Closures（闭包）

本节主要讲groovy中的一个核心语法：closurs，也叫闭包。闭包在groovy中是一个处于代码上下文中的开放的，匿名代码块。它可以访问到其外部的变量或方法。

1. 句法

1.1 定义一个闭包

{ [closureParameters -> ] statements }

其中[]内是可选的闭包参数，可省略。当闭包带有参数，就需要->来将参数和闭包体相分离。

下面看一些闭包的具体例子：

{ item++ }

{ -> item++ }

{ println it }

{ it -> println it }

{ name -> println name }

{ String x, int y ->

println "hey ${x} the value is ${y}"

}

{ reader ->

def line = reader.readLine()

line.trim()

}

1.2 闭包也是对象

闭包在groovy中是groovy.lang.Closure类的实例，这使得闭包可以赋值给变量或字段。

def listener = { e -> println "Clicked on $e.source" }

assert listener instanceof Closure

Closure callback = { println 'Done!' }

Closure<Boolean> isTextFile = {

File it -> it.name.endsWith('.txt')

}

1.3 闭包的调用

闭包有两种调用方式：

def code = { 123 }

assert code() == 123

assert code.call() == 123

闭包名+（）或者闭包名.call()来调用闭包。

2. 参数

2.1 正常参数

闭包的参数类型和前面讲的方法的参数类型一样，这里不多说。

2.2 含蓄的参数

当闭包没有显式声明参数时，其默认包含一个隐式的参数it。

def greeting = { "Hello, $it!" }

assert greeting('Patrick') == 'Hello, Patrick!'

2.3 参数列表

参数列表的用法与普通方法一样，这里不多赘述。

3. 委托策略

委托策略是groovy中闭包独有的语法，这也使得闭包较java的lambda更为高级。下面简单介绍一下groovy中的委托策略。

3.1 Owner，delegate和this

在理解delegate之前，首先先要了解一下闭包中this和owner的含义，闭包中三者是这么定义的：

this 表示定义闭包的外围类。

owner 表示定义闭包的直接外围对象，可以是类或者闭包。

delegate 表示一个用于处理方法调用和属性处理的第三方类。

3.1.1 This

闭包中，使用this关键字或者调用方法getThisObject()来获得其外围类：

class Enclosing {

void run() {

def whatIsThisObject = { getThisObject() }

assert whatIsThisObject() == this

def whatIsThis = { this }

assert whatIsThis() == this

}

}

class EnclosedInInnerClass {

class Inner {

Closure cl = { this }

}

void run() {

def inner = new Inner()

assert inner.cl() == inner

}

}

class NestedClosures {

void run() {

def nestedClosures = {

def cl = { this }

cl()

}

assert nestedClosures() == this

}

}

判断this表示的具体是哪个对象可以从this往外找，遇到的第一类就是this代表的类。

3.1.2 Owner

owner与this类似，只不过owner表示的是直接外围对象，可以是类也可以是闭包：

class Enclosing {

void run() {

def whatIsOwnerMethod = { getOwner() }

assert whatIsOwnerMethod() == this

def whatIsOwner = { owner }

assert whatIsOwner() == this

}

}

class EnclosedInInnerClass {

class Inner {

Closure cl = { owner }

}

void run() {

def inner = new Inner()

assert inner.cl() == inner

}

}

class NestedClosures {

void run() {

def nestedClosures = {

def cl = { owner }

cl()

}

assert nestedClosures() == nestedClosures

}

}

上述例子与this中的例子不同的就是NestedClosures，其中owner表示的是nestedClosures而不是NestedClosures。

3.1.3 Delegate

闭包中可以使用delegate关键字或者getDelegate()方法来得到delegate变量，它默认与owner一致，但可以由用户自定义其代表的对象。

class Enclosing {

void run() {

def cl = { getDelegate() }

def cl2 = { delegate }

assert cl() == cl2()

assert cl() == this

def enclosed = {

{ -> delegate }.call()

}

assert enclosed() == enclosed

}

}

闭包中的delegate可被指向任意对象，我们看下面这个例子：

class Person {

String name

}

class Thing {

String name

}

def p = new Person(name: 'Norman')

def t = new Thing(name: 'Teapot')

