

11章 块

作者 bluetea

网站 <https://github.com/bluetea>

当处理hash的迭代的时候, 可以用两种方式

```
1 h1 = {:x => "nihao", :y => "hello world"}
2 h1.each do |i|
3   puts "#{i[0]} = #{i[1]}"
4 end
5
6 h1.each do |x,y|
7   puts "#{x} = #{y}"
8 end
9
```

```
wangmjcdMBP:ruby wangmjcd$ ruby iterator.rb
x = nihao
y = hello world
x = nihao
y = hello world
```

11.2.2 隐藏常规处理

块除了用在迭代器以外, 还被用在其它地方, 例如另一个用法就是确保后处理被执行, 例如File.open方法, File.open 方法在接收块后, 会将File对象作为块变量, 并执行一次块, 例如:

```
1 File.open("line.txt") do |file|
2   file.each_line do |line|
3     puts line
4   end
5 end
```

```
wangmjcdMBP:ruby wangmjcd$ ruby iterator.rb
line 1 line
line 2 line line
line 3 line line line
wangmjcdMBP:ruby wangmjcd$
```

在File.open方法使用块时, 块内部的处理完毕并跳出块前, 文件会被自动关闭, 因此不需要使用File#close方法了

如果不用迭代的写法, 就会这样, 使用ensure来确保.close关闭

```
6
7 begin
8   f = File.open("line.txt")
9   f.each_line do |line|
10    puts line
11  end
12 ensure
13   f.close
14   puts "ensure is runed"
15 end
```

11.2.3 替换部分算法

自定义排列顺序

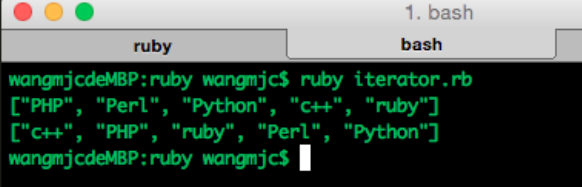
Array#sort方法没有指定块时, 会使用<=>运算符对各个元素进行比较, 并根据比较后的结果进行排序, <=>运算的返回值为 -1, 0, 1中的一个, 比较字符串时, 会按照字符编码的顺序进行比较, 比较字母是会按照先大写字母, 后小写字母的顺序排列。

```
[21] pry(main)> "cc" <=> "bb"
=> 1
[22] pry(main)> "cc" <=> "dd"
=> -1
[23] pry(main)> "cc" <=> "c"
=> 1
[24] pry(main)> "cc" <=> "cc"
=> 0
```

```
[25] pry(main)> ary = ["ruby", "Perl", "c++", "PHP", "Python"]
=> ["ruby", "Perl", "c++", "PHP", "Python"]
[26] pry(main)> ary.sort
=> ["PHP", "Perl", "Python", "c++", "ruby"]
[27] pry(main)> ary
=> ["ruby", "Perl", "c++", "PHP", "Python"]
[28] pry(main)> ary.sort!
=> ["PHP", "Perl", "Python", "c++", "ruby"]
[29] pry(main)> ary
=> ["PHP", "Perl", "Python", "c++", "ruby"]
```

我们也可以通过调用块来指定排列顺序，

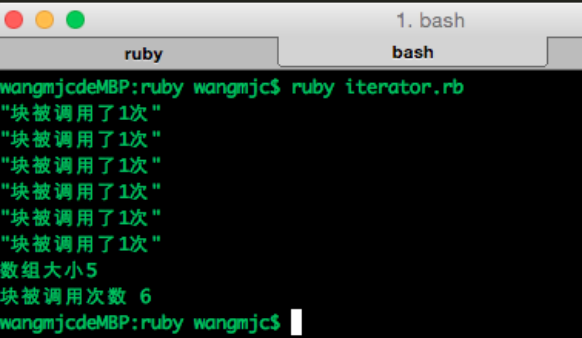
```
1 ary = ["ruby", "Perl", "c++", "PHP", "Python"]
2 print ary.sort{|a, b| a <=> b }, "\n"
3 print ary.sort{|a, b| a.length <=> b.length}, "\n"
```



```
wangmjcdMBP:ruby wangmjc$ ruby iterator.rb
["PHP", "Perl", "Python", "c++", "ruby"]
["c++", "PHP", "ruby", "Perl", "Python"]
wangmjcdMBP:ruby wangmjc$
```

