoginContext("test", callbackHandler).login();

   ...
}
```

因为伪装类是在一个测试方法中被设置，被伪装的 LoginContext类通过 MockLoginContext将只在特殊的测试中有效。

当构造方法被调用初始化 LoginContext的时候, 对应的 "$init" 伪装方法在 MockLoginContext 中将会被执行, 假定有正确的调用参数。同样的的，当LoginContext#login 方法被调用的时候，对应的伪装方法将会被执行，在这个例子中将会不做任何东西，由于这个方法没有参数和返回值。这些调用发生在测试的第一部分就被建立的伪装实例上。

以上的部分示例简单的验证了LoginContext 类被使用正确的参数和特定包含一个上下文名称和回调的句柄所构造所初始化。 如果真实对象完全没有被初始化，测试仍将通过(除非包含一些其他的条件导致其失败)。login 方法的调用对于这个测试的输出没有影响, 除了这个调用只是执行了一个空的伪装方法而不是真实的那个。

现在，要是我们想模拟一个不同的验证失败错误？LoginContext#login() 方法声明其将抛出一个 LoginException "如果验证失败", 我们可以很简单的实现(使用 JUnit 4 如下所示):

```
public static class MockLoginContextThatFailsAuthentication extends MockUp<LoginContext>
{
   @Mock
   public void $init(String name) {}

   @Mock
   public void login() throws LoginException
   {
      throw new LoginException();
   }
}

@Test(expected = LoginException.class)
public void settingUpAnotherMockClass() throws Exception
{
   new MockLoginContextThatFailsAuthentication();

   // Inside an application class:
   new LoginContext("test").login();
}
```

这个测试只有在 LoginContext#login() 方法抛出一个异常才会通过，，只有当对应的伪装方法被执行。

可以被伪装的方法种类

目前为止，我们只使用公开的伪装方法伪装了实例的公共方法。实际上任何在实际类上任何种类的方法都可以被伪装: 私有方法，保护或是包保护级别的, 静态方法, 最终方法, 和本地方法。(同样包括同步和 strictfp 方法, 但是这些修改只会影响方法的实现，而不是它们的接口。) 甚至一个实际类中的静态方法都可以被通过一个伪装实例方法进行修改，反之亦然（一个实例眞方法包含一个静态的伪装）; 这些都可以被使用到 final 修饰符中。

被伪装的方法需要包含一个实现，虽然不必要是字节码上(在本地方法的例子中)。因此，一个抽象的方法不能够被直接的伪装，对于Java 接口也一样。 (那就是说如下的 Mockups API 将会自动的生产一个代理对象，实现这个接口。)

内联的伪装类

通常对于各实际类的一个特定组的伪装方法只会在一个测试中有用。在这种情况下，我们将会建立一个匿名类在各自的测试方法中，正如下面的例子所展示的。

```
@Test
public void settingUpMocksUsingAnAnonymousMockClass() throws Exception
{
   new MockUp<LoginContext>() {
      @Mock void $init(String name) { assertEquals("test", name); }
      @Mock void login() {}
   });

   new LoginContext("test").login();
}
```

注意这里的伪装方法不需要申明为 public。

伪装接口

大多数情况下伪装类总是面对直接使用在一个实际的对象上。但是假如我们需要伪装一个实现确定接口的对象传入代码中进行测试时怎么办？如下示例显示了如何实现接口 javax.security.auth.callback.CallbackHandler。

```
@Test
public void mockingAnInterface() throws Exception
{
   CallbackHandler callbackHandler = new MockUp<CallbackHandler>() {
      @Mock
      void handle(Callback[] callbacks)
      {
         assertEquals(1, callbacks.length);
         assertTrue(callbacks[0] instanceof NameCallback);
      }
   }.getMockInstance();

   callbackHandler.handle(new Callback[] {new NameCallback("Enter name:")});
}
```

MockUp#getMockInstance() 方法返回了一个实现目标接口的代理对象。

伪装未被实现的类

为了展示这个功能，我们使用如下的测试代码。

```
public interface Service { int doSomething(); }
final class ServiceImpl implements Service { public int doSomething() { return 1; } }

    public final class TestedUnit
    {
       private final Service service1 = new ServiceImpl();
       private final Service service2 = new Service() { public int doSomething() { return

       public int businessOperation()
       {
          return service1.doSomething() + service2.doSomething();
       }
    }
```

我们想要测试的方法 businessOperation(), 实现了一个特定的接口Service的类。 这些实现是通过一个匿名的内部类实现的，而这些内容通过客户端代码是无法获取的(除了使用反射的方式)。

给定一个基类(可以使一个接口，一个抽象类，或者是任何种类的基类), 我们可以编写一个测试只需要知道这个基类，它的所有实现和扩展都将被伪装。为了做到这个，我们需要建立一个伪装，一个伪装类型只知道其基类，而被通过类型变量的方式进行使用。不仅仅已经被JVM载入的实现类可以被伪装，那些在测试执行的时候将要被JVM载入的其他类也可以被伪装。这些能力将被如下所示。

```
@Test
public <T extends Service> void mockingImplementationClassesFromAGivenBaseType()
{
   new MockUp<T>() {
      @Mock int doSomething() { return 7; }
   };

   int result = new TestedUnit().businessOperation();

   assertEquals(14, result);
}
```

在上面的测试中，所有对 Service#doSomething() 方法实现将被重定向到被伪装的实现中，而不是原来实际上实现接口的类。

调用次数的约束

所以的测试用例实例到目前为止只是使用了JUnit/TestNG 来判断样子调用的参数。 有时，我们可能想验证给定的依赖的方法或构造方法在整个测试用被调用。我们可能想验证实际上一个给定的伪装在测试中被调用了多少次，或定义如果大于或者是少于给定的调用次数将会测

试失败。为了实现这个，我们可以定义在一个给定伪装中调用次数的约束，如下例子所示。

