第 7 章

代码块与迭代

在 Ruby 中，代码块（或简称为“块”）指包括某些 Ruby 代码以及执行代码所必需的上下文的对象。代码块是 Ruby 在视觉上最与众不同的方面，也是令从其他语言转来的新用户最感困惑的。从本质上讲，一个 Ruby 代码块就是一个没有名字的方法。

大多数其他语言都具有类似Ruby代码块的东西：C的函数指针、C++的函数对象、Python 的 lambda 和链表推导式（list comprehension）、Perl 的匿名函数、Java 的匿名内部类。这些特性主要存在于这些语言的偏远角落里，避免新手程序员使用。没有代码块就无法编写 Ruby。在主流语言中，只有 Lisp 更像是面向代码块的。

与大多数其他语言不同，Ruby 令代码块易于创建，并且很少在其上强加限制。在本书的所有其他章节中，读者都将看到传递给方法的代码块似乎没什么了不起（但实际确实是个大问题）：

```
[1,2,3].each { |i| puts i}
# 1
# 2
# 3
```

在本章中，我们将介绍如果编写该类方法、有助于以上述方式编写的方法类型，以及何时、如何将代码块视为第一类对象。

Ruby为创建代码块提供了两种语法。当整个代码块在一行内能放下时，用花括号括起代码块是最易读的：

```
[1,2,3].each { |i| puts i }
# 1
# 2
# 3
```

当代码块比一行长时，代码块以关键字 do 开始并以关键字 end 结尾会更易读：

```
[1,2,3].each do |i|
  if i % 2 == 0
    puts "#{i} is even."
  else
    puts "#{i} is odd."
  end
end
# 1 is odd.
# 2 is even.
# 3 is odd.
```

有些人对代码块的返回值感兴趣，他们会使用括号语法，如果他们对代码块的副作用感兴趣，那么他们会使用 do...end 语法。

要记住，括号语法比 do...end 语法具有更高的优先级。考虑下列两段代码：

```
1.upto 3 do |x|
  puts x
end
# 1
# 2
# 3

1.upto 3 { |x| puts x }
# SyntaxError: compile error
```

在第二个示例中，代码块藏在数字 3 后面，而不是函数调用 1.upto 3 后面。单独的变量不能获取代码块，因此会得到一个编译错误。当拿不定主意的时候，可以使用圆括号。

```
1.upto(3) { |x| puts x }
# 1
# 2
# 3
```

传递给方法的代码块通常是就地创建的匿名对象。但是，通过调用 lambda 可以将代码块实例化为 Proc 对象。更多细节请参见 7.1 节。

```
hello = lambda { "Hello" }
hello.call
# => "Hello"

log = lambda { |str| puts "[LOG] #{str}" }
log.call("A test log message.")
# [LOG] A test log message.
```

与任何方法一样，代码块可以接收参数。代码块的参数在代码块开始的一个以逗号分隔的列表中定义，以通道符（“ | ”）括起：

```
{1=>2, 2=>4}.each { |k,v| puts "Key #{k}, value #{v}" }
# Key 1, value 2
# Key 2, value 4
```

传递给代码块的参数几乎类似于传递给方法的参数，只是存在一些限制：不能为代码块参数设置默认值，不能扩展散列或内置数组，以及代码块本身不能接收代码块参数（注1）。

由于Proc对象的创建与其他对象类似，因此可以创建其返回值被自定义为可执行Ruby代码段的工厂方法。下面是一个针对执行乘法的代码块的简单工厂方法：

```
def times_n(n)
  lambda { |x| x * n }
end
```

下列代码用工厂来创建和使用两种自定义的方法：

```
times_ten = times_n(10)
times_ten.call(5)                                    # => 50
times_ten.call(1.25)                                 # => 12.5

circumference = times_n(2*Math::PI)
circumference.call(10)        # => 62.8318530717959
circumference.call(3)                                # => 18.8495559215388
[1, 2, 3].collect(&circumference)
# => [6.28318530717959, 12.5663706143592, 18.8495559215388]
```

读者可能已经听人们谈起过有关Ruby的“闭包”。什么是闭包，它与代码块有什么区别？在Ruby中，闭包与代码块之间不存在差别。每个Ruby代码块也是一个闭包（注2）。

那么，什么使Ruby代码块成为闭包呢？Ruby代码块基本上会在定义它的上下文周围传递。代码块可以引用处于定义它的范围内的变量，即使那些变量后来超过了范围。下面是一个简单示例，更多示例请参见7.4节。

```
ceiling = 50
# Which of these numbers are less than the target?
[1, 10, 49, 50.1, 200].select { |x| x < ceiling }
# => [1, 10, 49]
```

注1： 在Ruby 1.9中，一个代码块是可以接收一个代码块参数的：`|arg1, arg2, &block|`。这样使诸如`Module#define_method`的方法变得更有用。在Ruby 2.0中，可以给默认值传递代码块参数。

注2： 有人质疑如果一个代码块没有实际使用它所携带的任何内容就不是一个“真正的”闭包：假设Ruby支持的话，使用一块“哑”代码块可以完成相同的工作。为了简化内容，我们没有讨论这点。

变量 ceiling 当代码块被定义时处于在范围之内，但它在执行流进入 select 方法时超出了范围。虽然如此，代码块仍可以从 select 内部访问 ceiling，因为它包含了围绕自己的上下文。这就是使其成为闭包的东西。

我们怀疑许多人在谈论 Ruby 代码块时说“闭包”仅仅是为了听起来很聪明。由于我们已经丧失了所有可能会令我们听起来很聪明的机会，因此我们决定在本书中只用普通的“块”来指代 Ruby 闭包。唯一例外的情况出现在少数我们必须讨论使 Ruby 代码块成为真正闭包、而不是“哑”代码块的上下文的地方。

7.1 代码块的创建与调用

问题

要将某段 Ruby 代码放入一个对象，使得可以传递并随后调用它。

解决方案

到目前为止，应当熟悉代码块是指以花括号括起的某段 Ruby 代码。读者可能会认为可以如下定义一个代码块对象：

```
aBlock = { |x| puts x }                 # WRONG

# SyntaxError: compile error
```

