

个数据库表中的信息) Person有几个字段(对应表中的列),比如firstName、lastName和cityOfResidence等,可能随时会加入其他字段。GORM允许Person类的用户调用诸如findByName()、findByName()和findByNameAndLastName()这样的方法,甚至如果age也是Person的一个字段,还可以调用findByNameAndAge()。Person类不会提前创建其中任何一个方法。每个方法都是在运行时第一次被调用的时候合成的。本章接下来将探讨在Groovy中如何合成方法。

在Groovy中,通过实现methodMissing(),可以拦截对不存在的方法的调用。同样,通过实现propertyMissing(),也可以拦截对不存在的属性的访问。在这些方法内,可以动态地为这些不存在的方法或属性实现相应逻辑。我们会基于所定义的约定推断其语义。比如,以find打头的方法名可能意味着查询,而以update打头的方法名则可能意味着保存,依此类推。

下面看一个合成方法的例子。jack是一个无聊的人(Person),只工作不玩耍,我们想把他变成一个多项全能运动员,能够玩各种各样的运动。

14

InjectionAndSynthesisWithMOP/MethodSynthesisUsingMethodMissing.groovy

```
class Person {
    def work() { "working..." }

    def plays = ['Tennis', 'VolleyBall', 'BasketBall']

    def methodMissing(String name, args) {
        System.out.println "methodMissing called for $name"
        def methodInList = plays.find { it == name.split('play')[1] }
        if (methodInList) {
            "playing ${name.split('play')[1]}..."
        } else {
            throw new MissingMethodException(name, Person.class, args)
        }
    }
}

jack = new Person()

println jack.work()
println jack.playTennis()
println jack.playBasketBall()
println jack.playVolleyBall()
println jack.playTennis()

try {
    jack.playPolitics()
} catch(Exception ex) {
    println "Error: " + ex
}
```

这段代码的输出如下：

```
working...
methodMissing called for playTennis
playing Tennis...
methodMissing called for playBasketBall
playing BasketBall...
methodMissing called for playVolleyBall
playing VolleyBall...
methodMissing called for playTennis
playing Tennis...
methodMissing called for playPolitics
Error: groovy.lang.MissingMethodException:
No signature of method: Person.playPolitics()
is applicable for argument types: () values: []
```

`work()`是`Person`上预先定义的唯一一个领域方法。调用`work()`会直接到达该方法。然而，如果调用的是不存在的方法，则会被路由到`methodMissing()`方法。在`methodMissing()`中，如果被调用的方法以`play`开头，并以`plays`数组中的一个名字结尾，就接受它。而且可以动态地修改`plays`数组，以添加我们想要的其他运动，给人留下`jack`又习得了新技能的印象。如果被调用的方法不被支持的，比如`playPolitics()`，就抛出`MissingMethodException`。

从调用者的视角看，调用普通的方法和合成的方法并无二致。

前面的实现极为动态，但是存在一个陷阱。重复调用一个不存在的方法，比如`playTennis()`，每次处理都会带来同样的性能问题。在第一次调用时注入该方法可以提高效率。再次说明，Graeme Rocher称这种技术为“拦截、缓存、调用”模式。我们将在第一次调用时合成方法，将其注入到`MetaClass`中缓存下来，然后调用这一注入的方法。代码如下：

```
InjectionAndSynthesisWithMOP/MethodSynthesisUsingMethodMissing2.groovy
class Person {
    def work() { "working..." }

    def plays = ['Tennis', 'VolleyBall', 'BasketBall']

    def methodMissing(String name, args) {
        System.out.println "methodMissing called for $name"
        def methodInList = plays.find { it == name.split('play')[1] }

        if (methodInList) {
            def impl = { Object[] args ->
                "playing ${name.split('play')[1]}..."
            }

            Person instance = this
            instance.metaClass."$name" = impl //以后再调用就会使用它
        }
    }
}
```

```

    impl(args)
} else {
    throw new MissingMethodException(name, Person.class, args)
}
}

jack = new Person()
println jack.playTennis()
println jack.playTennis()

```

前面代码的输出说明，合成的方法在第一次调用时被缓存了下来：

```

methodMissing called for playTennis
playing Tennis...
playing Tennis...

```

methodMissing()方法仅在第一次调用所支持的不存在方法时会被调用到。第二次调用（以及随后的调用）同一方法时，会直接抵达向实例的MetaClass中注入的实现（闭包）。

14

methodMissing与GroovyInterceptable

对于实现了GroovyInterceptable的对象，调用该对象上的任何方法，都会调用到invokeMethod()。与invokeMethod()不同的是，只有调用不存在的方法时，才会调用到methodMissing()。如果一个对象实现了GroovyInterceptable，不管被调用的方法是否存在，invokeMethod()都会被调用（如果存在的话）。只有对象将控制转移给其MetaClass的invokeMethod()时，methodMissing()才会被调用。

12.2节中使用了GroovyInterceptable来拦截方法调用。也可以将其与methodMissing()混合使用，来拦截对现有的方法和合成的方法的调用，如下所示：

InjectionAndSynthesisWithMOP/InterceptingMissingMethods.groovy

```

class Person implements GroovyInterceptable {
    def work() { "working..." }
    def plays = ['Tennis', 'VolleyBall', 'BasketBall']
    def invokeMethod(String name, args) {
        System.out.println "intercepting call for $name"

        def method = metaClass.getMetaMethod(name, args)

        if (method) {
            method.invoke(this, args)
        } else {
            metaClass.invokeMethod(this, name, args)
        }
    }
}

```

```

}

def methodMissing(String name, args) {
    System.out.println "methodMissing called for $name"
    def methodInList = plays.find { it == name.split('play')[1] }

    if (methodInList) {
        def impl = { Object[] args ->
            "playing ${name.split('play')[1]}..."
        }

        Person instance = this
        instance.metaClass."$name" = impl //以后再调用就会使用它

        impl(args)
    } else {
        throw new MissingMethodException(name, Person.class, args)
    }
}

jack = new Person()
println jack.work()
println jack.playTennis()
println jack.playTennis()

```

这段代码的输出如下：

```

intercepting call for work
working...
intercepting call for playTennis
methodMissing called for playTennis
playing Tennis...
intercepting call for playTennis
playing Tennis...

```

方法合成是Groovy最强大的特性之一。该特性在基于Groovy的库和框架中应用广泛，比如easyb和GORM。我也经常用到该特性，主要是为复杂的业务逻辑处理编写可扩展的代码，一般只需要几行代码。

14.2 使用ExpandoMetaClass合成方法

上一节介绍了如何合成方法。然而，如果我们无权编辑类的源文件，或者该类并非一个POGO，那种方法就行不通了。对于这类情况，可以使用ExpandoMetaClass来合成方法。

12.2节中讲过如何与MetaClass交互。不同于为每个领域方法提供一个拦截器，这里将在MetaClass上实现methodMissing()方法。仍以上一节的Person类（还有无聊的jack）为例，使用ExpandoMetaClass，如下所示：

InjectionAndSynthesisWithMOP/MethodSynthesisUsingEMC.groovy

```

class Person {
    def work() { "working..." }
}

Person.metaClass.methodMissing = { String name, args ->
    def plays = ['Tennis', 'VolleyBall', 'BasketBall']

    System.out.println "methodMissing called for $name"
    def methodInList = plays.find { it == name.split('play')[1] }

    if (methodInList) {
        def impl = { Object[] vargs ->
            "playing ${name.split('play')[1]}..."
        }
        Person.metaClass."$name" = impl //以后再调用就会使用它
        impl(args)
    } else {
        throw new MissingMethodException(name, Person.class, args)
    }
}
jack = new Person()
println jack.work()
println jack.playTennis()
println jack.playTennis()

try {
    jack.playPolitics()
} catch(ex) {
    println ex
}

```

14

前面代码的输出如下：

```

working...
methodMissing called for playTennis
playing Tennis...
playing Tennis...
methodMissing called for playPolitics
groovy.lang.MissingMethodException:
    No signature of method: Person.playPolitics()
    is applicable for argument types: () values: []

```

如果我们的类中也提供了methodMissing()方法，MetaClass的methodMissing()方法会优先被调用。类的MetaClass上的方法会覆盖掉类中的方法。

当在jack上调用work()时，Person的work()被直接执行。然而如果调用的是不存在的方法，则调用会被路由到Person的MetaClass中的methodMissing()。这个方法实现了与上一节的解决方案类似的逻辑。重复调用所支持的不存在方法不会产生开销，这在前面第二次调用playTennis()的输出中可以看到。因为第一次调用的实现已经被缓存了。

12.2节中还使用ExpandoMetaClass的invokeMethod()拦截了方法调用。我们可以混合使用invokeMethod()和methodMissing(), 来拦截对现有的方法和合成的方法的调用, 如下所示:

```
InjectionAndSynthesisWithMOP/MethodSynthesisAndInterceptionUsingEMC.groovy

class Person {
    def work() { "working..." }
}

Person.metaClass.invokeMethod = { String name, args ->
    System.out.println "intercepting call for ${name}"
}

def method = Person.metaClass.getMetaMethod(name, args)

if (method) {
    method.invoke(delegate, args)
} else {
    Person.metaClass.invokeMissingMethod(delegate, name, args)
}
}

Person.metaClass.methodMissing = { String name, args ->
    def plays = ['Tennis', 'VolleyBall', 'BasketBall']

    System.out.println "methodMissing called for ${name}"
    def methodInList = plays.find { it == name.split('play')[1] }

    if (methodInList) {
        def impl = { Object[] vargs ->
            "playing ${name.split('play')[1]}..."
        }

        Person.metaClass."$name" = impl //以后再调用就会使用它

        impl(args)
    } else {
        throw new MissingMethodException(name, Person.class, args)
    }
}

jack = new Person()
println jack.work()
println jack.playTennis()
println jack.playTennis()
```