```
@Test
public void specifyingInvocationCountConstraints() throws Exception
{
   new MockUp<LoginContext>() {
      @Mock(minInvocations = 1)
      void $init(String name) { assertEquals("test", name); }

      @Mock(invocations = 1)
      void login() {}

      @Mock(maxInvocations = 1)
      void logout() {}
   };

   new LoginContext("test").login();
}
```

在这个测试中我们使用了 @Mock 注释中三个相关调用次数的属性。第一个伪装定义了 LoginContext(String) 构造方法必须至少在测试中被调用一次。第二个定义了 login() 方法必须被调用只有一次, 但是第三个声明了 logout() 可以被调用不仅仅一次。

同样的可以在一个伪装中定义调用的最小和最大次数，为了约束调用次数在给定的范围中。

伪装方法的初始化

当一个生产代码中的类在一个或者是多个静态初始化块中做了一些工作，我们可能需要进行伪装为了避免在执行测试的时候进行这些初始化。我们可以通过顶一个特殊的伪装方法来做这些，通过如下的代码。

```
@Test
public void mockingStaticInitializers()
{
   new MockUp<ClassWithStaticInitializers>() {
      @Mock
      void $clinit()
      {
         // Do nothing here (usually).
      }
   };

   ClassWithStaticInitializers.doSomething();
}
```

特别需要注意的是如果一个类的静态初始化代码被伪装了，这个类中的所有静态代码块和对静态属性的赋值都将不被执行 (除了那些在编译时就被解析的，而不会产生任何可执行字节码)。由于JVM只会尝试初始化一次类，恢复一个被伪装的静态初始化代码是无效的。因此，如果你伪装了一个没有被JVM进行初始化的类的静态初始化，原始类中的初始化代码将在测试中不会被执行。这将导致任何的静态属性将运行时被表达式所覆盖，而不是他们缺省的初始化的值。

使用调用上下文

一个伪装的方法可以选择性的声明附加的参数在 mockit.Invocation 类型中, 使用第一个参数提供出来。 针对每一个真实的被执行的相应伪装的方法或者是构造方法，一个调用对象将被自动传入伪装方法的执行中。

调用上下文对象提供了一些 "getters" 方法，可以在伪装的方法中使用。 其中一个是 getInvokedInstance() 方法, 将会返回在调用发生时的伪装实例(null 如果这个方法是静态的)。其他的 getters 提供了一些调用(包括当前) 包括伪装的方法或构造方法，调用次数的约束(如果有的话) 在 @Mock 注释中被定义, 等等。 如下我们有一个测试的例子。

```
@Test
public void accessingTheMockedInstanceInMockMethods() throws Exception
{
   final Subject testSubject = new Subject();

   new MockUp<LoginContext>() {
      @Mock
      void $init(Invocation invocation, String name, Subject subject)
      {
         assertNotNull(name);
         assertSame(testSubject, subject);

         // Gets the invoked instance.
         LoginContext loginContext = invocation.getInvokedInstance();

         // Verifies that this is the first invocation.
         assertEquals(1, invocation.getInvocationCount());

         // Forces setting of private Subject field, since no setter is available.
         Deencapsulation.setField(loginContext, subject);
      }

      @Mock(minInvocations = 1)
      void login(Invocation invocation)
      {
         // Gets the invoked instance.
         LoginContext loginContext = invocation.getInvokedInstance();

         // getSubject() returns null until the subject is authenticated.
         assertNull(loginContext.getSubject());

         // Private field set to true when login succeeds.
         Deencapsulation.setField(loginContext, "loginSucceeded", true);
      }

      @Mock
      void logout(Invocation invocation)
      {
         // Gets the invoked instance.
         LoginContext loginContext = invocation.getInvokedInstance();

         assertSame(testSubject, loginContext.getSubject());
      }
   };

   LoginContext theMockedInstance = new LoginContext("test", testSubject);
   theMockedInstance.login();
   theMockedInstance.logout();
}
```

执行眞正的实现部分

一旦一个 @Mock 方法被执行了，任何其他的对伪装的方法的执行都将被重定向到对应的伪装方法中，导致它的实现是可重入的。 然后如果我们想执行被伪装方法的眞实实现是，我们可以通过调用proceed() 方法在伪装方法接收到的第一个调用上下文参数对象上。

如下的测试示例演示了 LoginContext 对象通过正常的方式被建立 (在创建的时候不需要任何的伪装), 使用一个未被明确定义的配置。 (想看完整测试版本，请看 mockit.MockAnnotationsTest 类。)

```
@Test
public void proceedIntoRealImplementationsOfMockedMethods() throws Exception
{
   // Create objects to be exercised by the code under test:
   LoginContext loginContext = new LoginContext("test", null, null, configuration);

   // Set up mocks:
   ProceedingMockLoginContext mockInstance = new ProceedingMockLoginContext();

   // Exercise the code under test:
   assertNull(loginContext.getSubject());
   loginContext.login();
   assertNotNull(loginContext.getSubject());
   assertTrue(mockInstance.loggedIn);

   mockInstance.ignoreLogout = true;
   loginContext.logout(); // first entry: do nothing
   assertTrue(mockInstance.loggedIn);

   mockInstance.ignoreLogout = false;
   loginContext.logout(); // second entry: execute real implementation
   assertFalse(mockInstance.loggedIn);
}

static final class ProceedingMockLoginContext extends MockUp<LoginContext>
{
   boolean ignoreLogout;
   boolean loggedIn;

   @Mock
   void login(Invocation inv) throws LoginException
   {
      try {
         inv.proceed(); // executes the real code of the mocked method
         loggedIn = true;
      }
      finally {
         // This is here just to show that arbitrary actions can be taken inside
         // the mock, before and/or after the real method gets executed.
         LoginContext lc = inv.getInvokedInstance();
```

```
            System.out.println("Login attempted for " + lc.getSubject());
        }
    }