这样做无法工作是因为代码块只有当它是对某个方法调用的参数时才是合法的 Ruby 语法。存在若干等价的方法可以获得一个代码块并返回它作为一个对象。最受人欢迎的方法是 Kernel#lambda（注 3）：

```
aBlock = lambda { |x| puts x }          # RIGHT
```

要调用该代码块，可使用 call 方法：

```
aBlock.call "Hello World!"
# Hello World!
```

讨论

将一小块代码块赋给一个变量的能力是非常强大的。它使得用户可以编写常规框架并在关键点插入特定代码段。

注 3： 名字 lambda 来源于 Lisp 的 lambda 运算（一种数学形式化系统）。

在7.2节中可以发现，通过在参数名上预先加&可以接收一个代码块作为某个方法的参数。使用这种方式，可以编写自己的lambda方法的平凡版本：

```
def my_lambda(&aBlock)
  aBlock
end

b = my_lambda { puts "Hello World My Way!" }
b.call
# Hello World My Way!
```

最近定义的代码块实际是一个Proc对象。

```
b.class                                 # => Proc
```

还可以使用Proc构造器或方法Kernel#proc初始化代码块。Kernel#lambda、Kernel#proc和Proc.new方法基本上可以完成相同的工作。下列三行代码几乎是等价的：

```
aBlock = Proc.new { |x| puts x }
aBlock = proc { |x| puts x }
aBlock = lambda { |x| puts x }
```

有什么不同呢？Kernel#lambda是创建代码块对象的首选方法，因为它令代码块对象的执行像Ruby方法一样。考虑一下当使用错误的参数数目调用一个代码块时的结果：

```
add_lambda = lambda { |x,y| x + y }

add_lambda.call(4)
# ArgumentError: wrong number of arguments (1 for 2)

add_lambda.call(4,5,6)
# ArgumentError: wrong number of arguments (3 for 2)
```

使用lambda创建的代码块执行类似于Ruby方法。如果没有指定正确的参数数目，那么就无法调用代码块。但是，使用Proc.new创建的代码块执行类似于传递给诸如Enumerable#each的方法的匿名代码块：

```
add_procnew = Proc.new { |x,y| x + y }

add_procnew.call(4)
# TypeError: nil can't be coerced into Fixnum

add_procnew.call(4,5,6)                                 # => 9
```

如果在调用代码块时没有指定足够的参数，那么剩余参数会赋为空。如果指定了过多的参数，那么余下的参数被忽略。除非用户希望这类行为存在，否则最好使用lambda。

在Ruby 1.8中，Kernel#proc执行类似于Kernel#lambda。在Ruby 1.9中，Kernel#proc执行类似于Proc.new，更与其名字相称。

参考

- 7.2 节“ 编写接收代码块的方法 ”
- 10.4 节“ 获得方法的引用 ”

7.2 编写接收代码块的方法

问题

要编写一个方法，能够接收和调用附加的代码块：一个类似于`Array#each`、`Fixnum#upto`以及其他内置 Ruby 方法的方法。

解决方案

用户无需做任何特别的事情即可令自己的方法具有接收代码块的能力。只要调用传递代码块, 任意方法都可以使用代码块。在自己方法的任何时间内都可以使用`yield`调用代码块：

```
def call_twice
  puts "I'm about to call your block."
  yield
  puts "I'm about to call your block again."
  yield
end

call_twice { puts "Hi, I'm a talking code block." }
# I'm about to call your block.
# Hi, I'm a talking code block.
# I'm about to call your block again.
# Hi, I'm a talking code block.
```

另一个示例：

```
def repeat(n)
 if block_given?
   n.times { yield }
  else
    raise ArgumentError.new("I can't repeat a block you don't give me!")
  end
end

repeat(4) { puts "Hello." }
# Hello.
# Hello.
# Hello.
# Hello.

repeat(4)
# ArgumentError: I can't repeat a block you don't give me!
```

讨论

由于 Ruby 对于迭代和接收代码块的其他方法非常重视，因而知晓如何在自己方法中使用代码块非常重要。

用户不必做特别处理使得自己的方法具有接收代码块的功能。调用者可以向任意 Ruby 方法传递代码块，如果方法从不调用 `yield`，那么即使传递了也没有用。

```
puts("Print this message.") { puts "And also run this code block!" }
# Print this message.
```

`yield` 关键字的行为类似于一个特殊的方法，替代传入的任意代码块。调用它时，它就如同代码块是一个 `Proc` 对象并且已经调用了它的 `call` 方法。

如果不熟悉传递代码块的实际方式，可能会感到有些神秘，但是，它在 Ruby 中通常是调用代码块的首选方法。如果更习惯于将接收代码块视为传入方法的“真正”的参数，那么可以参见 7.3 节。

可以向 `yield` 传递参数（它们将被传给代码块），并且可以使用 `yield` 语句的值（也即代码块中最后语句的值）工作。

下面是一个向其代码块传递参数并使用代码块值的方法：

```
def call_twice
  puts "Calling your block."
  ret1 = yield("very first")
  puts "The value of your block: #{ret1}"

  puts "Calling your block again."
  ret2 = yield("second")
  puts "The value of your block: #{ret2}"
end

call_twice do |which_time|
  puts "I'm a code block, called for the #{which_time} time."
  which_time == "very first" ? 1 : 2
end
# Calling your block.
# I'm a code block, called for the very first time.
# The value of your block: 1
# Calling your block again.
# I'm a code block, called for the second time.
# The value of your block: 2
```

下面是一个更现实的示例。方法 `Hash#find` 获取一个代码块，将散列的每个键 - 值对传递给代码块并返回令代码块计算为 true 的第一个键 - 值对。

```
squares = {0=>0, 1=>1, 2=>4, 3=>9}
squares.find { |key, value| key > 1 }          # => [2, 4]
```

假设我们希望一个方法的工作类似于Hash#find, 但返回一个包含令代码块计算为true的所有键 - 值对的新散列。这可以通过传递参数给 yield 语句并使用其结果完成：

```
class Hash
  def find_all
    new_hash = Hash.new
    each { |k,v| new_hash[k] = v if yield(k, v) }
    new_hash
  end
end

squares.find_all { |key, value| key > 1 }        # => {2=>4, 3=>9}
```

