Нумераторы, итераторы, лямбды

Андрей Васильев

2019

Нумераторы - объекты-итераторы

- Итераторы предлагают способ взаимодействия с внутренним состоянием объекта через вызов методов
- Очень сложно таким образом решить задачу синхронной обработки набора коллекций
- Иногда интересно передать итератор внутрь другого метода
- Класс `Enumerator` реализует «внешние итераторы»

```
a = [1, 3, "cat"]
h = {dog: "canine", fox: "vulpine"}
# Создаём нумераторы
enum_a = a.to_enum
enum_h = h.to_enum
enum_a.next # => 1
enum_h.next # => [:dog, "canine"]
enum_a.next # => 3
enum_h.next # => [:fox, "vulpine"]
```

Создание нумератора

- Вызов метода `to_enum` на коллекции или `enum_for` с указанием названия итератора
- Большинство стандартных итераторов возвращают нумераторы, если с ними не ассоциирован блок

```
a = [1, 3, "cat"]
enum_a = a.each # Созадаём нумератор
enum_a.next # => 1
enum_b = a.to_enum # Создаём нумератор
enum_c = a.enum_for(:each) # Создаём нумератор
```

Нумераторы являются объектами класса `Enumerator`, который включает в себя модуль `Enumerable`, что делает доступным для нумераторов всех «классных» методов

Метод `loop`

Задачей данного метода является бесконечный вызов блока. Если внутри блока используются нумераторы, то выход будет осуществлён, когда закончатся значения в нумераторе

```
short = [1, 2, 3].to_enum
long = ('a'..'z').to_enum
loop do
  puts " #{short.next} - #{long.next}"
end
```

Нумераторы являются объектами

Метод `each_with_index` определён в модуле `Enumerable`

```
result = []
['a', 'b', 'c'].each_with_index do |item, index|
  result << [item, index]
end
result # => [["a", 0], ["b", 1], ["c", 2]]
```

Метод `with_index` определён в классе `Enumerator`

```
result = []
"dog".each_char.with_index do |item, index|
  result << [item, index]
end
result # => [["d", 1], ["o", 2], ["g", 2]]
```

Создание нумераторов с enum_for

Методу enum_for можно передать название метода-итератора, который будет предоставлять значения последовательности

```
enum = "cat".enum_for(:each_char)
enum.to_a # => ["c", "a", "t"]
```

Если итератор ожидает аргументов, то их следует передать после имени метода

```
enum_in_threes = (1..7).enum_for(:each_slice, 3)
enum_in_threes.to_a # => [[1, 2, 3], [4, 5, 6],
                    # [7]]
```

Создание произвольных нумераторов

Нумераторы могут быть созданы на основе обычного блока, который предоставляет очередные значения

```
numbers = Enumerator.new do |yielder|
  number = 0
  count = 1
  loop do
    number += count
    count += 1
    yielder.yield number
  end
end
5.times { print numbers.next, " " }
puts numbers.first(5) # Доступны методы Enumerabl
```

Бесконечные последовательности

Если генерирующий блок способен предоставлять бесконечное число значений, то его надо указать “ленивым”

```
numbers = Enumerator.new do |yielder|
  number = 0
  loop do
    number += 1
    yielder.yield number
  end
end.lazy
puts numbers.all.first(10)
puts numbers.select { |val|
     val % 10 == 0 }.first(5)
puts numbers.select { |val|
     (val % 3).zero? }.first(10)
```

Создание собственного нумератора

Хорошей практикой при создании собственного итератора является возвращение нумератора в случае, когда блок не ассоциирован с данным методом

```
def iterator
  return enum_for(:iterator) unless block_given?
  ...
end
```

В результате ваш собственный итератор можно будет использовать как встроенные итераторы: либо в форме ассоциации с болоком, либо в форме получения нумератора

```
some = Some.new
some.iterator { ... }
numer = some.iterator
numer.each { ... }
```

