

Ruby元编程读书笔记 第三章 代码块

作者 bluetea

网站<https://github.com/bluetea>

块是一种控制作用域(scope)的强大手段,也是被称为“可调用对象”的一员,可调用对象还包括 proc lambda这样的对象
内容结构:

一.块的基础知识

二.作用域(scope)概述以及怎么想闭包(closure)一样通过块携带变量穿越作用域

三.通过传递块传递给 Instance_eval()方法来进一步控制作用域

四.把块转换为proc或者lambda这样的可调用对象(callable object),以便留待以后使用。

一块的基础知识

只有调用一个方法时,才可以定义一个块,块会被直接传递给这个方法,该方法用的是yield回调这个块。块可以有自己的参数,当回调块时,可以像调用方法那样为块提供参数值,像方法一样,块中最后一行代码的执行结果会作为返回值。

```
block.rb
1 def my_method(a,b)
2   a + yield(a, b)
3 end
4
5 p my_method(2, 3){|a, b| a + b}
```

```
1. bash
wangmjcdMBP:ruby wangmjc$ ruby block.rb
7
wangmjcdMBP:ruby wangmjc$
```

```
block.rb
1 def my_method(a,b)
2   a + yield(a, b)
3 end
4 p my_method(2, 3)
5
```

```
1. bash
wangmjcdMBP:ruby wangmjc$ ruby block.rb
block.rb:2:in `my_method': no block given (yield) (LocalJumpError)
    from block.rb:4:in `'
wangmjcdMBP:ruby wangmjc$
```

在方法中可以用block_given?()方法来询问当前方法的调用是否有块,以做出不同的处理,例如:

```
block.rb
1 def my_method(a,b)
2   if block_given?
3     a + yield(a, b)
4   else
5     return a
6   end
7 end
8
9
10 p my_method(2, 3){|a, b| a + b}
11 p my_method(2, 3)
```

```
1. bash
wangmjcdMBP:ruby wangmjc$ ruby block.rb
7
2
wangmjcdMBP:ruby wangmjc$
```

using关键字,模仿c#的using关键字写出一个using方法,不管对象是否出现异常,确保对象的关闭
测试方法如下

```
wangmjcdMBP:ruby wangmjc$ ruby block.rb
Loaded suite block
Started
..
Finished in 0.001365 seconds.

-----
2 tests, 3 assertions, 0 failures, 0 errors, 0 pendings, 0 omissions, 0 notifications
100% passed
-----
1465.20 tests/s, 2197.80 assertions/s
wangmjcdMBP:ruby wangmjc$
```

尽管我们不能为using定义一个关键字,但是可以通过内核方法(kernel模块的实例方法,就是内核方法)定义一个using,相当于关键字

了,这个Kernel#using方法以Resource对象作为参数,它还接受一个块,无论是否有块,ensure都会保证Resource#dispose方法被调用来模拟关闭。

如果发生了异常,那么Kernel#using会把这个异常重新抛给调用者。

3.3闭包

当定义一个块时他会获取当前环境中的绑定,并且把它传给一个方法时(调用块的方法),它会带着这些绑定一起进入方法,例如:

```
closure.rb
1 def my_method
2   x= "GoodBye"
3   yield("Curel")
4 end
5 x = "Hello"
6 p my_method{|y| "#{x} #{y} World!!"}
```

```
1. bash
bash
wangmjcdMBP:ruby wangmjc$ ruby closure.rb
"Hello Curel World!!"
wangmjcdMBP:ruby wangmjc$
```

块绑定的x是在块定义的时候,当前作用域(scope)的xmy_method里面的x对块是不可见的。基于这样的特性成为闭包。也就意味着一个块可以获取局部绑定,可以一直跟着他们。

作用域

在作用域中,你会看到到处都是绑定,会有一堆局部变量,你会发现自己站在一个对象中,有自己的方法和实例变量,还有一些常量,甚至还有一堆全局变量,这个对象就是当前对象,也成为self。

注意块的局部变量

块在定义的时候,会获取周围的绑定(获取当前作用域的绑定),可以在块的内部定义额外的绑定(变量),但是这些绑定在块结束时就会消失。

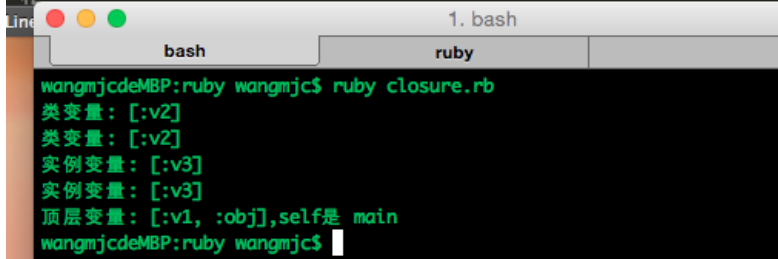
```
closure.rb
1 def my_mehtod
2   yield
3 end
4 top_level_variable = 1
5
6 my_mehtod do
7   top_level_variable += 2
8   block_local_variable = 3
9 end
10
11 p top_level_variable # => 3
12 p block_local_variable # => Error
```

```
1. bash
bash
wangmjcdMBP:ruby wangmjc$ ruby closure.rb
3
closure.rb:12:in `<main>': undefined local variable or method `block_local_variable' for main:Object (NameError)
wangmjcdMBP:ruby wangmjc$
```