结果证明，Hash#find 方法已经完成了我们所希望的逆转。通过取反我们代码块的结果，我们可以令Hash#delete_if做Hash#find_all的工作，只需要清理散列中的重复项，因为 delete_if 是一种破坏性方法：

```
squares.dup.delete_if { |key, value| key > 1 }   # => {0=>0, 1=>1}
squares.dup.delete_if { |key, value| key <= 1 }  # => {2=>4, 3=>9}
```

Hash#find_all 经过证明并非是必要的，但它促成了一个很好的示例。

通过在自己方法中调用Kernel#block_given?就可以编写一个接收可选代码块的方法。该方法仅在对该方法的调用者传入一个代码块时返回 true。如果它返回 false，那么可以引发一个异常，或者后退至不需要代码块以及不使用 yield 关键字时的动作。

如果方法调用 yield 而调用者没有传入一个代码块，那么 Ruby 会引发一个异常：

```
[1, 2, 3].each
# LocalJumpError: no block given
```

参考

- 7.3 节“ 将代码块参数与变量绑定 ”

7.3 将代码块参数与变量绑定

问题

已经编写了一个接收代码块的方法，但简单地使用yield调用代码块对用户而言不够用。现在希望以某种方式将代码块绑定至变量，以便于直接操作代码块。可能最希望将它作为代码块传递给另一方法。

解决方案

在方法的参数列表最后放入块变量的名称。使用&作为前缀，使得Ruby知晓它是一个代码块参数，而不是一个常规参数。

一个传入的代码块将被转换为一个`Proc`对象并绑定至块变量。可以将它传递给其他方法，使用`call`直接调用它，或者对它使用`yield`，就仿佛从来没有将它绑定至变量。下列三种方法几乎完成相同的工作：

```
def repeat(n)
  n.times { yield } if block_given?
end
repeat(2) { puts "Hello." }
# Hello.
# Hello.

def repeat(n, &block)
  n.times { block.call } if block
end
repeat(2) { puts "Hello." }
# Hello.
# Hello.

def repeat(n, &block)
  n.times { yield } if block
end
repeat(2) { puts "Hello." }
# Hello.
# Hello.
```

讨论

如果`&foo`是方法的最后一个参数的名字，那么它意味着方法接收一个名为`foo`的可选代码块。如果调用者选择传入一个代码块，那么它将被作为一个绑定至变量`foo`的`Proc`对象提供。由于它是一个可选参数，因此，如果实际没有传入代码块，那么`foo`为空。这样可以将用户从必须调用`Kernel#block_given?`来查看是否获得了从代码块中解放出来。

当调用一个方法时，可以通过在适当变量名前加`&`前缀，将任意`Proc`对象作为代码块传入。甚至可以在最初作为代码块传递给用户方法的`Proc`对象上这样做。

许多用于集合的方法，如`each`、`select`和`detect`都接收代码块。当自己的方法可以将代码块绑定至变量时，很容易封装这种方法。下列代码是一个名为`biggest`的方法，它查找集合中令给定代码块结果为true的最大元素：

```
def biggest(collection, &block)
  block ? collection.select(&block).max : collection.max
```

```
end

array = [1, 2, 3, 4, 5]
biggest(array) {|i| i < 3}                  # => 2
biggest(array) {|i| i != 5 }                # => 4
biggest(array)                              # => 5
```

这在需要为接收代码块的方法编写前端时也非常有用。封装器方法可以将传入的代码块绑定至一个变量，然后将它作为代码块传递给其他方法。

下列代码调用一个代码块有限次数，每次传入一个位于 `min` 和 `max` 之间的随机数：

```
def pick_random_numbers(min, max, limit)
  limit.times { yield min+rand(max+1) }
end
```

下列是针对 `pick_random_numbers` 的一个封装器方法，它调用一个代码块 6 次，每次传入一个从 1～49 的随机数：

```
def lottery_style_numbers(&block)
  pick_random_numbers(1, 49, 6, &block)
end

lottery_style_numbers { |n| puts "Lucky number: #{n}" }
# Lucky number: 20
# Lucky number: 39
# Lucky number: 41
# Lucky number: 10
# Lucky number: 41
# Lucky number: 32
```

代码块参数必须总是为某个方法定义的最后一个参数。这意味着如果方法接收的参数数目是可变的，那么代码块参数要位于用于该可变参数的容器之后：

```
def invoke_on_each(*args, &block)
  args.each { |arg| yield arg }
end

invoke_on_each(1, 2, 3, 4) { |x| puts x ** 2 }
# 1
# 4
# 9
# 16
```

参考

- 8.11 节“ 接收或传递参数的变量数目 ”
- 回忆章节介绍中的内容，在 Ruby 1.8 中，代码块本身不能接收代码块参数，这一点在 Ruby 1.9 中得以修正

7.4 作为闭包的代码块：在代码块内使用外部变量

问题

要在某个方法和该方法内定义的代码块之间共享变量。

解决方案

只需引用变量即可，Ruby会完成正确的工作。下面是一个为数组的每个元素增加特定数的方法：

```
def add_to_all(array, number)
  array.collect { |x| x + number }
end

add_to_all([1, 2, 3], 10)               # => [11, 12, 13]
```

`Enumerable#collect`不能直接访问`number`，但它传递了一个可以访问`number`的代码块，因为`number`位于定义代码块的范围内。

讨论

一个Ruby代码块就是一个闭包：它包含了它被定义时的上下文。这一点非常有用，因为它允许像一段普通代码一样定义一个代码块，然后匆匆完成并传递给预定义的一段代码进行处理。

一个Ruby代码块包含对变量绑定的引用，而不是值的副本。如果变量随后发生改变，那么代码块将会访问新值：

```
tax_percent = 6
position = lambda do
  "I have always supported a #{tax_percent}% tax on imported limes."
end
position.call
# => "I have always supported a 6% tax on imported limes."

tax_percent = 7.25
position.call
# => "I have always supported a 7.25% tax on imported limes."
```

这在两种方式下皆有效：可以重新绑定或修改来自代码块内的变量。

```
counter = 0
4.times { counter += 1; puts "Counter now #{counter}"}
```

```
# Counter now 1
# Counter now 2
# Counter now 3
# Counter now 4
counter                                                    # => 4
```