Блоки для описания транзакций

Зачастую необходимо выполнять связные действия, например открытый файл обязательно должне быть закрыт

```
class File
  def self.open_and_process(*args)
    f = File.open(*args)
    yield f
    f.close()
  end
end

File.open_and_process("testfile", "r") do |file|
  while line = file.gets
    puts line
  end
end
```

Интересные моменты из примера

- Методы, начинающиеся со слова `self`, относятся к классу, а не к объекту класса (“статические”)
- `*args` в аргументах метода `open_and_process` собирает все аргументы в массив `args`
- `*args` в вызове метода `open` раскрывает содержимое массива и записывает их как аргументы метода
- Есть также нотация `**opts` для обработки именованных аргументов
- Вы можете открыть существующие классы и добавить в них методы. Данная техника на настоящий момент не приветствуется, так как классы - глобальные переменные. Используйте наследование, если это необходимо.

Метод `File.open`

- Данный метод уже реализует необходимую функциональность, вы можете ассоциировать с ним блок
- Метод `block_given?` проверяет наличие блока и позволяет реализовать альтернативное поведение

```
if block_given?
  result = yield file
file.close
```

Данная техника применяется в итераторах Array, Hash, Enumerable и т.д. для обработки ситуации работы метода с блоком и без него. Если вы не ассоциировали блок с итератором, то он вернёт нумератор

Блоки могут быть объектами

Блоки похожи на анонимные методы, однако с ними можно общаться как с объектами: сохранять в переменные...

```
class ProcExample
  def pass_in_block(&action)
    @stored_proc = action
  end
  def use_proc(parameter)
    @stored_proc.call(parameter)
  end
end
eg = ProcExample.new
eg.pass_in_block do |param|
  puts "The parameter is  #{param}"
end
eg.use_proc(99)
```

Создание блоков-объектов

Блоки представлены классом Proc

```
reach = Proc.new do |param|
  puts "You called #{param}"
end
reach.call(42) # => You called 42
reach.call('scar') => You called scar
```

Объекты можно вернуть из методов

```
def create_block_object(&block)
  block
end
reach = create_block_object do |param|
  puts "You called #{param}"
end
```

Лямбда-блоки

Блоки можно создавать с помощью метода `lambda`

```
bo = lambda do |param1, param2|
  puts "You called me with #{param1}"
end
bo.call(42, 'reason')
```

Или использовать краткий синтаксис `->`

```
bo = -> (param1, param2) do
  puts "You called me with #{param1}"
end
bo.call('dog', 'barks')
```

Отличие лямбд от блоков

- При вызове лямбды проверяется количество аргументов
- При вызове блока аргументы обрабатываются так
  - Если блок был вызван с 1 аргументом - массивом, тогда его значения становятся значением параметров блока
  - Если блоку было передано меньше аргументов, чем нужно, то оставшиеся аргументы будут равны `nil`
  - Если блоку было передано больше аргументов, чем нужно, то оставшиеся параметры будут отброшены

вывод Используйте лямбды, т.к. они предоставляют простую модель использования и гарантии

- Внутри тела лямбды ключевое слово `return` выходит из лямбды, а не из метода, её вызвавшую.
- Внутри тела блока `return` приведёт к выбросу исключения, если будет вызвано вне связанного метода

вывод Не используйте ключевое слово `return` внутри блоков

Вызов лямбд и блоков

Лямбды и блоки определены в едином классе `Proc`

Для вызова блока он предоставляет следующие методы:

```
action = -> (param) { ... }
```

- `#call(params)`: `action.call(42)`
- `#.(params)`: `action.(42)`
- `#[params]`: `action.[42]`
- `#yield(params)`: `action.yield(42)`