切换作用域

根据前面的例子延时作用域,用Kernel#local_variable()方法来跟踪绑定的名字:

```
1 v1 = 1
2 class MyClass
3   v2 = 2
4   puts "类变量: #{local_variables}" # => [:v2]
5
6   def my_method
7     v3 = 3
8     puts "实例变量: #{local_variables}" # => [:v3]
9   end
10  puts "类变量: #{local_variables}" # => [:v2]
11 end
12 obj = MyClass.new
13 obj.my_method
14 obj.my_method
15 puts "顶层变量: #{local_variables},self是 #{self}" #[:v1, :obj]
```



```
wangmjcdMBP:ruby wangmjc$ ruby closure.rb
类变量: [:v2]
类变量: [:v2]
实例变量: [:v3]
实例变量: [:v3]
顶层变量: [:v1, :obj],self是 main
wangmjcdMBP:ruby wangmjc$
```

ruby中的作用域之间是截然分开的，一旦进入新的作用域，原先的绑定会被彻底替换为一组新的绑定(变量)。上面这个程序一共3个作用域，v1在顶级作用域，和v2在的MyClass作用域，还有v3的my_method作用域。只有生成了一个MyClass实例后，调用了my_method方法，才打开了my_method作用域。

程序调用了2此my_method方法，两个都是全新的作用域，不是同一个的

作用域门

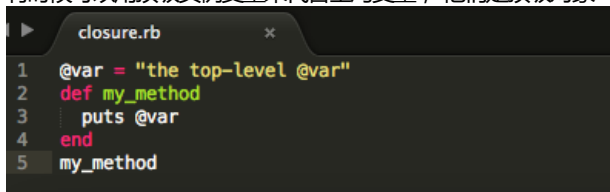
程序会在3个地方关闭前一个作用域，同时打开一个新的作用域：

- 类定义
- 模块定义
- 方法定义

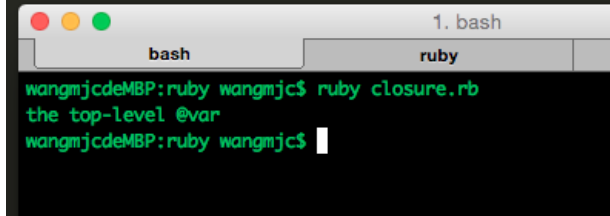
只要程序进入类定义，模块定义，方法定义都会发生做作用域切换，三个边界分别为 class, module, def 关键字，每个关键字都是一个作用域们

注意：全局变量可以在任何作用域访问，并且任何人都可以修改，所以基本无法追踪，所以不要使用全局变量。

有时候可以用顶级实例变量来代替全局变量，他们是顶级对象 main的实例变量。



```
closure.rb
1 @var = "the top-level @var"
2 def my_method
3   puts @var
4 end
5 my_method
```



```
wangmjcdMBP:ruby wangmjc$ ruby closure.rb
the top-level @var
wangmjcdMBP:ruby wangmjc$
```

只要main是self的时候都是可以访问这个顶级实例变量的，但是当其他变量成为self的时候，就会退出顶级作用域了

扁平化作用域

作用域门很难穿越，在进入另一个作用域时，局部变量会立刻失效，一次如何让局部变量穿越作用域们呢

穿越 class作和 def用域门

```
closure.rb
1 my_var = "success"
2 MyClass = Class.new do
3   puts "#{my_var} in the class definition" # => 可以成功用Class.new代替 class
4
5   define_method :my_method do
6     puts "#{my_var} in the method definition" # => 可以成功, 用define_method代替def
7   end
8 end
9 MyClass.new.my_method

1. bash
bash ruby
wangmjcdeMBP:ruby wangmjc$ clear
wangmjcdeMBP:ruby wangmjc$ ruby closure.rb
success in the class definition
success in the method definition
```

例如：

这种技巧叫做 嵌套文法作用域(nested lexical scopes)，也成为扁平作用域，

共享作用域

例如：想在一系列方法之间共享一个局部变量，但是又不希望其他方法可以访问者变量，就可以把这些方法定义在那个变量所在扁平作用域中，例如：

```
closure.rb
1 class MyClass
2   def my1
3     puts var = 1 #方法内部的局部变量
4     Kernel.send :define_method, :my2 do #创建的my2方法为Kernel#my2方法
5       puts var
6     end
7     Kernel.send :define_method, :my3 do |x|
8       puts var + x
9     end
10  end
11
12  def my4
13    puts var
14  end
15 end
16 obj = MyClass.new
17 obj.my1
18 obj.my2
19 obj.my3(10)
20 puts Kernel.instance_methods.grep /my3/
21 obj.my4

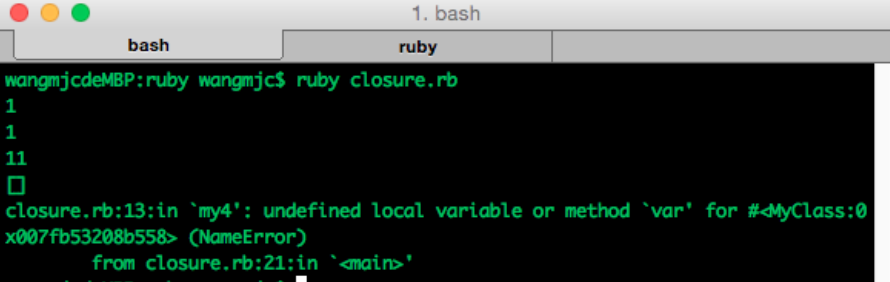
1. bash
bash ruby
wangmjcdeMBP:ruby wangmjc$ ruby closure.rb
1
1
11
my3
closure.rb:13:in `my4': undefined local variable or method `var' for #<MyClass:0x007fc0c4837598> (NameError)
    from closure.rb:21:in `<main>'
wangmjcdeMBP:ruby wangmjc$
```

使用了2个内核方法（不得不使用动态派发技术来访问Kernel的define_mehtod方法（私有方法）），所以创建的是Kernel的实例方法，所以obj实例也是可以访问的，Kernel#my2和Kernel#my3方法都可以看到局部变量var，而外部定义的my4则无法看到var变量