前面代码的输出如下:

```
intercepting call for work
working...
intercepting call for playTennis
methodMissing called for playTennis
```

```
playing Tennis...
intercepting call for playTennis
playing Tennis...
```

invokeMethod与methodMissing的对比

`invokeMethod()`是`GroovyObject`的一个方法。`methodMissing()`则是Groovy中后来引入的，是基于`MetaClass`的方法处理的组成部分。如果目标是处理对不存在的方法的调用，应该实现`methodMissing()`，因为它的开销较低。如果目标是拦截所有的方法调用，而不管方法存在与否，则应使用`invokeMethod()`。

14.3 为具体的实例合成方法

14

在13.3节中，我们知道了如何向某个类的具体实例中注入方法。通过向具体的实例提供专用的`MetaClass`，也可以将方法合成到这些实例中。下面是一个例子：

InjectionAndSynthesisWithMOP/SynthesizesInstance.groovy

```
class Person {}

def emc = new ExpandoMetaClass(Person)
emc.methodMissing = { String name, args ->
    "I'm Jack of all trades... I can $name"
}
emc.initialize()

def jack = new Person()
def paul = new Person()

jack.metaClass = emc

println jack.sing()
println jack.dance()
println jack.juggle()

try {
    paul.sing()
} catch(ex) {
    println ex
}
```

这段代码的输出如下：

```
I'm Jack of all trades... I can sing
I'm Jack of all trades... I can dance
I'm Jack of all trades... I can juggle
```

```
groovy.lang.MissingMethodException:  
    No signature of method: Person.sing()  
    is applicable for argument types: () values: []
```

能够在实例的层次合成方法，非常有用。在测试中，或者在一个Web应用的特定Web请求中，我们可以修改一个选定实例的行为，而不影响Java虚拟机中该实例所关联的类。

还有一点非常强大，就是能够基于实例的当前状态或实例所接受的输入创建动态的方法或行为。这为创建和实现高度动态的DSL铺平了道路，我们将在后面看到。

本章介绍了如何合成方法。下一章将探讨如何动态地创建类，并且感受一下如何在各种元编程技术中做出选择。

上一章介绍了如何合成方法，本章探讨如何合成一个完整的类。不同于提前创建显式的类，可以在运行时创建类，这样更灵活。虽然委托优于继承，但是委托在Java中并不好实现，而在本章中将看到，Groovy的MOP允许仅使用一行代码实现方法委托。本章最后将回顾前几章介绍的MOP技术。^①

15.1 使用 Expando 创建动态类

Groovy中可以完全在运行时创建一个类。假设要构建一个用于配置设备的应用，而我们对这些设备一无所知，只知道它们有些属性和配置脚本，那就无法在编写代码时奢侈地为每个设备创建一个类。因此，需要在运行时合成与这些设备打交道并完成配置的类。在Groovy中，类可以根据命令在运行时产生。

Groovy的Expando类提供了动态合成类的能力，也因其动态可扩展性而得名。可以在构建时使用一个Map为其指定属性和方法，也可以动态地随时指定。下面就从一个例子入手，合成一个Car类。这里会介绍两种使用Expando创建这个类的方法。

15

MOPpingUp/UsingExpando.groovy

```
carA = new Expando()
carB = new Expando(year: 2012, miles: 0)
carA.year = 2012
carA.miles = 10

println "carA: " + carA
println "carB: " + carB
```

输出如下：

```
carA: {year=2012, miles=10}
carB: {year=2012, miles=0}
```

^① 本章英文标题为“MOPping Up”，有双关之意。mop up本身有“清理、清扫”之意，MOP本身又是本书第三部分所探讨的“元对象协议”。——译者注

此处创建的第一个Expando实例——carA，没有任何属性或方法。之后向该实例中注入了year和miles属性。而创建的第二个Expando实例——carB，在构建时提供了初始化过的year和miles属性。

我们不仅可以定义属性，还可以定义方法，并像调用任何方法那样调用它们。下面来试一下。再次重申，我们既可以在构建时定义方法，也可以以后随意注入：

MOPpingUp/UsingExpando.groovy

```
car = new Expando(year: 2012, miles: 0, turn: { println 'turning...' })
car.drive = {
    miles += 10
    println "$miles miles driven"
}

car.drive()
car.turn()
```

这段代码的输出如下：

```
10 miles driven
turning...
```

假设有一个输入文件，其中保存了一些汽车用的数据，如下所示：

MOPpingUp/car.dat

```
miles, year, make
42451, 2003, Acura
24031, 2003, Chevy
14233, 2006, Honda
```

无需显式创建一个Car类，就能方便地使用Car对象（如下列代码所示）。解析car.dat文件的内容，首先提取出属性名。然后为输入文件中的每行数据创建一个Expando实例，并使用行中的属性值填充这些实例。甚至还以闭包的形式添加了一个方法，以计算截止到2008年，汽车每年驾驶的平均公里数。一旦对象创建出来，就可以动态地访问其属性或调用其方法了。也可以通过名字来使用方法或属性，下列代码的最后有演示。

MOPpingUp/DynamicObjectsUsingExpando.groovy

```
data = new File('car.dat').readLines()

props = data[0].split(", ")
data -= data[0]

def averageMilesDrivenPerYear = { miles.toLong() / (2008 - year.toLong()) }

cars = data.collect {
    car = new Expando()
    it.split(", ").eachWithIndex { value, index ->
```

```

        car[props[index]] = value
    }

    car.ampy = averageMilesDrivenPerYear

    car
}

props.each { name -> print "$name " }
println " Avg. MPY"

ampyMethod = 'ampy'
cars.each { car ->
    for(String property : props) { print "${car[property]} " }
    println car."$ampyMethod"()
}

// 你也可能想通过名字访问属性或方法
car = cars[0]
println "$car.miles $car.year $car.make ${car.ampy()}"

```

这段代码的输出如下：

```

miles year make Avg. MPY
42451 2003 Acura 8490.2
24031 2003 Chevy 4806.2
14233 2006 Honda 7116.5
42451 2003 Acura 8490.2

```

想在运行时合成类时，可以使用Expando。它是轻量级的，而且非常灵活。比如，当为单元测试创建模拟（Mock）对象时，该特性会大放光彩，18.8节将予以介绍。

15

15.2 方法委托：汇总练习

继承用来扩展一个类的行为，而委托依赖所包含或聚合的对象可以提供一个类的行为。如果想用一个对象替代另一个对象，应该选择继承；如果只是想简单地使用一个对象，则应该选择委托。请将继承保留给is-a或kind-of关系；大多数情况下，应该首选委托（参见*Effective Java [Blo08]*）。然而使用继承编写程序很容易，只需要一个extends关键字。委托编写起来就困难了，因为必须编写很多方法，用以将调用路由给所包含对象的方法。Groovy可以帮助我们做正确的事。通过使用MOP，用一行代码即可轻松实现委托，本节将会介绍。

在下面的例子中，一个Manager想把工作委托给一个Worker或一个Expert。使用methodMissing()和ExpandoMetaClass来实现该功能。如果在Manager上调用的一个方法并不存在，该实例的methodMissing()方法会将调用路由给Worker或Expert，只要其中有一个能够成功处理respondsTo()方法即可（参见11.2节）。如果这些委托对象都不能处理某个方法，则

Manager无法处理该方法。

MOPpingUp/Delegation.groovy

```

class Worker {
    def simpleWork1(spec) { println "worker does work1 with spec $spec" }
    def simpleWork2() { println "worker does work2" }
}

class Expert {
    def advancedWork1(spec) { println "Expert does work1 with spec $spec" }
    def advancedWork2(scope, spec) {
        println "Expert does work2 with scope $scope spec $spec"
    }
}

class Manager {
    def worker = new Worker()
    def expert = new Expert()
    def schedule() { println "Scheduling ..." }
    def methodMissing(String name, args) {
        println "intercepting call to $name..."
        def delegateTo = null

        if(name.startsWith('simple')) { delegateTo = worker }
        if(name.startsWith('advanced')) { delegateTo = expert }
        if (delegateTo?.metaClass.respondsTo(delegateTo, name, args)) {
            Manager instance = this
            instance.metaClass."${name}" = { Object[] varArgs ->
                delegateTo.invokeMethod(name, varArgs)
            }
            delegateTo.invokeMethod(name, args)
        } else {
            throw new MissingMethodException(name, Manager.class, args)
        }
    }
}
peter = new Manager()
peter.schedule()
peter.simpleWork1('fast')
peter.simpleWork1('quality')
peter.simpleWork2()
peter.simpleWork2()
peter.advancedWork1('fast')
peter.advancedWork1('quality')
peter.advancedWork2('prototype', 'fast')
peter.advancedWork2('product', 'quality')
try {

