

择：可以不提供参数（`foo()`），可以提供某个值（`foo(6)`），也可以提供一个Map（`foo(name: 'Brad', age: 12)`），还可以提供一个Map和一个值（`foo(name: 'Brad', age: 12, 6)`）。`BuilderSupport`提供了4个版本的`createNode()`，分别对应这4种选择。当在生成器实例上调用方法时，相应的方法会被调用。`setParent()`用于让生成器的作者知道当前所处理节点的父节点。不管`createNode()`返回的是什么，都会被看作一个节点，而且生成器支持将其作为一个参数发送给`nodeCompleted()`。

`BuilderSupport`不会像处理缺失的方法那样处理缺失的属性。然而仍然可以使用`propertyMissing()`方法来处理那些情况。

下面来看一下`TodoBuilderWithSupport`的代码，它扩展了`BuilderSupport`。待办事项列表所选择的格式仅支持无参数的方法调用（和属性）以及接受一个Map的方法调用。所以接受一个`Object`类型参数的`createNode()`版本会抛出一个异常，指示这是一种无效格式。在该方法的另外两个版本，以及`propertyMissing()`方法中，我们会通过增加`level`变量记录嵌套层次。在`nodeCompleted()`方法中要减少`level`，因为当退出一个嵌套层次时才会调用这个方法。`createNode()`方法返回所创建节点的名字，所以我们可以将这个名字与`nodeCompleted()`比较，以确定何时退出最顶层的节点`build`。如果需求更为复杂，可以亲自定制表示不同节点的类，并返回这个类的实例。当节点被创建时，如果需要执行一些其他操作，比如将子节点附到父节点上，可以使用`setParent()`来实现。该方法接受的参数是用作父节点和子节点的`Node`实例，也就是创建这些节点时`createNode()`所返回的节点对象。`TodoBuilderWithSupport`中的其余代码用于处理所发现的节点并创建期望的输出。

尝试一下，看看哪些方法被调用了，又是以什么样的顺序调用的。为理解其顺序，可以在这些方法中插入一些`println`语句。

UsingBuilders/TodoBuilderWithSupport.groovy

```
class TodoBuilderWithSupport extends BuilderSupport {
    int level = 0
    def result = new StringWriter()
    void setParent(parent, child) {}

    def createNode(name) {
        if (name == 'build') {
            result << "To-Do:\n"
            'buildnode'
        } else {
            handle(name, [:])
        }
    }
    def createNode(name, Object value) {
        throw new Exception("Invalid format")
    }
}
```

```

def createNode(name, Map attribute) {
    handle(name, attribute)
}
def createNode(name, Map attribute, Object value) {
    throw new Exception("Invalid format")
}

def propertyMissing(String name) {
    handle(name, [:])
    level--
}

void nodeCompleted(parent, node) {
    level--
    if (node == 'buildnode') {
        println result
    }
}

def handle(String name, attributes) {
    level++
    level.times { result << " "}
    result << placeXifStatusDone(attributes)
    result << name.replaceAll("_", " ")
    result << printParameters(attributes)
    result << "\n"
    name
}

def placeXifStatusDone(attributes) {
    attributes['status'] == 'done' ? "x" : "-"
}

def printParameters(attributes) {
    def values = ""
    if(attributes.size() > 0) {
        values += "["
        def count = 0
        attributes.each { key, value ->
            if (key == 'status') return
            count++
            values += (count > 1 ? " " : "") + "${key}: ${value}"
        }
        values += "]"
    }
    values
}
}

```

我们看到了将公用代码重构到BuilderSupport中的优势，但是还可以利用另一个层次的重构，下一节将会介绍。

17.6 使用FactoryBuilderSupport

如果要处理的是SwingBuilder中的button（按钮）、checkbox（复选框）和label（标签）等明确定义的节点名，将用到FactoryBuilderSupport。上一节介绍的BuilderSupport适合处理层次式结构，然而对于处理不同类型的节点并不方便。假设必须处理20种不同类型的节点，createNode()的实现就会变得非常复杂。基于节点名创建不同的节点，会导致出现大量麻烦的switch语句。我们可能很快就倾向于使用抽象工厂（参见*Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV95]）方式来创建这些节点。FactoryBuilderSupport就是这么做的。基于节点名，将节点的创建委托给不同的工厂。我们只需要将节点名映射到工厂。

FactoryBuilderSupport受到了SwingBuilder的启发，后来SwingBuilder又做了修改，扩展了FactoryBuilderSupport，而不再扩展BuilderSupport。下面看一个实现和使用扩展了FactoryBuilderSupport的生成器的例子。创建一个名为RobotBuilder的生成器，可以将其用于机器人的创建与编程。作为第一步，先考虑一下怎么使用这个生成器：

```
UsingBuilders/UsingFactoryBuilderSupport.groovy
def bldr = new RobotBuilder()

def robot = bldr.robot('iRobot') {
    forward(dist: 20)
    left(rotation: 90)
    forward(speed: 10, duration: 5)
}

robot.go()
```

我们希望RobotBuilder从代码中产生如下输出：

```
Robot iRobot operating...
move distance... 20
turn left... 90 degrees
move distance... 50
```

现在看一下这个生成器。RobotBuilder扩展了FactoryBuilderSupport。在它的实例初始化器中，使用了FactoryBuilderSupport的registerFactory()方法，将节点名robot、forward和left映射到了相应的工厂。RobotBuilder中就这么多了。像遍历节点层次、调用相应工厂这些苦差事，FactoryBuilderSupport都给干了。下面将会看到，其余的细节工作，将由工厂和节点负责：

```
UsingBuilders/UsingFactoryBuilderSupport.groovy
class RobotBuilder extends FactoryBuilderSupport {
}
```

```

registerFactory('robot', new RobotFactory())

registerFactory('forward', new ForwardMoveFactory())

registerFactory('left', new LeftTurnFactory())
};

}

```

下面显示的Robot、ForwardMove和LeftTurn等类分别表示robot、forward和left节点。

UsingBuilders/UsingFactoryBuilderSupport.groovy

```

class Robot {
    String name
    def movements = []

    void go() {
        println "Robot $name operating..."
        movements.each { movement -> println movement }
    }
}

class ForwardMove {
    def dist
    String toString() { "move distance... $dist" }
}

class LeftTurn {
    def rotation
    String toString() { "turn left... $rotation degrees" }
}

```

Robot有一个name属性和一个ArrayList——movements。其go()方法负责遍历每个移动并打印详细信息。其他两个类——ForwardMove 和LeftTurn——各有一个属性。虽然ForwardMove类只有一个名为dist的属性，但是在本节开头的代码中，已经将speed和duration两个属性指派给了left节点。工厂负责处理这些属性，一会将会介绍。

看一下这些工厂。FactoryBuilderSupport依赖于Factory接口。该接口提供了一些方法，分别用于控制节点的创建、处理节点属性的设置、设置节点间的父子关系以及确定该节点是否为叶节点等。Groovy中提供了Factory的一个默认实现，叫做AbstractFactory，如下所示：

```

// AbstractFactory.java代码片段，来自Groovy中的部分实现
public abstract class AbstractFactory implements Factory
{
    public boolean isLeaf() { return false; }

    public boolean onHandleNodeAttributes(FactoryBuilderSupport builder,
                                         Object node, Map attributes ) { return true; }

    public void onNodeCompleted(FactoryBuilderSupport builder,

```

```

        Object parent, Object node ) { }

    public void setParent(FactoryBuilderSupport builder,
        Object parent, Object child ) { }

    public void setChild(FactoryBuilderSupport builder,
        Object parent, Object child ) { }
}

```

`isLeaf()`的默认实现会返回`false`, 说明该节点可以有一个处理子节点的闭包。

`onHandle NodeAttributes()`之中适合对属性执行任何特殊的处理, 像`left`中的`duration`和`speed`。在这个方法内, 我们将从`attributes`中去除任何已经处理过的属性。如果像默认实现中那样返回`true`, `FactoryBuilderSupport`会将`attributes`中找到的剩余属性填充到节点实例中去。`onNode Completed()`方法会在节点处理完成时调用, 而且在节点创建结束时可以执行一些最终操作。`setParent()`会在子节点的工厂上调用, 所以可以设置任何父子关系。类似地, `setChild()`会在父节点的工厂上调用。`AbstractFactory`中唯一没有提供的`Factory`中的方法是`newInstance()`, 它负责实例化实际的节点。

这个例子中, `Robot`、`ForwardMove`和`LeftTurn`分别需要一个工厂, 它们对应的`RobotFactory`、`ForwardMoveFactory`和`LeftTurnFactory`如下:

```

UsingBuilders/UsingFactoryBuilderSupport.groovy

class RobotFactory extends AbstractFactory {
    def newInstance(FactoryBuilderSupport builder, name, value, Map attributes ) {
        new Robot(name: value)
    }

    void setChild(FactoryBuilderSupport builder, Object parent, Object child) {
        parent.movements << child
    }
}

class ForwardMoveFactory extends AbstractFactory {
    boolean isLeaf() { true }
    def newInstance(FactoryBuilderSupport builder, name, value, Map attributes ) {
        new ForwardMove()
    }

    boolean onHandleNodeAttributes(FactoryBuilderSupport builder,
        Object node, Map attributes) {
        if (attributes.speed && attributes.duration) {
            node.dist = attributes.speed * attributes.duration
            attributes.remove('speed')
            attributes.remove('duration')
        }
    }
}

```

```

        true
    }
}

class LeftTurnFactory extends AbstractFactory {
    boolean isLeaf() { true }

    def newInstance(FactoryBuilderSupport builder, name, value, Map attributes) {
        new LeftTurn()
    }
}