所以这种用来共享变量的技巧称为 共享作用域（shared scope）

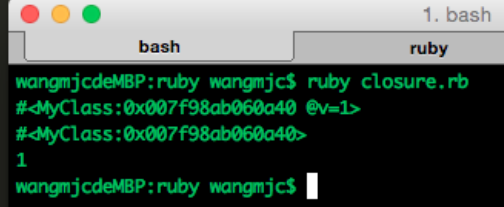
如果上面的例子，不想创建my2和my3为Kernel方法，只想是MyClass的实例方法用下面方式：

```
1 class MyClass
2   def my1
3     puts var = 1 #方法内部的局部变量
4     self.class.send :define_method, :my2 do #创建的my2方法为MyClass#my2方法
5       puts var
6     end
7     self.class.send :define_method, :my3 do |x|
8       puts var + x
9     end
10  end
11
12  def my4
13    puts var
14  end
15 end
16 obj = MyClass.new
17 obj.my1
18 obj.my2
19 obj.my3(10)
20 p Kernel.instance_methods.grep /my3/
21 obj.my4
```



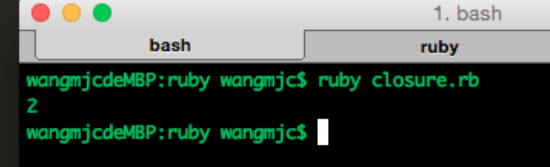
3.4instance_eval()方法--另一种混合代码和绑定的方式

```
1 class MyClass
2   def initialize
3     @v =1
4   end
5 end
6 obj = MyClass.new
7 p obj
8 obj.instance_eval do
9   puts self
10  puts @v
11 end
```



在运行时，给块的接受者为self，因此他可以访问接受者的私有方法和实例变量，例如@v, instance_eval甚至在不碰其它绑定的情况下修改self对象的变量和方法，例如:

```
1 class MyClass
2   def initialize
3     @v =1
4   end
5 end
6 obj = MyClass.new
7 v = 2
8 obj.instance_eval do
9   @v = v
10  puts @v
11 end
```



以上的代码是在扁平作用域中执行，因此可以访问局部变量v,但是块因为把运行他的对象（obj）作为self，所以它可以访问obj的实例变量@v，可以称instance_eval为上下文探针(context probe),它就像深入到对象中的代码片段，进行操作

和instance_eval相似的方法，为instance_exec(),类似，但是允许传递参数，例如

```
closure.rb
1 class MyClass
2   def initialize
3     @v1, @v2 = 1, 2
4   end
5 end
6 obj = MyClass.new
7 obj.instance_exec(5) do |i|
8   p @v1 + @v2 + i
9 end
```

```
wangmjcdeMBP:ruby wangmjc$ ruby closure.rb
8
```

打破封装

用 instance_eval 可以进入对象，查看内部细节，在调试时很有用
原书用的是RSpec的例子，但是没有看太明白

洁净室

有时候我们会创建一个对象，仅仅是为了在其中执行块，这样的对象成为**洁净室**
例如：

```
CleanRoom.rb
1 class CleanRoom
2   def complex_calculation
3   end
4
5   def do_something
6   end
7
8 end
9
10 cr = CleanRoom.new
11 cr.instance_eval do
12   if complex_calculation > 10
13     do_something
14   end
15 end
```

洁净室仅仅是一个用来执行块的环境，他通常会暴露若干有用的方法供块调用

3.5 可调用对象

从底层来看，使用块需要两步

第一步：将代码打包备用

共3中方法可以打包块，（1）使用proc。（2）使用lambda。（3）使用方法

第二步：调用块（通过yield）

Proc对象

在ruby中块不是对象，需要通过proc把块转换为对象，转换为对象后（就有了Proc#call实例对象方法了），这个存储的块可以公别人调用。

一个proc就是一个转换成对象的块，通过把块传给Proc.new就可以创建一个proc了，以后就可以用Proc#call方法调用了

```
[1] pry(main)> b = Proc.new {|x| x + 1}
=> #<Proc:0x007faa43a80cd8@pry:1>
[2] pry(main)> b.call(2)
=> 3
```

call的参数就是往块中传参数

这种技术成为**延迟执行**

Ruby还有2个内核方法（Kernel Method），可以把块转换为 lambda和proc，其实Proc.new(), lambda(), proc()三种方法是有区别的。

```
[6] pry(main)> dec = lambda {|x| x - 1}
=> #<Proc:0x007faa448e97c0@pry:5 (lambda)>
[7] pry(main)> dec.class
=> Proc
[8] pry(main)> dec.call(4)
=> 3
[9] pry(main)>
```

&操作符

块是方法的一个匿名参数，方法通过yield可以运行一个块，但是下面两种情况，yield不适用

1.想把这个块传递给另外一个方法

2.想把这个块转换为一个proc，必须立刻调用，无法转换为proc存起来

这2种情况下，都需要指着这个块说，我想要这个块，就是把块转换为&block变量，为了做到这一点，需要给块取个名字，可以给这个方法添加一个特殊的参数，这个参数必须是方法参数列表的最后一个，且以&符号开头。&操作符的真正意义是 这是一个Proc对象，我要把

它当成一个块来使用。

例如:

```
cleanroom.rb x
1 def math(a, b)
2   yield(a, b) #需要跟随一个块
3 end
4 def teach_math(a, b, &block)
5   #传入一个块对象, 但是不是自己用为了传递给另外一个方法
6   puts math(a, b, &block) #真正需要块的方法, 这个这个方法才能调用成功
7 end
8 teach_math(2,3){|a, b| a + b}
```

```
1. bash
ruby bash
bogon:ruby wangmjc$ ruby cleanroom.rb
5
bogon:ruby wangmjc$
```

