Итераторы в Ruby

Итераторы в Ruby

В Ruby практически не используются циклы, вся современная экосистема языка и приложения выстроены при помощи итераторов блоков.

1. Циклы Ruby

В Ruby как и в любом языке программирования имеются операторы цикла, цель которых, выполнение повторяющихся участков кода.

Например, для перебора элементов массива `[1, 2, 3, 4, 5]` можно воспользоваться циклом `for`:

```
for x in [1, 2, 3, 4, 5] do
  puts x
end
1
2
3
```

```
4
5
```

Цикл пробегает значения от начала массива до последнего элемента, помещая на каждой итерации текущее значение из массива в переменную `x`. Когда значения в цикле заканчиваются, цикл завершает работу.

Еще один оператор для выполнения циклов — это `while`. Например, следующий код выводит значения от 1 до 5:

```
i = 1
while i <= 5 do
  puts i
  i += 1
end
1
2
3
4
5
```

Цикл начинается с ключевого слова `while`, после которого размещено условие. Выражение с условием возвращает либо `true` (истина), либо `false` (ложь). Пока условие возвращает `true`, цикл выполняет выражения между ключевыми словами `do` и `end`. Как только условие возвращает `false`, а это происходит когда переменная `i` получает значение 6, он прекращает работу.

Кроме оператора `while` существует оператор `until`, который противоположен `while`, так как выполняет блок до тех пор, пока условие ложно:

```
i = 1
until i > 5 do
  puts i
  i += 1
end
1
2
3
4
5
```

2. Почему Ruby-исты не используют циклы?

Перечисленные выше операторы трудно обнаружить в Ruby-коде, по крайней мере в коде конечных приложений. Вместо них, рубисты часто прибегают к итераторам, специальным методам, которые позволяют обходить коллекции.

Рассмотрим типичный итератор `each`:

```
[1, 2, 3, 4, 5].each do |i|
  puts i
end
1
2
3
4
5
```

Метод `each` является именно методом объекта `[1, 2, 3, 4, 5]`, а не специальной конструкцией языка, чуть позже мы попробуем писать свои собственные методы-итераторы.

Конструкция между `do` и `end` называется блоком, и в отличие от одноименных конструкций в циклах имеет собственную область видимости, не может быть сокращена за счет удаление `do` (как в случае циклов), однако, может быть преобразована в краткую форму за счет использования фигурных скобок:

```
[1, 2, 3, 4, 5].each { |i| puts i }
1
2
3
4
5
```

К фигурным скобках обычно прибегают, когда блок состоит из одного выражения, в случае нескольких выражений используют полную форму блока, с использованием ключевых слов `do` и `end`.

Итератор `each` применим, не только для массивов, но и для хэшей:

```
{a: 'b', c: 'd'}.each { |key, value| puts "#{key}: #{value}" }
a: b
c: d
```

Здесь блок принимает вместо одного, два параметра, `key` под ключ (`:a`, `:c`), `value` – под значения (`'b'`, `'d'`).

Итераторы необязательно обслуживают коллекции, например, итератор `times` применяется к числам позволяет выполнить цикл указанное количество раз:

```
5.times { |i| puts i }
0
1
2
3
4
```

Причем для вещественного числа метод `times` уже не сработает:

```
(5.0).times { |i| puts i }
NoMethodError: undefined method `times' for 5.0:Float
```

Для итерирования от одного числа к другому можно воспользоваться методом `upto`:

```
5.upto(10) { |i| puts i }
5
6
7
8
9
10
```

Метод `downto` позволяет наоборот пробегать числа с шагом минус один:

```
10.downto(5) { |i| puts i }
10
9
8
7
6
5
```

Главное знать, какой итератор можно применять с текущим объектом.

3. Как определить какой из итераторов можно использовать?

Как видно из предыдущего раздела итераторы можно применять не ко всем объектам. При поиске подходящего итератора в документации следует помнить, что Ruby является полностью объектно-ориентированным языком. Это упрощает работу с документацией. Так как любая практически любая конструкция языка, за исключением небольшого количества ключевых слов (тех операторов цикла `for`, `while`, `until`), является либо объектом, либо методом объекта.