```

每个工厂的newInstance()方法负责实例化相应节点。在RobotFactory的setChild()方法中，向Robot的movements列表中添加了移动节点。因为forward和left是叶节点，其工厂中的isLeaf()方法会返回true。ForwardMoveFactory的onHandleNodeAttributes()中提供了对forward节点的特殊属性的支持。

花一分钟看一下isLeaf()方法带来的好处。在下面的例子中，我们向forward节点提供了一个闭包：

UsingBuilders/UsingFactoryBuilderSupport.groovy

```

def robotBldr = new RobotBuilder()
robotBldr.robot('bRobot') {
    forward(dist: 20) { }
}

```

FactoryBuilderSupport类意识到forward节点不能有嵌套的层次，于是作出拒绝，如下所示：

```
java.lang.RuntimeException: 'forward' doesn't support nesting.
```

要实现处理多个明确定义的节点的生成器，使用FactoryBuilderSupport要比使用BuilderSupport清晰很多。FactoryBuilderSupport还提供了其他便捷方法，可以拦截节点创建的生命周期，因此，如果需要的话，可以对节点遍历施加更多控制，比如可以使用preInstantiate()方法在工厂创建节点之前执行一些动作，或者是覆盖postNodeCompletion()方法，在节点处理完毕之后执行一些动作。如果需要在构建时执行其他任务，可以使用诸如FactoryBuilderSupport的getCurrentNode()和getParentNode()等便捷方法，轻松地处理正在创建的层次式结构。关于生成器及其API的更多细节，参见<http://groovy.codehaus.org/FactoryBuilderSupport>和<http://groovy.codehaus.org/api/groovy/util/FactoryBuilderSupport.html>。

本章介绍了如何使用Groovy的生成器。对于一些枯燥的任务，比如创建XML或HTML文档，生成器提供了用于执行它们的DSL语法。可以使用Groovy提供的生成器，也可以亲自创建生成器。而且如果我们创建了一个有用的生成器，应该考虑将其贡献给社区！

我们意识并体会到了Groovy之强大。Groovy的动态特性需要我们的自律——对代码进行自动化测试非常重要。下一章将探索如何编写单元测试。

第 18 章

单元测试与模拟

18

单元测试对于元编程至关重要。不管静态类型语言的编译器执行的类型检查多么弱，动态类型语言甚至连这种程度的支持都没有。这就是动态语言中单元测试存在的必要原因了。（参见*Test Driven Development: By Example* [Bec02]、*Pragmatic Unit Testing in Java with JUnit* [TH03]和*JUnit Recipes: Practical Methods for Programmer Testing* [Rai04]。）尽管在动态语言中可以轻松地利用动态能力和元编程，但我们必须花些时间，确保程序的表现符合我们的预期，而不仅仅是符合我们的输入——要知道输入也会犯错。

开发者对单元测试的认识已经较几年前有所进步，遗憾的是，其应用却还不够。单元测试类似于对软件进行锻炼，它能够改进代码的健康度。这一点大部分开发者都同意，然而他们总有种种不做单元测试的借口。

单元测试不只是Groovy编程的关键，在Groovy中进行单元测试也很容易，而且充满乐趣。JUnit内置到了Groovy中。元编程能力使得创建模拟对象非常容易。Groovy还有一个内置的模拟库。接下来看一下如何使用Groovy对Java和Groovy应用进行单元测试。

18.1 本书代码与自动化单元测试

我所介绍的单元测试并不是抽象的建议。本书中的全部代码都使用了自动化单元测试。这是因为我在使用的是一门一直在演进的语言，Groovy的特性会改变，实现也会改变，bug会被修复，新特性会加入进来，变化不一而足。就在写作这些章节、编写代码示例时，我机器上安装的Groovy更新了好多次。如果某次更新后，因为某个特性或实现的变更，一个代码示例无法工作了，我就得及时发现并理解其原因，不能花掉太多精力。此外，随着写作的进行，我重构了书中的一些例子。有了自动化单元测试，我知道在经历了语言更新和我自己的重构之后，书中的例子仍然能够工作，这样我就睡得更踏实了。

在编写了最初的一些示例不久之后，我决定休息一下，想办法将所有示例的测试自动化，同时保持其独立性，并且可以放在隔离的文件中。有些示例是函数，还有一些是独立的程序或脚本。有了Groovy的元编程能力，结合ExpandoMetaClass以及加载并执行脚本的能力，创建和执行自动化单元测试轻而易举。

我花了几个小时才知道该怎么做。每当我写完一个新的示例，我就花最多二分钟的时间为这个示例写好测试。没几天，投入的精力和时间就初见成效了，后来效果又好了几倍。随着我更新Groovy，有些示例会运行失败。更重要的是，这些测试能够确保其他的例子正常工作，而且是有效的。

这些测试至少给我带来了5个方面的好处：

- 促进了我对Groovy特性的理解；
- 在Groovy用户邮件列表中提出的问题帮助修复了一些Groovy的bug；
- 帮助找到并修复了Groovy文档中的一处不一致；
- 帮我确保所有的示例都是有效的，而且在最新版的Groovy中可以很好地工作；
- 让我有勇气随意、随时重构任何示例，而且让我可以充满自信地说：我的重构会改进代码的结构，但是不会影响预期行为。

18.2 对 Java 和 Groovy 代码执行单元测试

因为出色的Java-Groovy集成，任何基于Java的测试框架和模拟对象框架，如EasyMock、JMock和Mockito等，都可以结合Groovy使用。而且还不止这些，在安装Groovy时，已经自动获得了一个构建于JUnit之上的单元测试框架。可以使用它来测试Java虚拟机上的任何代码，包括Java代码、Groovy代码，等等。只需要从GroovyTestCase扩展出自己的测试类，并实现测试方法，然后准备运行测试就可以了。

单元测试必须满足FAIR条件

在编写单元测试时，要记住测试必须满足FAIR条件，这些条件是快速（fast）、自动化（automated）、隔离（isolated）和可重复（repeatable）。

测试必须要快。随着代码的演进和重构，需要快速得到代码仍然满足预期的反馈。如果测试很慢，开发者势必不愿费劲运行它们。因此需要一个非常快速的编辑和运行周期。

测试必须是自动化的。手工测试很累人，而且容易出错，也相当于减少了投入在重要任务上的时间。自动化测试就像立在我们肩上的天使，写代码的时候它们会安静地注视；如果代码不符合预期，它们会在我们的耳畔低语。它们会在代码开始走向崩溃的早期提供反馈。有一点可能都认同，我们宁可听计算机说我们的代码很糟糕，也不愿意听到同事这么说。自动化测试会让我们看上去很优秀，而且可以信赖。当我们说干完了的时候，我们知道代码会按预期方式工作。

测试必须隔离。如果碰到1031个编译错误，通常问题就是少了个分号，是吧？这没有实质性的帮助，一个小错误级联到一堆报告的错误是毫无意义的。我们想要的是隐藏的bug或错误与失败的测试用例之间的直接关联。这才能帮助我们快速找到并修复问题，而不是用大

量的失败测试把我们吓倒。隔离确保了一个测试不会遗留下可能会影响另一个测试的残余状态，从而就可以以任何顺序运行测试，可以运行所有测试、单个测试或是选定的一些测试。

测试必须是可以重复的。测试一定要能够运行任意多次，并且得到的是确定性的、可预测的结果。如果这次运行失败，未做任何修改，下次运行就通过了，这种测试是最坏的。线程原因可能会导致一些这种问题。再举一个例子，如果一个测试是向数据库中插入数据，而且存在唯一的列约束，那随后再运行同样的测试而不清理数据库，则测试会失败。不过如果该测试会回滚事务，这种情况就不会出现，该测试就是可重复的。测试的可重复性对于在快速演进代码的同时保持清醒非常关键。

先编写一个简单的测试：

```
UnitTestingWithGroovy/ListTest.groovy
class ListTest extends GroovyTestCase {
    void testListSize() {
        def lst = [1, 2]
        assertEquals "ArrayList size must be 2", 2, lst.size()
    }
}
```

即使Groovy是动态类型的，JUnit还是期望测试方法的返回类型为void。这意味着在定义测试方法时，必须显式地使用void代替def。Groovy的可选类型在这起到了帮助。要运行前面的代码，只需要像执行任何Groovy程序那样执行它。输入下列命令：

```
groovy ListTest
```

输出如下：

```
Time: 0.006
```

```
OK (1 test)
```

如果熟悉JUnit，这里的输出也就理解了——一项测试成功执行。

如果喜欢看到红绿条，可以在支持运行测试的集成开发环境（IDE）内运行单元测试。

也可以调用junit.swingui.TestRunner的run()方法，并将Groovy测试类的名字传给它，这样运行测试就能在Swing GUI内看到红绿条了。

可以使用JUnit中我们所熟悉的任何assert方法。为方便使用，Groovy添加了更多assert方法，且举几种，有assertArrayEquals()、assertLength()、assertContains()、assertToString()、assertInspect()、assertScript()和shouldFail()等。

在编写单元测试时，考虑编写3种类型的测试：正面测试、负面测试和异常测试。正面测试

可以帮助确定代码的表现符合预期。可以在正常的路径上调用这种测试。比如，在一个账户中存入100美元，然后检查余额是不是多了这么多。负面测试检查的是，代码能否按预期方式处理前置条件失效、无效输入等问题。存入一个负值，看看代码会做什么。如果账户关闭，又会怎么样呢？异常测试可以帮助确定的是，当异常情况出现时，代码是否会抛出正确的异常，以及表现是否符合预期。如果账户关闭后，自动取款操作发起，会怎么样？在这一点上一定要相信我，我遇到过一个“有创意”的银行就出现过这种场景。像这样的情况可以受益于异常测试。