注意：块中组后一个表达式的值就是块的执行结果，因此 `<=>` 运算符必须在最后一行使用

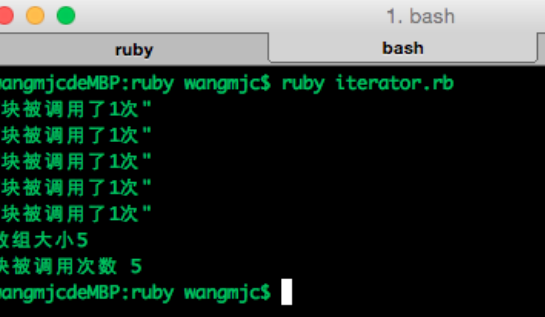
```
1 ary = ["ruby", "Perl", "c++", "PHP", "Python"]
2 call_times = 0
3 ary.sort do |a, b|
4   p "块被调用了1次"
5   call_times += 1
6   a.length <=> b.length
7 end
8 puts "数组大小#{ary.size}"
9 puts "块被调用次数 #{call_times}"
```



```
wangmjcdMBP:ruby wangmjc$ ruby iterator.rb
"块被调用了1次"
"块被调用了1次"
"块被调用了1次"
"块被调用了1次"
"块被调用了1次"
"块被调用了1次"
数组大小5
块被调用次数 6
wangmjcdMBP:ruby wangmjc$
```

元素数5个，块被调用了6次

```
1 ary = ["ruby", "Perl", "c++", "PHP", "Python"]
2 call_times = 0
3 ary.sort_by do |item|
4   p "块被调用了1次"
5   call_times += 1
6   item.length
7 end
8 puts "数组大小#{ary.size}"
9 puts "块被调用次数 #{call_times}"
```



```
wangmjcdMBP:ruby wangmjc$ ruby iterator.rb
"块被调用了1次"
"块被调用了1次"
"块被调用了1次"
"块被调用了1次"
"块被调用了1次"
数组大小5
块被调用次数 5
wangmjcdMBP:ruby wangmjc$
```

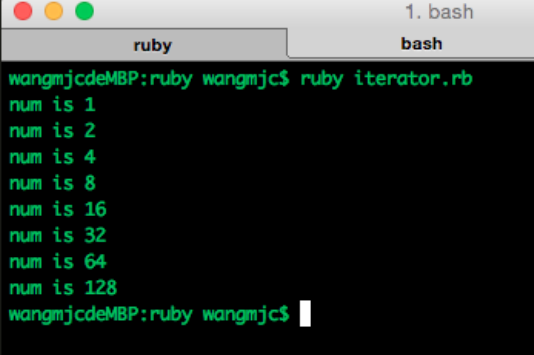
元素5个，块被调用5次，，所以sort_by函数效率更高

所以总结一下，元素排序算法中公共部分由方法本身提供，我们则可以用块来替换方法中排列的顺序（或者取得用于比较的顺序），或者根据不同的目的来替换需要更改的部分。

11.3定义带块的方法

11.3.1 执行块，缺点，需要在块中指定break

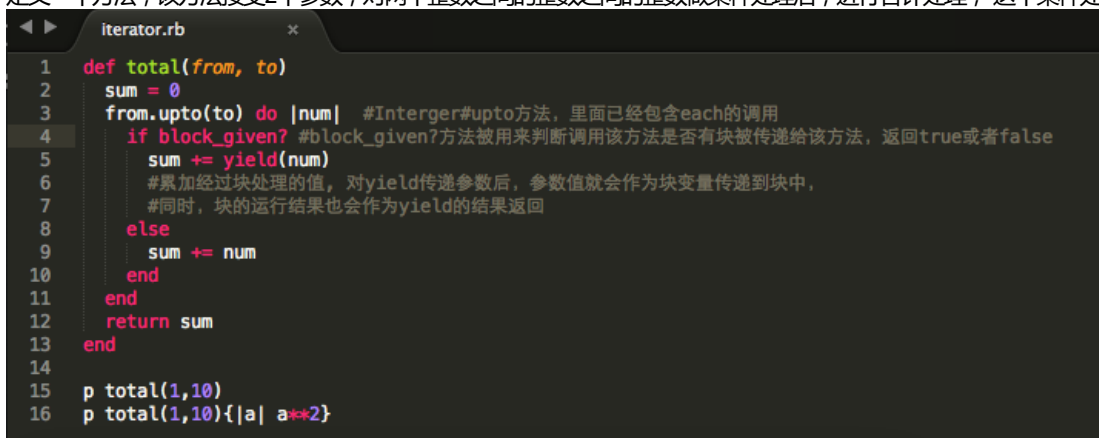
```
1 def myloop
2   while true
3     yield
4   end
5 end
6
7 num =1
8 myloop do
9   puts "num is #{num}"
10  break if num >100
11  num *= 2
12 end
```