Все формы равносильны между собой. Рекомендуется использовать `#call()`

Замыкания в блоках

Замыкание - возможность доступа к переменным, объявлённых вне блока

```
def n_times(thing)
  lambda {|n| thing * n }
end
p1 = n_times(23)
p1.call(3) # => 69
p1.call(2) # => 46
```

- Аргумент `thing` метода `n_times` находится в области видимости выражений блока
- При вызове лямбды происходит обращение к `thing`
- Вы можете изменять такие переменные, но осторожно

Генераторы на основе лямбд

Можно реализовать простой генератор следующим образом

```ruby
def power_proc_generator
  value = 1
  lambda { value += value }
end
power_proc = power_proc_generator
puts power_proc.call # => 2
puts power_proc.call # => 4
```

Синтаксис для создания лямбд

Современным способом описания коротких лямбд является

```
-> params { ... }
proc1 = -> arg {puts "In proc1 with #{arg}"}
```

- Список аргументов выносится перед телом блока
- Ключевое слово `lambda` длиннее `->`

```
def my_while(cond, &body)
  while cond.call
    body.call
  end
end
a = 0
my_while -> { a < 3 } do
  puts a
  a += 1
end
```

Преобразование объектов в блоки

Любой класс, реализующий метод `#to_proc` может быть преобразован в блок с помощью оператора &

```
class Greater
  def initialize(greating)
    @greating = greating
  end
  def to_proc
    proc {|name| "#{@greating}, #{name}!" }
  end
end

hi = Greater.new("Hi")
hey = Greater.new("Hey")
["Bob", "Jane"].map(&hi)  #=> ["Hi, Bob!", "Hi, J
["Bob", "Jane"].map(&hey) #=> ["Hey, Bob!", "Hey,
```

Поддержка со стороны стандартных классов

Символы

Создаёт блок, который будет вызывать метод с именем символа

```
:to_s.to_proc.call(1)    #=> "1"
[1, 2].map(&:to_s)       #=> ["1", "2"]
(1..3).map(&:succ)       #=> [2, 3, 4]
```

Хеши

Создаёт блок, который связывает ключи с их значением

```
h = {a:1, b:2}
hash_proc = h.to_proc
hash_proc.call(:a)    #=> 1
hash_proc.call(:b)    #=> 2
hash_proc.call(:c)    #=> nil
[:a, :b, :c].map(&h)  #=> [1, 2, nil]
```

Блоки для вызова методов объектов

Метод `Object#method(symbol)` позволяет создать объект класса `Method`, который позволяет вызывать данный метод в стиле лямбды

Интерфейс класса `Method` повторяет в ключевых моментах интерфейс класса `Proc` и де-факто может использоваться как альтернатива блокам и лямбдам

```
class Thing
  def square(n)
    n*n
  end
end
thing = Thing.new
square_method  = thing.method(:square)
square_method.call(9)                  #=> 81
[ 1, 2, 3 ].map(&square_method)  #=> [1, 4, 9]
```

Композиция лямбд (с версии 2.6)

Классы `Proc` и `Method` предоставляют методы << и >>, которые позволяют создать цепочки обработки данных

```
multiplication = lambda {|x| x * x }
addition = lambda {|x| x + x }

(multiplication << addition).call(2) #=> 16
(multiplication >> addition).call(2) #=> 8
```

Ограничением такого подхода является то, чтобы формат передаваемых данных был в точности тем, что ожидают блоки

```
result = multiplication.call(2).then do |mult|
  addition.call(mult)
end
puts result #=> 8
```

Каррирование блоков

Объекты классов `Proc` и `Method` поддерживают метод `curry`, который позволяет создавать блоки с предустановленными аргументами

```
summator = lambda { |x, y, z| x + y + z }
summator.curry.call(1).call(2).call(3) #=> 6

apply_math = -> (fn, a, b) { a.send(fn), b }
add = apply_math.curry.(:+)
add.(1, 2) # => 3
increment = add.curry.(1)
increment.(1) # => 2
increment.(5) # => 6
```