以这些术语来思考测试，有助于把实现的逻辑彻底想清楚。我们不仅要处理实现逻辑的代码，还要考虑常常会让我们陷入困境的边界条件和极端条件。

利用Groovy和JUnit提供的**assert**方法，可以轻松实现正面测试。实现负面测试和异常测试需要的工作会多一点，Groovy也有可以提供帮助的机制，下一节将会介绍。

即使主项目是用Java写的，也应该考虑用Groovy编写测试代码。因为Groovy是轻量级的，我们会发现，在Groovy中编写测试会更容易、更快、更有乐趣。这也是在以Java为主的项目中实践Groovy的一种不错的方式。

假设在src目录下有一个Java类Car，如以下代码所示。再假设我们已经使用javac将其编译到了classes目录下。

UnitTestingWithGroovy/src/Car.java

```
// Java代码
package com.agiledeveloper;

public class Car
{
    private int miles;
    public int getMiles() { return miles; }
    public void drive(int dist)
    {
        miles += dist;
    }
}
```

可以在Groovy中为这个类编写单元测试，而且不必编译测试代码来运行它。下面是Car类的一些正面测试。这些测试都位于test目录下的CarTest.groovy文件中。

18

UnitTestingWithGroovy/test/CarTest.groovy

```
class CarTest extends GroovyTestCase
{
    def car
    void setUp()
    {
        car = new com.agiledeveloper.Car()
```

```

void testInitialize()
{
    assertEquals 0, car.miles
}
void testDrive()
{
    car.drive(10)
    assertEquals 10, car.miles
}
}

```

`setUp()`方法和相应的`tearDown()`方法（前面的例子中没有显示出来）会将每个测试调用夹在中间。可以在`setUp()`中初始化对象，可以根据需要在`tearDown()`中执行清理或复位操作。这2个方法可以帮助避免复制代码，同时可以帮助将测试彼此隔离起来。

要运行该测试，请输入`groovy -classpath classes test/CarTest`命令。应该会看到如下输出：

```
Time: 0.003
```

```
OK (2 tests)
```

输出表明两项测试都执行了，而且不出意外，均成功通过。第一项测试确认了该`Car`一开始里程表上的公里数为0；驾驶一定的距离，里程表上会增加相应的公里数。现在编写一个负面测试：

```

void testDriveNegativeInput()
{
    car.drive(-10)
    assertEquals 0, car.miles
}

```

`drive()`的参数使用了负值——`-10`。我们判断，这种情况下`Car`一定会忽略我们的驾驶请求，所以预计里程数不会改变。然而Java代码没有处理这一条件，没有检查输入参数就修改了里程数。运行之前的测试，会出现一项失效：

```

...F
Time: 0.004
There was 1 failure:
1) testDriveNegativeInput(CarTest)
   junit.framework.AssertionFailedError:
   expected:<0> but was:<-10>
...
FAILURES!!!
Tests run: 3, Failures: 1, Errors: 0

```

输出表明，两个正面测试通过，但是负面测试失败了。可以修复Java代码来正确地处理这种情况。可见，使用Groovy测试Java代码相当直接，而且简单。

18.3 测试异常

现在看一下编写异常测试。可以将要测试的方法包在try-catch中。如果该方法抛出了预期的异常，也就是说进入了catch块，则一切正常。

如果代码没有抛出任何异常，会调用fail()来说明测试失败，如下所示：

UnitTestingWithGroovy/ExpectException.groovy

```
try {
    divide(2, 0)
    fail "Expected ArithmeticException ..."
} catch(ArithmeticException ex) {
    assertTrue true // 成功
}
```

前面的代码是Java风格的JUnit测试，在Groovy中也可以使用。不过，Groovy提供了一个shouldFail()方法，它可以优雅地将样板代码包起来，这使得编写异常测试更容易了。使用它来编写一个异常测试：

UnitTestingWithGroovy/ExpectException.groovy

```
shouldFail { divide(2, 0) }
```

shouldFail()方法接受一个闭包。它会在一个看守式的try-catch块中调用该闭包。如果没有抛出异常，它会通过调用fail()方法抛出一个异常。如果意在于捕获一个具体的异常，可以向shouldFail()方法指明这一信息：

UnitTestingWithGroovy/ExpectException.groovy

```
shouldFail(ArithmeticException) { divide(2, 0) }
```

在这个例子中，shouldFail()期望闭包抛出ArithmeticException。如果代码抛出了ArithmeticException，或者抛出的是扩展了该异常的某些异常类，测试正常通过。如果抛出的是其他某些异常，或者没有抛出异常，则shouldFail()失败。我们可以利用Groovy中括号的灵活性（参见19.9节），将前面的调用写成下面这样：

UnitTestingWithGroovy/ExpectException.groovy

```
shouldFail ArithmeticException, { divide(2, 0) }
```

18.4 模拟

要对存在依赖的大片代码进行单元测试，就算并非不可能，也是非常困难的。（怎么算大呢？在编辑器内，如果不往下滚动就没法看全，这就算大了——嘿，别忙着去调小字体哦。）单

元测试有一个优点，它会强制我们让代码单元小一些。代码越小，内聚性就越高。它还会强制我们将代码与其周边环境解耦，也就意味着降低耦合性。高内聚和低耦合是优秀设计的特质。本节将探讨处理依赖的不同方式，后面几节则将探讨存在依赖时的单元测试。

耦合有两种形式：有的代码会依赖我们的代码，有的代码是我们的代码所依赖的。在对我们的代码进行单元测试之前，需要解决这两种耦合。

必须将被测代码从其所在的应用中分离或解耦出来。假设有逻辑位于GUI内按钮的事件处理器中，这种逻辑很难进行单元测试。因此，为进行单元测试，必须将其代码分离到一个方法中。

再假设有逻辑严重依赖某个资源。那个资源可能响应很慢，使用代价高昂，不可预测，或者目前正处于开发之中。因此，在有效地进行单元测试之前，必须将这种依赖从代码中分离出去。这可以借助存根和模拟来实现。

存根与模拟的对比

存根用于代替真正的对象。当被测方法调用存根时，它会根据设定好的预期响应简单地应答。之所以要设置响应，是为了满足测试通过的需要。模拟对象所做的事情要比存根多得多。它可以帮助确定被测代码按预期方式与其依赖或协作对象交互。模拟对象可以记录代码中在该对象所代表的协作对象上进行的方法调用的次序和次数。它可以确保传递给方法调用的是正确的参数。存根验证状态，而模拟验证行为。当在测试中使用模拟时，它不仅验证状态，也验证代码与其依赖的交互行为^①。

Groovy同时支持创建存根和模拟，18.10节将会介绍。

我们的代码所依赖的代码被称作协作者，我们的代码与协作者合作完成其工作。协作者可以是一个组件、一个对象、一个层次或一个子系统。它可能是局部的，可能位于我们的对象内部，可能是作为参数出入的，甚至可能是远程的。没有协作者，我们的对象就无法行使其功能。然而为了满足测试需求，需要替换掉协作者。

模拟用于代替协作者，它不做任何真正的工作，而只是简单地向调用它的代码做出预期响应，以便让测试工作。

当运行应用时，我们希望代码依赖的是它所需要的真正对象（协作者）。进行集成测试时也是如此。然而在进行单元测试时，我们希望代码依赖模拟。因此，需要想个办法，通过开关控制我们的代码依赖模拟，还是依赖真正的对象。

^① Martin Fowler在“Mocks Aren’t Stubs”（模拟并非存根）一文中探讨了存根与模拟的不同，参见<http://martinfowler.com/articles/mocksArentStubs.html>。

在像Java这样的静态类型语言中，可以使用接口实现，如图18-1所示。Java中的框架（如EasyMock、JMock和Mockito等）简化了模拟，其中有一些甚至支持在不创建接口的条件下实现模拟。使用一种基于代理的机制，它们会拦截调用，并将请求路由给模拟对象，而不是真正的依赖对象。

