

21章 Proc类

作者 bluetea

网站 <https://github.com/bluetea>

20.1 所谓proc，就是使块对象化的类，Proc与块关系非常紧密，两种创建proc对象的方法

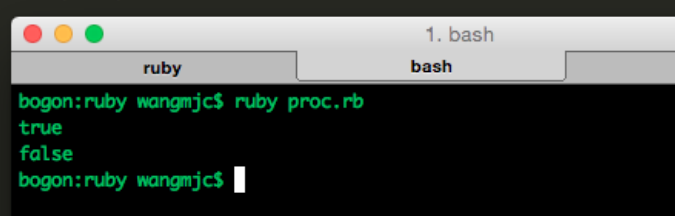
```
[29] pry(main)> hello = Proc.new do |name|
[29] pry(main)* puts "hello-#{name}"
[29] pry(main)* end
=> #<Proc:0x007fa76112ee80@pry:24>
[30] pry(main)> hello1 = proc { |name| puts name }
=> #<Proc:0x007fa7611060c0@pry:27>
```

通过调用Proc#call方法执行块，调用Proc#call方法时的参数会作为块变量，块中最后一个表达式的值会作为call的返回值，Proc#call还有一个名称叫Proc#[]

```
[36] pry(main)> hello1.call("wang")
=> 1
[37] pry(main)> hello1[2]
=> 1
[38] pry(main)>
```

例如判断给定的年是不是闰年

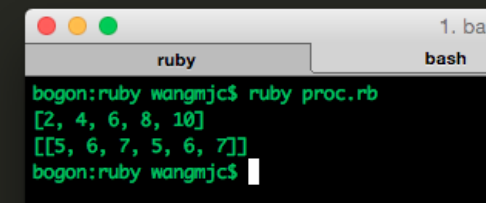
```
1 leap = proc do |year|
2   year % 4 == 0 && year % 100 != 0 || year % 400 == 0
3 end
4
5 p leap.call(2000)
6 p leap.call(2014)
```



```
1. bash
ruby
bagon:ruby wangmjc$ ruby proc.rb
true
false
bagon:ruby wangmjc$
```

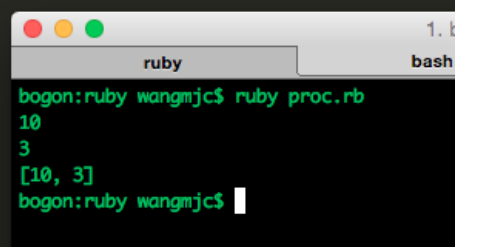
也可以将块变量设置为*数组，就可以想方法一样，以数组的形式接收可变数量的参数另外定义普通方法时可以使用的参数开式，如默认函数，关键字函数，几乎都可以用于块的定义

```
1 d = Proc.new do |*args|
2   args.map {|i| i*2 }
3 end
4
5 p d.call(1,2,3,4,5)
6 p d.call([5,6,7])
```



```
1. bash
ruby
bagon:ruby wangmjc$ ruby proc.rb
[2, 4, 6, 8, 10]
[[5, 6, 7, 5, 6, 7]]
bagon:ruby wangmjc$
```

```
1 d = Proc.new do |a=10, b|
2   p a, b
3 end
4
5 p d.call(3)
```



```
1. bash
ruby
bagon:ruby wangmjc$ ruby proc.rb
10
3
[10, 3]
bagon:ruby wangmjc$
```

21.1.1 lambda

lambda创建的proc更接近于方法，和proc的行为两点不同

1.lambda的参数检查更加严密,如果参数数量不正确，lambda会产生错误

```
proc.rb
1 prc1 = Proc.new do |a, b, c|
2   p [a, b, c]
3 end
4 prc1.call(2,3)

bagon:ruby wangmjc$ ruby proc.rb
[2, 3, nil]
bagon:ruby wangmjc$

proc.rb
1 prc2 = lambda do |a, b, c|
2   p [a, b, c]
3 end
4 prc2.call(2,3)

bagon:ruby wangmjc$ ruby proc.rb
proc.rb:1:in `block in <main>': wrong number of arguments (2 for 3) (ArgumentError)
    from proc.rb:4:in `call'
    from proc.rb:4:in `<main>'
bagon:ruby wangmjc$
```