    @Mock
    void logout(Invocation inv) throws LoginException
    {
        // We can choose to proceed into the real implementation or not.
        if (!ignoreLogout) {
            inv.proceed();
            loggedIn = false;
        }
    }
}
```

在上面的示例中，所有的在被测试的 LoginContext 类中方法将被执行, 即使一些方法被伪装 (login 和 logout)。 这个例子有点做作，实际上对真实实现的执行很少在测试中被使用，至少不会直接使用。

你可能会注意到使用 Invocation#proceed(...) 在一个伪装的方法中的效果和使用 advice (AOP 的概念) 对应于真实的方法。这是一个强大的能力特别是对于一些事情(考虑一下一个拦截器或装饰器)。

针对所有的可以使用的 mockit.Invocation 类中的方法描述, 可以查看它的 API 文档。

在测试时间重用伪装

大多数的测试可能只会使用专用的伪装类，针对每一个特殊的测试而构建。可能有事我们需要在多个测试线程或者是单个或者是真个测试套件中复用伪装对象。 我们将会看到不同的方式来构建伪装从而保证他们共享于测试组，和定义一些可重用的伪装类。

使用 **before/after** 方法

在一个给定的测试对象上，我们可以定义在每个测试方法运行之前后内容(即使测试抛出了一个错误或者是异常)。 使用 JUnit, 我们可以使用 @Before 和 @After 注释来完成一个或者多个随意的测试实例的实例方法。在TestNG，我们仍然可以同样的使用 @BeforeMethod 和 @AfterMethod 注释。

任何伪装类在一个测试方法中可以在"before" 方法中声明使用。 这个伪装类将会在这个测试类的所有测试方法中被共享。任何在before中的伪装同样在 "after" 方法中也有用的。

例如，我们想要伪装 LoginContext 类，使用在一系列的测试中，我们可以在测试类中使用如下的方法:

```
public class MyTestClass
{
   @Before
   public void setUpSharedMocks()
   {
      new MockUp<LoginContext>() {
         // shared mocks here...
      };
   }

   // test methods that will share the mocks set up above...
}
```

以上的例子是使用 JUnit, 但是在 TestNG中也是可以类似的相同使用的。

从基类中继承而来的类，可能选择性的定义了 "before" 和/或者 "after" 方法包含了对 Mockups API 的调用。

重用伪装类

命名的伪装类可以被设计成为了一个具体类(可选择为final)，从而可以在特定的测试用使用。当通过测试代码直接初始化的时候，这些伪装实例可以通过构造方法参数，属性，或者是非伪装函数的方法注入。此外，他们仍然可以被定义为基类，(可能是 abstract) 被具体的伪装类进行扩展在特定的测试类或方法中。

关于这章的测试例子来自于 JMockit 自己的测试套件。它们所使用的类，部分显示在这里:

```
public final class TextFile
{
   // fields and constructors that accept a TextReader or DefaultTextReader object...

   public List<String[]> parse()
   {
      skipHeader();

      List<String[]> result = new ArrayList<String[]>();

      while(true) {
         String  strLine = nextLine();

         if (strLine == null) {
            closeReader();
            break;
         }

         String[] parsedLine = strLine.split(",");
         result.add(parsedLine);
      }

      return result;
   }

   // private helper methods that call "skip(n)", "readLine()", and "close()"...

   public interface TextReader
   {
      long skip(long n) throws IOException;
      String readLine() throws IOException;
      void close() throws IOException;
   }

   static final class DefaultTextReader implements TextReader
   {
      DefaultTextReader(String fileName) throws FileNotFoundException { ...mocked... }
      public long skip(long n) throws IOException { ...mocked... }
      public String readLine() throws IOException { ...mocked... }
      public void close() throws IOException { ...mocked... }
   }
}
```

以上类的一些测试方法如下所示。

```
public final class TextFileUsingMockUpsTest
{
   // A reusable mock-up class to be applied in specific tests.
   static final class MockTextReaderConstructor extends MockUp<DefaultTextReader>
   {
      @Mock(invocations = 1)
      void $init(String fileName) { assertThat(fileName, equalTo("file")); }
   }

   @Test
   public void parseTextFileUsingDefaultTextReader() throws Exception
   {
      new MockTextReaderConstructor();
      new MockTextReaderForParse<DefaultTextReader>() {};

      List<String[]> result = new TextFile("file", 200).parse();

      // assert result from parsing
   }

   ...
```

以上的测试使用了两个可重用的伪装类。第一个包装了单个构造TextFile.DefaultTextReader嵌套类的伪装。 任何在 TextFile 类中的测试都将被调用这个构造方法。它通过简单的初始化在测试方法中。

第二个伪装类被使用在对DefaultTextReader 类的相同测试上。 接下来我们将看到, 它定义了对整个不同集合成员的通过 TextFile#parse() 方法的调用中伪装。

```
...

// A reusable base mock class to be extended in specific tests.
static class MockTextReaderForParse<T extends TextReader> extends MockUp<T>
{
   static final String[] LINES = { "line1", "another,line", null};
   int invocation;

   @Mock(invocations = 1)
   long skip(long n)
   {
      assertEquals(200, n);
      return n;
   }

   @Mock(invocations = 3)
   String readLine() throws IOException { return LINES[invocation++]; }

   @Mock(invocations = 1)
   void close() {}
}

...
```

以上伪装类，像 mockit.MockUp 类中它所继承的是泛型。 在这个特殊的例子中，这是必须的，由于被测试的 TextFile 类依赖于两种不同类型的"text reader" : TextFile.TextReader (一个可以通过客户端代码进行实现的接口), 和 TextFile.DefaultTextReader (一个内置的缺省的对接口的实现)。 前者的测试简单的通过定义一个匿名的子类定义了被伪装的具体的DefaultTextReader的实现的方法来使用这个伪装对象。第二个测试，通过传递一个TextReader的实现给 TextFile:

```
...

