Groovy闭包:this、owner、delegate（规格文件节选翻译）

Azur_wxj 关注
2017.10.13 19:59:25 字数 1,770 阅读 943

这一段翻译自Groovy的规格文件的 3.2 Owner, delegate and this

3.2. Owner, delegate and this

为了明白**delegate**的概念，我们必须首先解释清楚在闭包中关键字**this**的意义。一个闭包通常定义了一下三个量：

- **this** 指的是闭包定义所在的类
- **owner** 指的是闭包定义所在的对象，这个对象可能是类也可能是另一个闭包。【这是因为闭包是可以嵌套定义的。】
- **delegate** 指的是一个第三方的（委托）对象，当在闭包中调用的方法或属性没有定义时，就会去委托对象上寻找。

3.2.1. this的意义

在闭包中，调用`getThisObject`返回这个闭包定义所在的对象。它等价于使用一个显式的`this`：

```
class Enclosing {
    void run() {
        def whatIsThisObject = { getThisObject() }  //#1
        assert whatIsThisObject() == this           //#2
        def whatIsThis = { this }                       //#3
        assert whatIsThis() == this                     //#4
    }
}
class EnclosedInInnerClass {
    class Inner {
        Closure cl = { this }                           //#5
    }
    void run() {
        def inner = new Inner()
        assert inner.cl() == inner                  //#6
    }
}
class NestedClosures {
    void run() {
        def nestedClosures = {
            def cl = { this }                           //#7
            cl()
        }
        assert nestedClosures() == this             //#8
    }
}
```

1. 在`Enclosing`中定义一个闭包，并返回`getThisObject`
2. 调用闭包将返回Enclosing类的一个实例。
3. 通常你会使用这种便捷的语法
4. 并且它将会返回同一个对象
5. 如果闭包定义在一个内部类中
6. 则闭包中的`this`将返回这个内部类，而不是顶级类
7. 在闭包嵌套的情况中，如此处的`c1`闭包定义在`nestedClosures`闭包中

推荐阅读
全局Context无侵入式获取
阅读 4,911
五轮阿里面试题及答案
阅读 67,466
解决DialogFragment内存泄漏问题
阅读 794
为什么要用newInstance来实例化Fragment?
阅读 1,717
疫情结束了，我想去离婚
阅读 172,433

Azur_wxj 关注 赞赏支持

```groovy
class Person {
    String name
    int age
    String toString() { "$name is $age years old" }

    String dump() {
        def cl = {
            String msg = this.toString()              //#1
            println msg
            msg
        }
        cl()
    }
}
def p = new Person(name:'Janice', age:74)
assert p.dump() == 'Janice is 74 years old'
```

该闭包在`this`上调用了`toString`方法，即外层包裹类的`toString`方法，它返回了Person类实例的一个描述。

3.2.2. 闭包的Owner

闭包的`owner`属性和闭包的`this`非常相似，但是有一点微妙的差别：它返回这个闭包的直接包裹对象，可能是外层的嵌套闭包，也可能是一个类。

```groovy
class Enclosing {
    void run() {
        def whatIsOwnerMethod = { getOwner() }    //#1
        assert whatIsOwnerMethod() == this            //#2
        def whatIsOwner = { owner }                    //#3
        assert whatIsOwner() == this                  ///#4
    }
}
class EnclosedInInnerClass {
    class Inner {
        Closure cl = { owner }                         //#5
    }
    void run() {
        def inner = new Inner()
        assert inner.cl() == inner                       //#6
    }
}
class NestedClosures {
    void run() {
        def nestedClosures = {
            def cl = { owner }                                  //#7
            cl()
        }
        assert nestedClosures() == nestedClosures            //#8
    }
}
```

1. 定义在`Enclosing`类中的闭包，返回`getOwner`
2. 调用该闭包将返回`Enclosing`的实例
3. 通常会使用这种简洁的语法
4. 返回同一个对象
5. 如果闭包是在一个内部类中定义的
6. 则`owner`指定是内部类，而不是顶级类
7. 但是在嵌套闭包的情况下，如此处的`c1`闭包定义在`nestedClosures`闭包内
8. 则`owner`指的是**外层嵌套的闭包，这一点与`this`不同！！**