```

```
peter.simpleWork3()
} catch(Exception ex) {
    println ex
}
```

输出如下：

```
Scheduling ...
intercepting call to simpleWork1...
worker does work1 with spec fast
worker does work1 with spec quality
intercepting call to simpleWork2...
worker does work2
worker does work2
intercepting call to advancedWork1...
Expert does work1 with spec fast
Expert does work1 with spec quality
intercepting call to advancedWork2...
Expert does work2 with scope prototype spec fast
Expert does work2 with scope product spec quality
intercepting call to simpleWork3...
groovy.lang.MissingMethodException:
    No signature of method: Manager.simpleWork3()
    is applicable for argument types: () values: []
```

前面实现了一种委托调用的方式，但是工作量非常大。我们不希望每次想使用委托的时候都花上这么大的力气。可以重构这段代码，以便复用。先来看看当重构之后的代码用于Manager类中时，看上去是什么样子的：

MOPpingUp/DelegationRefactored.groovy

```
class Manager {
    { delegateCallsTo Worker, Expert, GregorianCalendar }

    def schedule() { println "Scheduling ..." }
}
```

这段代码短小、漂亮。初始化块中调用了一个尚未实现的方法`delegateCallsTo()`，并将想用于委托未实现方法的类的名字传给了它。如果想在另一个类中使用委托，现在只需要拿走初始化块中的这行代码。下面看看精巧的`delegateCallsTo()`方法：

MOPpingUp/DelegationRefactored.groovy

```
Object.metaClass.delegateCallsTo = { Class... klassOfDelegates ->
    def objectOfDelegates = klassOfDelegates.collect { it.newInstance() }
    delegate.metaClass.methodMissing = { String name, args ->
        println "intercepting call to $name..."
        def delegateTo = objectOfDelegates.find {
            it.metaClass.respondsTo(it, name, args) }
        if (delegateTo) {
```

```
        delegate.metaClass."${name}" = { Object[] varArgs ->
            delegateTo.invokeMethod(name, varArgs)
        }
        delegateTo.invokeMethod(name, args)
    } else {
        throw new MissingMethodException(name, delegate.getClass(), args)
    }
}
```

当在类的实例初始化器内调用**delegateCallsTo()**方法时，该方法会向类中添加一个**methodMissing()**方法。这个类在闭包内被称作**delegate**。该闭包会接受作为一个参数提供给**delegateCallsTo()**的**Class**列表，并创建一个由用于委托的类组成的列表，这些类负责实现委托方法。在**methodMissing()**中，调用会被路由给这些类中可以对所调用方法做出响应的那个。如果这些类都无法响应，则调用失败。在提供给**delegateCallsTo()**方法的类的列表中，类出现的先后顺序也代表了其优先级：第一个优先级最高。当然，我们肯定要实际看看效果，运行如下代码，测试前面编写的**Manager**：

MOPpingUp/DelegationRefactored.groovy

```
peter = new Manager()
peter.schedule()
peter.simpleWork1('fast')
peter.simpleWork1('quality')
peter.simpleWork2()
peter.simpleWork2()
peter.advancedWork1('fast')
peter.advancedWork1('quality')
peter.advancedWork2('prototype', 'fast')
peter.advancedWork2('product', 'quality')
println "Is 2008 a leap year? " + peter.isLeapYear(2008)
try {
    peter.simpleWork3()
} catch(Exception ex) {
    println ex
}
```

输出如下：

```
Scheduling ...
intercepting call to simpleWork1...
worker does work1 with spec fast
worker does work1 with spec quality
intercepting call to simpleWork2...
worker does work2
worker does work2
intercepting call to advancedWork1..
Expert does work1 with spec fast
Expert does work1 with spec quality
```

```

intercepting call to advancedWork...
Expert does work2 with scope prototype spec fast
Expert does work2 with scope product spec quality
intercepting call to isLeapYear...
Is 2008 a leap year? true
intercepting call to simpleWork3...
groovy.lang.MissingMethodException:
    No signature of method: Manager.simpleWork3()
    is applicable for argument types: () values: []

```

可以基于这个想法实现我们的需求。比如，如果想混入一些已经创建好的对象，可以将其作为一个数组发送给`delegateCallsTo()`的第一个参数，然后将它们与委托类中创建的对象一起使用。前面的例子演示了如何使用Groovy的MOP来实现诸如方法委托这样的动态行为。

从本节的例子中可以学到如何在运行时动态地修改实例的行为。如果喜欢，可以基于对象的当前状态修改委托。如果委托是静态的，则可以提前确定，没必要在运行时修改。这种情况可以简单地使用2.10.2节介绍的`@Delegate`注解（这是一种编译时元编程技术）。

15.3 MOP 技术回顾

第三部分介绍了很多可用于拦截、注入和合成方法的选项。本节要解决的问题是，了解哪个选项适合我们。

15.3.1 用于方法拦截的选项

第12章和13.1节探讨了方法拦截，我们可以使用`GroovyInterceptable`、`ExpandoMetaClass`或分类。

如果有权修改类的源代码，可以在想要拦截方法调用的类上实现`GroovyInterceptable`接口。做起来像实现`invokeMethod()`方法一样简单。

如果无法修改类，或者那是个Java类，则可以使用`ExpandoMetaClass`或分类。`ExpandoMetaClass`显然非常适合这种情况，因为一个`invokeMethod()`方法就可以负责拦截类上的任何方法。而分类则需要为每个要拦截的方法提供一个单独的方法。此外，如果使用分类，就必须使用`use()`块。

15.3.2 用于方法注入的选项

13.1节探讨了方法注入，可以使用分类或`ExpandoMetaClass`来实现。

在方法注入方面，分类完全可以与`ExpandoMetaClass`相媲美。如果使用分类，可以控制注入方法的位置。通过使用不同的分类，可以轻松实现不同版本的方法注入。我们还可以轻松地嵌

入和混用多个分类。而分类所提供的控制——即方法注入尽在use()块内起作用，而且被限制在执行线程上，也可以认为是一个限制。如果想在任意位置使用注入的方法，或者想注入静态方法和构造器，ExpandoMetaClass是更好的选择。不过还请注意，ExpandoMetaClass不是Groovy中的默认MetaClass。

借助ExpandoMetaClass，可以向某个类的具体实例注入方法，而不影响整个类。

15.3.3 用于方法合成的选项

14.1节探讨了方法合成，我们可以在Groovy对象或ExpandoMetaClass上使用methodMissing()。

如果有权修改类的源代码，能够在类上为想要合成的方法实现methodMissing()方法，可以通过在第一次调用时注入方法来改进性能。如果同时需要拦截方法，实现GroovyInterceptable接口即可。

如果无法修改类，或者那是个Java类，可以将methodMissing()方法添加到类的ExpandoMetaClass中。如果同时想拦截方法，可以在ExpandoMetaClass上实现invokeMethod()。

借助ExpandoMetaClass，可以将方法合成到某个类的具体实例中，而不影响整个类。

元编程是Groovy得以大放异彩的一个关键特性。它使在运行时扩展程序，以及利用该语言的动态能力成为现实。到目前为止所学到的元编程技术，为我们提供了在运行时修改程序行为的能力。Groovy还支持在编译时修改程序的行为。下一章将探讨编译时元编程。

应用编译时元编程

16

与某些只有运行时能力的动态类型语言不同，Groovy同时提供了运行时和编译时元编程。

利用前面几章研究的运行时元编程，可以将与类和实例上方法的拦截、注入甚至合成相关的决策推迟到运行时。就元编程而言，大部分情况下，有这些就够了。编译时元编程是一种高级特性，可用于某些特殊情况，现在主要是框架或工具的编写者使用。

借助Groovy，可以在编译时分析和修改程序的结构。这可以为应用带来高度的可扩展现行，同时支持添加新的横切特性。比如，无需修改源代码，即可为线程安全、日志消息和在代码的不同部分执行前置和后置检验操作等，对类进行检查。

编译时元编程，正是某些强大的特性和基于Groovy的工具背后的魔力所在。比如，Groovy 2 中的类型检查器就实现为了一个抽象语法树（AST）变换。优雅的单元测试工具Spock使用这种方式来支持流畅的测试用例^①。用于探测潜在的错误和代码不良味道的代码分析工具CodeNarc也大量使用了该特性^②。该特性支撑起了Groovy的注解，比如2.10节介绍的@Delegate和@Immutable。

16

本章将介绍如何使用编译时元编程来分析代码结构、拦截方法及注入行为。

16.1 在编译时分析代码

高级开发人员和软件架构师往往会倡导编码标准，而且会尽力确保其团队遵循一致的编程实践。可以利用Groovy的强大将代码审查自动化，以发现代码中的异味^③和不良实践。本节将介绍如何使用编译时元编程来检查代码异味。

尽管命名变量有些困难，而且需要努力，但是使用只有一个字母的变量名肯定是不对的。与其靠人工监管代码，不如利用编译时元编程来检查差劲的变量名和方法名。

① <https://github.com/spockframework>

② <http://codenarc.sourceforge.net>

③ 程序开发领域，代码中的任何可能导致深层次问题的症状都可以叫作代码异味，具体可以参考<http://zh.wikipedia.org/wiki/%E4%BB%A3%E7%A0%81%E5%BC%82%E5%91%B3>。——译者注

AST/CodeAnalysis/smelly.groovy

```

def canVote(a) {
    a > 17 ? "You can vote" : "You can't vote"
}

def p(instance) {
    //用于打印实例的代码
}