@Test
public void parseTextFileUsingProvidedTextReader() throws Exception
{
   TextReader textReader = new MockTextReaderForParse<TextReader>() {}.getMockInstance();

   List<String[]> result = new TextFile(textReader, 200).parse();

   // assert result from parsing
}

...
```

在这个例子中接口的实现是一个伪装的代理对象，通过调用MockUp#getMockInstance() 方法获得。

最后，我们将使用一个更有趣的例子，具体的伪装子类真正的覆盖了基伪装类的实现:

```
...

@Test
public void doesNotCloseTextReaderInCaseOfIOFailure() throws Exception
{
   new MockTextReaderConstructor();

   new MockTextReaderForParse<DefaultTextReader>() {
      @Override @Mock
      String readLine() throws IOException { throw new IOException(); }

      @Override @Mock(invocations = 0)
      void close() {}
   };

   TextFile textFile = new TextFile("file", 200);

   try {
      textFile.parse();
      fail();
   }
   catch (RuntimeException e) {
      assertTrue(e.getCause() instanceof IOException);
   }
}
```

测试中强制一个 IOException 被抛出在首次调用readLine()的时候。 (这个异常将被在解析方法中包装成一个 RuntimeException异常。) 它还通过定义调用次数约束，保证 close() 方法没有被调用。 这里不仅仅展示了继承的伪装可以被覆盖，而且通过@Mock 注释所定义的元数据也可以被覆盖。

在测试类和测试套件级别使用伪装

正如我们所看到的，伪装类通常被使用在独立的测试专用。有时可能我们需要伪装类作为整个测试类内(指的是其中所有的测试方法)或者是整个测试套件中使用(指的是其中所有测试类)。它仍可能定义模拟类在一个整体的测试中通过外部的配置，通过定义一个JVM级别的系统属性或者是添加一个外部的 jmockit.properties 文件在运行时的 classpath下。

伪装程序化的应用在更广阔的范围

为了伪装一个类在整个测试类的范围内（所有测试）, 我们简单的使用内置的 @BeforeClass 方法 (在 JUnit 或 TestNG)。 为了使用伪装在一个测试套件中，我们需要使用 TestNG中的 @BeforeSuite 方法, 或在 JUnit 套件类中。接下来的例子将会显示一个JUnit 4 测试套件的配

置应用的伪装。

```
@RunWith(Suite.class)
@Suite.SuiteClasses({MyFirstTest.class, MySecondTest.class})
public final class TestSuite
{
   @BeforeClass
   public static void setupMocks()
   {
      new LoggingMocks();