这在希望与陌生迭代器协同模拟`inject`或`collct`时特别有用。可以在代码块外部创建一个存储对象，并在代码块内向其中添加内容。下列代码模拟`Enumerable#collect`，但它以反序收集数组的元素：

```
accumulator = []
[1, 2, 3].reverse_each { |x| accumulator << x + 1 }

accumulator                  # => [4, 3, 2]
```

`accumulator`变量不在`Array#reverse_each`的范围内，但它在代码块的范围内。

7.5 在数据结构上编写迭代器

问题

已经创建一个自定义的数据结构，并希望为它实现一个`each`方法，或者希望实现一种在现有数据结构上进行迭代的特别方法。

解决方案

复杂数据结构通常在基本数据结构范围外构造：散列、数组，等等。所有的基本数据结构都定义了`each`方法。如果数据结构完全由标量值和这些简单数据结构组成，那么可以按照其构成成分的`each`方法编写一个新`each`方法。

下面是一个简单的树数据结构。一棵树包含一个单独的值和一系列孩子（每个孩子是一个较小的树）。

```
class Tree
  attr_reader :value
  def initialize(value)
    @value = value
    @children = []
  end

  def <<(value)
    subtree = Tree.new(value)
    @children << subtree
    return subtree
  end
end
```

下列代码创建一棵特定树（图 7-1）：

```
t = Tree.new("Parent")
child1 = t << "Child 1"
child1 << "Grandchild 1.1"
child1 << "Grandchild 1.2"
child2 = t << "Child 2"
child2 << "Grandchild 2.1"
```

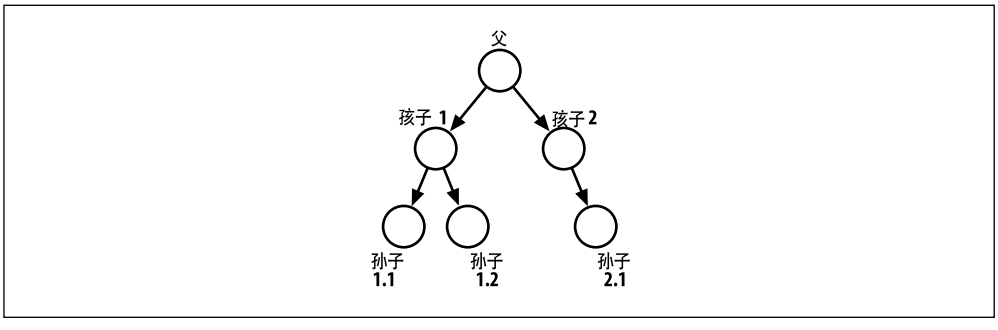

图 7-1　一棵简单的树

我们如何在这种数据结构上迭代呢？由于树是递归定义的，因此在其上进行递归迭代才有意义。`Tree#each` 的这种实现产生树中存储的值，然后在其孩子（子树存储在已经支持 `each` 的数组中）上迭代并在每个子树上递归地调用 `Tree#each`。

```
class Tree
  def each
    yield value
    @children.each do |child_node|
      child_node.each { |e| yield e }
    end
  end
end
```

`each` 方法以看起来正确的方式遍历树：

```
t.each { |x| puts x }
# Parent
# Child 1
# Grandchild 1.1
# Grandchild 1.2
# Child 2
# Grandchild 2.1
```

讨论

构建一个迭代器的最简单方法是递归：使用较小的迭代器直到遍历用户数据结构中的每

个元素。但是，如果这些迭代器不存在怎么办？最可能的情况是如果存在但以错误序列给出元素怎么办？需要下降一层并编写某些循环。

循环在 Ruby 中的地位有些低，因为迭代器更合乎语言习惯，但是，当编写迭代器时，除了使用循环可能别无选择。下面是对 4.1 节中迭代器的翻版，它举例说明了如何使用循环在一个数组的两端同时迭代：

```
class Array
 def each_from_both_sides(  )
    front_index = 0
    back_index = self.length-1
    while front_index <= back_index
      yield self[front_index]
      front_index += 1
      if front_index <= back_index
        yield self[back_index]
        back_index -= 1
      end
    end
  end
end

%w{Curses! been again! foiled I've}.each_from_both_sides { |x| puts x }
# Curses!
# I've
# been
# foiled
# again!
```

下列是两个更简单的迭代器。第一个在一行中多次生成每个元素：

```
module Enumerable
  def each_n_times(n)
    each { |e| n.times { yield e } }
  end
end

%w{Hello Echo}.each_n_times(3) { |x| puts x }
# Hello
# Hello
# Hello
# Echo
# Echo
# Echo
```

下一个以随机顺序返回 `Enumerable` 的元素；有关以更有效方式完成顺序打乱的内容请参见 4.10 节。

```
module Enumerable
  def each_randomly
    (sort_by { rand }).each { |e| yield e }
```

```
  end
end
%w{Eat at Joe's}.each_randomly { |x| puts x }
# Eat
# Joe's
# at
```

参考

- 4.1 节“ 在数组上迭代 ”
- 4.10 节“ 打乱数组 ”
- 5.7 节“ 在散列上迭代 ”
- 7.6 节“ 更改对象的迭代方式 ”
- 7.8 节“ 停止迭代 ”
- 7.9 节“ 并行地在多个迭代变量上循环 ”

7.6 更改对象的迭代方式

问题

要使用数据结构作为 `Enumerable`，但对象的 `#each` 实现不按用户希望的方式迭代。由于 `Enumerable` 的所有方法都基于 `each`，因而使得它们对用户皆无用处。

解决方案

下列是一个具体的示例：一个简单的数组。

```
array = %w{bob loves alice}
array.collect { |x| x.capitalize }
# => ["Bob", "Loves", "Alice"]
```

假设我们希望在该数组上调用 `collect`，但不希望 `collect` 使用 `each`：我们希望它使用 `reverse_each`。假定类似的这种方法为 `collect_reverse`：

```
array.collect_reverse { |x| x.capitalize }
# => ["Alice", "Loves", "Bob"]
```

事实上，定义一个 `collect_reverse` 方法会添加重要的新代码并且只能解决部分问题。我们将使用调用 `reverse_each` 的一个单独方法来重写数组的 `each` 实现，但这是种小技巧并且肯定会带来不希望出现的副作用。