```

`canVote()`方法接受一个表示年龄的参数`a`，确定这个年龄的人是否可以投票。我们希望自动地探测出代码中存在异味的参数`a`和古怪的方法名`p()`。要尽可能早地探测出来，那就在编译代码时。因为这段代码在语法上是正确的，编译器不会检查出其中的异味，但是我们可以。我们可以命令编译器在遇到这种存在异味的代码时不予通过，尽管代码在语法上是正确的。

16.1.1 理解代码结构

要检查代码异味，需要遍历代码结构，分析类名、方法名、字段名和参数名等信息。这可以通过编写一个解析器来实现，但是，编译器已经解析并分析过代码，倒不如依靠编译器，尽量减少人为的工作。

Groovy编译器允许我们进入其编译阶段，一窥其所处理的AST（抽象语法树）。AST树结构描述了程序中的表达式和语句，它是使用节点表示的。随着编译过程的进行，程序的AST会被变换，包括节点的插入、删除和重新排列。在编译过程中，我们可以随着AST的演进对它进行检查，加以修改，以及命令编译器去标记警告或错误。

`canVote()`方法很短，只是返回三元操作符的结果，但是其AST却异常丰富，包含很多细粒度的细节信息（参见图16-1）。

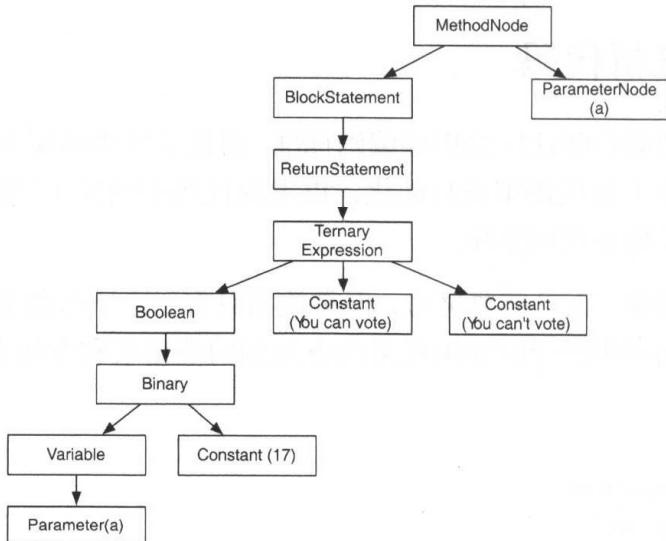


图16-1 `canVote()`方法的AST

要使用编译时元编程，必须理解和使用AST。这项任务非常复杂，但幸运的是有个帮手。`groovyConsole`工具有一个很棒的功能，它可以显示代码的AST。在这个工具中打开Groovy源代码，选择Script菜单下的Inspect AST菜单项。`groovyConsole`工具会显示这段存在异味的代码的AST结构，如图16-2所示。

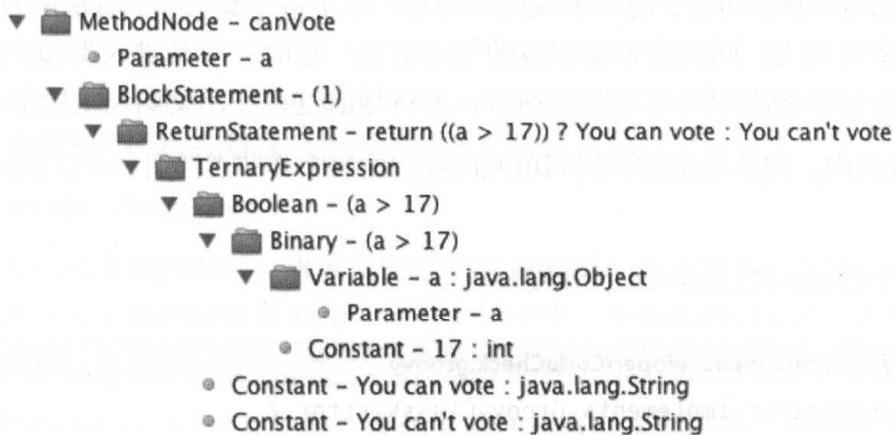


图16-2 在`groovyConsole`的AST浏览器视图中查看`canVote()`方法

16.1.2 在代码结构中导航

既然已经掌握了可能存在异味的代码的AST，是时候通过在这种结构中导航来检查代码了。Groovy中提供的AST变换应用编程接口（API）使这一任务更易达成了。为了在代码中导航，现在创建一个名为`CodeCheck`的类，并实现`ASTTransformation`接口。

16

```

AST/CodeAnalysis/com/agiledeveloper/CodeCheck.groovy
@GroovyASTTransformation(phase = CompilePhase.SEMANTIC_ANALYSIS)
class CodeCheck implements ASTTransformation {
    void visit(ASTNode[] astNodes, SourceUnit sourceUnit) {
        sourceUnit.ast.classes.each { classNode ->
            classNode.visitContents(new OurClassVisitor(sourceUnit))
        }
    }
}
  
```

如果想检查AST，首先使用注解`GroovyASTTransformation`告知编译器。Groovy编译器包括多个阶段，而且支持开发者在任何阶段中介入：初始化、解析、转换、语义分析、规范化、指令选择、class生成、输出和结束^①。首个合理的时机是在语义分析阶段之后，AST在此时被创建出来。如果想使用信息更多的AST，可以在更靠后的阶段介入。上面的例子指明该AST变换必须在语义分析阶段（`CompilePhase.SEMANTIC_ANALYSIS`）之后应用。

^① 关于这些编译阶段的更多信息，可以参考<http://groovy.codehaus.org/Compiler+Phase+Guide>。——译者注

随着编译器到达指定阶段，它将调用用于变换的类的visit()方法，并向该方法提供一个由ASTNode实例组成的列表，以及一个表示被编译代码的SourceUnit引用。可以在visit()方法内迭代这些AST节点来检查各种元素，如果愿意，甚至可以修改这些节点。

在这个例子中，我们想访问整个结构来查找代码异味。AST变换API提供了一个名为GroovyClassVisitor的访问者，简化了我们的操作。对于每个类节点、方法节点、字段节点等，该访问者的方法将被调用，同时被告知相应的节点信息。这时，与其进入类和方法的层次结构，不如坐在原处，在这些方法中采取相应的动作，而将导航这种累活交给API去做。

在visit()方法内，向从给定的源代码单元内找到的每个类节点注册一个GroovyClassVisitor的实现。

现在实现这个GroovyClassVisitor接口。

```
AST/CodeAnalysis/com/agiledeveloper/CodeCheck.groovy
class OurClassVisitor implements GroovyClassVisitor {
    SourceUnit sourceUnit
    OurClassVisitor(theSourceUnit) { sourceUnit = theSourceUnit }
    private void reportError(message, lineNumber, columnNumber) {
        sourceUnit.addError(new SyntaxException(message, lineNumber, columnNumber))
    }

    void visitMethod(MethodNode node) {
        if(node.name.size() == 1)
            reportError "Make method name descriptive, avoid single letter names",
                        lineNumber, node.columnNumber

        node.parameters.each { parameter ->
            if(parameter.name.size() == 1)
                reportError "Single letter parameters are morally wrong!",
                            parameter.lineNumber, parameter.columnNumber
        }
    }

    void visitClass(ClassNode node) {}
    void visitConstructor(ConstructorNode node) {}
    void visitField(FieldNode node) {}
    void visitProperty(PropertyName node) {}
}
```

当导航到一个类中的类元素时，AST变换API将调用访问者的方法；对于构造器、字段、方法等，也是如此。因为现在目的是发现代码异味，所以在visitMethod()方法中检查只有一个字母的方法名，如果找到，则添加一条错误消息。编译器也相应报告该错误，并且让编译失败。也可以不使用错误，而是将其报告为警告。

除了发现方法名中的异味，这里还检查了方法的参数名。可以继续扩展，通过在其他方法中（如`visitField()`和`visitProperty()`）实现代码，来发现字段名、属性名等元素中的异味。

这样就几乎实现了一个小型代码异味检查器，但是在编译过程中，还必须帮助编译器发现并应用这个AST变换。这里将使用一种叫作全局变换的方式，此时变换可以应用于任何代码片段中，不需要代码上的任何特殊标记。Groovy编译器将在`classpath`中的每个JAR文件中查找这样的全局变换。为使查找更为高效，编译器期望我们在各个JAR文件中的一个特殊的清单文件（`manifest/META-INF/services/org.codehaus.groovy.transform.ASTTransformation`）中声明这个变换的类名。下面就是用于代码检查器这个例子的清单文件的内容：

```
com.agiledveloper.CodeCheck
```

总结一下这节已经学到的东西。清单文件告知编译器用于变换的类的名字。在变换类中，使用注解指明它应该在哪个编译阶段被调用。在这个例子中，变换类的`visit()`方法借助一个访问者，调用相应的动作。

现在需要编译`CodeCheck`类，并使用生成的类文件和清单文件创建一个JAR文件。如果该JAR文件在`classpath`下，Groovy编译器将应用这个代码异味检查器。下面看一下实现该功能需要的步骤：

```
$ groovyc -d classes com/agiledveloper/CodeCheck.groovy
$ jar -cf checkcode.jar -C classes com -C manifest .
$ groovyc -classpath checkcode.jar smelly.groovy
```