如果没有块传递给这个math会怎样呢, 是不符合math方法的参数定义需要一个块对象的

```
cleanroom.rb
1 def math(a, b)
2   yield(a, b) #需要跟随一个块
3 end
4 def teach_math(a, b)
5   #传入一个块对象, 但是不是自己用为了传递给另外一个方法
6   puts math(a, b) #真正需要块的方法, 这个这个方法才能调用成功
7 end
8 teach_math(2,3)
```

```
1. bash
ruby bash
bogon:ruby wangmjc$ ruby cleanroom.rb
cleanroom.rb:2:in `math': no block given (yield) (LocalJumpError)
      from cleanroom.rb:6:in `teach_math'
      from cleanroom.rb:8:in `'
```

```
cleanroom.rb x
1 def math(a, b)
2   yield(a, b) #需要跟随一个块
3 end
4 def teach_math(a, b, &block)
5   #传入一个块对象, 但是不是自己用为了传递给另外一个方法
6   puts math(a, b, &block) #真正需要块的方法, 这个这个方法才能调用成功
7 end
8 teach_math(2, 3)
```

```
1. bash
ruby bash
bogon:ruby wangmjc$ ruby cleanroom.rb
cleanroom.rb:2:in `math': no block given (yield) (LocalJumpError)
      from cleanroom.rb:6:in `teach_math'
      from cleanroom.rb:8:in `'
```

或者直接用proc把块对象化, 把这个块传递给这个方法但是格式要对, teach_math需要的是&格式的, 所以也要传&格式参数例如:

```
cleanroom.rb x
1 def math(a, b)
2   yield(a, b) #需要跟随一个块
3 end
4 def teach_math(a, b, &block)
5   #传入一个块对象, 但是不是自己用为了传递给另外一个方法
6   puts math(a, b, &block) #真正需要块的方法, 这个这个方法才能调用成功
7 end
8 b = lambda{|a, b| a + b}
9 teach_math(2, 3, b)
```

```
1. bash
ruby bash
bogon:ruby wangmjc$ ruby cleanroom.rb
cleanroom.rb:4:in `teach_math': wrong number of arguments (3 for 2) (ArgumentError)
      from cleanroom.rb:9:in `'
```

&操作符的真正意义是 将块转为proc, 然后传递到内部。简单的去掉&就会再得到一个proc对象。方法生成的是proc对象


```
cleanroom.rb
1 def my_method(&block) #需要的参数是一个proc对象（直接的block或proc对象）
2   block #有点类似于yield，需要跟随block
3 end
4 p = my_method{|name| puts "name is #{name}"}
5 puts p.class
6 p.call("wang")
7
8 block = Proc.new {|sex| puts "sex is #{sex}"}
9 p = my_method(&block) #再用一次&把proc对象转换为块了，而my_method需要块
10 puts p.class
11 p.call("female")
```

```
1. bash
ruby bash
bogon:ruby wangmjc$ ruby cleanroom.rb
Proc
name is wang
Proc
sex is female
bogon:ruby wangmjc$ ruby cleanroom.rb
Proc
name is wang
Proc
sex is female
```

把块转换为proc的方式，再用一次&，就是把proc对象转换回块。

```
cleanroom.rb
1 def my_method(greeting)
2   puts "#{greeting}, #{yield}"
3 end
4 my_proc = proc {"Bill"} #定义一个proc对象
5 my_method("Hello", &my_proc) #将proc对象转换为块传递给方法
```

```
1. bash
ruby bash
bogon:ruby wangmjc$ ruby cleanroom.rb
Hello , Bill
```

来自HighLine的例子

这个gem可以帮助你输入，输出自动化控制台，例如：可以让HighLine来收集用户用逗号分隔的输入，然后转换成数组，例如

```
cleanroom.rb
1 require "highline"
2
3 h1 = HighLine.new
4 friends = h1.ask("Friends?", lambda {|s| s.split(',')}) #需要的是block或者proc对象
5 puts "you are friends with : #{friends.inspect}"
```

```
1. bash
ruby bash
bogon:ruby wangmjc$ ruby cleanroom.rb
Friends?
wang, zhang, zhao,li
you are friends with : ["wang", " zhang", " zhao", "li"]
bogon:ruby wangmjc$
```

```
5 # Raises EOFError if input is exhausted.
6 #
7 def ask( question, answer_type = String, &details ) # :yields: question
8   @question ||= Question.new(question, answer_type, &details)
9
10   return gather if @question.gather
```

实际在HighLine源码中，这里ask需要的是一个proc对象，并把这个proc传递给一个Question类的对象。

proc与lambda的对比

主要区别：

1.与return关键字有关

return在lambda和proc中的对比

lambda的return只从lambda中返回


```
cleanroom.rb
def double(callable_object)
  callable_object.call(1) * 2
end

l = lambda {|x| return x * 10}
puts double(l)
```

```
1. bash
ruby
bagon:ruby wangmjc$ ruby cleanroom.rb
20
```

```
cleanroom.rb
1 def double(callable_object)
2   callable_object.call(1) * 2
3 end
4
5 l = proc {|x| return x * 10}
6 puts double(l)
```

```
1. bash
ruby
bagon:ruby wangmjc$ ruby cleanroom.rb
cleanroom.rb:5:in `block in <main>': unexpected return (LocalJumpError)
from cleanroom.rb:2:in `call'
from cleanroom.rb:2:in `double'
from cleanroom.rb:6:in `<main>'
```