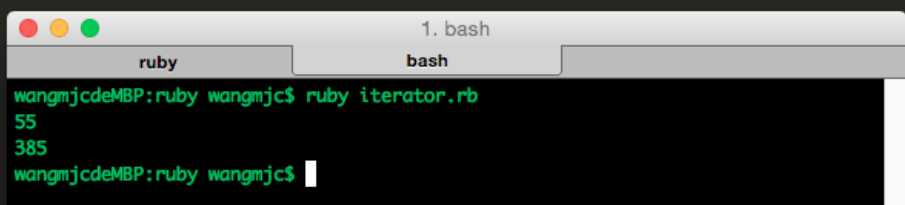
myloop方法在执行while循环的时候执行了yield关键字，yield关键字的作用就是执行块的方法，因为这个while循环条件固定为true，所以会无限循环下去，但只要在块中调用break，就会中断mylooo的循环

11.3.2传递块参数，获取块的值

定义一个方法，该方法接受2个参数，对两个整数之间的整数之间的整数做某种处理后，进行合计处理，这个某种处理是由块指定的

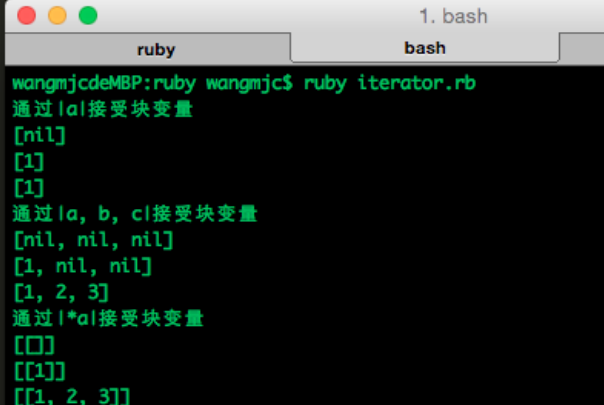


```
1 def total(from, to)
2   sum = 0
3   from.upto(to) do |num| #Integer#upto方法，里面已经包含each的调用
4     if block_given? #block_given?方法被用来判断调用该方法是否有块被传递给该方法，返回true或者false
5       sum += yield(num)
6       #累加经过块处理的值，对yield传递参数后，参数值就会作为块变量传递到块中，
7       #同时，块的运行结果也会作为yield的结果返回
8     else
9       sum += num
10    end
11  end
12  return sum
13 end
14
15 p total(1,10)
16 p total(1,10){|a| a**2}
```



测试yeild传递0个，1个，3个等多个参数时，对应的块变量是如何接收的

```
1 def block_args_test
2   yield() #会把nil传递给后面的块
3   yield(1) #会把1传递给后边的块
4   yield(1,2,3) #会把1, 2, 3分别传递给后面的块
5 end
6
7 puts "通过|a|接受块变量"
8 block_args_test do |a|
9   p [a]
10 end
11
12 puts "通过|a, b, c|接受块变量"
13 block_args_test do |a, b, c|
14   p [a, b, c]
15 end
16
17 puts "通过|*a|接受块变量"
18 block_args_test do |*a|
19   p [a]
20 end
21
```



通过上面可以看出，块的变量较多的时候，多出来的为nil，当块的变量不足的时候，则不能接收所有变量，

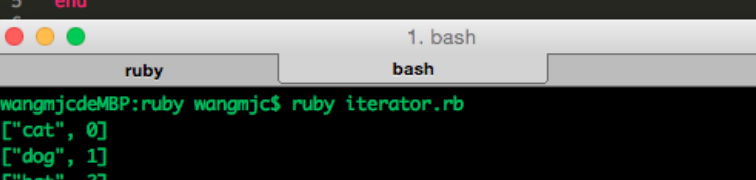
抽取嵌套数组的元素规则，同样也适用于块变量，例如Hash#each_with_index方法的块变量有2个，并yield ([键, 值], 索引) 的形式传递

```
1 hash = {a: 100, b: 200, c:300}
2 array = ["cat", "dog", "hat"]
3 hash.each_with_index do |key,value,index| #把值只传递给后面的2个变量，第三个都会赋值为nil，key包含2个内容
4   p [key, value, index]
5 end
6 hash.each_with_index do |(key,value),index| #因为前两个用 () 包含，所以会分别接收key和value
7   p [key, value, index]
8 end
```