幸运地是，存在一个没有副作用的极佳解决方案：在 Enumerator 中封装对象。如果用它替换其 each 方法，那么它会给出一个行为类似于旧对象的新对象：

```
require 'enumerator'
reversed_array = array.to_enum(:reverse_each)
reversed_array.collect { |x| x.capitalize }
# => ["Alice", "Loves", "Bob"]

reversed_array.each_with_index do |x, i|
  puts %{#{i}=>"#{x}"}
end
# 0=>"alice"
# 1=>"loves"
# 2=>"bob"
```

要注意，不能像实际数组一样对我们的数组使用 Enumerator。只有 Enumerable 的方法才被支持：

```
reversed_array[0]
# NoMethodError: undefined method `[]' for
#<Enumerable::Enumerator:0xb7c2cc8c>
```

讨论

只要试图重新实现 Enumerable 的某个方法，都可以试用 Enumerator 来代替。它看起来修改了对象的 each 方法，但没有影响原始对象。

这样能够节省用户的许多工作。假设有一个提供了三种不同迭代风格的数据结构：each_prefix、each_postfix 和 each_infix。与其为所有这三种迭代风格实现 Enumerable 的方法，不如令 each_prefix 作为 each 的默认实现，并在需要不同行为 Enumerable 时调用 tree.to_enum(:each_postfix) 或 tree.to_enum(:each_infix)。

一个基本对象可以拥有多个 Enumerable 对象。下面是第二个用于我们简单数组的 Enumerable，其中 each 的行为类似于 each_with_index 对原始数组的操作：

```
array_with_index = array.enum_with_index
array_with_index.each do |x, i|
  puts %{#{i}=>"#{x}"}
end
# 0=>"bob"
# 1=>"loves"
# 2=>"alice"

array_with_index.each_with_index do |x, i|
  puts %{#{i}=>#{x.inspect}}
end
# 0=>["bob", 0]
```

```
# 1=>["loves", 1]
# 2=>["alice", 2]
```

当请求‘enumerator’时，Enumerable给出两种额外的枚举方法：each_cons和each_slice。这些方法令数据结构上的大量迭代变得很容易。下列示例是展示其工作过程的最好方式：

```
sentence = %w{Well, now I've seen everything!}

two_word_window = sentence.to_enum(:each_cons, 2)
two_word_window.each { |x| puts x.inspect }
# ["Well,", "now"]
# ["now", "I've"]
# ["I've", "seen"]
# ["seen", "everything!"]

two_words_at_a_time = sentence.to_enum(:each_slice, 2)
two_words_at_a_time.each { |x| puts x.inspect }
# ["Well,", "now"]
# ["I've", "seen"]
# ["everything!"]
```

要注意，传入to_enum的所有参数是如何随传入迭代方法本身的参数进行传递的。

在Ruby 1.9中，Enumerable::Enumerator类是Ruby核心的组成部分，不需要请求声明。另外，each_cons和each_slice都是Enumerable的内置方法。

参考

- 7.9节“并行地在多个迭代变量上循环”
- 20.6节“在很多对象上同时运行代码块”

7.7 编写分类或收集的代码块方法

问题

Ruby标准库自带的基本代码块方法对用户而言不够用，用户希望定义自己的方法对枚举中元素进行分类（类似Enumerable#detect和Enumerable#find_all），或者对枚举中的每个元素进行转换（类似Enumerable#collect）。

解决方案

通常可以使用inject编写方法对对象的枚举进行搜索或分类。使用inject可以编写方法的自己版本，例如detect和find_all：

```
module Enumerable
  def find_no_more_than(limit)
    inject([]) do |a,e|
      a << e if yield e
      return a if a.size >= limit
      a
    end
  end
end
```

下列代码发现某个列表中至多有三个偶数：

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
a.find_no_more_than(3) { |x| x % 2 == 0 }            # => [2, 4, 6]
```

如果发现自己需要编写类似collect的方法，那么可能是由于对于个人目的而言，collect本身产生的元素顺序不对。用户不能使用inject，因为它产生的元素顺序与collect相同。

需要查找或编写一个迭代器，能够按自己希望的顺序产生元素。一旦这样做了，存在两种选择：可以在迭代器方法上面编写一个与collect等价的方法，或者使用迭代器方法构建一个Enumerable对象并调用其collect方法（参见 7.6 节）。

讨论

我们在第 4 章中更详细地讨论了这些代码块方法，因为数组是最简单、最普通的枚举数据类型，并且是最常见的。但是，几乎所有数据结构都可以枚举，并且稍复杂的数据结构可以不同方式进行枚举。

如 9.4 节所示，Enumerable的方法，如detect和inject，实际是以each的方式实现。detect和inject方法为代码块传递由each产生的所有元素。yield语句的值用于确定元素是否与某种标准相匹配。

在诸如detect的方法中，一旦发现匹配的元素迭代可能会停止。在诸如find_all的方法中，迭代会遍历所有元素，收集所有匹配的元素。

诸如collect的方法以相同方式工作，但是，它们在新数据结构中收集代码块返回的值并在迭代完成后返回该数据结构，而不是根据代码块的要求返回元素的某个子集。

如果使用的是一个特殊对象并希望其collect方法使用一个不同的迭代器，那么应当将对象转换为Enumerator并调用其collect方法。但是，如果正在编写一个类并希

望编写一个类似collect的新方法，那么必须定义一个新方法（注4）。在这种情况下，最好的解决方案可能是编写一个返回自定义的Enumerator的方法：这样，方法的用户就可以使用Enumerable的所有方法，而不仅是collect了。

参考

- 4.5节“数组排序”
- 4.11节“获取数组的N个最小项”
- 4.15节“集合划分或分类”
- 7.6节“更改对象的迭代方式”
- 如果读者希望的只是令自己的用户数据结构支持Enumerable的方法，那么可以参见9.4节“实现Enumerable：编写一个方法，获得22种免费方法”

7.8 停止迭代

问题

希望在传入的代码块内部中止迭代。

解决方案

中止执行的最简单方法是使用break。break语句会跳出当前方法中定义的最近闭循环：