运行上述命令，检查前面所创建的例子中的代码异味。因为发现了代码异味，编译器将输出一些错误消息，并使编译失败。

16

```
org.codehaus.groovy.control.MultipleCompilationErrorsException:
  startup failed:
smelly.groovy: 1: Single letter parameters are morally wrong! @
  line 1, column 13.
    def canVote(a) {
      ^
smelly.groovy: 5: Make method name descriptive, avoid single letter names @
  line 5, column 1.
    def p(instance) {
      ^
2 errors
```

代码检查发生在编译阶段，而非运行期间。这需要我们花些时间来熟悉。研究并修改这个例子，随着在AST中导航，设置一些输出信息，仔细了解这种变换如何应用，以及何时应用。

本节介绍了如何使用AST变换来检查代码。如果只是想找出一些常见的代码异味，其实不必这么大费周章。可以使用CodeNarc这款Groovy代码质量工具。它能方便地使用CodeNarc所提供的标准检查方法，甚至可以扩展其规则，以监视想检查的代码异味。现在看的这个例子可以帮我们

见识到AST变换之强大，如果想在代码中执行领域特定的约束检查，AST变换将会派上用场。

AST变换的强大远不止分析代码这么简单。可以在编译时拦截方法调用，甚至向程序中注入代码，下面将予以介绍。

16.2 使用AST变换拦截方法调用

假设正在开发某款银行软件，业务人员丢给我们一个出乎意料的问题。他们希望，在活期存款账户中，每笔金额超过10 000美元的存款、取款和转账业务都需要经过审计。对此需要做出快速的反应，所以先来考虑几个方案。

最差的方案是找到在所有活期存款账户上执行事务的地方，然后加以修改。但即使凭借我们最喜爱的强大集成开发环境，找到并修改这些调用也不是那么有趣。此外，每次调用其中的一个方法时，还都要记得执行审计操作。

另一个方案是修改活期存款账户类中的方法来执行额外的操作。与第一个方案相比，这个方案更为可靠，而且需要的工作更少。然而，它会导致代码冗余，而且还要当心新加入的函数。

第12章介绍了如何使用运行时元编程拦截方法调用。与前面的两个方案相比，这种技术优势极大。它没有代码冗余，而且对于新加入的方法，这种方案也很容易扩展。但是方法拦截仅发生于运行时，所以会对执行造成轻微影响。本节将介绍如何通过在编译时拦截方法来避免这一不利影响。

看一下包含了新添加的audit()方法的CheckingAccount类。当这个类中的其他方法被调用时，我们希望该方法也适时调用。

AST/InterceptingCalls/UsingCheckingAccount.groovy

```
class CheckingAccount {
    def audit(amount) { if(amount > 10000) print "auditing..." }
    def deposit(amount) { println "depositing ${amount}..." }
    def withdraw(amount) { println "withdrawing ${amount}..." }
}

def account = new CheckingAccount()
account.deposit(1000)
account.deposit(12000)
account.withdraw(11000)
```

现在使用AST变换，在CheckingAccount类的每个方法中都加入一个对audit()方法的调用（除了该方法本身）。为其编写一个变换类：

AST/InterceptingCalls/com/agiledeveloper/InjectAudit.groovy

```
@GroovyASTTransformation(phase = CompilePhase.SEMANTIC_ANALYSIS)
class InjectAudit implements ASTTransformation {
    void visit(ASTNode[] astNodes, SourceUnit sourceUnit) {
```

```

def checkingAccountClassNode =
    astNodes[0].classes.find { it.name == 'CheckingAccount' }
injectAuditMethod(checkingAccountClassNode)
}

```

InjectAudit类实现了ASTTransformation接口，并提供了必需的visit()方法。使用@GroovyASTTransformation(phase = CompilePhase.SEMANTIC_ANALYSIS)注解告诉编译器在语义分析阶段的最后应用该变换。

因为这是一个全局变换，编译器会在被编译的代码中发现的所有节点上触发该变换。在visit()方法内，使用find()方法将代表CheckingAccount类的节点提取出来，之后使用一个辅助方法injectAuditMethod()采取相应的动作：

```

AST/InterceptingCalls/com/agiledeveloper/InjectAudit.groovy
static void injectAuditMethod(checkingAccountClassNode) {
    def nonAuditMethods =
        checkingAccountClassNode?.methods.findAll { it.name != 'audit' }
    nonAuditMethods?.each { injectMethodWithAudit(it) }
}

```

使用findAll()方法提取出了除audit()之外的所有方法。之后通过辅助方法injectMethodWithAudit()，在每个选中的方法开头加入了一个对audit()方法的调用。

真正的动作在injectMethodWithAudit()中。我们希望在每个方法的开头放置一个对audit()方法的调用。遗憾的是，所需的步骤并不是调用一下这个方法那么简单。必须为方法调用创建AST，并将其插入到目标方法AST中的语句列表中。下面实现这一功能：

```

AST/InterceptingCalls/com/agiledeveloper/InjectAudit.groovy
static void injectMethodWithAudit(methodNode) {
    def callToAudit = new ExpressionStatement(
        new MethodCallExpression(
            new VariableExpression('this'),
            'audit',
            new ArgumentListExpression(methodNode.parameters)
        )
    )

    methodNode.code.statements.add(0, callToAudit)
}

```

要理解这里为方法调用创建的AST，可以使用groovyConsole来查看一下该调用对应的内部AST结构。这可以帮助你联想到并弄清楚需要创建什么。像audit(amount)这样的一个简单调用，可能需要很多行代码、一系列表达式对象，如上面的代码所示。

MethodCallExpression在AST层次表示这个调用。其第一个参数——VariableExpression，

指出这个调用要在此执行环境中的当前对象(`this`)上进行。第二个参数指出了要调用的方法的名字。第三个参数代表要传给这个方法的参数，在这个例子中就是一个使用外围方法节点的参数创建的`ArgumentListExpression`实例。

最后，将所创建的表达式节点插入到目标方法的语句列表中。

在将该变换投入使用之前还有最后一步，即必须让编译器知道它。创建清单文件META-INF/services/org.codehaus.groovy.transform.ASTTransformation，在其中列出该变换的类名。

```
com.agiledeveloper.InjectAudit
```

首先编译该变换，并将其打入JAR包。

```
$ groovyc -d classes com/agiledeveloper/InjectAudit.groovy
$ jar -cf injectAudit.jar -C classes com -C manifest .
```

用于将方法调用加入到`CheckingAccount`类中的变换就准备好了。为研究这一变换的作用，首先不使用该变换来运行`UsingCheckingAccount.groovy`：

```
$ groovy UsingCheckingAccount.groovy
```

调用3个方法的结果就是直接调用：

```
depositing 1000...
depositing 12000...
withdrawning 11000...
```

之后再使用这一变换来修改其行为，将`injectAudit.jar`包含在`classpath`中。

```
$ groovy -classpath injectAudit.jar UsingCheckingAccount.groovy
```

编译器将识别出`injectAudit.jar`中的这一变换，并加入对`audit()`方法的调用：

```
depositing 1000...
auditing...depositing 12000...
auditing...withdrawning 11000...
```

每次调用`CheckingAccount`类上的一个方法，都会先调用`audit()`，但是这一变换不会影响`audit()`方法的任何直接调用。

现在是以脚本形式运行的代码，要使变换生效，每次运行时都要将`injectAudit.jar`包含在`classpath`中。为避免该问题，可以预编译这段代码。只需要使用`groovyc`编译这个脚本，并将`injectAudit.jar`放到`classpath`中。生成的字节码中将包含对`audit()`的适当调用。之后就可以使用`groovy`或`java`命令运行字节码了。

本节使用AST变换在编译时添加了方法调用。与运行时元编程相比，性能更好，然而要做的工作多出很多。下面将介绍缓解这一问题的不同方法。

像`this.audit()`这样一个简单的调用，在变换期间都需要创建多个对象、很多行代码。对于更复杂的调用，想想就令人生畏，甚至最有激情的程序员都会很快望而却步。谢天谢地，

ASTBuilder类可以减轻我们的负担。

ASTBuilder提供了3种创建AST子树的不同方式：`buildFromSpec()`、`buildFromString()`和`buildFromCode()`。下面使用它们实现`injectMethodWithAudit()`方法。

实例化表达式等类的实例这种新操作有很多噪音，`buildFromSpec()`方法可以帮助减少这些噪音。简单地使用一个`methodCall`块来创建一个`MethodCallExpression`实例，并使用`variable`来定义一个变量，如下面这个版本的`injectMethodWithAudit()`所示：

AST/EasingThePain/com/agiledeveloper/InjectAudit.groovy

```
static void injectMethodWithAudit(methodNode) {
    List<Statement> statements = new AstBuilder().buildFromSpec {
        expression {
            methodCall {
                variable 'this'
                constant 'audit'
                argumentList {
                    methodNode.parameters.each { variable it.name }
                }
            }
        }
    }
    def callToCheck = statements[0]
    methodNode.code.statements.add(0, callToCheck)
}
```

16

使用`buildFromSpec()`方法创建AST非常流畅，但是这种方式也引入了一些复杂性——必须熟悉该API所期望的DSL语法。为此我们必须知道正在创建的AST的结构，毕竟该API只是使语法更流畅了。

与其花费这么大力气，不如简单地使用`buildFromString()`方法，从嵌入在字符串中的一段代码获得一个AST变换。下面使用这一生成器方法重写`injectMethodWithAudit()`方法。