3.2.3. 闭包的delegate

推荐阅读

全局Context无侵入式获取
阅读 4,911

五轮阿里面试题及答案
阅读 67,466

解决DialogFragment内存泄漏问题
阅读 794

为什么要用newInstance来实例化Fragment?
阅读 1,717

疫情结束了，我想去离婚
阅读 172,433

```groovy
class Enclosing {
    void run() {
        def cl = { getDelegate() }           //#1
        def cl2 = { delegate }                       //#2
        assert cl() == cl2()                                 //#3
        assert cl() == this                             //#4
        def enclosed = {
            { -> delegate }.call()                           //#5
        }
        assert enclosed() == enclosed                   //#6
    }
}
```

1. 通过调用`getDelegate()`得到闭包的`delegate`对象
2. 或使用`delegate`属性
3. 二者都返回同一个对象
4. 都是外围的包裹类或闭包
5. 尤其是在嵌套闭包的情况
6. 此时`delegate`就是`owner`

闭包的`delegate`属性可以被修改成任何对象。让我们通过创建两个类来说明这一点，这两个类都不是互相的子类但是它们都定义了一个名为`name`的属性：

```groovy
class Person {
    String name
}
class Thing {
    String name
}

def p = new Person(name: 'Norman')
def t = new Thing(name: 'Teapot')
```

然后我们定义一个闭包，它从`delegate`上取`name`属性并返回：

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

通过改变`delegate`，我们可以看到目标对象将被改变：

```groovy
upperCasedName.delegate = p
assert upperCasedName() == 'NORMAN'
upperCasedName.delegate = t
assert upperCasedName() == 'TEAPOT'
```

此时。这种*效果*和使用一个定义在闭包的词法作用域内的`target`对象没有差别：

```groovy
def target = p
def upperCasedNameUsingVar = { target.name.toUpperCase() }
assert upperCasedNameUsingVar() == 'NORMAN'
```

但是本质上却有重要区别：

- `target`是一个局部变量的引用
- `delegate`使用时可以不带有前缀

3.2.4. 委托策略

闭包内任何时候，只要使用一个没有被引用的属性，就会委托给delegate对象

推荐阅读

全局Context无侵入式获取
阅读 4,911

五轮阿里面试题及答案
阅读 67,466

解决DialogFragment内存泄漏问题
阅读 794

为什么要用newInstance来实例化Fragment?
阅读 1,717

疫情结束了，我想去离婚
阅读 172,433

Azur_wxj 关注 赞赏支持

```
}
def p = new Person(name:'Igor')
def cl = { name.toUpperCase() }//#1
cl.delegate = p            //#2
assert cl() == 'IGOR'    //#3
```

1. `name` 没有被闭包所在的词法作用域内任何一个变量引用
2. 改变 `delegate`，使之指向一个 `Person` 类实例
3. 方法调用成功

这段代码能够工作的原因是因为 `name` 属性将会被委托给 `delegate` 对象执行！这是在闭包内解析属性或方法调用的一种很有效的方法，从而没有必要设置一个显式的 `delegate.receiver`：根据默认的闭包委托策略，这种行为是可以的。一个闭包定义了多种可供选择的解析策略：

- `Closure.OWNER_FIRST` 是默认策略。如果一个属性/方法存在于 `owner` 上，那么就会在 `owner` 上调用；否则，就会委托给 `delegate`。
- `Closure.DELEGATE_FIRST` 和上面策略刚好相反： `delegate` 首先被委托，其次再委托给 `owner`。
- `Closure.OWNER_ONLY` 即只在 `owner` 上解析属性/方法， `delegate` 被忽略。
- `Closure.DELEGATE_ONLY` 即只在 `delegate` 上解析属性/方法， `owner` 被忽略。
- `Closure.TO_SELF` 可以被那些需要高级的元编程技术并却希望实现自定义的解析策略的开发者使用：解析并不是在 `delegate` 或者 `owner` 上进行，而是只在闭包类本身上进行。只有实现了自己的 `Closure` 子类，这么使用才是被允许的。

让我们用代码解释这个 `owner first` 的默认策略：

```
class Person {
    String name
    def pretty = { "My name is $name" }        //#1
    String toString() {
        pretty()
    }
}
class Thing {
    String name                                //#2
}

def p = new Person(name: 'Sarah')
def t = new Thing(name: 'Teapot')

assert p.toString() == 'My name is Sarah'           //#3
p.pretty.delegate = t                               //#4
assert p.toString() == 'My name is Sarah'   //#5
```