```
1.upto(10) do |x|
  puts x
  break if x == 3
end
# 1
# 2
# 3
```

讨论

break语句虽然简单，但有许多限制。不能在使用Proc.net或（在Ruby 1.9及以上版本）Kernel#proc定义的代码块中使用break。如果这对用户是个问题的话，那么可以使用lambda代替：

注4： 当然，在这些场景以外，用户的方法可以创建一个适当的Enumerator并调用它的collect实现。

```
aBlock = Proc.new do |x|
  puts x
  break if x == 3
  puts x + 2
end

aBlock.call(5)
# 5
# 7

aBlock.call(3)
# 3
# LocalJumpError: break from proc-closure
```

更严重的是，不能使用break同时从多个循环中跳出。一旦某个循环已经运行，那么就无法知道它是正常结束的还是使用 break 结束的。

关于这一问题最简单的方法是用一个描述符号名在一个catch代码块中括起希望跳出的代码，然后可以在希望跳至catch代码块尾部的时候抛出相应的符号。这样就可以从任意多嵌套循环和方法调用中跳出了。

throw/catch 语法不是异常处理——异常使用 raise/rescue 语法。这是一种特殊的流控制结构，设计目的是使用流控制代替异常（有时在 Java 程序中也存在）。它有点像老式的全局 GOTO，能够突然将执行移动至程序的远处部分。但是，它比 GOTO 更能保持代码的可读性，因为它是受限的：throw 只能跳至相应 catch 代码块的尾部。

catch..throw 语法的最好示例是 6.12 节中描述的 Find.find 函数。当向 Find.find 传递一个代码块时，它会被迫放弃特定目录中的所有目录和文件。当给定代码块一个目录时，它能够通过调用Find.prune抛出一个:prune符号来阻止find继续向该目录递归查找。使用 break 会完全停止 find 操作，而抛出一个符号可令 Find.prune 知道只跳出一个目录。

下面是 Find.find 和 Find.prune 代码的简化版：

```
def find(*paths)
  paths.each do |p|
    catch(:prune) do
      # Process p as a file or directory...
    end
    # When you call Find.prune you'll end up here.
  end
end

def prune
  throw :prune
end
```

当调用Find.prune时，执行跳转位置紧随catch(:prune)代码块其后。Find.find然后开始处理下一文件或目录。

参考

- 6.12 节“遍历目录树”
- ri Find

7.9 并行地在多个迭代变量上循环

问题

要同时遍历多个迭代方法，可能要协调若干不同数组中的相应元素。

解决方案

生成器库中定义的SyncEnumerator类能够很容易地在多个数组或其他Enumerable对象上并行执行迭代，它的each方法产生一系列数组，每个数组包含来自每个潜在Enumerable对象的一个项：

```
require 'generator'

enumerator = SyncEnumerator.new(%w{Four seven}, %w{score years},
                                %w{and ago})
enumerator.each do |row|
  row.each { |word| puts word }
  puts '---'
end
# Four
# score
# and
# ---
# seven
# years
# ago
# ---

enumerator = SyncEnumerator.new(%w{Four and}, %w{score seven years ago})
enumerator.each do |row|
  row.each { |word| puts word }
  puts '---'
end
# Four
# score
# ---
# and
```

```
# seven
# ---
# nil
# years
# ---
# nil
# ago
# ---
```

可以通过在一个 `Generator` 对象中封装自己所有的 `Enumerable` 对象令 `SyncEnumerator` 重新工作。下列代码的运行类似于 `SyncEnumerator#each`，只是它产生每个单独项，而不是包含来自每个 `Enumerable` 中一个项的数组：

```
def interosculate(*enumerables)
  generators = enumerables.collect { |x| Generator.new(x) }
  done = false
  until done
    done = true
    generators.each do |g|
      if g.next?
        yield g.next
        done = false
      end
    end
  end
end

interosculate(%w{Four and}, %w{score seven years ago}) do |x|
  puts x
end
# Four
# score
# and
# seven
# years
# ago
```

讨论

任何实现 `each` 方法的对象都能够被封装在 `Generator` 对象中。如果读者使用过 Java，那么可以将 `Generator` 想象成 Java 的 `Iterator` 对象，它保持用户在某个数组结构上执行特定迭代的轨迹。

一般而言，当向一个诸如 `each` 的迭代器方法传递一个代码块时，该代码块将不中断地在迭代器中的每个元素上被调用。在迭代器完成迭代之前，该代码块之外的代码不会被执行。用户可以通过在代码块内编写 `break` 语句来停止迭代，但无法随后在相同位置重新启动被打断地迭代——除非使用 `Generator`。

可以将诸如each的迭代器方法视为一台糖果自动售货机，一旦按下按钮，它就会以稳定流吞出其所有糖果。Generator类允许用户将糖果自动售货机转变为每个按下按钮只分配一部分糖果，用户可以拿着这个新自动售货机到处走动并更容易地分配糖果。

在Ruby 1.8中，Generator类使用延拓实现这一技巧。它为跳出之后再跳回迭代设置了书签，当调用Generator#next时，生成器“抽取”迭代一次（产生一个单独的元素），设置一个书签并将控制返回至用户代码。下次调用Generator#next时，生成器跳回它先前设置的书签并再“抽取”迭代一次。

Ruby 1.9使用一种基本线程的更高效实现。该实现调用每个Enumerable对象的each方法（触发糖果的永不结束流），但它针对每个对象在一个单独线程中完成这一工作。每批糖果出来后，Ruby冻结时间（暂停线程）直至下次调用Generator#next为止。

在一个生成器中封装一个数组很容易，但如果要将所有数组放入生成器中，就不必使用Generators、甚至SyncEnumerable浪费时间了。将一个索引放入每个数组并在希望从特定数组中获取另一个项时令索引增长就可以很容易地模拟SyncEnumerable对数组的行为。Generator方法将任意类型的迭代转换为单项糖果自动售货机的功能确实非常有用。

假设希望使用生成器的功能在数组上执行迭代，但心里想使用的迭代却不是普通的。例如，考虑类似下列的一个数组：

```
l = ["junk1", 1, "junk2", 2, "junk3", "junk4", 3, "junk5"]
```

我们说，用户希望在列表上迭代但跳过“junk”条目。在一个生成器对象中封装该列表是无效的，因为它会给出所有的条目：