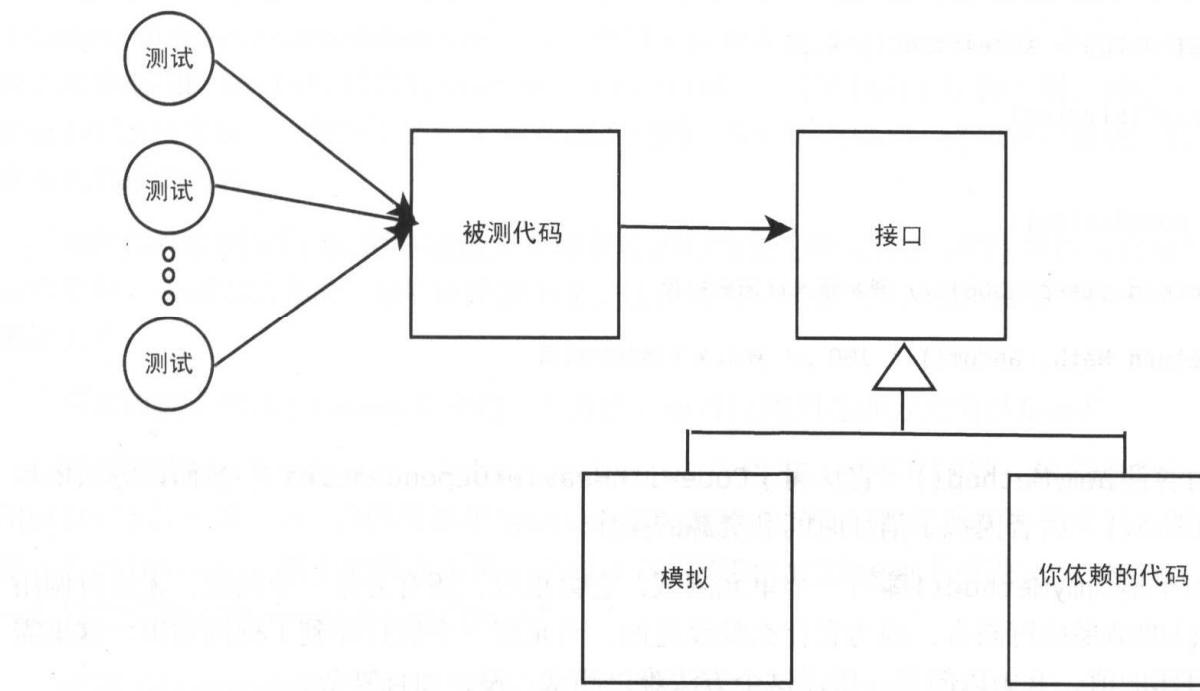


图18-1 单元测试中的模拟

Groovy的动态特性与元编程能力在这方面提供了很大的优势。在Groovy中有很多种创建模拟的方式。我们可以使用下列特性：

- 方法覆盖
- 分类
- ExpandoMetaClass
- Expando
- Map
- Groovy的Mock Library

下面几节将探讨在Groovy中创建和使用模拟的技术。

18.5 使用覆盖实现模拟

假设有一个类，它依赖一个做某个重要工作的方法，而这个方法在执行时要消耗大量的时间和资源，比如下面的myMethod()：

UnitTestingWithGroovy/com/agiledeveloper/CodeWithHeavierDependencies.groovy

```

package com.agiledeveloper

public class CodeWithHeavierDependencies
{
    public void myMethod()
    {
        def value = someAction() + 10

        println(value)
    }

    int someAction()
    {
        Thread.sleep(5000) // 模拟消耗时间的动作

        return Math.random() * 100 // 模拟某个动作的结果
    }
}

```

现在打算测试myMethod()（它从属于CodeWithHeavierDependencies）。然而该方法依赖于someAction()，后者模拟了消耗时间和资源的操作。

如果简单地为myMethod()编写一个单元测试，它会很慢。还有另外一个问题，无法对调用myMethod()的结果执行断言，因为它什么都没返回，只是将一个值打印到了标准输出。这里需要捕获它打印的值，并施以断言。所以这个方法难以测试：慢，而且复杂。

为解决这些问题，下面覆盖这个讨厌的方法：

UnitTestingWithGroovy/TestByOverriding.groovy

```

import com.agiledeveloper.CodeWithHeavierDependencies

class TestCodeWithHeavierDependenciesUsingOverriding extends GroovyTestCase {
    void testMyMethod() {
        def testObj = new CodeWithHeavierDependenciesExt()

        testObj.myMethod()

        assertEquals 35, testObj.result
    }
}

class CodeWithHeavierDependenciesExt extends CodeWithHeavierDependencies {
    def result

    int someAction() { 25 }

    def println(text) { result = text }
}

```

运行这段代码，并确保测试快速通过：

```
Time: 0.015
```

```
OK (1 test)
```

这段代码中创建了一个用于模拟的新类——`CodeWithHeavierDependenciesExt`，它扩展了`CodeWithHeavierDependencies`。新类模拟了`someAction`和`println()`方法。（Groovy的习惯是简单地用`println()`代替`System.out.println()`，这里利用了这种习惯，提供了一个局部的`println()`实现——明白了吧？）运行测试代码，看看能否成功。这次没有延迟，标准输出也没有乱糟糟的内容。

现在仍然在测试行为，但是通过将非确定性的行为变为确定性的，就能够针对它编写断言了。必须想出一个聪明的办法，将依赖模拟出来，这样就可以将精力集中到对我们代码的行为做单元测试上了。

前面的例子测试了Groovy类中的一个方法，也可以使用这种方式测试Java类。

通过覆盖Java类中的方法（如`someAction()`）来模拟，也不是问题。然而和Groovy代码调用`println()`不同，Java代码要调用`System.out.println()`。因此，在用于模拟的派生类中创建一个`println()`，起不到什么帮助。不过可以扩展`PrintStream`并替换`System.out`。下面看一个与之前测试的Groovy类等价的Java类：

UnitTestingWithGroovy/com/agiledveloper/JavaCodeWithHeavierDependencies.java

```
package com.agiledveloper;

public class JavaCodeWithHeavierDependencies
{
    public int someAction()
    {
        try
        {
            Thread.sleep(5000); // 模拟消耗时间的动作
        }
        catch(InterruptedException ex) {}

        return (int) (Math.random() * 100); // 模拟某个动作的结果
    }

    public void myMethod()
    {
        int value = someAction() + 10;

        System.out.println(value);
    }
}
```

用于测试上述Java代码的Groovy代码如下：

```
UnitTestingWithGroovy/TestJavaByOverride.groovy
import com.agiledveloper.JavaCodeWithHeavierDependencies

class TestCodeWithHeavierDependenciesUsingOverriding extends GroovyTestCase {
    void testMyMethod() {
        def testObj = new ExtendedJavaCode()

        def originalPrintStream = System.out
        def printMock = new PrintMock()
        System.out = printMock

        try {
            testObj.myMethod()
        } finally { System.out = originalPrintStream }

        assertEquals 35, printMock.result
    }
}

class ExtendedJavaCode extends JavaCodeWithHeavierDependencies {
    int someAction() { 25 }
}

class PrintMock extends PrintStream {
    PrintMock() { super(System.out) }

    def result

    void println(int text) { result = text }
}
```

输出符合预期，测试通过：

Time: 0.026

OK (1 test)

被测试的myMethod()方法是JavaCodeWithHeavierDependencies类的一部分。这里创建了ExtendedJavaCode类，它扩展了JavaCodeWithHeavierDependencies，并覆盖了someAction()方法。此外还创建了一个扩展了PrintStream的PrintMock类，并将System.out指派给了这个类的实例。这可以帮助拦截对System.out.println()调用，并直接转向我们的模拟实现。

18.6 使用分类实现模拟

13.1节探讨了分类可以在Groovy中提供可控的AOP。本节将介绍如何使用分类实现模拟。

UnitTestingWithGroovy/TestUsingCategories.groovy

```

import com.agiledeveloper.CodeWithHeavierDependencies

class TestUsingCategories extends GroovyTestCase {
    void testMyMethod() {
        def testObj = new CodeWithHeavierDependencies()

        use(MockHelper) {
            testObj.myMethod()

            assertEquals 35, MockHelper.result
        }
    }
}

class MockHelper {
    def static result

    def static println(self, text) { result = text }

    def static someAction(CodeWithHeavierDependencies self) { 25 }
}

```

MockHelper有两个静态方法，分别对应我们想模拟的方法——someAction()和println()。在这个测试内，通过使用use(MockHelper)，让分类来拦截对方法的调用，并在适当的情况下替换这两个方法。这很像AOP中使用的建议（Advice）。

这段代码让人放心地通过了测试，如下所示：

Time: 0.027

OK (1 test)

分类仅在Groovy代码中才有用，无助于模拟在编译好的Java代码中调用的方法。

上一节介绍的覆盖方式在Java和Groovy代码中均可使用。然而，如果被测试的类是final的，这种方式就无法使用了。而分类方式可以应用于那种情况。

18.7 使用 ExpandoMetaClass 实现模拟

在Groovy中还可以以另外一种方式拦截方法调用，也就是使用ExpandoMetaClass（参见13.2节和13.3节）。第13章中的两节介绍的两种方式都不需要创建一个单独的类。但是，可以为每个想要模拟的方法创建一个闭包，然后将其设置到被测试实例的MetaClass中。来看一个例子。

为被测试的实例创建一个ExpandoMetaClass实例。这个MetaClass将提供协作方法的模拟实现。

下面这段代码中，为模拟 `println()` 创建了一个闭包，并将其设置到了用于 `ClassWithHeavierDependencies` 的一个 `ExpandoMetaClass` 实例中，见第7行。类似地，这里也为模拟 `someAction()` 创建了一个闭包，见第8行。专门为被测试的实例创建一个 `ExpandoMetaClass` 实例，其优点是，不会对 `CodeWithHeavierDependencies` 的元类造成全局的影响。也就是说，如果还有其他测试，这里模拟的方法不会影响它们（别忘了，要保持测试彼此隔离）。

UnitTestingWithGroovy/TestUsingExpandoMetaClass.groovy

```

Line 1 import com.agiledeveloper.CodeWithHeavierDependencies

- class TestUsingExpandoMetaClass extends GroovyTestCase {
- void testMyMethod() {
5   def result
- def emc = new ExpandoMetaClass(CodeWithHeavierDependencies, true)
- emc.println = { text -> result = text }
- emc.someAction = { -> 25 }
- emc.initialize()
10
- def testObj = new CodeWithHeavierDependencies()
- testObj.metaClass = emc

- testObj.myMethod()
15
- assertEquals 35, result
- }
- }

```

输出证实测试通过：

Time: 0.031

OK (1 test)

在这个例子中，当 `myMethod()` 调用 `println()` 和 `someAction()` 这 2 个方法时，`ExpandoMetaClass` 会拦截这些调用，并将其路由到模拟实现。再次提醒，这与 AOP 中的建议很类似。

创建模拟，设定预期，然后将其应用于测试中，这些步骤在前面的例子中都很好地包含了。没有创建额外的类。如果有其他测试，可以简洁地创建必要的模拟，以满足那些测试的需要。

通过 `ExpandoMetaClass` 实现模拟的这种方式，只能在 Groovy 代码中使用；对于在编译好的 Java 代码内调用的方法，则没什么帮助。

18.8 使用 Exando 实现模拟

到目前为止，本章介绍的都是如何模拟在另一个实例方法内调用到的实例方法，接下来将介绍如何模拟我们的代码所依赖的其他对象。

先来看一个例子。假设想要测试某个类中的方法，这些方法依赖一个文件（`File`）。这使得单元测试很难编写。为了实现快速、自动化的单元测试，需要想办法模拟这个文件对象：