在proc中的return不是从proc中返回，而是从定义这个proc中的作用域中返回

所以这个错误一目了然：

```
cleanroom.rb
1 def double(callable_object)
2   callable_object.call(1) * 2
3 end
4
5 l = proc {|x| return x * 10}
6 puts double(l)
```

```
1. bash
ruby
bagon:ruby wangmjc$ ruby cleanroom.rb
cleanroom.rb:5:in `block in <main>': unexpected return (LocalJumpError)
from cleanroom.rb:2:in `call'
from cleanroom.rb:2:in `double'
from cleanroom.rb:6:in `<main>'
```

程序试图从定义的作用域中返回，但是已经是在顶级作用域了，所以程序就会失败，但是可以用隐式的return来规避这个问题。

```
cleanroom.rb
1 def double(callable_object)
2   callable_object.call(1) * 2
3 end
4
5 l = proc {|x| x * 10}
6 puts double(l)
```

```
1. bash
ruby
bagon:ruby wangmjc$ ruby cleanroom.rb
20
```

2. 对参数检验的严格程度

例如：proc会自动调整自己期望的参数形式

```
[19] pry(main)> p = Proc.new{|a, b| [a, b]}
=> #<Proc:0x007f8a728ebdf0@pry:17>
[20] pry(main)> p.arity
=> 2
[21] pry(main)> p.call(1,2,3)
=> [1, 2]
[22] pry(main)> p.call(1)
=> [1, nil]
```

lambda会抛出一个ArgumentError错误

```
[23] pry(main)> l = lambda {la, bl [a, b]}
=> #<Proc:0x007f8a72413738@pry>:21 (lambda)>
[24] pry(main)> l.call(1,2)
=> [1, 2]
[25] pry(main)> l.call(1,2,3)
ArgumentError: wrong number of arguments (3 for 2)
from (pry):21:in `block in __pry__'
[26] pry(main)> l.call(1)
ArgumentError: wrong number of arguments (1 for 2)
from (pry):21:in `block in __pry__'
```

lambda更像方法，严格的检查传递的参数，可以用return返回，所以推荐使用

Kernel#proc方法

把块转换成对象的proc和Proc.new方法很想，而proc就是Kernel#proc方法，实际在1.9之后，proc就是 Proc.new的别名

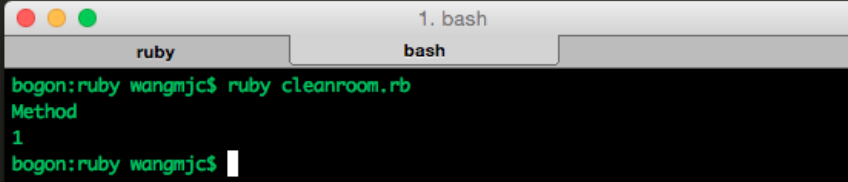
lambda的简单定义

```
[31] pry(main)> p = ->(x){x+1}
=> #<Proc:0x007f8a72542668@pry>:29 (lambda)>
[32] pry(main)> p.call(2)
=> 3
```

重访方法

方法，同样可调用对象家族的

```
1 class MyClass
2   def initialize(value)
3     @x = value
4   end
5
6   def my_method
7     @x
8   end
9 end
10
11 obj = MyClass.new(1)
12 m = obj.method :my_method #获得一个用Method对象表示的方法，通过Mehtod#call调用
13 puts m.class
14 puts m.call
```



```
1. bash
ruby bash
bagon:ruby wangmjc$ ruby cleanroom.rb
Method
1
bagon:ruby wangmjc$
```

通过Object#method方法可以获得一个用Method对象表示的方法，以后可以用Mehtod#call对它进行调用， Method对象类似于lambda，但最重要的是，Lambda会在定义它的作用域执行（他是一个闭包），但是Mehtod对象会在它自身的所在的对象作用域执行

```
[35] pry(main)> Object.instance_methods.grep /^method/
=> [:methods, :method]
```

用Method#unbind方法把一个方法跟他的对象分离，返回的是一个UnboundMethod对象,这个UnboundMehtod对象不能执行，需要把他绑定到同一个类的对象，才能执行，例如：

```
cleanroom.rb
1 class MyClass
2   def initialize(value)
3     @x = value
4   end
5
6   def my_method
7     @x
8   end
9 end
10
11 obj = MyClass.new(1)
12 m = obj.method :my_method #获得一个用Method对象表示的方法，通过Method#call调用
13 puts m.class
14 puts m.call
15
16 unbound = m.unbind
17 puts unbound.class
18 obj1 = MyClass.new(2)
19 m1 = unbound.bind(obj1)
20 puts m1.class
21 puts m1.call
```

```
1. bash
ruby
bagon:ruby wangmjc$ ruby cleanroom.rb
Method
1
UnboundMethod
Method
2
```

可以用Method#to_proc方法把Method对象转化为proc对象，也可以用define_method把块转化为方法，例如：

```
cleanroom.rb
1 class MyClass
2   def initialize(value)
3     @x = value
4   end
5
6   def my_method
7     @x
8   end
9 end
10
11 obj = MyClass.new(1)
12 m = obj.method :my_method #获得一个用Method对象表示的方法，通过Method#call调用
13 puts m.class
14 p = m.to_proc
15 puts p.class
16 puts p.call
```

```
1. bash
ruby
bagon:ruby wangmjc$ ruby cleanroom.rb
Method
Proc
1
bagon:ruby wangmjc$
```

用define_method把块转化为方法，例如

```
1 class MyClass
2   define_method :my_mehtod do
3     puts "把块转换为方法"
4   end
5 end
6
7 a = MyClass.new
8 a.my_mehtod
```

```
1. bash
ruby
bagon:ruby wangmjc$ ruby cleanroom.rb
把块转换为方法
bagon:ruby wangmjc$
```