Например даже обычное сложение

```
5 + 2
7
```

следует рассматривать как объектно-ориентированную операцию, заключающуюся в вызове метода с именем + в отношении объекта 5, с передачей методу аргумента 2

```
5.+(2)
7
```

Разумеется на практике отдается предпочтение более привычной арифметической форме записи 5 + 2, хотя в случае Ruby вторая форма записи более каноническая и более точно отражает, что стоит за реальным вызовом.

Почти каждый объект, с которым приходится иметь дело в Ruby, имеет метод `methods`, который возвращает список методов объекта. Например:

```
5.methods
[:%, :&, :*, :+, :-, :/, :<, :>, :^, :|, :~, :-@, :**, :<=>, :<<, :>>, :<=,
:>=, :==, :===, :[],
 :inspect, :size, :succ, :to_s, :to_f, :div, :divmod, :fdiv, :modulo, :abs,
:magnitude, :zero?,
 :odd?, :even?, :bit_length, :to_int, :to_i, :next, :upto, :chr, :ord,
:integer?, :floor, :ceil,
 :round, :truncate, :downto, :times, :pred, :to_r, :numerator, :denominator,
:rationalize, :gcd,
 :lcm, :gcdlcm, :+@, :eql?, :singleton_method_added, :coerce, :i, :remainder,
:real?, :nonzero?,
 :step, :positive?, :negative?, :quo, :arg, :rectangular, :rect, :polar,
:real, :imaginary, :imag,
 :abs2, :angle, :phase, :conjugate, :conj, :to_c, :between?, :iterator,
:instance_of?, :public_send,
 :instance_variable_get, :instance_variable_set, :instance_variable_defined?,
 :remove_instance_variable, :private_methods, :kind_of?, :instance_variables,
:tap, :is_a?, :extend,
```

```
 :define_singleton_method, :to_enum, :enum_for, :=~, :!~, :respond_to?,
:freeze, :display, :send,
 :object_id, :method, :public_method, :singleton_method, :nil?, :hash,
:class, :singleton_class,
 :clone, :dup, :itself, :taint, :tainted?, :untaint, :untrust, :trust,
:untrusted?, :methods,
 :protected_methods, :frozen?, :public_methods, :singleton_methods, :!, :!=,
:__send__, :equal?,
 :instance_eval, :instance_exec, :__id__]
```

Полученный массив можно сортировать

```
5.methods.sort
[:!, :!=, :!~, :%, :&, :*, :**, :+, :+@, :-, :-@, :/, :<, :<<, :<=, :<=>,
:==, :===, :=~, :>, :>=,
 :>>, :[], :^, :__id__, :__send__, :abs, :abs2, :angle, :arg, :between?,
:bit_length, :ceil, :chr,
 :class, :clone, :coerce, :conj, :conjugate, :define_singleton_method,
:denominator, :display, :div,
 :divmod, :downto, :dup, :enum_for, :eql?, :equal?, :even?, :extend, :fdiv,
:floor, :freeze, :frozen?,
 :gcd, :gcdlcm, :hash, :i, :imag, :imaginary, :inspect, :instance_eval,
:instance_exec, :instance_of?,
 :instance_variable_defined?, :instance_variable_get, :instance_variable_set,
:instance_variables,
 :integer?, :is_a?, :iterator, :itself, :kind_of?, :lcm, :magnitude, :method,
:methods, :modulo,
 :negative?, :next, :nil?, :nonzero?, :numerator, :object_id, :odd?, :ord,
:phase, :polar, :positive?,
 :pred, :private_methods, :protected_methods, :public_method,
:public_methods, :public_send, :quo,
 :rationalize, :real, :real?, :rect, :rectangular, :remainder,
:remove_instance_variable, :respond_to?,
 :round, :send, :singleton_class, :singleton_method, :singleton_method_added,
:singleton_methods,
 :size, :step, :succ, :taint, :tainted?, :tap, :times, :to_c, :to_enum,
:to_f, :to_i, :to_int, :to_r,
 :to_s, :truncate, :trust, :untaint, :untrust, :untrusted?, :upto, :zero?,
:|, :~]
```

Или отфильтровать при помощи метода `grep`

```
5.methods.grep :downto
[:downto]
```

Метод `grep` допускает использование регулярных выражений, например, следующий вызов вернет список методов, начинающихся с символа `d`

```
5.methods.grep /^d.*/
[:div, :divmod, :downto, :denominator, :define_singleton_method, :display,
:dup]
```

Кроме того, всегда можно запросить у текущего объекта его класс при помощи одноименного метода `class`:

```
5.class
Fixnum
```

Получив имя класса, в документации всегда можно уточнить список его методов.