针对array的变量，只有2个参数可传递

```
1 hash = {a: 100, b: 200, c:300}
2 array = ["cat", "dog", "hat"]
3 array.each_with_index do |value, index| #把值只传递给后面的2个变量
4   p [value, index]
5 end
```



11.3.3 如何控制块的执行

```
1 def total(from, to)
2   sum = 0
3   from.upto(to) do |num| #Integer#upto方法, 里面已经包含each的调用
4     if block_given? #block_given?方法被用来判断调用该方法是否有块被传递给该方法, 返回true或者false
5       sum += yield(num)
6       #累加经过块处理的值, 对yield传递参数后, 参数值就会作为块变量传递到块中,
7       #同时, 块的运行结果也会作为yield的结果返回
8     else
9       sum += num
10    end
11  end
12  return sum
13 end
14
15 n = total(1, 10) do |num|
16   if num == 5
17     break #在块中使用break的时候, 程序会马上返回调用的地方, 所有块内的返回值会被忽略
18   end
19   num
20 end
21
22 p n
```

1. bash

ruby bash

wangmjcdeMBP:ruby wangmjc\$ ruby iterator.rb
nil
wangmjcdeMBP:ruby wangmjc\$

当你希望返回某个值的时候, 就可以在break后跟元素(数字, 字符都可以), 就会把这个元素返回

```
1 def total(from, to)
2   sum = 0
3   from.upto(to) do |num| #Integer#upto方法, 里面已经包含each的调用
4     if block_given? #block_given?方法被用来判断调用该方法是否有块被传递给该方法, 返回true或者false
5       sum += yield(num)
6       #累加经过块处理的值, 对yield传递参数后, 参数值就会作为块变量传递到块中,
7       #同时, 块的运行结果也会作为yield的结果返回
8     else
9       sum += num
10    end
11  end
12  return sum
13 end
14
15 n = total(1, 10) do |num|
16   if num == 5
17     break "hello" #在块中使用break的时候, 程序会马上返回调用的地方, 所有块内的返回值会被忽略
18   end
19   num
20 end
21
22 p n
```

1. bash

ruby bash

wangmjcdeMBP:ruby wangmjc\$ ruby iterator.rb
"hello"

此外, 如果在块中使用next, 程序就会中断当前处理, 报错TypeError, 是因为返回的是nil, 无法进行处理

```

1  def total(from, to)
2    sum = 0
3    from.upto(to) do |num| #Integer#upto方法, 里面已经包含each的调用
4      if block_given? #block_given?方法被用来判断调用该方法是否有块被传递给该方法, 返回true或者false
5        sum += yield(num)
6        #累加经过块处理的值, 对yield传递参数后, 参数值就会作为块变量传递到块中,
7        #同时, 块的运行结果也会作为yield的结果返回
8      else
9        sum += num
10     end
11   end
12   return sum
13 end
14
15 n = total(1, 10) do |num|
16   if num == 4
17     next #如果为next后面没有指明任何参数, 就会返回nil,
18         #在这个total中无法进行算术操作, 所以会报错, 如果可以处理, 程序继续下轮进行
19   end
20   num
21 end
22
23 p n

```

```

1. bash
ruby      bash
wangmjcdMBP:ruby wangmjc$ ruby iterator.rb
iterator.rb:5:in `+': nil can't be coerced into Fixnum (TypeError)
    from iterator.rb:5:in `block in total'
    from iterator.rb:3:in `upto'
    from iterator.rb:3:in `total'
    from iterator.rb:15:in `'
wangmjcdMBP:ruby wangmjc$

```

如果next后面跟上常量, 就会把这个常量当做本轮循环的返回值, 并继续下面的处理例如:

```

1  def total(from, to)
2    sum = 0
3    from.upto(to) do |num| #Integer#upto方法, 里面已经包含each的调用
4      if block_given? #block_given?方法被用来判断调用该方法是否有块被传递给该方法, 返回true或者false
5        sum += yield(num)
6        #累加经过块处理的值, 对yield传递参数后, 参数值就会作为块变量传递到块中,
7        #同时, 块的运行结果也会作为yield的结果返回
8      else
9        sum += num
10     end
11   end
12   return sum
13 end
14
15 n = total(1, 10) do |num|
16   if num == 4
17     next 1000 #next后面参数作为本轮循环的返回值, 返回给yield
18   end
19   num
20 end
21
22 p n