UnitTestingWithGroovy/com/agiledeveloper/ClassWithDependency.groovy

```
package com.agiledeveloper

public class ClassWithDependency
{
    def methodA(val, file)
    {
        file.write "The value is ${val}."
    }
    def methodB(val)
    {
        def file = new java.io.FileWriter("output.txt")
        file.write "The value is ${val}."
    }
    def methodC(val)
    {
        def file = new java.io.FileWriter("output.txt")
        file.write "The value is ${val}."
        file.close()
    }
}
```

代码中有三个方法，存在不同形式的依赖。`methodA()`接受一个实例，看上去像`File`。其他2个方法——`methodB()`和`methodC()`，则在内部实例化了一个`FileWriter`实例。`Exando`类只能帮助处理第一个方法。考虑到这一点，本节将只探讨`methodA()`的测试。至于如何测试另2个方法，则将在18.10节介绍。

`methodA()`使用`File`的`write()`方法向给定文件对象中写入了一条消息。我们想测试`methodA()`，但是不希望真把消息写入物理文件，之后再读回来用于断言。

这里可以利用Groovy的动态类型，因为`methodA()`没有指明参数的类型。因此，可以发送任何对象，只要它具备预期参数的能力，比如说提供了`write()`方法（参见3.4节）。现在就动手吧。创建一个包含`write()`方法的`HandTossedFileMock`类。不必担心真正的`File`类中的所有属性和方法，只需要关心被测试的方法实际调用到的。代码如下：

UnitTestingWithGroovy/TestUsingAHandTossedMock.groovy

```

import com.agiledeveloper.ClassWithDependency
class TestWithExpando extends GroovyTestCase {
    void testMethodA() {
        def testObj = new ClassWithDependency()
        def fileMock = new HandTossedFileMock()
        testObj.methodA(1, fileMock)

        assertEquals "The value is 1.", fileMock.result
    }
}

class HandTossedFileMock {
    def result
    def write(value) { result = value }
}

```

这段代码的输出证实测试通过：

```
Time: 0.015
```

```
OK (1 test)
```

在这段代码中，`HandTossedFileMock`类的模拟实现`write()`，只是简单地将接受的参数保存到一个`result`属性中。用这个模拟类的实例代替真正的`File`，发送给`methodA()`。多亏了动态类型特性，`methodA()`能够非常开心地使用这个模拟实例。

实现并不难，然而如果不弄出一个单独的类，那就更好了。这就该`Expando`大显身手了（参见15.1节）。

只需创建一个`Expando`实例，为其提供一个`text`属性和一个`write()`方法的模拟实现。然后将这个实例传递给`methodA()`。看一下代码：

UnitTestingWithGroovy/TestUsingExpando.groovy

```

import com.agiledeveloper.ClassWithDependency

class TestUsingExpando extends GroovyTestCase {
    void testMethodA() {
        def fileMock = new Expando(text: '', write: { text = it })

        def testObj = new ClassWithDependency()
        testObj.methodA(1, fileMock)
        assertEquals "The value is 1.", fileMock.text
    }
}

```

输出如下：

```
Time: 0.022
```

```
OK (1 test)
```

在前面的两个例子中，调用methodA()方法时都没有创建真正的物理文件。单元测试运行很快，而且测试完毕后我们不用读取或清理任何文件。

当向被测试方法传递依赖的对象时，Expando很有用。然而，如果方法会在内部创建依赖的对象（比如methodB()和methodC()），它就于事无补了。18.10节将解决这类问题。

18.9 使用 Map 实现模拟

上一节介绍了一个使用Expando实现模拟对象的例子，其实也可以使用Map。众所周知，Map中有键和与键关联的值。其中的值可以是对象，甚至可以是闭包。利用这一点，可以使用一个Map来代替协作对象。

将上一节使用Expando的例子，使用Map重写：

UnitTestingWithGroovy/TestUsingMap.groovy

```
import com.agiledeveloper.ClassWithDependency

class TestUsingMap extends GroovyTestCase {
    void testMethodA() {
        def text = ''
        def fileMock = [write : { text = it }]

        def testObj = new ClassWithDependency()
        testObj.methodA(1, fileMock)

        assertEquals "The value is 1.", text
    }
}
```

输出如下：

```
Time: 0.029
```

```
OK (1 test)
```

与Expando类似，当向被测方法传递依赖的对象时，Map也很有用。但如果协作对象是在被测方法内部创建的，它就帮不上什么忙了。下一节会解决这个问题。

18.10 使用 Groovy Mock Library 实现模拟

Groovy Mock Library是在groovy.mock.interceptor包中实现的，用于模拟较深的依赖，

也就是模拟被测方法内创建的协作对象或依赖对象。StubFor和MockFor是负责这一功能的两个类。下面依次看一下。

StubFor和MockFor的意图是像分类那样拦截方法调用(参见18.6节)。然而与分类不同的是,不必为模拟创建单独的类。在使用时,只需要在StubFor或MockFor的实例上引入模拟方法,这些类会负责替换我们要模拟的对象的MetaClass。

18.4节探讨了存根和模拟的不同。下面就先从一个使用StubFor的例子入手,理解存根的优势与不足;之后再使用MockFor,看一下模拟的优势。

18.10.1 使用 StubFor

使用Groovy的StubFor为File类创建存根:

UnitTestingWithGroovy/TestUsingStubFor.groovy

```
Line 1 import com.agiledeveloper.ClassWithDependency

-
- class TestUsingStubFor extends GroovyTestCase {
-     void testMethodB() {
5         def testObj = new ClassWithDependency()

-
-         def fileMock = new groovy.mock.interceptor.StubFor(java.io.FileWriter)
-         def text
-         fileMock.demand.write { text = it.toString() }
10        fileMock.demand.close {}

-
-         fileMock.use {
-             testObj.methodB(1)
-         }
15        assertEquals "The value is 1.", text
-     }
- }
```

在创建StubFor的实例时,首先提供想为其创建存根的类——这里是java.io.FileWriter。之后为write()方法的存根实现创建一个闭包。第12行在该存根上调用了use()方法。此时,它会将FileWriter的MetaClass替换为一个ProxyMetaClass。在所附的闭包内,对FileWriter实例的任何调用都会被路由到该存根。

然而存根和模拟不会帮助拦截对构造器的调用。在上述例子中,FileWriter的构造器被调用了,结果是磁盘上创建了一个名为output.txt的文件。

StubFor帮助我们测试的是,methodB()方法是否创建了一个正常的FileWriter实例,并将期望的内容写到了这个实例中。不过它是有局限性的。它没有测试当关闭文件时,这个方法的表

现是否正常。即使在存根上要求了`close()`方法（参见代码第10行），它也不会检查该方法是不是真被调用了。存根只是简单地代替协作者并验证状态。要验证行为，必须使用模拟，具体而言，就是使用`MockFor`类。

18.10.2 使用 MockFor

对前面的测试代码做一处修改：

```
UnitTestingWithGroovy/TestUsingMockFor.groovy
//def fileMock = new groovy.mock.interceptor.StubFor(java.io.FileWriter)
def fileMock = new groovy.mock.interceptor.MockFor(java.io.FileWriter)
```

将`StubFor`替换为`MockFor`，这是唯一的修改。现在运行测试，输出如下：

```
.F
Time: 0.093
There was 1 failure:
1) testMethod1(TestUsingStubFor)junit.framework.AssertionFailedError:
verify[1]: expected 1..1 call(s) to 'close' but was never called.
```

与存根不同，模拟会指出，纵然代码产生了指定结果，但是其表现与预期不符。也就是说，这段代码没有调用使用`demand`在测试预期中设置的`close()`方法。

`methodC()`所做的事情与`methodB()`相同，但是它调用了`close()`。使用`MockFor`测试这个方法：

```
UnitTestingWithGroovy/TestMethodCUsingMock.groovy
import com.agiledeveloper.ClassWithDependency

class TestMethodCUsingMock extends GroovyTestCase {
    void testMethodC() {
        def testObj = new ClassWithDependency()

        def fileMock = new groovy.mock.interceptor.MockFor(java.io.FileWriter)
        def text
        fileMock.demand.write { text = it.toString() }
        fileMock.demand.close {}
        fileMock.use {
            testObj.methodC(1)
        }
        assertEquals "The value is 1.", text
    }
}
```

在这种情况下，模拟会告诉我们，与协作者的合作非常愉快。测试通过，输出如下：

```
Time: 0.088
OK (1 test)
```

前面几个例子中，被测方法仅创建了一个被模拟对象FileWriter的实例。如果该方法会创建多个实例，又会怎么样呢？这个模拟就会代表所有实例，但是必须为每个实例创建预期要求，也就是要通过demand有所体现。来看一个使用了两个FileWriter实例的例子。下列代码中的useFiles()方法将给定参数复制到第一个文件中，并将参数的大小写到第二个文件中：

```
class TwoFileUser {
    def useFiles(str) {
        def file1 = new java.io.FileWriter("output1.txt")
        def file2 = new java.io.FileWriter("output2.txt")
        file1.write str
        file2.write str.size()
        file1.close()
        file2.close()
    }
}
```

相应的测试代码如下：

```
UnitTestingWithGroovy/TwoFileUserTest.groovy
class TwoFileUserTest extends GroovyTestCase {
    void testUseFiles() {
        def testObj = new TwoFileUser()
        def testData = 'Multi Files'
        def fileMock = new groovy.mock.interceptor.MockFor(java.io.FileWriter)
        fileMock.demand.write() { assertEquals testData, it }
        fileMock.demand.write() { assertEquals testData.size(), it }
        fileMock.demand.close(2..2) {}
        fileMock.use {
            testObj.useFiles(testData)
        }
    }
    void tearDown() {
        new File('output1.txt').delete()
        new File('output2.txt').delete()
    }
}
```