4. Итераторы коллекций

Так как в Ruby практически все является объектом, сам язык и его библиотеки предоставляют большое количество предопределенных объектов. Многие из них содержат итераторы. Особенно много итераторов содержат объекты коллекций, так как это основной и предпочтительный способ для манипулированием содержимым коллекции.

Среди итераторов выделяют несколько основополагающих, знать которые должен любой Ruby-разработчик:

- each
- map (синоним collect)
- select (синоним find_all)
- reject (синоним delete_if)
- reduce (синоним inject)
- tap

Давайте посмотрим, как они работают. Итератор `map` очень похож на `each`, однако, если `each` всегда возвращает исходный массив, то `map` возвращает массив, состоящий из такого же количества элементов, что и исходный, однако, в качестве элементов используется содержимое, вычисленное в блоке, например:

```
[1, 2, 3, 4, 5].each { |x| x + 1 }
[1, 2, 3, 4, 5]

[1, 2, 3, 4, 5].map { |x| x + 1 }
[2, 3, 4, 5, 6]
```

или вообще так

```
[1, 2, 3, 4, 5].map { |x| 1 }
[1, 1, 1, 1, 1]
```

У `map` есть синоним `collect`, однако рубисты в массе своей предпочитают более короткий в написании `map`. Это вообще общее правило, чем короче конструкция в написании тем она более популярна.

Таким образом map возвращает массив с тем же количеством элементов, что и исходный, но в качестве элементов которого выступает содержимое блоков.

Для того, чтобы отфильтровать содержимое массива используется пара итераторов `select` и `reject`. Блоки в этих итераторах должны возвращать `true` или `false`, `select` – возвращает те элементы коллекции, для которых блок возвращает `true`, а `reject` – `false`. Давай посмотрим это на примере уже рассмотренного выше примера, отбирая четные номера.

```
[1, 2, 3, 4, 5].select { |x| x.even? }
[2, 4]
```

Чтобы было понятнее, давайте, заменим `select` на `map`:

```
[1, 2, 3, 4, 5].map { |x| x.even? }
[false, true, false, true, false]
```

Т.е. `select` отбирает второй и четвертый элемент коллекции. Итератор `reject`, наоборот, выбирает `false`:

```
[1, 2, 3, 4, 5].reject { |x| x.even? }
[1, 3, 5]
```

Когда в блоке вызывается единственный метод, как в примере выше, можно воспользоваться сокращенной формой:

```
[1, 2, 3, 4, 5].reject(&:even?)
[1, 3, 5]
```

Пока будем избегать такой записи, однако, в реальных приложениях она часто встречается, в силу того, что короче полного варианта.

Итераторы можно комбинировать друг с другом, например, если требуется извлечь квадраты нечетных чисел, можно воспользоваться следующей комбинацией методов `map` и `select`:

```
[1, 2, 3, 4, 5].reject { |x| x.even? }.map { |x| x * x}
[1, 9, 25]
```

Еще один итератор, который обязательно нужно рассмотреть — это `reduce`, суть его это накопление результатов, по мере обхода коллекции. Например, перемножив элементы коллекции друг на друга можно получить факториал:

```
[1, 2, 3, 4, 5].reduce { |fact, x| fact * x }
120
```

Можем перепроверить, перемножив элементы коллекции без итератора

```
1 * 2 * 3 * 4 * 5
120
```

Итератор `reduce`, на первый взгляд может показаться магическим: непонятно как инициализируется переменная `fact` и как вообще она ведет себя в итераторе. Дело в том, что в такой форме переменная-аккумулятор `fact` инициализируется первым элементом коллекции. На каждой новой итерации, переменной присваивается результат вычисления блока. В качестве результата итератор возвращает значение переменной `fact`. Можно и явно инициировать переменную-аккумулятор, если передать итератору `reduce` аргумент:

```
[1, 2, 3, 4, 5].reduce(2) { |fact, x| fact * x }
240
```