第二个不同点，lambda可以用return将值从块中返回，注意是用return将值返回，如果用默认返回，lambda和proc是没区别的

```
proc.rb
1 prc2 = proc do |a, b, c|
2   a+b+c
3 end
4 p prc2.call(2,3,4)

bagon:ruby wangmjc$ ruby proc.rb
9
bagon:ruby wangmjc$

proc.rb
1 prc2 = lambda do |a, b, c|
2   a+b+c
3 end
4 p prc2.call(2,3,4)

bagon:ruby wangmjc$ ruby proc.rb
9
bagon:ruby wangmjc$ ruby proc.rb
9
bagon:ruby wangmjc$
```

如下：

```
proc.rb
1 def sum(n)
2   lambda do |a|
3     a+n
4   end
5 end
6 p1 = sum(2)
7 p p1.call(1)

bagon:ruby wangmjc$ ruby proc.rb
3
bagon:ruby wangmjc$
```

```
proc.rb
1 def sum(n)
2   Proc.new do |a|
3     a+n
4   end
5 end
6 p1 = sum(2)
7 p p1.call(1)
```

```
1. bash
ruby
bogon:ruby wangmjc$ ruby proc.rb
3
bogon:ruby wangmjc$
```

```
proc.rb
1 def sum(n)
2   proc do |a|
3     a+n
4   end
5 end
6 p1 = sum(2)
7 p p1.call(1)
```

```
ruby
bogon:ruby wangmjc$ ruby proc.rb
3
bogon:ruby wangmjc$
```

真正的区别是用return的时候

```
proc.rb
1 def sum(n)
2   proc do |a|
3     return a+n
4   end
5 end
6 p1 = sum(2)
7 p p1.call(1)
```

```
1. bash
ruby
bogon:ruby wangmjc$ ruby proc.rb
proc.rb:3:in `block in sum': unexpected return (LocalJumpError)
    from proc.rb:7:in `call'
    from proc.rb:7:in `<main>'
bogon:ruby wangmjc$
```

```
proc.rb
1 def sum(n)
2   lambda do |a|
3     return a+n
4   end
5 end
6 p1 = sum(2)
7 p p1.call(1)
```

```
1. b
ruby
bogon:ruby wangmjc$ ruby proc.rb
3
bogon:ruby wangmjc$
```

这是因为使用proc的方法的时候在块中使用return后，程序就会跳出当前的执行块，从而直接从这个块的方法返回了，普通的块也可以从中返回的。

注意：

用proc方法创建proc对象的情况下，由于这些方法都接受块，在调用proc#call方法的时候并没有适当的返回对象，因此会发生错误，而lambda的情况下则与return一样，将值返回给Proc#call对象。

```

[43] pry(main)> def sum(n)
[43] pry(main)*   proc do |a|
[43] pry(main)*     return a+n
[43] pry(main)*   end
[43] pry(main)* end
=> :sum
[44] pry(main)> p1 = sum(2)
=> #<Proc:0x007fa760832d90@pry>:42>
[45] pry(main)> p p1.call(1)
LocalJumpError: unexpected return
from (pry):43:in `block in sum'
[46] pry(main)> def sum(n)
[46] pry(main)*   lambda do |a|
[46] pry(main)*     return a+n
[46] pry(main)*   end
[46] pry(main)* end
=> :sum
[47] pry(main)> p1 = sum(2)
=> #<Proc:0x007fa76114d560@pry>:49 (lambda)>
[48] pry(main)> p p1.call(1)
3
=> 3

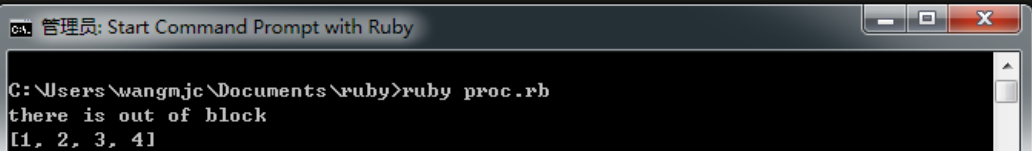
```

在block中用return会作为整个方法的返回值，不会执行后面的内容了，例如下面两个程序的对比

```

proc.rb
1 def prefix(ary, object)
2   result = []
3   ary.each do |item|
4     result << item
5     result if object == item #当没有return的时候，最后一行会作为返回值，然后继续执行后面的程序
6   end
7   puts "there is out of block"
8   p result
9 end
10
11 prefix([1,2,3,4], 3)