可调用对象的总结

- 1.块 虽然不是真正的对象，但是是可以调用的：在定义时候的闭包(定义时的作用域)执行
- 2.proc：Proc类对象，跟块一样，也在定义时候的闭包执行（定义时的作用域）
- 3.lambda：也是Proc类的对象，但是跟普通的proc有些细微区别，也是闭包。
- 4.方法：绑定于对象，在所绑定对象的作用域中执行，也可以与这个对象解除绑定，再重新绑定到一个对象的作用域中执行区别：
 - 1.在方法和Lambda中，return语句会从可调用对象中返回，但是在块和proc中会从可调用对象的原始上线文中返回

2.方法和Lambda对传入的参数检查严格，但是Proc.new和proc要宽松很多。
我们可以将一种可调用对象转换为另外一种可调用对象。

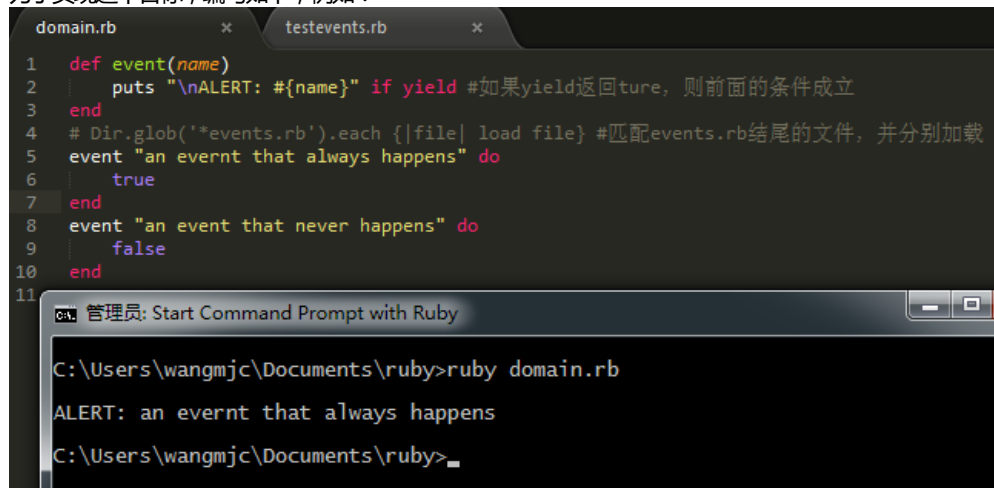
3.6 编写一种领域专属语言

为别人编写一个可以简单的 领域专属语言，例如

```
domain.rb
1 even "we are earning wads of money" do
2   recent_orders = ...
3   recent_orders > 1000
4 end
```

要定义一个事件，需要提供一个描述事件的名字以及一个代码块，如果这个块运行结果返回ture，则系统会调用邮件程序发送警告，如果返回false，则不会发生任何事情，系统没几分钟就检查一次所有事件

为了实现这个目标，编写如下，例如：

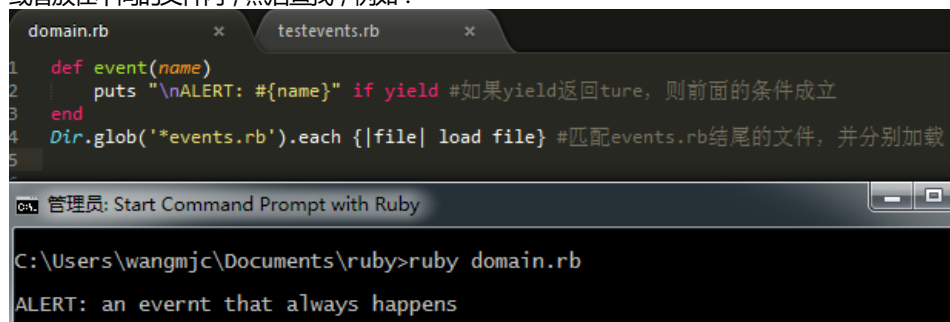


```
domain.rb      x  testevents.rb  x
1 def event(name)
2   puts "\nALERT: #{name}" if yield #如果yield返回ture，则前面的条件成立
3 end
4 # Dir.glob('*events.rb').each {|file| load file} #匹配events.rb结尾的文件，并分别加载
5 event "an evernt that always happens" do
6   true
7 end
8 event "an event that never happens" do
9   false
10 end
11
```

管理员: Start Command Prompt with Ruby

```
C:\Users\wangmjc\Documents\ruby>ruby domain.rb
ALERT: an evernt that always happens
C:\Users\wangmjc\Documents\ruby>_
```

或者放在不同的文件内，然后查找，例如：



```
domain.rb      x  testevents.rb  x
1 def event(name)
2   puts "\nALERT: #{name}" if yield #如果yield返回ture，则前面的条件成立
3 end
4 Dir.glob('*events.rb').each {|file| load file} #匹配events.rb结尾的文件，并分别加载
5
```

