**Groovy学习笔记（四）：特殊注解**

**1.@Canonical**

如果要编写的toString()方法只是简单地显示以逗号分隔的字段值，则可以使用@Canonical变换让Grooovy编译器帮来干这个活。

默认情况下，它生成的代码会包含所有字段。不过可以让它仅包含特定字段，而去掉其他字段。

@Canonical

class Student {

String firstName

String lastName

int age

String address

}

def student = new Student(firstName: "Zhang",

lastName: "Sanfeng",

age: 16,

address: "China")

println student

* 1

输出：

Student(Zhang, Sanfeng, 16, China)

不管如何，这个注解在打印Log等情况下，可堪一用。

**2.@Delegate**

只有当派生类是真正可替换的，而且可代替基类使用时，继承才显示出其优势。从纯粹的代码复用角度看，对于其他大部分用途，**委托要优于继承**。

然而在Java中我们不太愿意使用委托，因为会导致代码冗余，而且需要更多工作。Groovy使委托变得非常容易，所以我们可以做出正确的设计选择。

class Worker {

def work() { println 'get work done' }

def analyze() { println 'analysis...'}

def writeReport() { println 'get report written' }

}

class Expert {

def analyze() { println 'expert analysis...' }

}

class Manager {

@Delegate

Expert expert = new Expert()

@Delegate

Worker worker = new Worker()

}

def manager = new Manager()

manager.analyze()

manager.work()

manager.writeReport()

在编译时，Groovy会检查Manager类，如果该类中没有被委托类中的方法，就把这些方法从被委托类中引入进来。因此，首先它会引入Expert类中的analyze()方法。

而从Worker类中，只会把work()和writeReport()方法因为进来。这时候，因为从Expert类带来的analyze()方法已经出现在Manager类中，所以Worker类中的analyze()方法会被忽略。

对于引入的每个方法，Groovy会简单地把对该方法的调用路由给实例上的相应方法，就像这样：publicObjectanalyze(){ expert.analyze() }。委托类会对新获得的方法做出响应，在下面的输出中可以看到：

expert analysis…  
get work done  
get report written

因为有了@Delegate注解，Manager类是可扩展的。如果在Worker或Expert类上添加或去掉了方法，不必对Manager类做任何修改，相应的变化就会生效。只需要重新编译代码，剩下的事Groovy会处理。

**3.@Immutable**

不可变对象天生是线程安全的，将其字段标记为final是很好的实践选择。**如果用@Immutable注解标记一个类，Groovy会将其字段标记为final的**，并且额外为我们创建一些便捷方法，从而使得“做正确的事情”变得更容易了。

@Immutable

class ImmutableStudent {

String name

int age

}

println new ImmutableStudent("jack", 24)

作为反馈，Groovy给我们提供了一个构造器，其参数以类中字段定义的顺序依次列出。在构造时间过后，字段就无法修改了。此外，Groovy还添加了hashCode()、equals()和toString()方法。

运行所提供的构造器和toString()方法:

ImmutableStudent(jack, 24)

**可以使用@Immutable注解轻松地创建轻量级的不可变值对象**。在基于Actor模型的并发应用中，线程安全是个大问题，而这些不可变值对象是作为消息传递的理想实例。

**4.@Lazy**

我们想把耗时对象的构建推迟到真正需要时。完全可以懒惰与高效并得，编写更少的代码，同时又能获得惰性初始化的所有好处。

下面的例子将推迟创建Heavy实例，直到真正需要它时。既可以在声明的地方直接初始化实例，也可以将创建逻辑包在一个闭包中。

class Heavy {

def size = 10

Heavy() { println "Creating Heavy : Size = $size" }

}

class AsNeed {

def value

@Lazy Heavy heavy1 = new Heavy()

@Lazy Heavy heavy2 = { new Heavy(size: value) }()

AsNeed() { println 'Created AsNeed' }

}

def need = new AsNeed(value: 1000)

println need.heavy1.size

println need.heavy1.size

println need.heavy2.size

**Groovy不仅推迟了创建，还将字段标记为volatile，并确保创建期间是线程安全的**。实例会在第一次访问这些字段的时候被创建，在输出中可以看到：

Created AsNeed  
Creating Heavy : Size = 10  
10  
10  
Creating Heavy : Size = 10  
1000

另一个好处是，@Lazy注解提供了一种轻松实现**线程安全**的**虚拟代理模式**（virtual proxy pattern）的方式。

**5.@Newify**

在Groovy中，经常会按照传统的Java语法，使用new来创建实例。然而，在创建DSL时，去掉这个关键字，表达会更流畅。

@Newify注解可以帮助创建类似Ruby的构造器，在这里，new是该类的一个方法。该注解还可以用来创建类似Python的构造器（也类似Scala的applicator），这里可以完全去掉new。要创建类似Python的构造器，必须向@Newify注解指明类型列表。

只有将auto=false这个值作为一个参数设置给@Newify，Groovy才会创建Ruby风格的构造器。可以在不同的作用域中使用@Newify注解，比如类或方法，如下面例子所示：

@Newify(value = [Student, ImmutableStudent])

def fluentCreate() {

println Student.new(firstName: "Zhang", lastName: "Sanfeng",

age: 24, address: "China")

println ImmutableStudent.new("LiHua", 29)

}

fluentCreate()

结果：

Student(Zhang, Sanfeng, 24, China)  
ImmutableStudent(LiHua, 29)

在创建DSL时，@Newify注解非常有用，它可以使得实例创建更像是一个隐式操作。

**6.@Singleton**

要实现单件模式，正常来讲，我们会创建一个静态字段，并创建一个静态方法来初始化该字段，然后返回单件实例。我们必须确保该方法是线程安全的，同时还要决定是否要惰性创建该单件。

而通过使用@Singleton变换则完全可以避免这种麻烦，如下面例子所示：

@Singleton(lazy = true, strict = false)

class President {

private President() {

println 'Instance'

}

def hello() {

println 'hello'

}

}

President.instance.hello()

President.instance.hello()

* 1

输出：

Instance  
hello  
hello

这里使用@Singleton注解标记了TheUnique类，以生成静态的getInstance()方法。因为此处将lazy属性的值设为了true，所以会将实例的创建延迟到请求时。

Groovy不仅将实例创建延迟到了最后责任时刻，还保证创建部分是线程安全的。