定义了两个拥有相同属性name的类Person和Thing。接着定义一个闭包，其作用是通过delegate来获得name属性。

def upperCasedName = { delegate.name.toUpperCase() }

接着改变闭包的delegate的指向，我们可以看到闭包调用结果也不同：

upperCasedName.delegate = p

assert upperCasedName() == 'NORMAN'

upperCasedName.delegate = t

assert upperCasedName() == 'TEAPOT'

3.1.4 Delegate策略

在闭包中，当一个属性没有指明其所有者的时候，delegate策略就会发挥作用了。

class Person {

String name

}

def p = new Person(name:'Igor')

def cl = { name.toUpperCase() } //❶

cl.delegate = p //❷

assert cl() == 'IGOR' //❸

可以看到❶处的name没有指明其所有者。即这个name属性压根不知道是谁的。在❷处指明cl的delegate为p，这时候在❸处调用成功。

以上代码之所以可以正常运行是因为name属性会被delegate处理。这是一个十分强大的方式用于解决闭包内的属性的访问或方法的调用。在❶处没有显示的使用delegate.name是因为delegate策略已经在程序运行的时候帮助我们这样做了。下面我们看看闭包拥有的不同的delegate策略：

Closure.OWNER\_FIRST 这是默认的策略，优先从owner中寻找属性或方法，找不到再从delegete中寻找。上面的例子就是因为在owner中没有找到name，接着在delegate中找到了name属性。

Closure.DELEGATE\_FIRST 与OWNER\_FIRST相反。

Closure.OWNER\_ONLY 只在owner中寻找。

Closure.DELEGATE\_ONLY 只在delegate中寻找。

Closure.TO\_SELF 在闭包自身中寻找。

下面我们看一下默认的Closure.OWNER\_FIRST的用法：

class Person {

String name

def pretty = { "My name is $name" }

String toString() {

pretty()

}

}

class Thing {

String name

}

def p = new Person(name: 'Sarah')

def t = new Thing(name: 'Teapot')

assert p.toString() == 'My name is Sarah'

p.pretty.delegate = t //❶

assert p.toString() == 'My name is Sarah' //❷

尽管在❶处将delegate指向了t，但因为是owner first的缘故，还是会优先使用Person的name属性。

略做修改：

p.pretty.resolveStrategy = Closure.DELEGATE\_FIRST

assert p.toString() == 'My name is Teapot'

这时候就会访问t的name属性了。

下面再来看一个例子：

class Person {

String name

int age

def fetchAge = { age }

}

class Thing {

String name

}

def p = new Person(name:'Jessica', age:42)

def t = new Thing(name:'Printer')

def cl = p.fetchAge

cl.delegate = p

assert cl() == 42

cl.delegate = t

assert cl() == 42

cl.resolveStrategy = Closure.DELEGATE\_ONLY

cl.delegate = p

assert cl() == 42

cl.delegate = t

try {

cl()

assert false

} catch (MissingPropertyException ex) {

// "age" is not defined on the delegate

}

当使用了Closure.DELEGATE\_ONLY后，若delegate中找不到age属性，则会直接报错。

4. GStrings中的闭包

先来看一下下面这段代码：

def x = 1

def gs = "x = ${x}"

assert gs == 'x = 1'

OK,运行没有问题，那如果加两行代码呢？

x = 2

assert gs == 'x = 2'

这里就会报错了，错误原因有两：

GString只是调用了字符串的toString方法来获得值。

${x}这种写法并不是一个闭包，而是一个表达式等价于$x,当GString被创建的时候该表达式会被计算。

所以当给x赋值2的时候，gs已经被创建，表达式也已经被计算，结果是x = 1,所以gs得值就是固定的x = 1。

如果要在GString使用闭包也是可以的，如下：

def x = 1

def gs = "x = ${-> x}"

assert gs == 'x = 1'

x = 2

assert gs == 'x = 2'

总结

到这里groovy中闭包的基本用法结束了，更多闭包的用请参考

Closures

还记得我们学习groovy的目的是什么吗？对了，就是gradle。而且在gradle中使用了大量闭包的概念，所以在学习gradle之前还请好好掌握闭包这一节内容。😀

博客新地址www.zqlite.com