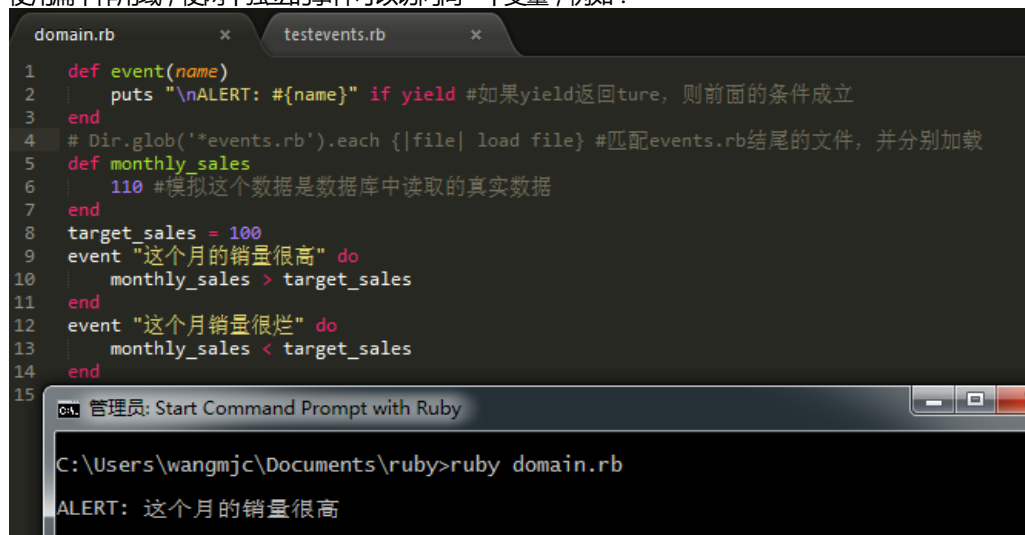
管理员: Start Command Prompt with Ruby

```
C:\Users\wangmjc\Documents\ruby>ruby domain.rb
ALERT: an evernt that always happens
```

这样，其他维护人员可以自己添加*events.rb文件，不需要程序员全程参与后续的维护

共享事件

使用扁平作用域，使两个独立的事件可以访问同一个变量，例如：



```
domain.rb      x  testevents.rb  x
1 def event(name)
2   puts "\nALERT: #{name}" if yield #如果yield返回ture，则前面的条件成立
3 end
4 # Dir.glob('*events.rb').each {|file| load file} #匹配events.rb结尾的文件，并分别加载
5 def monthly_sales
6   110 #模拟这个数据是数据库中读取的真实数据
7 end
8 target_sales = 100
9 event "这个月的销量很高" do
10   monthly_sales > target_sales
11 end
12 event "这个月销量很烂" do
13   monthly_sales < target_sales
14 end
15
```

管理员: Start Command Prompt with Ruby

```
C:\Users\wangmjc\Documents\ruby>ruby domain.rb
ALERT: 这个月的销量很高
```

更新上面的这个RedFlag DSL程序

增加一个setup指令来设置 变量

```
domain.rb  x  testevents.rb  x
1  def event(name)
2    puts "\nALERT: #{name}" if yield #如果yield返回ture, 则前面的条件成立
3  end
4  def setup
5    yield
6  end
7  # Dir.glob('*events.rb').each {|file| load file} #匹配events.rb结尾的文件, 并分别加载
8  setup do
9    puts "seting up sky"
10   @sky_height = 100
11 end
12 setup do
13   puts "seting up mountains"
14   @mountains_height = 200
15 end
16
17 event "it's getting closer" do
18   @sky_height < @mountains_height
19 end
```

```
管理员: Start Command Prompt with Ruby
C:\Users\wangmjc\Documents\ruby>ruby domain.rb
seting up sky
seting up mountains
ALERT: it's getting closer
```

setup为@前缀变量赋值, 事件来读取这些值, 这样所有的变量在setup中进行初始化。

版本迭代, 当前的DSL会立刻执行块, 新版本改为按照特定的顺序智选哪个块和事件
先自己了解下&格式proc传递规则:

```
test.rb  x
1  def method1(&block) #需要的是一个&格式proc对象
2    block             #延迟执行bloc
3  end
4  p l1 = lambda {puts "hello"}
5  p a1 = method1(&l1) #传递&格式proc对象, 然后生成proc类型a1
6  a1.call
7  puts "1-----"
8
9  def method2(&block) #需要的是一个&格式proc对象
10   # &block #这种模式是错误的, 将proc对象传递进来又不利用, 又不传递给其它方法
11 end
12 p l2 = lambda {puts "hello"}
13 p a2 = method1(&l2) #传递一个&格式的proc对象
14 a2.call
15 puts "2-----"
16
17 def method3(&block) #需要的是&格式的proc对象
18   block.call #在方法内部就执行了, 不延迟执行
19 end
20 p l3 = lambda {puts "hello"}
21 p a3 = method3(&l3)
22
23 puts "3-----"
24 def method4(block) #需要的是没有&格式的proc对象
25   block.call
26 end
27 p l4 = lambda {puts "hello"}
28 p a4 = method4(l4)
29 puts "4-----"
```

```
1. bash
ruby
bash
bogon:ruby wangmjc$ ruby test.rb
#<Proc:0x007fe0b9276e28@test.rb:4 (lambda)>
#<Proc:0x007fe0b9276e28@test.rb:4 (lambda)>
hello
1-----
#<Proc:0x007fe0b9276c48@test.rb:12 (lambda)>
#<Proc:0x007fe0b9276c48@test.rb:12 (lambda)>
hello
2-----
#<Proc:0x007fe0b9276ab8@test.rb:20 (lambda)>
hello
nil
3-----
#<Proc:0x007fe0b9276928@test.rb:27 (lambda)>
hello
nil
4-----
bogon:ruby wangmjc$
```

在每个执行中都建立一个洁净室, 然后在里面setup, 然后执行相应的event

```
dsl.rb  x
1
2  def event(name, &block) #传递&格式的proc对象
3    @events[name] = block #并延迟执行
4  end
5
6  def setup(&block)
7    @setups << block
8  end
9  Dir.glob("*events.rb").each do |file| #查找文件夹内的任何event文件的内容
10   @events = {}
11   @setups = []
12   load file
13   @events.each_pair do |name, event|
14     env = Object.new #每一次循环都创建一个新的env作用域, 相当于创建了一个洁净室
15     @setups.each do |setup|
16       env.instance_eval &setup #在这个作用域中执行setup的块, 设置变量
17     end
18     puts "ALERT: #{name}" if env.instance_eval &event #在每个新的env作用域判断event
19   end
20 end
21
```