```



```

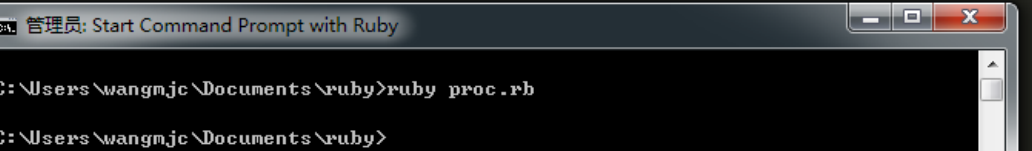
C:\Users\wangmjc\Documents\ruby>ruby proc.rb
there is out of block
[1, 2, 3, 4]

```

```

proc.rb
1 def prefix(ary, object)
2   result = []
3   ary.each do |item|
4     result << item
5     if object == item
6       return result #用了return后，会跳过块，然后返回值会作为整个方法的返回值，不会再执行后面的内容
7     end
8   end
9   puts "there is out of block"
10  p result
11 end
12
13 prefix([1,2,3,4], 3)

```



```

C:\Users\wangmjc\Documents\ruby>ruby proc.rb
C:\Users\wangmjc\Documents\ruby>

```

下面是lambda和proc方式的return的比较，lambda可以return出来，作为方法的返回值，而return在proc中，虽然块内的return应该从sum方法返回，但是程序由于运行时sum的上下文会消失，所以程序就会报错，还有一个原因是用proc方法建立的proc对象，用Proc#call方法是没有适当的返回对象的，但是用lambda方法家里proc对象的时候，将返回值给Proc#call方法。

```

proc.rb
1  def sum(n)
2    lambda do |i|
3      return n + i
4    end
5  end
6  s = sum(5)
7  p s.call(2)

```

管理员: Start Command Prompt with Ruby

```

C:\Users\wangmjc\Documents\ruby>ruby proc.rb
7
C:\Users\wangmjc\Documents\ruby>

```

```

proc.rb
1  def sum(n)
2    proc do |i|
3      return n + i
4    end
5  end
6  s = sum(5)
7  p s.call(2)

```

管理员: Start Command Prompt with Ruby

```

C:\Users\wangmjc\Documents\ruby>ruby proc.rb
proc.rb:3:in 'block in sum': unexpected return <LocalJumpError>
      from proc.rb:7:in 'call'
      from proc.rb:7:in '<main>'
C:\Users\wangmjc\Documents\ruby>

```

break用于控制迭代器，这个命令会想接受块的方法的调用者返回结果值，例如：

```

irb(main):002:0* [1,2,3].collect do |item|
irb(main):003:1*   break [1,2]
irb(main):004:1> end
=> [1, 2]
irb(main):005:0>

```

next方法的作用在于中断一次块的执行，例如：

```

proc.rb
1  def prefix(ary, object)
2    result = []
3    ary.each do |item|
4      if object == item
5        next
6      end
7      result << item
8    end
9
10   p result
11 end
12
13 prefix([1,2,3,4], 3)

```

管理员: Start Command Prompt with Ruby

```

C:\Users\wangmjc\Documents\ruby>ruby proc.rb
[1, 2, 4]
C:\Users\wangmjc\Documents\ruby>

```

lambda的另外一种写法

```

irb(main):005:0> a = ->{<i><p i*2>
=> #<Proc:0x000000002f898d0e@irb>:5 <lambda>>
irb(main):006:0> a.call(4)
8
=> 8

```

21.1.2 通过proc参数接受块

在调用带块的方法的时候，通过proc参数的形式制定块后，该块就会作为proc对象被方法接受。

```
proc.rb
1 def total2(from, to, &block)
2   result = 0
3   from.upto(to).each do |num|
4     if block
5       result += block.call(num)
6     else
7       result += num
8     end
9   end
10  return result
11 end
12
13 p total2(2,5)
14 p total2(2,5){|i| i*2}
```

```
C:\Users\wangmjc\Documents\ruby>ruby proc.rb
14
28
```