AST/EasingThePain2/com/agiledeveloper/InjectAudit.groovy

```
static void injectMethodWithAudit(methodNode) {
    def codeAsString = 'audit(amount)'
    List<Statement> statements = new AstBuilder().buildFromString(codeAsString)

    def callToAudit = statements[0].statements[0].expression
    methodNode.code.statements.add(0, new ExpressionStatement(callToAudit))
}
```

简单地将想插入的语句丢到了一个字符串中，剩下的活交给生成器处理。在创建AST时，`buildFromString()`给我们省了很多力气，但是遗憾的是，生成的结果被包在了一个`return`语

句中。必须再花点力气将所需的内容提取出来，在将其插入到目标方法的语句中之前，需要先将其放到一个**ExpressionStatement**中。

`buildFromString()`方法还有一个问题，它要求我们将代码放到一个字符串中。处理转义字符和多行代码会很麻烦。就算借助here文档语法（参见5.3节），也是非常困难。

`buildFromCode()`优于其他方法。可以像自然编写代码那样编写代码，并将其放到一个代码块中。`buildFromCode()`就像一位善良的撒玛利亚人^①，接收这个代码块，并生成AST变换。下面使用这个方法重写`injectMethodWithAudit()`方法：

```
AST/EasingThePain3/com/agiledeveloper/InjectAudit.groovy
static void injectMethodWithAudit(methodNode) {
    List<Statement> statements = new AstBuilder().buildFromCode { audit(amount) }
    def callToAudit = statements[0].statements[0].expression
    methodNode.code.statements.add(0, new ExpressionStatement(callToAudit))
}
```

这种方式对我们帮助很大。

- 不必再为要创建的节点的AST结构而挣扎。
- 不必担心AST结构是否会在未来的Groovy版本中发生改变；`AstBuilder`会随之演化，将我们从修改AST结构的工作中解放出来。
- 显而易见，所要生成的代码没有迷失在AST结构的细节之中。

`buildFromCode()`方法非常吸引人，但在使用时有一些注意事项：它没有完全解放我们，我们仍然需要对AST结构有所了解；仍然必须从产生的AST中提取正确的部分，并需要知道将其置于何处。对于可以使用该方法生成的代码，也存在一些限制：生成的代码会放在该变换编译后的代码中，躲不开窥视的眼睛。最后，`AstBuilder`的构建方法本身也要经过一次编译时的AST变换，这限制我们只能使用Groovy编写变换代码，而没有使用`AstBuilder`的变换则可以使用任何JVM语言编写。

本节介绍了如何在编译时拦截方法并向其中添加行为。也可以使用该技术向类中添加新的方法和字段，下一节将予以介绍。

16.3 使用AST变换注入方法

上一节介绍了如何向已有的方法中注入代码。通过使用AST变换，也可以向类中注入方法和字段。这样无需第三方库，即可在编译时发挥AOP的全部威力。

^① Samaritan，中东种族撒马利亚人，因心肠好而著称。——译者注

4.5节介绍的Execute Around Method模式中，使用了一个静态方法来辅助创建和清理实例。该方法支持在两个操作之间随便使用它生成的实例。下面使用AST变换来实现该模式。与其让程序员手动实现use()方法，不如我们来创建它，程序员只要请求调用即可。漂亮！

到目前为止，本章介绍的AST变换都是全局变换。它们会被应用于被编译的所有代码上。在变换内确定是要执行某些变换，还是简单地跳过。那种方式是非侵入式的。被变换的代码无须关注该变换，也不需要任何特殊处理。

然而对于有些可能遇到的问题，那种方式太过全面。这里需要知道把use()方法注入到哪个类中。本节的亮点就在于此。下面编写一个局部变换，该变换只应用于程序员使用所提供的特殊注解标记的选定部分。局部变换有一个优点：不必创建额外的清单文件。

先创建会触发变换的注解。

```
AST/EAM/com/agiledeveloper/EAM.groovy

@Retention(RetentionPolicy.SOURCE)
@Target([ElementType.TYPE])
@GroovyASTTransformationClass("com.agiledeveloper.EAMTransformation")

public @interface EAM {
}
```

使用Target，指明这个注解只能放在类上。使用GroovyASTTransformationClass，告知编译器，参数中提到的变换com.agiledeveloper.EAMTransformation应该应用于使用这个EAM注解标记的任何类。

下面将静态的use()方法插入到被注解的类中。这听上去有点吓人，但是编写局部变换与编写全局变换并没有多大区别，而后者我们已经会了。因为变换只在目标类上调用，所以可以直接在visit()方法中处理：

```
AST/EAM/com/agiledeveloper/EAMTransformation.groovy

@GroovyASTTransformation(phase = CompilePhase.SEMANTIC_ANALYSIS)
class EAMTransformation implements ASTTransformation {
    void visit(ASTNode[] astNodes, SourceUnit sourceUnit) {

        astNodes.findAll { node -> node instanceof ClassNode }.each { classNode ->

            def useMethodBody = new AstBuilder().buildFromCode {
                def instance = newInstance()
                try {
                    instance.open()
                    instance.with block
                } finally {
                    instance.close()
                }
            }
        }
    }
}
```

```

15      }

16      def useMethod = new MethodNode(
17          'use', ACC_PUBLIC | ACC_STATIC, ClassHelper.OBJECT_TYPE,
18          [new Parameter(ClassHelper.OBJECT_TYPE, 'block')] as Parameter[],
19          [] as ClassNode[], useMethodBody[0])

20      classNode.addMethod(useMethod)
21  }
22}

25 }

```

EAM模式的核心是我们想注入到类中的特殊的use()方法。在这个方法中，需要将实例发送给一个使用该实例的闭包。闭包调用本身需要包在一个try-finally块中，finally块中将调用清理代码。

在visit()方法内，使用ASTBuilder的buildFromCode()方法来创建use()方法，并将其注入到类中。这里假设目标类有一个open()方法和一个close()方法。如果缺少这两个方法，则将抛出运行时错误。如果愿意，也可以抛出编译时错误。为此，必须遍历该类的AST节点，如果没有找到这些方法，则像16.1节所做的那样报告错误。

在visit()方法中，第7行使用buildFromCode()创建的只是use()方法的方法体。之后还必须将方法体附到一个表示use()方法的方法节点上。第17行创建了一个MethodNode实例，并将方法体附了上去。

下面仔细看一下MethodNode的创建。第一个参数指明了方法名（use）。第二个参数指明了该方法应该使用public和static修饰符。第三个参数指明该方法的返回类型（Object）。方法一般都要接收参数，但是这里的use()方法期待的是一个闭包。代码中使用第四个参数指明了这一点，该参数是一个列表，需要列出use()方法所需的每个参数的类型和名字。第五个参数指明方法可能会抛出的异常，这个例子中没有。最后一个参数指向的是所创建方法的方法体。

最后一步，使用addMethod()方法将刚创建的这个方法添加到目标类中。得到这段简洁的代码真是费了不少劲，现在已经设计出一种方式，实现了向使用EAM注解标记的任何类中注入use()方法。下面找个类试试，不过首先需要编译变换代码，并将其打包到一个JAR文件中。

```

$ groovyc -d classes \
  com/agiledeveloper/EAM.groovy \
  com/agiledeveloper/EAMTransformation.groovy
$ jar -cf eam.jar -C classes com

```

创建一个可以注入use()方法的Resource类。

AST/EAM/resource.groovy

```

@com.agiledeveloper.EAM
class Resource {

```

```

private def open() { print "opened..." }
private def close() { print "closed" }
def read() { print "read..." }
def write() { print "write..." }
}
println "Using Resource"
Resource.use {
    read()
    write()
}

```

Resource类提供了期望的open()和close()方法。代码中调用了期望的use()方法。不要担心这个方法不存在，因为使用EAM注解标记了Resource类，前面编写的变换会向这个类中注入use()方法。最后，当编译Resource类时，要确保eam.jar在classpath下：

```
$ groovy -classpath eam.jar resource.groovy
```

从该命令的输出可以看出，通过编译时元编程实现的EAM模式起作用了。

```

Using Resource
opened...read...write...closed

```

4.5节创建了use()方法，而这一节通过AST变换，将该方法注入到了Resource类或任何使用@EAM注解的类中。一旦驾驭了AST变换，就可以使用这种技术实现非常强大的变换。

警告：在使用如此强大的工具时，需要确保变换的表现确实符合预期。好在这方面我们也有一些帮手。Groovy提供了一个可以使用@ASTTest注解调用的AST变换，以帮助测试其他AST变换，以及在不同的AST节点上对预期结果施加断言。^①

元编程是最强大的概念之一。如果使用得当，它可以帮助创建高度可扩展的软件。像Grails这样的框架就大量使用了元编程。Groovy的特殊之处在于，它同时提供了运行时和编译时的元编程能力。本章探讨的内容不只是如何借助这种能力使用Groovy语言，还包括如何灵活地使用这种能力向现有代码中注入行为。

本书的第三部分介绍了如何立即创建类、方法和属性。可以拦截对已有的方法的调用，甚至还可以拦截对不存在的方法的调用。使用元编程的程度取决于应用特定的需求。不过我们知道，当应用需要元编程时，可以快速实现。第四部分将介绍一些元编程可以起到关键作用的场景，比如使用模拟对象进行单元测试、创建生成器和创建DSL等。