建立的测试events.rb

```
test_events.rb
1 event "the sky is falling" do
2   @sky_height < 300
3 end
4
5 event "it's getting closer" do
6   @sky_height < @mountains_height
7 end
8
9 setup do
10  puts "setting up sky height"
11  @sky_height = 100
12 end
13
14 setup do
15  puts "setting up mountain height"
16  @mountains_height = 200
17 end
```

运行结果：

```
ruby bash
bogon:ruby wangmjc$ ruby dsl.rb
setting up sky height
setting up mountain height
ALERT: the sky is falling
setting up sky height
setting up mountain height
ALERT: it's getting closer
bogon:ruby wangmjc$
```

event()和setup()方法会使用&操作符把块转换为proc，然后非别存在@events和@setups中，这个顶级实例变量会被event()和setup()和主程序共享。

主程序加载以events.rb为结尾的文件，把每个文件的代码用event()和setup()方法调用，将块转换为proc对象存在相应的@events和@setups中。

然后load file加载文件内容，就会加载所有的event和setup，然后在一个Object的对象中执行（一个洁净室），然后在setup和event中的实例变量（@sky_height等）实际编程了env这个对象的实例变量，（这样每个event可以共享所有的setup里地实例变量的。）

在这个例子中，有@setups和@events这些实例变量（谁都可以读，有点类似于全局变量了），下面的例子就是消灭这些顶级变量

为了消除全局变量，可以使用一个共享作用域，很复杂啊

```
dsl.rb
1
2 lambda {
3   setups = [] #设置2个局部变量，不能被外部调用，第13行段和第20段有魔法
4   events = {}
5   Kernel.send :define_method, :event do |name, &block| #把方法跟随的块保存成proc对象保存起来
6     events[name] = block
7   end
8
9   Kernel.send :define_method, :setup do |&block|
10    #把方法跟随的块保存成proc对象保存起来
11    setups << block
12  end
13
14  Kernel.send :define_method, :each_event do |&block|
15    #定义一个新的内核方法，为了把局部变量内容传递出来
16    events.each_pair do |name, event|
17      #为了把自己的每对name和event传递给最后Dir方法调用时候的跟随的块
18      block.call(name, event) #用call相当于events里面内容都传递出来了
19    end
20  end
21
22  Kernel.send :define_method, :each_setup do |&block|
23    setups.each do |setup|
24      block.call(setup) #用call相当于把setups里面的内容都传递给外面block
25    end
26  end
27 }.call
28
29 Dir.glob("*events.rb").each do |file|
30   load file
31   #加载文件后，就会调用 :event和:setup两个内核方法，会将各个块保存在上面的2个局部变量内
32   each_event do |name, event|
33     #因为不在一个作用域，所以无法直接调用，但是each_event可以传递出来
34     env = Object.new
35     each_setup do |setup|
36       #因为不在一个作用域，所以无法直接调用，但是each_event可以传递出来
37       env.instance_eval &setup
38     end
39     puts "Alertt: #{name}" if env.instance_eval &event
40   end
41 end
```

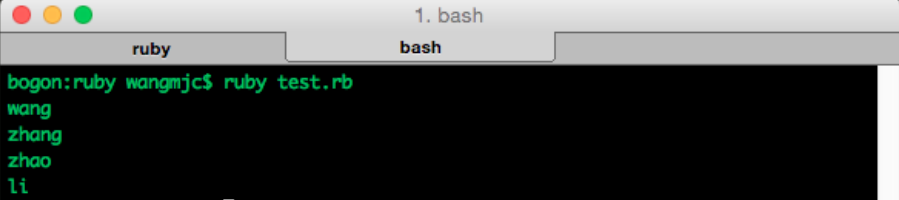
```
1. bash
ruby bash
bogon:ruby wangmjc$ ruby dsl.rb
setting up sky height
setting up mountain height
Alertt: the sky is falling
setting up sky height
setting up mountain height
Alertt: it's getting closer
bogon:ruby wangmjc$ open test_events.rb
bogon:ruby wangmjc$
```

```
test_events.rb
1 event "the sky is falling" do
2   @sky_height < 300
3 end
4
5 event "it's getting closer" do
6   @sky_height < @mountains_height
7 end
8
9 setup do
10  puts "setting up sky height"
11  @sky_height = 100
12 end
13
14 setup do
15  puts "setting up mountain height"
16  @mountains_height = 200
17 end
```

说明，共享作用域包含在lambda中，这个lambda会立即被调用，lambda中的代码中定义的是核心方法（内核方法）这就意味着所有的对象都会集成这些方法，包括main对象（运行Dir的self）。他们都共享2个局部变量 setups和events，在内核方法定义each_event和each_setup方法只是为了将局部变量setups和events的内容传递出来，然后外面的块就可以接受传递出来的内容了。但是在Kernel内定义一个

根据这个思想自己写了个test，测试下，如下：

```
test.rb
1 lambda {
2   names_collect = %w(wang zhang zhao li) #定义一个局部变量，是不能被外部访问的
3   Kernel.send :define_method, :each_name do |&block| #定义内核方法，就是为了让所有人都能调用
4     #需要一个封装为proc的对象
5     names_collect.each do |name|
6       block.call(name)
7       #这种方式可以将局部变量的内容都传递外面的块，相当于即使没有全局变流量也能把内容和别人共享
8     end
9   end
10 }.call
11
12 each_name do |x|
13   puts x
14 end
15
```



```
1. bash
ruby bash
bogon:ruby wangmjc$ ruby test.rb
wang
zhang
zhao
li
```