```

```

1. bash
ruby      bash
wangmjcdMBP:ruby wangmjc$ ruby iterator.rb
1051

```

11.3.4将块封装为对象

在定义接受块的方法时, 可以使用yield 关键字

在Ruby中, 还可以把块当做对象处理, 就可以在接收块的方法之外的其他地方执行块, 或者把块交给其他方法执行。

这种情况下需要用到proc对象, proc对象是能让块作为对象在程序中使用的类, proc对象调用call方法, 块中定义的程序会被执行

```

1  hello = Proc.new do |name|
2    puts "hello, #{name}"
3  end
4
5  hello.call("wang")

```

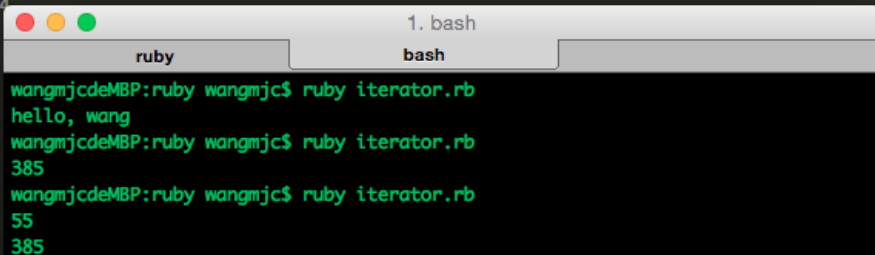
```

1. bash
ruby      bash
wangmjcdMBP:ruby wangmjc$ ruby iterator.rb
hello, wang
wangmjcdMBP:ruby wangmjc$

```

把块从一个方法传给另外一个方法时，首先会通过变量将块作为Proc对象接受，然后再传递给另外一个方法，在方法定义时，如果末尾参数使用“&参数名”的形式，ruby会自动把调用时跟的块，封装为proc对象，在方法内这个proc对象调用call方法时，就会把变量传递给后面的块。

```
1 def total(from, to, &block)
2   sum = 0
3   from.upto(to) do |num|
4     if block #判断block是否跟随
5       sum += block.call(num) #block是proc对象，proc对象的call方法会把num传递给后面的block的
6     else
7       sum += num
8     end
9   end
10  return sum
11 end
12 p total(1, 10)
13 p total(1, 10){|num| num**2}
```

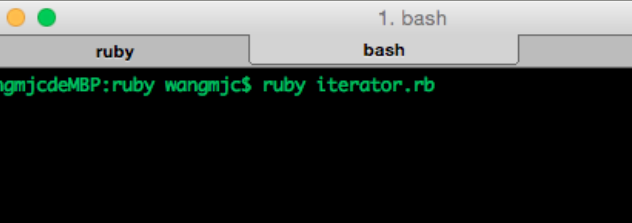


```
wangmjcdMBP:ruby wangmjcd$ ruby iterator.rb
hello, wang
wangmjcdMBP:ruby wangmjcd$ ruby iterator.rb
385
wangmjcdMBP:ruby wangmjcd$ ruby iterator.rb
55
385
```

方法中定义的&block参数称为proc参数，如果在方法调用的时候没有传递块，那么这个proc参数为nil，因此可以通过判断这个值判断是否有block被传入，

我们也能将proc对象在方法内部调用，也可以将proc对象传递给他去方法处理，这是，只需要在调用方法时，用“&proc对象”的形式定义参数就可以了，例如，重新封装Array#each参数

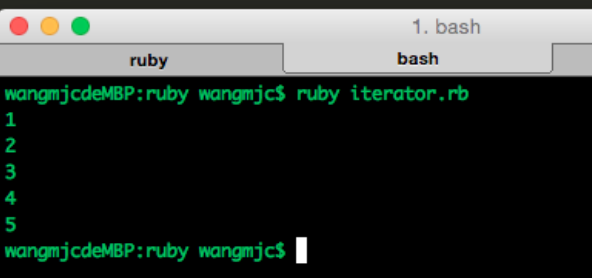
```
1 def call_each(ary, &block) #要求传递一个块，这块是proc对象
2   ary.each(&block) #把这个proc参数不展开，传递给each方法
3 end
4
5 array = [1,2,3,4,5]
6 call_each(array) {|item| puts item}
```



```
wangmjcdMBP:ruby wangmjcd$ ruby iterator.rb
1
2
3
4
5
```