21.1.3 to_proc方法

有些对象有to_proc方法，在方法中指定块时，如果以&对象的形式传递参数，对象to_proc就会被自动调用，进而生成proc对象（就是包装块的对象）

其中 Symbol#to_proc方法比较典型，且经常被用到，例如to_i 使用Symbol#to_proc方法，就是像这样生成proc对象

Proc.new{|i| i.to_i}

这个对象什么时候使用呢，例如把数组所有元素都转换为整数类型，一般做法如下：

```
irb(main):013:0> %w<2 4 6 8>.map {|i| i.to_i}
=> [2, 4, 6, 8]
irb(main):014:0> p = Proc.new{|i| i.to_s}
=> #<Proc:0x00000002ffb3e00@irb>:14>
irb(main):015:0> %w<2 4 6 8>.map(&p)
=> ["2", "4", "6", "8"]
irb(main):016:0> %w<2 4 6 8>.map(&:to_i)
=> [2, 4, 6, 8]
irb(main):018:0> [Integer, String, Array, Hash, File, IO].sort_by(&:name)
=> [Array, File, Hash, IO, Integer, String]
```

21.2 Proc的特征

虽然 Proc对象可以作为匿名函数或方法使用，但是它并不是单纯的对象化，例如

```
proc.rb
1 def counter
2   c = 0
3   Proc.new do
4     c += 1
5   end
6 end
7
8 c1 = counter
9 print "c1 = #{c1.call}\n "
10 print "c1 = #{c1.call}\n "
11 print "c1 = #{c1.call}\n "
12
13 c2 = counter
14 print "c2 = #{c2.call}\n "
15 print "c2 = #{c2.call}\n "
16
17 print "c1 = #{c1.call}\n "
```

```
C:\Users\wangmjc\Documents\ruby>ruby proc.rb
c1 = 1
c1 = 2
c1 = 3
c2 = 1
c2 = 2
c1 = 4

C:\Users\wangmjc\Documents\ruby>
```

通过例子可以看出，变量c1和c2引用的proc对象，是分别保存，处理调用counter方法时初始化的本地变量，与此同时，proc对象也会将处理内容，本地变量的作用域等定义块时的状态-----这种像proc对象这样讲处理内容，变量等环境同时进行保存的对象，称为闭包closure。

就像刚才的计数器的例子那样，proc对象（必须是proc对象才行，下面的例子就不行）可被用来对少量代码实现的功能做对象化处理，每个对象会保存一份闭包，另外由于Ruby中大量使用了块，因此在有一定规模的程序开发中，特别是像调用和传递带块的方法时的方法

```
proc.rb
1  def counter(&block)
2      c = 0
3      block.call(c)
4  end
5  b = proc do |x|
6      x += 1
7  end
8  p c1 = counter(&b)
9  p c1
10 p c1
```

```
C:\> 管理员: Start Command Prompt with Ruby

C:\Users\wangmjc\Documents\ruby>ruby proc.rb
1
1
1

C:\Users\wangmjc\Documents\ruby>
```

21.3 proc类的实例方法

1.prc.call(args...)

2.prc[args ...]

3.prc.yield(args ...)

4.prc === arg

上述几个方法都执行proc对象prc

```
proc.rb
1  prc = proc{|a,b| a+b}
2  p prc.call(5,6)
3  p prc[5,6]
4  p prc.yield(5,6)
5  p prc.(5,6)
6  p prc === [5,6]
```

```
C:\> 管理员: Start Command Prompt with Ruby

C:\Users\wangmjc\Documents\ruby>ruby proc.rb
11
11
11
11
11

C:\Users\wangmjc\Documents\ruby>
```

由于受到语法的限制，用过===指定的参数只能为1个，

注意：proc对象方法会作为case语句的条件使用，因此在创建这样的proc对象时，比较恰当的是，只接受一个参数，并返回true或者false

用proc对象设置复杂的case语句的条件使用，根据proc对象的返回值 true或者false使用

```
proc.rb
1  fizz = proc{|n| n % 3 == 0}
2  buzz = proc{|n| n % 5 == 0}
3  fb = proc{|n| n % 3 == 0 && n % 5 == 0}
4
5  1.upto(100).each do |i|
6    case i
7      when fb then puts "Fizz Buzz"
8      when fizz then puts "Fizz"
9      when buzz then puts "Buzz"
10     else puts i
11   end
12 end
```