      new MockUp<SomeClass>() {
         @Mock someMethod() {}
      };
   }
}
```

在这个例子中，我们使用 LoggingMocks 伪装类和一个内联的伪装类; 这些伪装将会作用到整个测试套件最后一个测试执行结束。

通过一个系统属性的外部应用

jmockit-mocks 系统属性支持以逗号分隔的全名称定义的伪装类列表。 如果是定义在 JVM启动的时候，所有的这些类（继承于MockUp）将被自动在整个测试中使用。在启动中定义的伪装类将会一直作用到测试运行结束，针对所有的测试类。

注意一个系统的属性可以通过标准的"-D" 命令行格式传递给JVM。 Ant/Maven/等 构建脚步都包含它们自己的方式进行系统属性的定义，可以检测它们的详细文档。

通过 jmockit.properties 文件的外部应用

伪装类同样的可以被定义为系统属性通过一个 独立的jmockit.properties 文件, 必须放在 classpath 的根下。如果有多个这个文件被包含在 classpath下 (无乱是在jars中或者是在纯路径下), 所有的类名列表将被添加到一起。这允许奖励可重用的伪装类打包在一个jar文件中，其中它包含了自己的 properties 文件; 当其被添加到 测试套件的 classpath 下,这些伪装类将会被自动在启动时候被使用。

为了方便，在 properties文件中项可以不需要写 "jmockit-" 的前缀。

代码覆盖

1. 行覆盖
2. 路径覆盖
3. 数据覆盖
4. 覆盖输出的类型
    i. 调用节点
5. 配置覆盖工具
6. 聚集多个测试结果报告
    i. 多个数据文件生成聚集报告
    ii. 从单个数据文件的每次测试之后生成聚集报告
7. 检测最小的覆盖
8. 在maven项目中激活覆盖
    i. 在maven站点中包含html 报告
9. 关闭覆盖输出
10. 独立模式

其他的java覆盖工具包括 EMMA, Clover, Cobertura, 和 JaCoCo. (后者更经常通过 EclEmma Eclipse 插件的方式进行使用。) 这些工具提供了两种的独立的覆盖测试度量，分别是语句和分支覆盖。第一种也被叫做行覆盖，尽管他们都没有眞实的尝试度量有多少的独立的可执行代码被覆盖。第二种度量有多少个可选的分支基于判断点(if 或者是 switch 语句) 在一个测试的时候被执行。JMockit覆盖使用一个不同的，但是相关的度量集合。代码覆盖包含了一系列软件度量方式，从而告诉你调有多少的生产代码被一个测试套件锁覆盖。这是相当具有度量性， 而不仅仅是说生产代码或者是测试代码的质量。也就是说，代码覆盖检测有时候能够发现那些可以删除的不可到达的代码。更重要的是，这些报告能够被当做发现缺少测试的引导。这不仅对于编写已经存在的生产代码的测试有帮助，而且对于首先编写测试，例如TDD的开发模式。（测试驱动开发）

JMockit 覆盖工具提供三种不同完备的代码覆盖度量方式：行覆盖，路径覆盖和数据覆盖。一个关于所有度量的覆盖的示例报告可以在线找到。

行覆盖

行覆盖测试度量告诉我们在一个已经被测试的源码文件中有多少执行代码被覆盖。每一个可执行的代码行可以不被覆盖，覆盖或者是部分覆盖。在第一个情况中，没有任何的可执行代码被执行。在第二个情况中，所有的代码被完整的最少执行一次。第三种情况，只有部分的

部分的可执行代码行被执行。例如这通常发生在代码行中包括多个逻辑条件在复杂的条件表达式中。JMockit 覆盖工具能够标识出所有的三种情况，计算每一个可以执行行的覆盖比例：0% 表示未覆盖行, 100% 表示覆盖的行, 或者一些值在之间标识部分覆盖行。

一个分支节点存在于程序包含两个可以执行路径上。任何包含逻辑条件的代码行都能被至少分为两个可执行的段，每一个段属于一个独立的分支。一个没有包含分支节点的代码中包含一个独立段。包含一个或者多个的分支节点的代码中包含两个或者是更多可执行的代码段，在行上被连续的分支节点所分开。

在一个给定行中我们称 NS >= 1 表示可执行的代码段数目。如果 NE 使我们在测试中至少执行一次的段数目。(也就说是 他们是被覆盖的段),然后我们能够使用 100 * NE / NS来计算出被覆盖的代码段比率。

同样的，针对整个源文件的行覆盖比率可以通过计算总的可执行的段和总的被覆盖的段，考虑到所有可执行的在文件中代码行。同样的，一个包的比率可以通过总的和覆盖的在包中源码文件的覆盖段数目进行计算。最后总的代码覆盖比率可以通过使用通用的公式应用到所有的包中。

路径覆盖

一个完全不同的度量方式是路径覆盖，是通过计算方法和构造函数，而不是代码的行或者是段数目。它告诉我们多少个可能被执行的路径通过方法或构造函数在一个测试运行中至少被执行过一次，从入口到结束。

注意每一个方法或构造函数都包含一个独立的入口节点，但是却又多个退出。一个退出发生在一个返回或者是异常抛出语句。同样的包括普通的退出。一个方法或者是构造函数的执行可能被抛出一个调用异常，一个对空的引用，或者是一些其他的能够导致未可知的程序错误的情况下被终结退出。

每一个可能的路径可以被执行（覆盖）或者是不执行（覆盖）。路径可能被部分执行通过被简单的认为未覆盖（例如， 他们被终结退出）。

针对一个方法或者是构造函数体的路径覆盖比率可以通过一个简单的同行覆盖类似的计算方式获得。如果 NP是可能的通过实现体的路径，NPE 表示从开始到结束时的执行路径数目，度量可以通过 100 * NPE / NP进行计算。 与在相同的行覆盖度量中一样的，我们将这个公式拓展到所有文件，整个包中，测试运行所覆盖到的整个包集合中。

数据覆盖

通过度量有多少个实例和静态非final的属性字段被测试所使用。在整个执行周期内，一个属性必须至少包含最后一次被读取所赋的值。这个比率通过100 * NFE / NF公式进行计算, 其中 NF 表示非final属性的数目，而 NFE 表示的是完全使用过的属性数目。

覆盖输出的类型

JMockit 覆盖工具可以生成如下类型的输出:

1. HTML 报告: 多页的HTML报告被写入"coverage-report" 路径下, 在当前的工作目录下(如果需要可以顶一个不同的新的输出目录)。 如果目录不存在将会被新建;如果之前已经被生成内容将会被覆盖。报告中包括了测试用例所覆盖的所有java远吗。通常来说工具会查找存在src命名的文件下的所有的以.java为后缀的源码，或者是直接在当前工作目录下; 任何在src和顶级包目录之间的子目录都将会被会搜索。
2. 覆盖数据文件: 单个命名为"coverage.ser"序列化的文件将会被写入到当前的工作目录下或者是一个特定的输出目录。如果这个文件已经存在，它的内容通过定义既可以重写或者是追加到当前测试运行时的内存结果。这些文件能够被读取或者是处理通过外部的工具。 mockit.coverage.data.CoverageData.readDataFromFile(文件) 方法将会建立一个新的覆盖数据实例包含在指定的序列化文件中的所有的可用覆盖数据。 关于更多的内容，可以通过查看 保存在jmockit-coverage.jar中的API 文档。

调用节点

当使用覆盖工具运行测试套件的时候，一个可选的调用节点信息能够被可以被用户选择性收集。一个调用节点指定是测试指定生产代码行中的测试代码点。

使用这些附加的信息生成覆盖信息将会消耗更多的时间和产生更大的输出; 在另一方面来说, 它将有助于了解在给定生产代码被测试用例运行时那些行被执行。当这些内容被包含在 HTML 报告中, 调用节点列表首次将会被隐藏，但是可以通过简单的店家每一个可执行的代码行进行查看。

配置覆盖工具

为了启用 JMockit 覆盖工具在 JUnit/TestNG 测试运行时, 添加 jmockit.jar 和 jmockit-coverage.jar 到运行时的类路径下。使用 JUnit, 确保 jmockit.jar 出现在路径之前。 (使用JMocikt运行测试用例的细节，可以查看相关的入门教程。)

当没有使用JMockit mocking API的时候， 代码覆盖仍然能够不需要添加jar的方式激活，只需要在运行时候添加 "-javaagent:/jmockit-coverage.jar" 作为 JVM 初始化的参数。

在大多数的例子中，代码覆盖工具不需要额外的配置进行使用。然后有一些工具的工作方式能够被配置。这些可以通过设定一个或者是多个"jmockit-coverage-xyz" 系统属性来进行JVM实例配置运行测试套件。为了方便可以将 "jmockit-" 前缀忽略，因此"coverage-xyz"也是可用的。

注意你可以很容易在 Ant target, 或者是 Maven surefire plugin 配置, 或是 在IDE中的test run 配置属性, 无论是 JUnit 或是 TestNG; 不需要特定的 JMockit 插件。

可用的配置属性包括：

1. [jmockit-]coverage-output: 一个或者是多个用逗号分隔的html, html-nocp ("nocp" 达标分调用点), 序列化, 和 序列化添加 , 这些被选定在测试运行的最终结果输出。缺省如果不配置将输出基础的HTML 报告 (html-nocp)。 "html" 和 "html-nocp" 值是互斥的，正如序列化和序列化添加那样。然而可以同时包含每个中的一个。在这种情况下，运行测试之后就会输出两种格式。 使用 "serial" 或是 "serial-append" 将会生成一个命名为"coverage.ser";在 "serial-append"的情况下, 当前测试用例覆盖收集到的数据将会被添加到之前已经存在的数据文件中(如果文件不存在，效果和"serial"一样)。
2. [jmockit-]coverage-outputDir: 绝对或者是相对的路径针对输出的目录，被用来输出 "coverage.ser" 或者是 "index.html" 文件 (加上 ".html"后缀的html报告文件将自动被生成的子目录下)。 缺省情况下，当前的被使用JVM工作目录将被使用，所有的 以".html"后缀的 html报告文件将被生成到"coverage-report"子目录下。 [jmockit-]coverage-srcDirs: 当生成html报告的时候，逗号分隔的java源文件夹目录将被搜索。(这些通常与序列化数据文件无关)每一个目录将被定义为一个绝对的或者是相对的路径。如果没有定义路径，所有的在当前工作路径下的src目录将被搜索。
3. [jmockit-]coverage-classes: 无论是类似系统的正则表达式 (使用传统的 *""* 和 *"?"* 通配符*),* 或者是一个 *java.util.regex-conformable* 正则表达式。 这给定的表达式将会被用来选择类*(*通过全限定名称*)* 从需要被覆盖测试的产品代码中。缺省情况下，所有的生产代码中的类将在测试的时候被载入，在*jar*文件中的类不被考虑。 例如*"some.package.*" 选择所有的在some.package 或者是 任何子 包中的类。 作为一个特殊的例子，如果属性被定义为 "loaded", 则所有的类将会被考虑，而不仅仅是只有在JVM运行测试是载入的类; 代码库中的部分不被载入的类将会被忽略。这对于只包含较少的基于代码库的子集合测试的情况，非常有用。
4. [jmockit-]coverage-excludes: 和之前的属性类似，只是定义代码覆盖测试时候不包含的类名。这个属性可以和 coverage-classes 或者是 它自身相结合使用。 缺省情况下，将没有类被排除。
5. [jmockit-]coverage-metrics: 一个或是多个逗号分隔的在行(默认), 路径，数据或者是所有的选项，用来表示代码覆盖中所有要收集的度量类型。
6. [jmockit-]coverage-check: 一个或者是多个分好分隔的规则用来定义在测试之后的最小的覆盖测试检测。缺省的奖没有任何检测将被使用。具体的可以查看检测最小的覆盖章节。

聚集多个测试结果报告

当覆盖工具在一个测试运行之后生成一个报告，它通常会覆盖之前的报告。通常来说，在测试报告中的覆盖数据只会被当前测试所收集的数据影响。现在假定你包含多个测试套件或者是多个运行时的测试配置，你想聚集所有的测试集合的结果到一个html报告中。这就是 "coverage.ser" 序列化数据文件排上用场的时候。

为了激活生成这些文件，我们简单的设定输出系统属性包含 "serial" 或 "serial-append"。正如这两个值所代表的，有多种不同的方式用来组合多分覆盖数据文件。以下的子章节将会详细的介绍每一种情况。

多个数据文件生成聚集报告

假如我们想要聚合多个测试运行结果合并到一个 HTML 报告中。每一个测试需要生成自己的 coverage.ser 文件, 之后它们才可以在最后一个阶段合并成一个报告; 因此每一个测试运行应当被配置为 "coverage-output=serial"。注意，为了保护之前每一个测试生成的 coverage.ser 输出文件，你需要将它们生成或者是复制到不同的路径中。

假设有两个或者是更多的 coverage.ser 文件在分隔的路径下, 一个聚集的报告可以通过执行 mockit.coverage.CodeCoverage.main 方法 (一个普通的 Java "main" 方法)。为了方便操作, jmockit-coverage.jar 文件可执行的。示例，如下的 Ant 任务可能被使用:

```
<java fork="yes" dir="myBaseDir" jar="jmockit-coverage.jar">
   <jvmarg line="-Djmockit-coverage-output=html"/>
   <arg line="module1-outDir anotherOutDir"/>
</java>
```

上面一个例子中使用了 "myBaseDir" 作为基地址，当分隔的JVM 实例运行的时候。 两个输出目录包含 "coverage.ser" 数据文件被定义，像使用命令行参数那样。其他的配置参数也可以通过 "coverage-xyz" 系统属性进行配置。这些独立的 JVM 实例将会读取每一个 "coverage.ser" 数据文件,合并这些内存中的覆盖数据，在退出之前然后生成一个聚集的 HTML 报告。

从单个数据文件的每次测试之后生成聚集报告

另外一种方法用来获取聚集多个测试运行的覆盖报告是从所有的测试中获取覆盖的数据进行聚集到一个数据文件中。这能够通过使用所有的测试运行在同一个工作目录货值通过指定覆盖的输出路径为共享路径，当使用 coverage-output=serial-append 针对每一个测试运行的时候。此外，最后的测试应该指定 html 或者是 html-nocp 对于输出 coverage-output 属性 和 serial-append。 当然第一个测试用例运行时候不需要从这个文件读取数据；因此在第一个测试运行的时候需要删除文件或者是通过设定第一个测试用例的属性为coverage-output=serial。

总的来说, 关于输出模式 "serial" 和 "serial-append" 的不同是首先在当我们有多个"coverage.ser" 文件的时候(每一个在不容路径的测试使用不同的路径), 然而之后我们将所有的测试共享同一个数据文件。

检测最小的覆盖

如果需要的情况， JMockit Coverage 可以检测最终的覆盖比率在测试运行之后需要满足一个最小值。 这些检测可以通过一个或者是多个检测规则设定在 "coverage-check" 系统属性中(当包含多个的时候，我们需要使用 ";" 字符进行分割)。

每一个检测规则必须以 "[scope:]最小行比率[,最小路径比率[,最小数据比率]]"格式。 包含三种不同的范围：

- 总和: 当没有范围设定的时候的默认值。它指定了对于每一个度量的总比率。例如 规则 80 规定了总的行覆盖必须大于 80%, 没有其他的度量的最小值。一个定义了三种度量的例子是 "70,60,85"。注意0可以定义为没有最小值。
- 每个文件: 定义了每一个文件需要满足的最小比率。如果一个或者是多个文件都以一个很低的比率结束，这个检测就是失败的。例如: "perFile:50,0,40", 意味着每一个源码必须至少 50%以上的行覆盖和至少40%以上的数据覆盖。
- 包: 定义了在给定包情况下的最小需要满足的总比率，包括子包。例如规则"com.important:90,70"定义了总的文件中的行覆盖在"com.important"包下需要至少90%以上的行覆盖, 而总的路径覆盖需要至少70%。 所有的检测都是在测试运行之后进行的(实际上是在虚拟机关闭之后)。其他格式的输出(HTML 报告, 序列化文件) 都不会被影响。当一个独立的检测失败了，一个描述信息将会被打印到标准的输出中。如果一个或者是多个检测失败，两个最终的行动将会被执行: 第一个, 一个空的命名为"coverage.check.failed"文件将会被建立在当前工作目录下;第二,一个错误将会被抛出(定义为 AssertionError)。当所有的检测都被执行通过了当前目录下的"coverage.check.failed"会被删除。

通过一个文件来表明覆盖检测是否成功或者是失败，可以允许构建工具做出对应的操作，典型的是当失败的时候建立一个文件。例如我们可以通过如下的Ant 构建脚步来执行:

```
<fail message="Coverage check failed">
   <condition><available file="coverage.check.failed"/></condition>
</fail>
```

或者是其他的 Maven pom.xml 文件:

```
<plugin>
   <artifactId>maven-enforcer-plugin</artifactId>
   <executions>
      <execution>
         <id>coverage.check</id>
         <goals><goal>enforce</goal></goals>
         <phase>test</phase>
         <configuration>
            <rules>
               <requireFilesDontExist>
                  <files><file>coverage.check.failed</file></files>
               </requireFilesDontExist>
            </rules>
         </configuration>
      </execution>
   </executions>
</plugin>
```

在maven项目中激活覆盖

如果你使用的是 Maven's "test" 目标, 你需要添加以下的依赖在 pom.xml 文件中 (假设 "jmockit.version" 属性被恰当的定义了):

```
<dependency>
   <groupId>org.jmockit</groupId>
   <artifactId>jmockit</artifactId>
   <version>${jmockit.version}</version>
   <scope>test</scope>
</dependency>
<dependency>
   <groupId>org.jmockit</groupId>
   <artifactId>jmockit-coverage</artifactId>
   <version>${jmockit.version}</version>
   <scope>runtime</scope>
</dependency>
```

在 Maven 2/3, surefire插件是通常用来实际上运行测试的。为了配置覆盖工具，需要定义适当的"coverage-xyz" 系统属性。例如, 生成文件的输出路径可以通过 coverage-outputDir 属性进行配置。

```
<plugin>
   <artifactId>maven-surefire-plugin</artifactId>
   <configuration>
      <systemPropertyVariables>
         <coverage-outputDir>target/my-coverage-report</coverage-outputDir>
         <!-- other properties, if needed -->
      </systemPropertyVariables>
   </configuration>
</plugin>
```

最后，如果测试并没有使用 JMockit mocking APIs,仍然可以使用覆盖工具。在这种情况下, 只需要依赖于"jmockit-coverage"。此外还需要如下配置 surefire 插件:

```
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <configuration>
      <argLine>
    -javaagent:"${settings.localRepository}"/org/jmockit/jmockit-coverage/${jmockit.versi
    jmockit-coverage-${jmockit.version}.jar
        <!-- coverage properties, if any are needed -->
      </argLine>
    </configuration>
</plugin>
```

在maven站点中包含html 报告

为了将JMockit Coverage HTML 报告包含在生成的 Maven site 文档中 , src/site/site.xml 描述文件需要被提供, 使用如下的类似的内容。