```
g = Generator.new(l)
g.next                                  # => "junk1"
g.next                                  # => 1
g.next                                  # => "junk2"
```

编写一个跳过junk的迭代器方法并不难。现在，我们不希望用迭代器方法——我们希望用Generator对象——但迭代器方法是一个良好起点，至少它可以证明我们希望的迭代可以用Ruby实现。

```
def l.my_iterator
  each { |e| yield e unless e =~ /^junk/ }
end

l.my_iterator { |x| puts x }
# 1
# 2
# 3
```

下面是一个小技巧：当用户在Generator或SyncEnumerable对象中封装一个数组时，实际上已经封装了数组的each方法。Generator不仅仅按each产生元素的顺序产生元素：它实际调用each，但在每次调用Generator#next之后使用continuation（或线程）欺骗暂停迭代。

通过定义一段适当的代码块并传递给Generator构造器，用户可以在任意迭代代码段（不仅仅是每个方法）之外构造一个generation对象。生成器会知道调用并打断该代码块，就如同在向构造器传递数组时它知道调用并打断each一样。下面是一个按我们希望的方式在数组上迭代的生成器：

```
g = Generator.new { |g| l.each { |e| g.yield e unless e =~ /^junk/ } }
g.next                                                  # => 1
g.next                                                  # => 2
g.next                                                  # => 3
```

Generator构造器可以将接收生成器对象的代码块本身作为参数。该代码块执行用户希望在生成器中封装的迭代。要注意代码块与l#my_iterator方法体的基本类似处。不同之处在于不使用调用Generator#yield函数的yield关键字，该函数处理涉及到设置和跳转至continuation的工作（Generator#next处理其他的continuation工作）。

明白了这一工作原理之后，就可以通过在Generator中封装l#my_iterator方法本身从而消除某些重复代码：

```
g = Generator.new { |g| l.my_iterator { |e| g.yield e } }
g.next                                                  # => 1
g.next                                                  # => 2
g.next                                                  # => 3
```

下面是能够封装方法以及数组的interosculate方法的一种版本，该方法接收Enumerable对象和Method对象的任意组合，将每个对象转换为一个Generator对象，并在所有Generator对象上循环，每个从一个对象中获取一个元素：

```
def interosculate(*iteratables)
  generators = iteratables.collect do |x|
    if x.is_a? Method
      Generator.new { |g| x.call { |e| g.yield e } }
    else
      Generator.new(x)
    end
  end
  done = false
  until done
    done = true
    generators.each do |g|
      if g.next?
        yield g.next
```

```
          done = false
        end
      end
    end
  end
```

下面，我们向`interosculate`传递一个数组和一个`Method`对象，这样就可以相反方向在两个数组上执行迭代：

```
words1 = %w{Four and years}
words2 = %w{ago seven score}
interosculate(words1, words2.method(:reverse_each)) { |x| puts x }
# Four
# score
# and
# seven
# years
# ago
```

参考

- 7.5 节“ 在数据结构上编写迭代器 ”
- 7.6 节“ 更改对象的迭代方式 ”

7.10 隐藏块方法中的设置与清除

问题

有一个总需要在自定义代码之前运行的设置方法，或者有一个需要在代码之后运行的清除方法。用户不相信编写代码的人（可能是他自己）能记住调用设置和清除方法。

解决方案

创建一个运行设置代码的方法，产生一个代码块（包含自定义的代码），然后运行清除代码。为了确保即使自定义代码抛出一个异常清除代码也始终运行，可以使用`begin/finally`块。

```
def between_setup_and_cleanup
  setup
  begin
   yield
  finally
    cleanup
  end
```

```
end
```

下面是一个具体示例。它向一个 HTML 文档的开头添加一个 DOCTYPE 和一个 HTML 标记。最后，它关闭之前打开的 HTML 标记。这样能够在生成 HTML 文件时节省用户的少量工作。

```
def write_html(out, doctype=nil)
  doctype ||= %{<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional/
/EN"
            "http://www.w3.org/TR/html4/loose.dtd">}
  out.puts doctype
  out.puts '<html>'
  begin
    yield out
  ensure
    out.puts '</html>'
  end
end

write_html($stdout) do |out|
  out.puts '<h1>Sorry, the Web is closed.</h1>'
end
# <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
#     "http://www.w3.org/TR/html4/loose.dtd">
# <html>
# <h1>Sorry, the Web is closed.</h1>
# </html>
```

讨论

这一有用的技术最经常出现在稀有资源（例如文件句柄或数据库链接）必须在使用完后关闭以免资源用尽的情况中。让编程人员记住这些资源的语言由于程序员的懒散而有资源泄漏的倾向。Ruby 很容易地实现了懒散仍能正确地完成工作。

用户在使用`Kernel#open`和`File#open`方法打开磁盘上的文件时可能已经用过这一技术。这些方法接收一个操作已经打开的文件的代码块，它们打开文件，调用用户代码块，并在完成工作后关闭文件：

```
open('output.txt', 'w') do |out|
  out.puts 'Sorry, the filesystem is also closed.'
end
```

Ruby 的标准`cgi`模块以`write_html`示例展示了其逻辑结论（注 5）。用户可以通过在

注 5： 但是如果使用模板代替用 Ruby 代码编写 HTML 的话，代码会更具可维护性。

内部互相嵌套代码块来构造一个完整的HTML文档。下面是一个小Ruby CGI，它输出的文档与上述`write_html`示例大致相同。

```
#!/usr/bin/ruby

# closed_cgi.rb
require 'cgi'
c = CGI.new("html4")

c.out do
   c.html do
      c.h1 { 'Sorry, the Web is closed.' }
  end
end
```

要注意代码块的多个层次：传递给`CGI#out`的代码块简单地调用`CGI#html`生成DOCTYPE和`<html>`标记。`<html>`标记包含对`CGI#h1`调用的结果，`CGI#h1`在`<h1>`标记中封装某些纯文本。程序产生如下输出：

```
Content-Type: text/html
Content-Length: 137

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<HTML><H1>Sorry, the Web is closed.</H1></HTML>
```

Ruby的`builder` gem中的`XmlMarkup`类以相同方式工作：可以编写类似于它创建的文件结构的Ruby代码：