^① 参见<http://groovy.codehaus.org/gapi/groovy/transform/ASTTest.html>。

第四部分

使用元编程

本部分内容

- 第 17 章 Groovy 生成器
- 第 18 章 单元测试与模拟
- 第 19 章 在 Groovy 中创建 DSL

Groovy生成器

17

生成器是内部的DSL，为处理某些特定类型的问题提供了方便。举个例子，如果需要使用嵌入的、层次式结构，比如树结构、XML、HTML或JSON（JavaScript Object Notation）等表示形式，生成器会非常有用。生成器提供的语法不会把使用者紧紧地绑定到底层的结构或实现上。生成器也不会替换掉底层实现；相反，它们只是为处理底层实现提供了一种优雅的方式。

Groovy可以用于很多日常任务，包括处理XML、JSON、HTML、DOM、SAX、Swing甚至Ant。在本章中，我们将通过一些任务来感知生成器，之后再研究2种创建生成器的技术。

17.1 构建 XML

程序员中的大部分人都讨厌XML。随着文档的增大，处理XML会越来越困难，工具和API支持也不尽如人意。我的理论是，XML就像人：小时候聪明可爱，越大越招人烦。

XML可能是一种很适合机器处理的格式，但是人直接处理很不方便。没有人会真喜欢处理XML，但是又不得不做。而Groovy几乎使得处理XML变成了一种乐趣，极大地缓解了其中的痛苦。

下面看一个例子，在Groovy中使用生成器创建XML文档：

UsingBuilders/UsingXMLBuilder.groovy

```
bldr = new groovy.xml.MarkupBuilder()
bldr.languages {
    language(name: 'C++) { author('Stroustrup') }
    language(name: 'Java') { author('Gosling') }
    language(name: 'Lisp') { author('McCarthy') }
}
```

这段代码使用`groovy.xml.MarkupBuilder`来创建XML文档。当在生成器上调用任意的方法或属性时，它会根据调用的上下文，体贴地假定我们引用的是所生成文档中的元素名或属性名。上述代码的输出如下：

```
<languages>
  <language name='C++'>
    <author>Stroustrup</author>
  </language>
  <language name='Java'>
    <author>Gosling</author>
  </language>
  <language name='Lisp'>
    <author>McCarthy</author>
  </language>
</languages>
```

我们调用了一个名为`languages()`的方法，但该方法在`MarkupBuilder`类的实例上并不存在。不过生成器并没有拒绝它，而是聪明地假定这次调用其实是想定义XML文档的一个根元素，这种假定可真不错。

附在这个方法调用上的闭包现在提供了一个内部的上下文。领域特定语言是与上下文有关的。在这个闭包内，被调用到的任何不存在的方法，都会被假定为是一个子元素的名字。如果在调用方法时传递的是`Map`参数（如`language(name: value)`），它们会被当作元素的属性。任何单个的参数值（如`author(value)`），表示的是元素内容，而非属性。可以研究一下前面的代码和相关输出，看看`MarkupBuilder`是如何推断代码的。

在前面的示例中，进入XML文档的数据都是硬编码的，而且生成器只是将结果写到了标准输出中。而在实际的项目中，这两种情况都很少用到。我们的数据可能来自一个集合，而集合又可能是通过一个数据源或输入流填充的。此外，我们可能还想把XML内容写入到一个`Writer`中，而不是写入到标准输出中。

生成器上可以轻松地附上一个`Writer`，将其作为构造器的一个参数。所以，可以将一个`StringWriter`附到生成器上。数据可以来自任何源，比如来自数据库（参见9.3节）。下面例子从一个`Map`中取到数据，创建了一个XML文档，并将文档写到了一个`StringWriter`中：

17

UsingBuilders/BuildXML.groovy

```
langs = ['C++' : 'Stroustrup', 'Java' : 'Gosling', 'Lisp' : 'McCarthy']

writer = new StringWriter()
bldr = new groovy.xml.MarkupBuilder(writer)
bldr.languages {
  langs.each { key, value ->
    language(name: key) {
      author (value)
    }
  }
}
println writer
```

这段代码的输出如下：

```
<languages>
    <language name='C++'>
        <author>Stroustrup</author>
    </language>
    <language name='Java'>
        <author>Gosling</author>
    </language>
    <language name='Lisp'>
        <author>McCarthy</author>
    </language>
</languages>
```

MarkupBuilder十分适合小到中型的文档。然而，如果文档非常大（若干兆字节），我们可以使用StreamingMarkupBuilder，它的内存占用情况更好一些。下面使用StreamingMarkupBuilder重写前面的示例，为增加一些趣味性，我们把命名空间和XML注释包含了进来：

UsingBuilders/BuildUsingStreamingBuilder.groovy

```
langs = ['C++' : 'Stroustrup', 'Java' : 'Gosling', 'Lisp' : 'McCarthy']

xmlDocument = new groovy.xml.StreamingMarkupBuilder().bind {
    mkp.xmlDeclaration()
    mkp.declareNamespace(computer: "Computer")
    languages {
        comment << "Created using StreamingMarkupBuilder"
        langs.each { key, value ->
            computer.language(name: key) {
                author (value)
            }
        }
    }
}
println xmlDocument
```

新版本的代码产生的输出如下：

```
<?xml version="1.0"?>
<languages xmlns:computer='Computer'>
    <!--Created using StreamingMarkupBuilder-->
    <computer:language name='C++'>
        <author>Stroustrup</author>
    </computer:language>
    <computer:language name='Java'>
        <author>Gosling</author>
    </computer:language>
    <computer:language name='Lisp'>
        <author>McCarthy</author>
    </computer:language>
</languages>
```

利用StreamingMarkupBuilder，借助该生成器支持的属性mkp，可以声明命名空间、XML注释等内容。一旦定义了一个命名空间，要将元素与命名空间关联起来，在前缀上使用点记号(.)即可，如computer.language，这里computer就是一个前缀。

XML的生成器，语法简单且优雅。我们不必使用XML的复杂语法来创建XML文档。创建XML输出也非常容易。然而如果要创建的是JSON输出，Groovy也考虑到了，下一节将会介绍。

17.2 构建 JSON

当创建Web服务，需要生成JSON格式的对象时，Groovy也提供了方便的解决方案^①。只需将实例发送给groovy.json.JsonBuilder的构造器，这个生成器会处理余下的工作，就这么简单。通过调用writeTo()方法，可以将生成的JSON格式写入到一个Writer中，如下面的例子所示：

UsingBuilders/BuildJSON.groovy

```
class Person {
    String first
    String last
    def sigs
    def tools
}
john = new Person(first: "John", last: "Smith",
    sigs: ['Java', 'Groovy'], tools: ['script': 'Groovy', 'test': 'Spock'])
bldr = new groovy.json.JsonBuilder(john)
writer = new StringWriter()
bldr.writeTo(writer)
println writer
```

该生成器使用字段的名字以及它们的值作为JSON格式的键和值，如下所示：

```
{"first": "John", "last": "Smith", "tools": {"script": "Groovy", "test": "Spock"},  
"sigs": ["Java", "Groovy"]}
```

产生输出毫不费力。如果想对输出加以定制，也需要多加几步。利用生成器流畅地创建指定的输出，如下面的例子所示。

UsingBuilders/BuildJSON.groovy

```
bldr = new groovy.json.JsonBuilder()
bldr {
    firstName john.first
    lastName john.last
    "special interest groups" john.sigs
    "preferred tools" {
        numberOfTools john.tools.size()
```

^① <http://www.json.org/>

```

        tools john.tools
    }
}
writer = new StringWriter()
bldr.writeTo(writer)
println writer

```

这里没有直接使用实例的属性名，而是为每个属性选择了不同的名字。还可以添加新属性，比如这个例子中的`numberOfTools`。生成器会使用我们提供的DSL语法来创建输出，其内容如下：

```
{"firstName":"John", "lastName": "Smith",
"special interest groups": ["Java", "Groovy"],
"preferred tools": {"numberOfTools": 2,
"tools": {"script": "Groovy", "test": "Spock"}}}
```

`JsonBuilder`可以从JavaBean、`HashMap`和列表生成JSON格式的输出。JSON格式的输出被保存在内存中，可以稍后再将其写入到一个流中，或是将其用于进一步处理。如果不将数据保存在内存中，而想在创建时就直接将其变为流，可以使用`StreamingJsonBuilder`代替`JsonBuilder`。

在Groovy中，反方向的处理也很容易；比如我们可以利用Groovy提供的`JsonSlurper`，从JSON数据创建`HashMap`。可以使用`parseText()`方法读取包含在`String`中的JSON数据。也可以使用`parse()`方法从`Reader`或文件中读取JSON数据。

下面我们解析一下前面例子中创建的JSON输出，假设现在输出保存在`person.json`文件中。

UsingBuilders/person.json

```
{"first": "John", "last": "Smith", "tools": {"script": "Groovy", "test": "Spock"},  
"sigs": ["Java", "Groovy"]}
```

除了来自文件，JSON数据也有可能来自一个Web服务。一旦以一个`Reader`实例的形式获得了数据流，就可以像下面例子中这样将其传给`parse()`方法。下面是处理`person.json`文件中的JSON数据的代码：

UsingBuilders/ParseJSON.groovy

```
def sluper = new JsonSlurper()  
def person = sluper.parse(new FileReader('person.json'))  
  
println "$person.first $person.last is interested in ${person.sigs.join(', ')}"
```