```
<?xml version="1.0" encoding="UTF-8"?>
<project
   xmlns="http://maven.apache.org/DECORATION/1.3.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/DECORATION/1.3.0
                       http://maven.apache.org/xsd/decoration-1.3.0.xsd">
   <body>
      <menu ref="reports"/>
      <menu>
         <item name="Code Coverage Report" href="../../coverage-report/index.html"/>
      </menu>
   </body>
</project>
```

关闭覆盖输出

有时我们可能需要关闭覆盖输出针对一个特殊的测试运行，而不想通过从classpath 下删除 jar。这可以通过两种不同的方法。

其一，我们可以管理相关输出文件的只读属性，当一个已经生成的时候。这个特殊的文件通常在工作目录下， "coverage.ser" 针对序列化的输出 或者是 "coverage-report/index.html" 针对 HTML 输出。文件的属性将在 JMockit 启动时候被使用; 如果一个只读的文件不能被覆盖，JMockit 将彻底避免尝试。

注意工作路径通常是被独立选择的在 Java IDE的测试配置中。同样的 , Java IDE 通常提供一个简单的机制来切换项目中文件的只读属性: 在 IntelliJ IDEA中，通过双击状态栏，当在编辑器中打开一个文件; 在 Eclipse 中，有一个 "Read only" 复选框在 "Properties" 面板中 (可以通过快捷键 "Alt+Enter" 打开) ，当在编辑器中选中一个文本文件的时候。

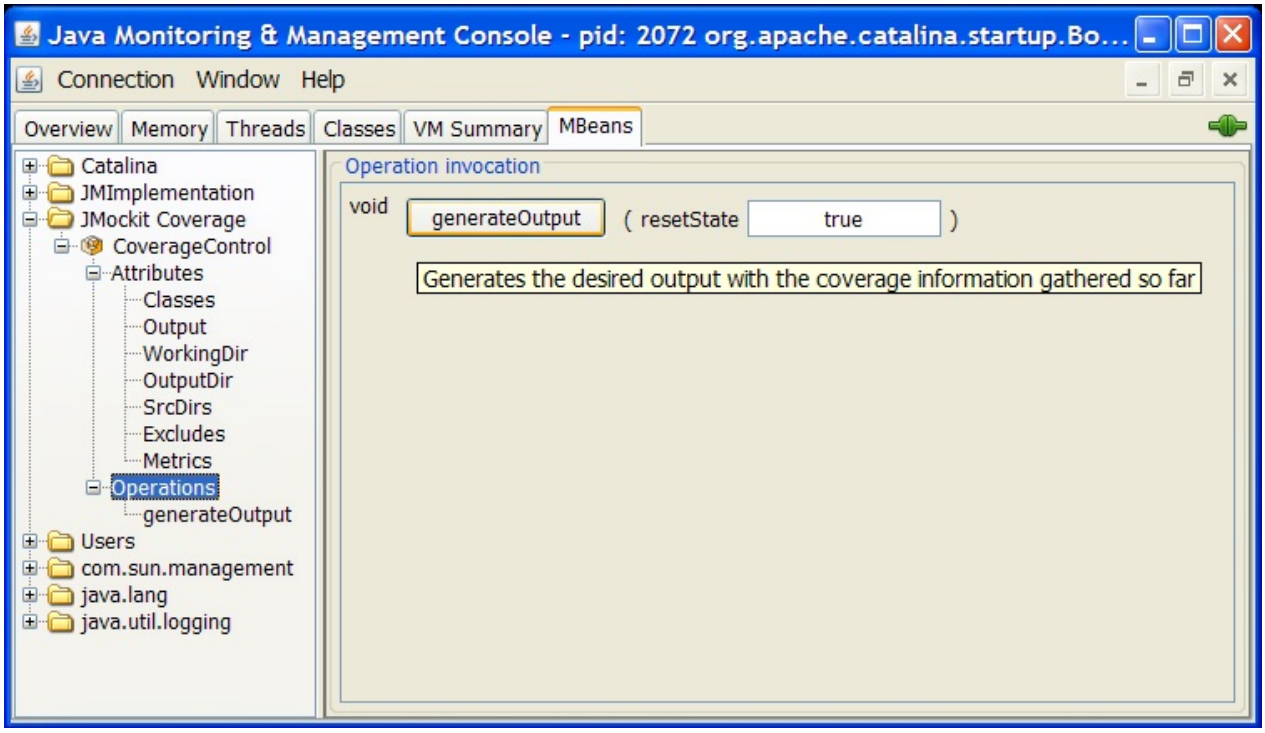
另外一个切换覆盖的开关是通过简单的设定 coverage-output 系统属性为一个未知的输出格式，例如 "-Dcoverage-output=none"。

独立模式

在前面的章节中我们描述了大部分传统的在使用JUnit/TestNG测试用启用测试覆盖度量的方法。这个工具可以被适用于更加通用的上下文中，尽管: 独立模式可以使得其附加到任何的 Java 6+ 的程序中用来度量行和路径覆盖在指定的类中，无论是什么代码调用了这些类。

为了启用独立模式，目标 JVM 实例必须添加 "-javaagent:jmockit-coverage.jar" 命令行参数。就这样就可以了，不需要 JMockit toolkit jars 出现在目标程序的路径下。覆盖工具的初始化的配置设定可以通过使用之前描述的 "coverage-xyz" 系统属性, 但是它完全是可选的; 配置的属性可以通过一个专用的UI在之后进行修改。

一旦目标进程通过 JMockit Coverage Java 客户端启动, 用户可以通过使用 JMX 客户端进行连接操作其中任意的 "MBeans"。通常来说，标准的 JConsole 工具在 Java 6+ JDK 中将会被使用。 JMockit Coverage MBean 提供了一些配置属性 (和那些可以在命令行中使用 "-D" 的一样), 通过操作可以获取想要生成的输出。JConsole 提供的用户界面在下面显示, 例子中运行覆盖工具的进程是 Tomcat 7 服务实例。



通过JMockit Coverage agent启动的用户界面通过JConsole tool展示, 在连接到tomcate实例之后。

配置属性 (通过一个属性的方式显示在 "CoverageControl" MBean 中) 入之前所述, 除了 "SrcDirs" (代表着覆盖的源码路径)。如果这个属性没有被指定，将会找不到需要进行覆盖的源码文件。

如果 MBean UI 没有被使用, 覆盖输出将会在JVM关闭的时候输出,通过覆盖的属性配置。

企业应用

1. 基于场景的测试
2. 一个例子
    i. 使用 Java EE
    ii. 使用 Spring framework

一个针对特定业务对象的企业应用，通常拥有一个可以给并发多个用户使用，一个多种实体类型的应用数据库的用户界面，同时它还集成了其他的应用在组织的内外部。在 Java中, Java EE APIs 或者是 Spring 框架常被用来构建这种应用。

在这个章节中我们将描述一个方法来进行java企业级应用的测试，通过使用基于场景的测试用例。其中每一个测试都定义了一个描述完善的场景（通常作为用例或者是使用场景）。在传统的分层架构中，测试用例是通过在高层的组件中进行公用的调用，（通常是展现层或者是UI层）这些调用将向下调用底层。

基于场景的测试

一个例子

使用 **Java EE**

使用 **Spring framework**