运行这个测试，输出如下：

```
UnitTestingWithGroovy/TwoFileUserTest.output
```

```
Time: 0.091
```

```
OK (1 test)
```

这里创建的测试预期要求要由被测方法中创建的两个对象共同满足。模拟可以灵活地支持多个对象。当然，如果有大量的对象需要创建，模拟实现起来也很困难。可以利用下文将探讨的指

定多次调用功能。

对于模拟`FileWriter`类的方法，`MockFor`游刃有余；但是对于构造器，它却无法阻止其运行。因此，非常遗憾，在运行测试时，会有两个名字分别为`output1.txt`和`output2.txt`的空文件被创建出来。`tearDown()`方法对此做了清理。

模拟记录了一个方法被调用的次序和次数，如果被测代码没有严格满足所要求的测试预期，模拟将抛出一个异常，致使测试失败。

对于同一方法的多次调用，可以轻松地设定测试预期。这有一个例子：

```
def someWriter() {
    def file = new FileWriter('output.txt')
    file.write("one")
    file.write("two")
    file.write(3)
    file.flush()
    file.write(file.getEncoding())
    file.close()
}
```

假设只想测试我们的代码与协作者之间的交互，需要为如下操作设定测试预期：依次调用3次`write()`，1次`flush()`，1次`getEncoding()`，1次`write()`，最后再调用1次`close()`。

可以在`demand`中使用范围来指定一个调用的基数或重数。比如`mock.demand.write(2..4){...}`的意思是，预期该方法至少被调用2次，但最多4次。可以用这种方式为前面的方法编写一个测试，看看表达对多次调用和返回值的预期，以及对接受到的参数值是否符合预期施加断言是多么容易。

```
void testSomeWriter() {
    def fileMock = new groovy.mock.interceptor.MockFor(java.io.FileWriter)
    fileMock.demand.write(3..3) {} // 如果想表达最多3次，使用0..3
    fileMock.demand.flush {}
    fileMock.demand.getEncoding { return "whatever" } // return是可选的
    fileMock.demand.write { assertEquals 'whatever', it.toString() }
    fileMock.demand.close {}

    fileMock.use {
        testObj.someWriter()
    }
}
```

在这个例子中，模拟会断言`write()`被调用3次；然而它未能对传入的参数施加断言。可以修改其代码，以断言参数，如下所示：

```
def params = ['one', 'two', 3]
def index = 0
fileMock.demand.write(3..3) { assert it == params[index++] }
// 如果想表达最多3次，使用0..3
```

本章介绍了Groovy库中内置的单元测试和模拟设施。某些强大和漂亮的第三方库使单元测试变得更容易、更有趣味了。对于模拟，可以看看gmock^①。测试工具Spock带来了更高的流畅度，编写和表达单元测试也更容易了^②。在Spock中，可以像expect: expected == expression这样简单地编写断言，可以提供一个包含输入值和预期值的数据表格，轻松地创建模拟。

单元测试需要很大的自律，然而收益大于成本。与静态类型语言相比，动态类型语言提供了更大的灵活性，所以单元测试非常关键。

本节介绍了使用存根和模拟管理依赖的技术。可以使用Groovy对Java代码进行单元测试。可以使用现有的单元测试和模拟框架，并且通过覆盖方法来模拟Groovy和Java代码。要对Groovy代码进行单元测试，可以使用分类和ExpandoMetaClass。二者都支持通过拦截方法调用实现模拟。而且如果使用ExpandoMetaClass，我们不必创建额外的类，测试会很简洁。

对于参数对象的简单模拟，可以使用Map或Expando。如果想对多个方法设置测试预期，以及想模拟被测方法内部的依赖，可以使用StubFor。要测试状态以及行为，则可以使用MockFor。

我们看到了Groovy的动态特性结合其元编程能力，是如何使单元测试变成一件乐事的。随着代码的演进、重构，以及我们对应用需求理解的加深，使用Groovy进行单元测试，可以帮助我们保持较快的开发速度。因为单元测试让我们有这种自信——我们的应用会继续满足预期。随着应用开发复杂性的提升，可以将单元测试当作一个安全钩。

① <http://code.google.com/p/gmock/>

② <http://code.google.com/p/spock>

在Groovy中创建DSL

19

领域特定语言（Domain-Specific Language, DSL）针对的是“某一特定类型的问题”，可以参考附录A中引用的Martin Fowler有关DSL的讨论。其语法聚焦于指定的领域或问题。我们不会像使用Java、Groovy或C++等语言那样将其应用于一般性编程任务，因为DSL的应用领域和能力都非常有限。

一门DSL会很小，很简单（尽管设计起来可能并不简单），很有表现力，聚焦于一个问题区域或领域。DSL有两大特点：上下文驱动，非常流畅。

DSL由来已久。很有可能我们已经接触过它了，比如在与外部应用通信时使用的某种特殊的关键字输入文件。Ant和Gant（参见附录A）也是DSL的例子。具体来说，Gant是一个Ant的包装器，它使用Groovy代替了XML，用于指定构建任务。

Groovy的动态特性与元编程能力，使其对于构建DSL很有吸引力。本章将探讨DSL，以及如何使用Groovy构建DSL。

19.1 上下文

上下文（Context）是DSL的特点之一。上下文也是人类赖以交流的基础，人之所以能够高效交流，上下文在谈话中所提供的连续性功不可没。前几天，听到我的朋友Neal喊道：“Venti latte with two extra shots！”^①。他这里用的就是星巴克的DSL。他完全没提到咖啡，但毫无疑问，他会得到一份，而且是价格较高的。这就是上下文驱动。

下面看看订购比萨的Java代码。这段代码缺乏上下文。引用joesPizza会被重复使用：

CreatingDSLs/OrderPizza.java

```
//Java代码  
package com.agiledeveloper;
```

19

^①超大杯拿铁再加两份浓缩咖啡。星巴克的饮品按容量划分为中杯（Tall）、大杯（Grande）和超大杯（Venti）三种杯型。——译者注

```

public class OrderPizza {
    public static void main(String[] args) {
        PizzaShop joesPizza = new PizzaShop();
        joesPizza.setSize(Size.LARGE);
        joesPizza.setCrust(Crust.THIN);
        joesPizza.setTopping("Olives", "Onions", "Bell Pepper");
        joesPizza.setAddress("101 Main St., ...");
        int time = joesPizza.setCard(CardType.VISA, "1234-1234-1234-1234");
        System.out.printf("Pizza will arrive in %d minutes\n", time);
    }
}

```

因为有了with()方法(参见7.1节),使用Groovy编写的同功能代码就没这么杂乱:

CreatingDSLs/OrderPizza.groovy

```

import com.agiledeveloper.*

PizzaShop joesPizza = new PizzaShop()
joesPizza.with {
    setSize(Size.LARGE)
    setCrust(Crust.THIN)
    setTopping("Olives", "Onions", "Bell Pepper")
    setAddress("101 Main St., ...")
    int time = setCard(CardType.VISA, "1234-1234-1234-1234")
    printf("Pizza will arrive in %d minutes\n", time)
}

```

因为在Groovy中类型是可选的,括号也几乎总是可选的(参见19.9节),所以前面的代码可以更轻巧一些:

CreatingDSLs/OrderPizza2.groovy

```

import com.agiledeveloper.*

PizzaShop joesPizza = new PizzaShop()
joesPizza.with {
    setSize Size.LARGE
    setCrust Crust.THIN
    setTopping "Olives", "Onions", "Bell Pepper"
    setAddress "101 Main St., ..."
    time = setCard(CardType.VISA, "1234-1234-1234-1234")
    printf "Pizza will arrive in %d minutes\n", time
}

```

上下文使一切变得更紧凑了(好的方面),也减少了混乱,提升了实际效果。

19.2 流畅

流畅是DSL的另一特点。它改进了代码的可读性,同时使代码自然地流动。其实要在设计上

实现流畅并不容易，但我们应该让用户使用起来很流畅。下面将探讨一些与流畅有关的例子，并探索几种在Groovy中编写循环的方式：

CreatingDSLs/FluentLoops.groovy

```
// 传统循环
for(int i = 0; i < 10; i++) {
    println(i);
}
// Groovy方式
for(i in 0..9) { println i }

0.upto(9) { println it }

10.times { println it }
```

上述所有循环都会产生同样的结果。在众多特性之中，Groovy为循环提供了流畅性。然而，流畅性并非Groovy专有。在Java中，EasyMock（Groovy的模拟库就是受其启发）在设置模拟的预期表现时就表现出了流畅性：

```
// Java代码
expect(alarm.raise()).andReturn(true);
expect(alarm.raise()).andThrow(new InvalidStateException());
```

这段代码表明，在第一个调用上，alarm模拟对象应该返回true，而在第二个调用上则是抛出一个异常。

在Grails/GORM中可以找到DSL的另一个很好的例子。比如，使用下面的Groovy语法，可以在一个对象的属性上指定数据约束条件：

```
class State
{
    String twoLetterCode
    static constraints = {
        twoLetterCode unique: true, blank: false, size: 2..2
    }
}
```

对于表达约束的这种流畅而且表现力很好的语法，Grails能够聪明地识别，进而为前端和后端生成验证逻辑。

Groovy生成器（参见第17章）是DSL的很好的例子，它们非常流畅，而且是基于上下文构建的。

19.3 DSL 的分类

19

在设计一门DSL时，必须确定要设计的是哪种类型的DSL：外部的还是内部的。

外部DSL定义了一门新语言。我们可以灵活地选择语法，之后是解析新语言中的命令并执行动作。当我刚开始做我的第一份工作时，公司让我维护一门DSL，它需要大量使用lex和yacc。