В случае `inject` тоже есть сокращенные варианты и их тоже стоит иметь в виду, когда вы читаете код Ruby-проектов:

```
[1, 2, 3, 4, 5].reduce(:*)
```

Итератор `tap` предназначен для выполнения побочного действия в цепочке других итераторов. Для того чтобы узнать промежуточное значение, которое возвращает итератор `reject` в следующем примере, можно воспользоваться итератором `tap`:

```
[1, 2, 3, 4, 5].reject { |x| x.even? }.tap{ |x| p x }.map { |x| x * x}
[1, 3, 5]
=> [1, 9, 25]
```

5. Антипатерны

Очень часто, особенно при первом знакомстве с итератором начинающие разработчики используют один итератор. Чаще всего итератор `each`, в результате получается код, который можно сделать короче

```
arr = []
[1, 2, 3, 4, 5].each do |x|
  arr << x * x
end
p arr
[1, 4, 9, 16, 25]
```

В случае когда пустой массив, заполняется внутри какого либо итератора, как правило, есть более подходящий итератор или комбинация итераторов, позволяющих сократить код, при помощи итератора `map`, `select` или `reject`:

```
arr = [1, 2, 3, 4, 5].map { |x| x * x }
p arr
[1, 4, 9, 16, 25]
```

Если внутри метода изменению подвергается хэш и в конце метода возникает явно вернуть переменную хэша

```
def change_hast(params)
  params[:page] = 1
  params
end
```

Как правило можно сократить код, воспользовавшись итератором tap, который всегда возвращает значение своего объекта

```
def change_hast(params)
  params.tap{ |p| p[:page] = 1 }
end
```

6. Итераторы изнутри: оператор yield и модуль Enumerable

Для того, чтобы получить возможность создавать свои собственные итераторы, потребуется задействовать оператора `yield`, который позволяет передать управление из текущего метода во внешний код:

```
def iterator
  yield 'hello'
  yield 'world'
end
```

В методе `iterator` оператор `yield` вызывается два раза. Это приводит к тому, что во-первых метод не работает без блока:

```
iterator
LocalJumpError: no block given (yield)
```

Во-вторых, если блок методу передается, он выполняется ровно два раза, по количеству вызовов оператора `yield`:

```
iterator { |word| puts word }
hello
world
```

Оператор `yield` может принимать переменное количество аргументов.

```
def iterator2
  yield 'hello', 'world'
end
```

Сколько аргументов передано `yield`, такое количество параметров должно быть в вызове блока:

```
iterator2 { |hello, word| puts “#{hello} #{word}” }
hello world
```

Таким образом, для создания собственного итератора, например, того же `each` нужно просто вызвать оператор `yield` для каждого из элементов коллекции. Создадим для этого специальный класс `Coll`.

```
class Coll
  def initialize(coll = [])
   @arr = coll
  end

  def each
    for x in @arr do
      yield x
    end
```

```
    end
  end
```

Обычно в языках программирования для описания свойств и поведения переменных используется тип. Класс выполняет схожую функцию, предоставляя код описания объектов. Метод `initialize` является конструктором и вызывается до всех других методов объекта, во время создания объекта методом `new`.

Единственная задача конструктора из примера, инициировать переменную объекта или как говорят инстанс-переменную `@arr`. Такие переменные – это переменные на уровне объекта, фактически это более короткий вариант `self.arr`. Сейчас не будем подробно останавливаться на объектно-ориентированной модели Ruby, так как язык полностью объектно-ориентированный и его объектно-ориентированные возможности мягко говоря богаты.

Помимо конструктора в классе определен метод `each`, который ведет себя как итератор, т.е. последовательно пробегает от первого элемента массива до последнего, передавая управление во внешний блок при помощи оператора `yield`. Рассмотрим как работает класс `Coll`, для этого создадим его объект.

```
obj = Coll.new([1, 2, 3, 4, 5])
```

Теперь можно воспользоваться итератором `each`:

```
obj.each { |x| puts x + 1 }
2
3
4
5
6
```

Воспользоваться итератором map уже не получится, так как он просто не объявлен в классе `Coll`.

```
obj.map { |x| x + 1 }
NoMethodError: undefined method `map' for #<Coll:0x007feaf383b1a8 @arr=[1, 2,
3, 4, 5]>
```