管理员: Start Command Prompt with Ruby

```
C:\Users\wangmjc\Documents\ruby>ruby proc.rb
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
Fizz Buzz
16
```

prc.arity

返回proc对象的call方法的参数的块变量个数，如果这个proc对象的参数是通过|*args|定义的，则返回-1

```
proc.rb
1  prc0 = proc{nil}
2  prc1 = proc{|a| a}
3  prc2 = Proc.new{|a,b| a + b}
4  prc3 = lambda{|a,b,c| a + b + c}
5  prcn = Proc.new{|*args| args}
6
7  p prc0.arity
8  p prc1.arity
9  p prc2.arity
10 p prc3.arity
11 p prcn.arity
12
```

管理员: Start Command Prompt with Ruby

```
C:\Users\wangmjc\Documents\ruby>ruby proc.rb
0
1
2
3
-1
C:\Users\wangmjc\Documents\ruby>
```

prc.parameters 返回关于块变量的详细信息，返回值为【种类，变量名】

种类的有几个返回值，注意Proc.new的为可选的，lambda的为必须的

1. opt 可省略的变量
2. :req 必须的变量
3. :rest 以*args形式表示的变量
4. :key 关键字参数形式的变量
5. :keyrest 以**args形式表示的变量 **args的形式是为了解决调用方法时，制定了未定义的关键字参数时而报错，所以就可以通过**args的形式来接受未定义的参数
6. :block 块


```
proc.rb
1 prc0 = proc{nil}
2 prc1 = proc{|a| a}
3 prc2 = Proc.new{|a,b| [a, b]}
4 prc3 = lambda{|a,b=1,c| [a,b, c]}
5 prc4 = lambda{|a,&block| [a, block]}
6 prc5 = lambda{|a: 1, b: 2| [a, b]}
7 prc6 = lambda{|a: 1, b: 2, **args| [a, b, args]}
8 prcn = Proc.new{|*args| args}
9
10 puts "prc0's parameters = #{prc0.parameters}"
11 puts "prc1's parameters = #{prc1.parameters}"
12 puts "prc2's parameters = #{prc2.parameters}"
13 puts "prc3's parameters = #{prc3.parameters}"
14 puts "prc4's parameters = #{prc4.parameters}"
15 puts "prc5's parameters = #{prc5.parameters}"
16 puts "prc6's parameters = #{prc6.parameters}"

C:\Users\wangmjc\Documents\ruby>ruby proc.rb
prc0's parameters = []
prc1's parameters = [[:opt, :a]]
prc2's parameters = [[:opt, :a], [:opt, :b]]
prc3's parameters = [[:req, :a], [:opt, :b], [:req, :c]]
prc4's parameters = [[:req, :a], [:block, :block]]
prc5's parameters = [[:key, :a], [:key, :b]]
prc6's parameters = [[:key, :a], [:key, :b], [:keyrest, :args]]
```

prc.lambda?可以判断一个块是否为lambda类型的

```
irb(main):049:0> a = lambda {|a| a}
=> #<Proc:0x00000003011af00c@irb>:49 <lambda>
irb(main):050:0> a.lambda?
=> true
```

prc.source_location

返回定义的prc的程序代码的位置, 返回值为【代码文件名, 行编号】形式的数组, prc由扩展库等生成, 当ruby脚本不存在时返回nil
在文件中返回文件名, 如果在irb中返回 代码文件名返回 irb

```
proc.rb
1
2 prc2 = Proc.new{|a,b| [a, b]}
3 prc6 = lambda{|a: 1, b: 2, **args| [a, b, args]}
4
5 p prc2.source_location
6 p prc6.source_location

C:\Users\wangmjc\Documents\ruby>ruby proc.rb
["proc.rb", 2]
["proc.rb", 3]

C:\Users\wangmjc\Documents\ruby>
irb(main):064:0* prc2 = Proc.new{|a,b| [a, b]}
=> #<Proc:0x00000002cc14900c@irb>:64
irb(main):065:0> prc6 = lambda{|a: 1, b: 2, **args| [a, b, args]}
=> #<Proc:0x000000033c2e100c@irb>:65 <lambda>
irb(main):066:0> prc2.source_location
=> ["<irb>", 64]
irb(main):067:0> prc6.source_location
=> ["<irb>", 65]
```