(起初我还以为,之所以让我做,是因为我比较优秀。后来才明白,只是没有人愿意做而已。)解析真是充满“乐趣”!可以使用诸如C++和Java等语言,使用它们提供的解析功能及库,以及利用它们提供的大量解析功能和库的支持来处理繁重的任务。比如可以使用ANTLR来构建DSL(参见Terence Parr的*The Definitive ANTLR Reference: Building Domain-Specific Languages*[Par07])。

内部的DSL,也称作嵌入式DSL,同样定义了一门语言,但是语法受到现有语言的约束。不必使用任何解析器,但必须巧妙地将语法映射到底层语言中的方法和属性等构造。内部DSL的用户可能意识不到自己正在使用的是一门更广的语言的语法。然而,为了让底层的语言为我们工作,创建内部的DSL需要在设计上付出巨大的努力,而且还需要一些聪明的技巧。

前面提到过Ant和Gant:前者使用的是XML,这是外部DSL的一个例子;而后者则使用Groovy来解决同样的问题,但它是内部DSL的一个例子。

19.4 设计内部的DSL

动态语言很适合设计和实现内部的DSL。这些语言提供了很好的元编程能力和灵活的语法,便于轻松地加载和执行代码片段。

然而并非所有的动态语言都是一样的。

比如,我发现在Ruby中创建DSL非常容易。Ruby是动态类型的,括号可选,可以用冒号代替双引号引用字符串,等等。Ruby的优雅为创建内部DSL提供了极大的支持。

在Python中创建DSL就面临一些挑战。空白在Python中是有意义的,这可能会成为创建DSL的障碍。

Groovy的动态类型和元编程能力对于创建内部的DSL帮助很大。然而,如果挑剔的话,Groovy对括号的处理,以及它没有Ruby提供的优雅的冒号符号,都是其不足。对于这些限制,必须采取一些变通措施,后面将会介绍。

设计一门内部DSL,需要耗费很多时间,也需要付出极多的耐心与精力。要想成功,一定要有创造力,巧妙地处理问题,并且愿意做出妥协。

19.5 Groovy与DSL

Groovy的很多核心功能对创建内部DSL很有帮助,包括:

- 动态类型与可选类型(参见3.5节);
- 动态加载、操纵和执行脚本的灵活性(参见10.8节);
- 因为分类和ExpandoMetaClass,可以打开类(参见第13章);
- 闭包为执行提供了很好的上下文(参见第4章);
- 操作符重载有助于自由地定义操作符(参见2.8节);

- 生成器支持（参见第17章）；
- 灵活的括号。

Groovy对括号的灵活处理，让人喜忧参半。调用需要参数的方法时，Groovy不要求括号；但如果调用的方法没有参数，则括号不可或缺。19.9节介绍了一个简单的技巧，用于解决这一烦恼。

本章接下来将介绍一个在Groovy中使用这些功能创建DSL的例子。

19.6 使用命令链接特性改进流畅性

利用Groovy支持将命令或方法调用链接起来的特性，可以实现一定程度的流畅性。当调用的方法需要参数时，Groovy不要求使用括号。此外，如果方法会返回一个结果，不使用点符号（.），就能在返回的这个实例上进行连续的调用。使用简单的、最普通的Groovy代码，不需要元编程的魔力，就可以像下面这样执行流畅的代码：

CreatingDSLs/CommandChain.groovy

```
move forward and then turn left
jump fast, forward and then turn right
```

看上去像个数据文件，但这是百分之百可以执行的Groovy代码。下面分析这段代码，并确定它执行所需的其余Groovy代码。

第一行没有一个逗号。Groovy将读取move和forward，并假定我们是在调用一个move()方法，forward是一个参数。我们定义一个move()方法，同时提供一个名为forward的变量，其中保存着我们期望的值，如forward。在Groovy处理完前两个单词后，它期待有一个可以调用and()方法的对象。为促成这一点，可以从move()方法返回一个支持and()方法的对象。因为第二行使用了一个逗号，jump()方法将接收两个参数。可以继续分析，以确定需要的方法、变量和参数。要处理前面像数据一样的代码，需要像下面这样创建一堆变量和方法：

CreatingDSLs/CommandChain.groovy

```
def (forward, left, then, fast, right) =
['forward', 'left', '', 'fast', 'right']

def move(dir) {
    println "moving $dir"
    this
}

def and(then) { this }

def turn(dir) {
    println "turning $dir"
    this
}
```

```
def jump(speed, dir) {
    println "jumping $speed and $dir"
    this
}
```

定义所需变量时使用了多赋值。`move()`、`and()`、`turn()`和`jump()`均返回`this`，也就是调用这些方法的对象。这使将方法在这两行中漂亮地链接起来成为可能。

将前面的代码与那两行流畅的文字放到一个文件中，使用`groovy`命令执行，输出如下：

```
moving forward
turning left
jumping fast and forward
turning right
```

Groovy中的命令链接特性，使得创建相当流畅的简单DSL非常容易。要创建更复杂的DSL，并在一个上下文内执行它们，还需要其他能力，下一节将予以介绍。

19.7 闭包与DSL

`with()`方法可以在一个闭包内辅助实现委托调用，并提供一个执行上下文。可以利用这种方式创建自己的方法，兼具上下文和流畅性。

再来看一下订购比萨的例子。假如想创建一种自然流动的语法，但是不想创建一个`PizzaShop`实例，因为这样就太偏于实现细节了。我们希望上下文是隐式的。看看下面的代码（下一节将介绍如何让这段代码更流畅，更符合上下文驱动风格）：

```
CreatingDSLs/ClosureHelp.groovy
time = getPizza {
    setSize Size.LARGE
    setCrust Crust.THIN
    setTopping "Olives", "Onions", "Bell Pepper"
    setAddress "101 Main St., ..."
    setCard(CardType.VISA, "1234-1234-1234-1234")
}

printf "Pizza will arrive in %d minutes\n", time
```

`getPizza()`方法接受一个闭包，闭包内使用了`PizzaShop`类的实例方法来实现订购比萨功能。不过这里的`PizzaShop`实例是隐式的。`delegate`（参见4.9节）负责将方法路由到隐式的实例，这一点在下面`getPizza()`方法的实现中可以看到：

```
CreatingDSLs/ClosureHelp.groovy
def getPizza(closure) {
    PizzaShop pizzaShop = new PizzaShop()
    closure.delegate = pizzaShop
```

```

closure()
}

```

执行调用getPizza()的代码，输出如下：

```
Pizza will arrive in 25 minutes
```

稍等一下，输出中打印的time的值是怎么得到的？因为getPizza()中的最后一条语句是调用闭包，所以闭包返回什么，这个方法就返回什么。而闭包内的最后一条语句是setCard()，所以该方法的结果会被返回给调用者。这里DSL强加了顺序信息：在订购比萨时，setCard()必须是最后调用的方法。可以致力于改进接口，让订购更为显而易见。此外，也可以将像setSizeSize.LARGE这样的设置语句修改为像size = Size.LARGE这样的赋值语句。

19.8 方法拦截与 DSL

不使用PizzaShop类，也可以实现订购比萨的DSL，比如可以完全靠拦截方法调用来实现。下面从订购比萨的代码入手（保存在一个名为orderPizza.dsl的文件中）：

Creating DSLs/orderPizza.dsl

```

size large
crust thin
topping Olives, Onions, Bell_Pepper
address "101 Main St., ..."
card visa, '1234-1234-1234-1234'

```

这怎么看都不像代码，倒是更像一个数据文件。然而，这就是纯正的Groovy代码，而且我们正打算执行它（文件中可见的一切，除了双引号中的字符串，其他不是方法名就是变量名）。但是在此之前，还必须玩些技巧，也就是设计自己的DSL。

首先创建一个名为GroovyPizzaDSL.groovy的文件，在其中定义large、thin和visa等变量（可以随意定义其他变量，比如small、thick和masterCard等）。现在定义一个acceptOrder()方法，用于调用最终执行DSL的闭包。此外，实现methodMissing()方法，对于不存在的方法，该方法会被调用（在DSL文件orderPizza.dsl中调用的方法几乎都不存在）。

Creating DSLs/GroovyPizzaDSL.groovy

```

def large = 'large'
def thin = 'thin'
def visa = 'Visa'
def Olives = 'Olives'
def Onions = 'Onions'
def Bell_Pepper = 'Bell Pepper'

orderInfo = [:]
def methodMissing(String name, args) {
    orderInfo[name] = args
}

```

```

}

def acceptOrder(closure) {
    closure.delegate = this
    closure()
    println "Validation and processing performed here for order received:"
    orderInfo.each { key, value ->
        println "${key} -> ${value.join(', ')}"
    }
}

```

必须想办法把这两个脚本放到一起执行。这可以非常简单地实现（参见10.8节），如下所示。调用GroovyShell，加载前面的两个脚本，将它们聚合到一起，形成一个脚本，然后计算处理。

CreatingDSLs/GroovyPizzaOrderProcess.groovy

```

def dslDef = new File('GroovyPizzaDSL.groovy').text
def dsl = new File('orderPizza.dsl').text

def script = """
${dslDef}
acceptOrder {
    ${dsl}
}
"""

new GroovyShell().evaluate(script)

```

前面代码的输出如下：

```

Validation and processing performed here for order received:
size -> large
crust -> thin
topping -> Olives, Onions, Bell Pepper
address -> 101 Main St., ...
card -> Visa, 1234-1234-1234-1234