```
require 'rubygems'
require 'builder'
xml = Builder::XmlMarkup.new.message('type' => 'apology') do |b|
  b.content('Sorry, Web Services are closed.')
end
puts xml
# <message type="apology">
#  <content>Sorry, Web Services are closed.</content>
# </message>
```

参考

- 6.13节“文件加锁”使用这一技术创建一个锁文件的方法，并在用户使用完后自动地解锁
- 11.9节“创建并修改XML文档”
- 20.11节“避免死锁”使用这一技术令用户线程以正确顺序加锁多个资源，并在用完后解锁

7.11 使用回调的松耦合系统

问题

希望联合不同类型的对象，但不使用充满相互引用的硬编码。

解决方案

使用回调系统，其中对象彼此间注册代码块以便在需要时执行。一个对象可以在需要时调用它注册的回调，或者可以在进行工作时向回调发送通知。

为了实现一个回调系统，要编写一个接收代码块的“ 注册 ”或“ 预订 ”方法。在数据结构内将注册的代码存储为 `Proc` 对象：可能是数组（如果仅有一种类型的回调的话）或散列（如果有多种类型的话）。当需要调用回调时，可以在数据结构上迭代并调用每个注册的代码块。

下面是一个混合模块，为某个类的每个实现给定其拥有的“ 侦听者 ”回调代码块的散列。一个外部对象可以通过使用事件名和一段代码块调用`subscribe`来侦听某个特定事件。分配程序本身负责在适当时间使用适当的事件名调用`notify`，而外部对象负责传递它希望“ 侦听 ”的事件名。

```
module EventDispatcher
  def setup_listeners
    @event_dispatcher_listeners = {}
  end

  def subscribe(event, &callback)
    (@event_dispatcher_listeners[event] ||= []) << callback
  end

  protected
  def notify(event, *args)
    if @event_dispatcher_listeners[event]
      @event_dispatcher_listeners[event].each do |m|
        m.call(*args) if m.respond_to? :call
      end
    end
    return nil
  end
end
```

下面是一个`Factory`类，保持侦听者集合。一个外部对象可以选择在每个`Factory`对象被创建时得到通知，或者在每个 `Factory` 对象产生一个窗口小部件时得到通知：

```
class Factory
  include EventDispatcher
```

```
  def initialize
    setup_listeners
  end

  def produce_widget(color)
    #Widget creation code goes here...
    notify(:new_widget, color)
  end
end
```

下面是一个侦听者类，它对使用 `Factory` 对象的结果感兴趣：

```
class WidgetCounter
  def initialize(factory)
    @counts = Hash.new(0)
    factory.subscribe(:new_widget) do |color|
      @counts[color] += 1
      puts "#{@counts[color]} #{color} widget(s) created since I started
watching."
    end
  end
end
```

最后侦听类的运行：

```
f1 = Factory.new
WidgetCounter.new(f1)
f1.produce_widget("red")
# 1 red widget(s) created since I started watching.

f1.produce_widget("green")
# 1 green widget(s) created since I started watching.

f1.produce_widget("red")
# 2 red widget(s) created since I started watching.

# This won't produce any output, since our listener is listening to
# another Factory.
Factory.new.produce_widget("blue")
```

讨论

回调是令代码可扩展的一种重要技术。该技术有许多种名字（回调、hook 方法、插件、发布/预订，等等），但无论使用哪种术语，它都是相同的。一个对象在某种条件达到时请求另一个对象调用一段代码（回调），这一技术甚至在两个对象彼此几乎互无所知的时候也有效，理想地使大型、高内聚系统重新分解为较小、松耦合系统。

在一个纯侦听者系统（诸如解决方案一节中给出的）中，回调建立一直从事件分配程序

向侦听者移动的通信线路。这在用户拥有一个主对象（诸如 Factory）且众多从对象（诸如 WidgetCounter）从中获取所有尾接指令时非常有用。

但是，在许多松耦合系统中，信息是双方向移动的：分配程序调用回调，然后使用返回的结果。考虑原型的 Web 门户：一个用户可定制的由 HTML 方框构成的网页，包含体育成绩、天气预报，等等。由于总有新方框被添加到系统中，因此核心门户软件不应当必须知道所有有关特定方框的信息。方框也应当尽量少知道有关核心软件的信息，以便于修改核心不需要对所有方框进行修改。

对 EventDispatcher 类的简单修改可令分配程序使用注册回调的返回值。EventDispatcher#notify 的原始实现调用注册的代码块，但忽略其返回值。EventDispatcher#notify 的这一版本产生返回值给传递通知的代码块：

```
module EventDispatcher
  def notify(event, *args)
      if @event_dispatcher_listeners[event]
      @event_dispatcher_listeners[event].each do |m|
        yield(m.call(*args)) if m.respond_to? :call
      end
    end
    return nil
  end
end
```

下面是个非常简单的门户显示引擎，它令方框在HTML表内注册为在门户网页的两行中的一行上显示：

```
class Portal
  include EventDispatcher

  def initialize
    setup_listeners
  end

  def render
    puts '<table>'
    render_block = Proc.new { |box| puts "  <td>#{box}</td>" }
    [:row1, :row2].each do |row|
      puts ' <tr>'
      notify(row, &render_block)
      puts ' </tr>'
    end
  puts '</table>'
  end
end
```

下面是一个显示特定用户的门户规划的显示引擎。该用户喜欢看到左侧是股票行情计时

器和天气报告，右侧是一个新方框。要注意，这里甚至没有任何针对这些方框的类，它们简单到了可以代为匿名代码块实现的地步：

```
portal = Portal.new
portal.subscribe(:row1) { 'Stock Ticker' }
portal.subscribe(:row1) { 'Weather' }
portal.subscribe(:row2) { 'Pointless, Trivial News' }

portal.render
# <table>
#  <tr>
#   <td>Stock Ticker</td>
#   <td>Weather</td>
#  </tr>
#  <tr>
#   <td>Pointless, Trivial News</td>
#  </tr>
# </table>
```

如果希望在某个类的所有实例之间共享注册的侦听者,那么可以将侦听者作为一个类变量，将预订作为一个模块的方法。这在希望侦听者在类的任何新实例被创建时得到通知时最有用处。