1. 定义一个闭包成员变量，它引用了 `name` 属性
2. `Thing` 类和 `Person` 类都定义了name属性
3. 使用默认策略， `name` 属性会在 `owner` 上解析（断言为真）
4. 改变 `delegate` 使之指向 `Thing` 的实例
5. 并没有什么被改变，依然在 `owner` 上解析（断言为真）

然而我们可以改变这种默认策略：

```
p.pretty.resolveStrategy = Closure.DELEGATE_FIRST
assert p.toString() == 'My name is Teapot'
```

通过改变 `resolveStrategy`，我们可以修改Groovy解析"隐式 `this` "引用的方式：在这种情况下，`name` 会首先在 `delegate` 上查询，如果没找到，则到 `owner` 上。因为 `name` 在 `delegate`（引用 `Thing` 类的实例）上有定义，所以被使用。

推荐阅读

全局Context无侵入式获取
阅读 4,911

五轮阿里面试题及答案
阅读 67,466

解决DialogFragment内存泄漏问题
阅读 794

为什么要用newInstance来实例化Fragment?
阅读 1,717

疫情结束了，我想去离婚
阅读 172,433

Groovy闭包:this、owner、delegate（规格文件节选翻译）

Azur_wxj 关注 赞赏支持

```groovy
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
}
```

在这个例子中，我们定义了两个类，它们都含有`name`属性但是只有`Person`类声明了`age`属性。`Person`类同时也声明了一个闭包，其中引用了`age`属性。我们可以将默认解析策略从"owner first"改变到"delegate only"。因为这个闭包的`owner`是`Person`类，所以我们可以检查`delegate`是不是`Person`类的一个实例，如果是，那么就能成功调用这个闭包，但是我们将它的`delegate`改成了`Thing`类的实例，所以运行时会报`groovy.lang.MissingPropertyException`异常。尽管这个闭包被定义在Person类中，但是owner并没有被使用。

> 关于如何利用这一特性来开发DSL（领域特定语言）的详细说明可以参照：dedicated section of the manual.

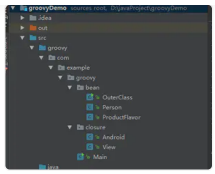
Azur_wxj
总资产11 (约0.85元) 共写了12.7W字 获得139个赞 共46个粉丝
关注

Groovy闭包:this、owner、delegate（规格文件节选翻译）

Azur_wxj 关注 赞赏支持

[Android Gradle] 搞定Groovy闭包这一篇就够了

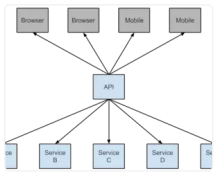
努力的人，应该像好色那样好学 做Android开发的同学，对Gradle肯定不陌生，我们用它配置、构建工程，可能还...



HitenDev 阅读 5,951 评论 6 赞 31

Spring Cloud

Spring Cloud为开发人员提供了快速构建分布式系统中一些常见模式的工具（例如配置管理，服务发现，断路器，智...



卡卡罗2017 阅读 80,112 评论 12 赞 120

iOS面试题300+

*面试心声:其实这些题本人都没怎么背,但是在上海 两周半 面了大约10家 收到差不多3个offer,总结起来就是把...

Dove_iOS 阅读 21,681 评论 40 赞 454

Groovy 闭包

本文介绍了Groovy闭包的有关内容。闭包可以说是Groovy中最重要的功能了。如果没有闭包，那么Groovy除了...

乐百川 阅读 4,739 评论 3 赞 11

百战程序员V1.2——尚学堂旗下高端培训_ Java1573题

百战程序员_ Java1573题 QQ群：561832648489034603 掌握80%年薪20万掌握50%年薪...

Albert陈凯 阅读 11,213 评论 2 赞 31

推荐阅读

全局Context无侵入式获取
阅读 4,911

五轮阿里面试题及答案
阅读 67,466

解决DialogFragment内存泄漏问题
阅读 794

为什么要用newInstance来实例化Fragment?
阅读 1,717

疫情结束了，我想去离婚
阅读 172,433