我们创建了一个`FileReader`，去读取文件中的数据。之后将其传给`parse()`方法，该方法会返回一个包含数据的`HashMap`实例。既可以像这里这样使用`HashMap`中的键和值，也可以从这些数据创建一个Groovy对象——还记得吗，可以以`HashMap`作为构造器的参数创建Groovy对象（参见2.2节）。

解析JSON数据的代码的输出如下：

```
John Smith is interested in Java, Groovy
```

解析JSON数据不费什么劲，简单得让人不安正是其便利性所在。

Groovy的生成器不仅仅能够生成数据，它们甚至可以让Swing编程体验变得相当漂亮，下一节将会介绍。

17.3 构建 Swing 应用

生成器之优雅并不局限于XML结构，Groovy还为创建Swing应用提供了一个生成器。当使用Swing时，我们需要执行一些很乏味的任务，比如创建组件（如按钮）、注册事件处理器等。通常，要实现一个事件处理器，要编写一个匿名内部类，而在实现处理器方法时，有些参数即使我们并不关心（如ActionEvent），也得接受它们。SwingBuilder，结合Groovy的闭包，帮我们去掉了这种苦差事。

可以使用生成器提供的嵌套或层次化结构来创建一个容器（如JFrame）及其组件（如按钮、文本框等）。Groovy提供的灵活的键值对可以初始化设施来初始化组件。定义事件处理器是小菜一碟，只需要向生成器提供一个闭包。尽管正在构建的Swing应用并不陌生，但是我们会发现，与用Java编写相比，用Groovy编写需要的代码要少一些。这有助于快速修改、试验并获得反馈。尽管仍在使用底层的Swing API，但语法有很大的不同。我们是在使用Groovy的方言与Swing对话^①。现在使用SwingBuilder类来创建一个Swing应用：

UsingBuilders/BuildSwing.groovy

```
bldr = new groovy.swing.SwingBuilder()

frame = bldr.frame(
    title: 'Swing',
    size: [50, 100],
    layout: new java.awtFlowLayout(),
    defaultCloseOperation:javax.swing.WindowConstants.EXIT_ON_CLOSE
) {
    lbl = label(text: 'test')
    btn = button(text: 'Click me', actionPerformed: {
        btn.text = 'Clicked'
        lbl.text = "Groovy!"
    })
}

frame.show()
```

图17-1显示了前面代码的输出。

^① <http://blog.agiledeveloper.com/2007/05/its-not-languages-but-their-idioms-that.html>

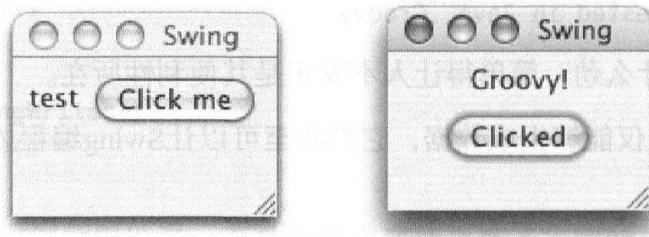


图17-1 使用SwingBuilder创建的一个小型Swing应用

这里初始化了一个JFrame实例，指定了它的title(标题)、size(大小)和layout(布局)，并设置了默认的关闭操作，一个简单的语句就搞定了这一切。这等价于Java中5条单独的语句。此外，注册事件处理器也很简单，只需要向button(这里代表JButton)的actionPerformed属性提供一个闭包。在Java中，要创建一个匿名内部类，并使用ActionEvent参数实现actionPerformed()方法，在Groovy中则不必这么辛苦。当然，这里有很多语法糖，但是看上去非常优雅，而且减少了代码量，这使得Swing API更易用了。

SwingBuilder生成器向我们演示了Groovy强大的表现力。它非常迷人，但是如果要创建任何比较大型的Swing应用，建议研究一下Griffon项目^①。Griffon是构建于Groovy之上的一一个框架，用于使用“约定优于配置”(Convention Over Configuration)的原则创建Swing应用。它不仅减轻了创建GUI的痛苦，还可以跨多个线程正确地处理事件。

17.4 使用元编程定制生成器

前面介绍过，有一些使用了嵌套或层次化结构或格式的专门化而且非常复杂的任务，对于这种任务，生成器提供了一种创建内部DSL的方式。当在应用中处理专门化的任务时，可以检查一下，是不是有生成器可以解决这个问题。如果没有找到任何生成器，可以自行创建。

定制生成器有两种创建方式：利用Groovy的元编程能力，一切自己来，本节采用的就是这种方式；使用Groovy提供的BuilderSupport(参见17.5节)或FactoryBuilderSupport(参见17.6节)。

为帮助理解BuilderSupport的优势，下面构建一个待办事项列表。下面的代码中使用了本节即将创建的生成器：

```
UsingBuilders/UsingTodoBuilder.groovy
bldr = new TodoBuilder()

bldr.build {
    Prepare_Vacation (start: '02/15', end: '02/22') {
        Reserve_Flight (on: '01/01', status: 'done')
    }
}
```

^① <http://griffon.codehaus.org>.

```

    Reserve_Hotel(on: '01/02')
    Reserve_Car(on: '01/02')
}
Buy_New_Mac {
    Install_QuickSilver
    Install_TextMate
    Install_Groovy {
        Run_all_tests
    }
}
}

```

在创建完ToDoBuilder之后，将会看到这段代码的输出如下：

```

To-Do:
- Prepare Vacation [start: 02/15 end: 02/22]
  x Reserve Flight [on: 01/01]
- Reserve Hotel [on: 01/02]
- Reserve Car [on: 01/02]
- Buy New Mac
  - Install QuickSilver
- Install TextMate
- Install Groovy
  - Run all tests

```

完成的任务使用x标记。缩进说明了任务的嵌套层次，而诸如开始日期等任务参数会紧跟在任务名称后面。

在上述用于待办事项列表的DSL中，我们创建了以诸如Reserve Car为名称的条目，不过这里用下划线代替了空格，这样就可以将其用作Groovy中的方法名了。`build()`是其中唯一一个提前确定的方法。其余的方法和属性都是通过`methodMissing()`和`propertyMissing()`来处理的，如下所示：

17

UsingBuilders/TodoBuilder.groovy

```

class TodoBuilder {
    def level = 0
    def result = new StringWriter()
    def build(closure) {
        result << "To-Do:\n"
        closure.delegate = this
        closure()
        println result
    }

    def methodMissing(String name, args) {
        handle(name, args)
    }

    def propertyMissing(String name) {

```

```

Object[] emptyArray = []
handle(name, emptyArray)
}

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

    if (args.length > 0 && args[-1] instanceof Closure) {
        def theClosure = args[-1]
        theClosure.delegate = this
        theClosure()
    }

    level--
}
def placeXifStatusDone(args) {
    args.length > 0 && args[0] instanceof Map &&
        args[0]['status'] == 'done' ? "x" : "-"
}

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

```

几乎全部是标准直接的Groovy代码，而且很好地应用了元编程。当被调用到不存在的方法或属性时，就假定它是一个条目。为检查调用时是不是提供了闭包，这里以-1为索引，获得了args中的最后一个参数，并对它进行了测试。之后将当前闭包的delegate设置为生成器，并调用该闭包向下遍历嵌套的任务。

创建定制生成器并不困难，不要犹豫。对于嵌套层次较深，而且大量使用Map和普通参数的

非常复杂的情况，下一节要介绍的BuilderSupport会有所帮助。

17.5 使用 BuilderSupport

上一节介绍了如何使用methodMissing()和propertyMissing()创建定制的生成器。如果要创建的生成器不止一个，可能要将某些方法识别代码重构到一个公共基类中。好在Groovy已经这么做了，BuilderSupport类提供了用于识别节点结构的便捷方法。这样就不用亲自编写处理结构的逻辑了，而只需要简单地监听调用，因为Groovy会遍历结构并采取相应的动作。扩展抽象类BuilderSupport感觉就像使用SAX（Simple API for XML，一个流行的事件驱动的XML解析器）。在解析与识别文档中的元素和属性时，它会触发我们提供的处理器上的事件。

在探索如何实现生成器之前，先来看看它能干什么：

UsingBuilders/UsingTodoBuilderWithSupport.groovy

```
bldr = new TodoBuilderWithSupport()

bldr.build {
    Prepare_Vacation (start: '02/15', end: '02/22') {
        Reserve_Flight (on: '01/01', status: 'done')
        Reserve_Hotel(on: '01/02')
        Reserve_Car(on: '01/02')
    }
    Buy_New_Mac {
        Install_QuickSilver
        Install_TextMate
        Install_Groovy {
            Run_all_tests
        }
    }
}
```

17

在创建完毕ToDo-BuilderWithSupport后再来运行前面的代码，输出如下：

```
To-Do:
- Prepare Vacation [start: 02/15 end: 02/22]
  x Reserve Flight [on: 01/01]
  - Reserve Hotel [on: 01/02]
  - Reserve Car [on: 01/02]
- Buy New Mac
  - Install QuickSilver
  - Install TextMate
  - Install Groovy
    - Run all tests
```

BuilderSupport期望我们实现两组具体的方法：setParent()和重载版本的createNode()。也可以视情况实现其他方法，比如nodeCompleted()。请记住在调用方法时可做的选