如果是自己把proc对象调用，就要这么写

```
1 def call_each(ary, &block)
2   ary.each do |i|
3     block.call(i)
4   end
5 end
6
7 array = [1,2,3,4,5]
8 call_each(array) {|item| puts item}
```

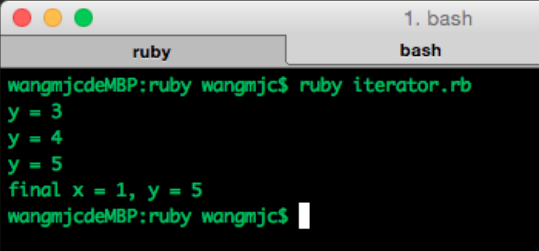


```
wangmjcdMBP:ruby wangmjcd$ ruby iterator.rb
1
2
3
4
5
wangmjcdMBP:ruby wangmjcd$
```

11.4局部变量与块变量

块内部命名空间与块外部是共享的，在块外部定义的局部变量，在块内部也可以继续使用，而块使用的那个传递用的变量（|x|），即使与外面的同名，ruby也会认为他们两个是不同的变量，例如

```
1 x = 1
2 y = 1
3 ary = [3,4,5]
4 ary.each do |x|
5   y = x
6   puts "y = #{y}"
7 end
8 puts "final x = #{x}, y = #{y}"
```

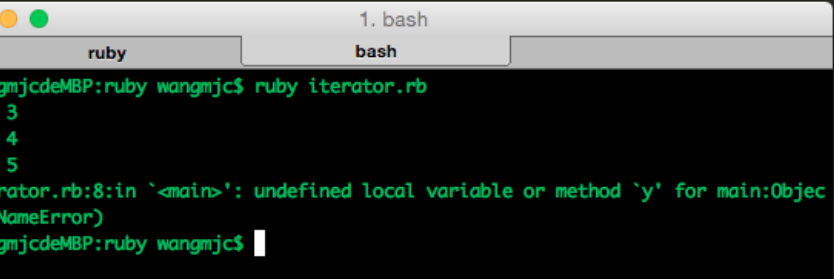


```
wangmjcdMBP:ruby wangmjc$ ruby iterator.rb
y = 3
y = 4
y = 5
final x = 1, y = 5
wangmjcdMBP:ruby wangmjc$
```

块内使用的变量x 和，外面的x同名，但是ruby会认为他们不同，不影响最后的值

如果只是在块内部第一次使用的局部变量，在外部是不能访问的，例如：

```
1 x = 1
2 #y = 1
3 ary = [3,4,5]
4 ary.each do |x|
5   y = x
6   puts "y = #{y}"
7 end
8 puts "final x = #{x}, y = #{y}"
```

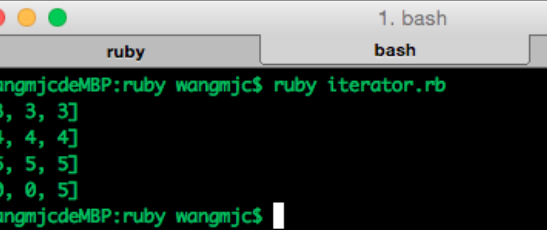


```
wangmjcdMBP:ruby wangmjc$ ruby iterator.rb
y = 3
y = 4
y = 5
iterator.rb:8:in `<main>': undefined local variable or method `y' for main:Object (NameError)
wangmjcdMBP:ruby wangmjc$
```

块中变量的作用域之所以这么设计，是为了通过与块外部共享局部变量，从而扩展变量的有效范围，，所以在块内部给局部 变量赋值的时候，要时刻注意与块外部同名变量的关系，一定要注意这个ruby陷阱。

块变量是只能在块内部使用，

```
1 x = y = z = 0
2 ary = [3,4,5]
3 ary.each do |x; y| #使用块变量x， 块局部变量y
4   y = x
5   z = x
6   p [x, y, z]
7 end
8 p [x, y, z]
```



```
wangmjcdMBP:ruby wangmjc$ ruby iterator.rb
[3, 3, 3]
[4, 4, 4]
[5, 5, 5]
[0, 0, 5]
wangmjcdMBP:ruby wangmjc$
```