```

可见，如果知道如何利用Groovy的MOP能力，在Groovy中设计与执行DSL相当容易（就像我们在orderpizza.dsl中所做的那样）。

19.9 括号的限制与变通方案

先放下比萨的例子，来看一个简单的计数器。计数器是一个支持计算总数的设备，本节将演示如何为一个简单的计数器创建一门DSL。下面是第一次尝试：

CreatingDSLs/Total.groovy

```

value = 0
def clear() { value = 0 }
def add(number) { value += number }

```

```
def total() { println "Total is $value" }

clear()
add 2
add 5
add 7
total()
```

前面代码的输出如下：

```
Total is 14
```

这段代码中写的是total()和clear()，而不是total和clear。下面去掉括号，尝试调用total：

CreatingDSLs/Total.groovy

```
try {
    total
} catch(Exception ex) {
    println ex
}
```

执行这段代码，得到如下结果：

```
groovy.lang.MissingPropertyException:
No such property: total for class: Total
```

Groovy认为对total的调用引用了一个（不存在的）属性。使用一门语言来设计DSL就像陪两岁的孩子玩耍，当孩子发脾气时，不要和他争，得让他点。因此，在这种情况下，告诉Groovy一切正常，然后把问题处理掉。简单地创建它想要的属性即可：

```
value = 0
def getClear() { value = 0 }
def add(number) { value += number }
def getTotal() { println "Total is $value" }
```

通过编写getTotal()和getClear()方法，实现了名为total和clear的属性。现在Groovy会非常高兴（像个孩子一样）地跟我们玩了，我们也可以不用括号调用这些属性了：

```
clear
add 2
add 5
add 7
total
clear
total
```

输出如下：

```
Total is 14
Total is 0
```

前面介绍了创建流畅的语法的不同方式，下面将探讨如何拦截和合成DSL中的方法调用。

19.10 分类与DSL

使用分类，可以以可控的方式拦截方法调用（参见13.1节）。在创建DSL时也可以使用分类。现在想办法实现下面这种流畅的调用：`2.days.ago.at(4.30)`。

`2`是一个`Integer`实例，而`days`不是这个实例上的属性。这里就使用分类将其注入为一个属性（对应`getDays()`方法）。`days`在这里看就是噪音，但是在需要区分五天以前还是五分钟以前的另一个上下文中，可能就是有用的了。在`two days ago at 4.30`这个句子中，它起到的是连接作用。可以将`getDays()`方法实现为接受`Integer`并返回接受的对象。`getAgo()`方法对应`ago`属性，它接受一个`Integer`实例，使用`Calendar`上的操作，根据当前的日期计算`Integer`实例所指的天数之前的日期，然后将这个日期返回。最后，`at()`方法将该日期上的时间设置为参数（`4.30`）指定的时间，然后返回一个`Date`实例。这一切可以都在`use()`块内执行，如下面代码所示。（这里没有在作为参数提供的时间上执行错误检查，如果喜欢，可以发送`4.70`来代替`5:10`；这是一个没有写入文档的特性。此外，我们可能希望复制调用`at()`方法的`Calendar`实例，以避免任何副作用。）

```
CreatingDSLs/DSLUsingCategory.groovy
class DateUtil {
    static int getDays(Integer self) { self }

    static Calendar getAgo(Integer self) {
        def date = Calendar.instance
        date.add(Calendar.DAY_OF_MONTH, -self)
        date
    }

    static Date at(Calendar self, Double time) {
        def hour = (int)(time.doubleValue())
        def minute = (int)(Math.round((time.doubleValue() - hour) * 100))
        self.set(Calendar.HOUR_OF_DAY, hour)
        self.set(Calendar.MINUTE, minute)
        self.set(Calendar.SECOND, 0)
        self.time
    }
}

use(DateUtil) {
    println 2.days.ago.at(4.30)
}
```

前面代码的输出如下：

Thu Jan 31 04:30:00 MST 2008

这里创建的DSL语法还有最后一个问题是：用的是`2.days.ago.at(4.30)`。使用`4:30`来代替

4.30会更自然，因此最好是修改为使用`2.days.ago.at(4:30)`。Groovy可以接受Map作为方法的一个参数。

通过将`at()`方法的参数定义为Map，而非Double，可以实现上面的需求：

CreatingDSLs/DSLUsingCategory2.groovy

```
class DateUtil {
    static int getDays(Integer self) { self }

    static Calendar getAgo(Integer self) {
        def date = Calendar.instance
        date.add(Calendar.DAY_OF_MONTH, -self)
        date
    }

    static Date at(Calendar self, Map time) {
        def hour = 0
        def minute = 0
        time.each {key, value -> hour = key.toInteger()
            minute = value.toInteger()
        }
        self.set(Calendar.HOUR_OF_DAY, hour)
        self.set(Calendar.MINUTE, minute)
        self.set(Calendar.SECOND, 0)
        self.time
    }
}

use(DateUtil) {
    println 2.days.ago.at(4:30)
}
```

这段代码的输出如下：

```
Thu Jan 31 04:30:00 MST 2008
```

分类方式唯一的限制是，只能在`use()`块内使用该DSL。这一限制可能实际上也是优势，因为方法注入是可控的。一旦离开代码块，注入的方法就会被从上下文中丢弃，不再可用，这可能比较理想。下一节将介绍如何使用`ExpandoMetaClass`实现同样的语法。

19.11 ExpandoMetaClass 与 DSL

分类只能应用于`use`块内，而且其效果被限制在了作用域内。如果希望方法注入在整个应用内都有效果，可以使用`ExpandoMetaClass`来代替分类。下面使用`ExpandoMetaClass`实现上一节介绍的DSL语法：

CreatingDSLs/DSLUsingExpandoMetaClass.groovy

```

Integer.metaClass{
    getDays = { ->
        delegate
    }

    getAgo = { ->
        def date = Calendar.instance
        date.add(Calendar.DAY_OF_MONTH, -delegate)
        date
    }
}

Calendar.metaClass.at = { Map time ->
    def hour = 0
    def minute = 0
    time.each {key, value ->
        hour = key.toInteger()
        minute = value.toInteger()
    }

    delegate.set(Calendar.HOUR_OF_DAY, hour)
    delegate.set(Calendar.MINUTE, minute)
    delegate.set(Calendar.SECOND, 0)
    delegate.time
}

println 2.days.ago.at(4:30)

```

这里将想要的方法添加到了Integer类和Calendar类的ExpandoMetaClass中。调用这些流畅的方法，会被路由到我们添加的方法，如下所示：

Fri Feb 03 04:30:00 MST 2012

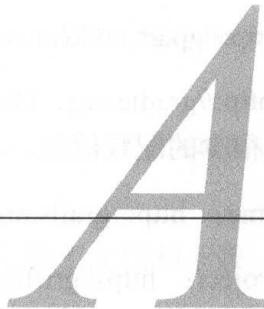
使用ExpandoMetaClass添加方法这种解决方案，要比编写分类所用的静态方法清晰得多。

现在知道了在Groovy中创建内部的DSL相当容易。动态特性和可选类型对于创建流畅的接口帮助很大。闭包可以帮助创建上下文。Groovy的分类和ExpandoMetaClass对于方法调用与属性的注入、拦截和合成也很有帮助。最后，Groovy能够加载和执行任意脚本，这在执行DSL时可以派上用场。

希望本书中对Groovy这一强大的动态语言及其功能的介绍会使您的阅读和学习犹如一段颇为享受的美好旅程。我自己已经将Groovy应用到一切任务中，小到执行自动化例行任务的小型脚本，大到运行企业级应用。Groovy的简洁以及支持与Java轻松集成吸引了我，而由此带来的开发效率的提高留住了我。由衷地希望本书中的概念能够帮助你使用Java虚拟机上的这门强大到不可思议的语言编写可靠的程序，同时获得动态语言的开发效率。祝你一切顺利！

附录 A

Web 资源



A Bit of Groovy History: <http://glaforge.free.fr/weblog/index.php?itemid=99>, Guillaume Laforge谈Groovy历史的一篇博客文章。

API for FactoryBuilderSupport: <http://groovy.codehaus.org/api/groovy/util/FactoryBuilderSupport.html>, FactoryBuilderSupport类的API, 它是SwingBuilder的新基类。

ASTTest Annotation: <http://groovy.codehaus.org/gapi/groovy/transform/ASTTest.html>, 用于测试和调试AST变换的Groovy注解。

CodeNarc: <http://codenarc.sourceforge.net>, CodeNarc是一款基于Groovy的静态代码分析工具。

Crash of the Mars Orbiter: <http://www.cnn.com/TECH/space/9909/30/mars.metric.02>, CNN有关火星轨道探测器坠毁的报道。

Duck Typing.: <http://c2.com/cgi/wiki?DuckTyping>, 什么是鸭子类型。

easyb: <http://www.easyb.org> easyb, 是一款自动测试工具, 支持流畅地进行功能测试和继承测试。

Eclipse Plug-in for Groovy: <http://groovy.codehaus.org/Eclipse+Plugin Eclipse>, IDE中用于支持Groovy开发的插件。

FactoryBuilderSupport : <http://groovy.codehaus.org/FactoryBuilderSupport> , Groovy 的FactoryBuilderSupport类, 它是SwingBuilder的新基类。

Gant Home: <http://gant.codehaus.org>, Gant的网站, 它是一款类似Ant的工具, 不过它使用的是Groovy, 而非XML。

The GDK.: <http://groovy.codehaus.org/groovy-jdk>, 列出了Groovy JDK中的方法。

Getting Started with Grails: <http://www.infoq.com/minibooks/grails>, Jason Rudolph介绍Grails使用的一本书。

Good, Bad, and Ugly of Java Generics: <http://www.agiledeveloper.com/articles/GenericsInJava>