По идее нам нужно объявлять каждый итератор, который может потребоваться в дальнейшей работе. Если мы сейчас попробуем обратиться к `map`, то не сможем обнаружить его. К счастью, стандартная библиотека Ruby облегчает нам работу, достаточно реализовать один единственный метод each и подключить к классу модуль `Enumerable` и все базовые итераторы, `map`, `select`, `reject` и т.д. будут построены из метода `each` автоматически.

```
class Coll
  include Enumerable

  def initialize(coll = [])
     @arr = coll
  end

  def each
      for x in @arr do
      yield x
      end
  end
end
obj = Coll.new([1, 2, 3, 4, 5])
obj.map { |x| x + 1 }
[2, 3, 4, 5, 6]
```

Когда мы говорим о создании собственных итераторов, следует упомянуть метод `block_given?`, который проверяет передан ли текущему методу блок или нет. Это позволяет вместо завершении метода выбросом исключения совершить какое-то разумное действие. Например, вместо передачи управления в блок, просто вернуть коллекцию в виде массива. Вот здесь представлен метод `block`, который при помощи метода `block_given?` и оператора `if` позволяет обработать ситуацию передачи блока.

```
def block(arr = [])
  if block_given?
    arr.each { |x| yield x }
  else
    arr
  end
end
```

7. Реальный пример обход дерева каталога

Рассмотрим более сложный пример, пусть у нас имеется дерево с директориями и названиями файлов. Нам требуется рекурсивно обойти дерево, отобрав только файлы. Чтобы было интереснее пусть у нас сложилась ситуация, когда мы не можем воспользоваться стандартными средствами, классами `Dir` или `Find` и вообще дерево сформировано в виде массива, в котором элементы могут быть либо строками `String`, либо хэшами в которых ключи названия папок, а массивы — список файлов в папке или опять же подкаталоги.

```
tree = [
  'index.rb',
  {
    'src' => [
      'file01.rb',
      'file02.rb',
      'file03.rb'
    ]
  },
  {
    'doc' => [
      'file01.md',
      'file02.md',
      'file03.md',
      {
        'details' => [
          'index.md',
          'arch.md'
        ]
      }
    ]
  }
]
```

Обход дерева при помощи итератора each не очень интересен, так как он затрагивает только самый верхний уровень массива.

```
tree.each { |x| puts x }
index.rb
{"src"=>["file01.rb", "file02.rb", "file03.rb"]}
{"doc"=>["file01.md", "file02.md", "file03.md", {"details"=>["index.md",
"arch.md"]}]}
```

Однако, при помощи метода `is_a?` мы можем определить тип объекта, например, следующий код позволяет определить, что перед нами "файл" или "папка".

```
tree.each do |x|
  puts 'Dir' if x.is_a? Hash
  puts 'File' if x.is_a? String
end
```

Таким образом на верхнем уровне, можно различать файлы и папки. Если перед нами файл, можно передать его во внешний блок при помощи `yield`, если перед нами папка – можно повторить операцию сканирования. Таким образом, мы приходим к тому, что итератор должен быть рекурсивным, т.е. вызывать самого себя. Пробежались по элементам коллекции: для строк вернули блок, для массивов опять вызывали это же самый метод и так, до тех пор, пока не доберемся до самых дальних уголков дерева.

Сложность заключается в том, что при вызове оператора `yield` управление передается лишь непосредственно в точку вызова.

```
def walk(arr = [], &proc)
  arr.each do |el|
    proc.call(el) if el.is_a? String
    el.each { |_dir, files| walk(files, &proc) } if el.is_a? Hash
  end
end
```

Т.е. во второй строке each-блока, там где итератор `walk` вызывается для очередного списка файлов `files` необходимо вызывать блок. Для того, чтобы этого не делать, а передать блок наружу, пригодятся альтернативный синтаксис передачи блоков. Для этого последний параметр метода `walk` предваряется амперсандом `&proc`, в результате внутри функции блок становится именованным Proc-объект.

Во-первых его можно передать в другие методы или вглубь рекурсивного вызова, как в случае walk-итератора.

Во-вторых вместо оператора `yield` можно использовать proc-объект, вызвав у него метод `call`, т.е.

```
def walk(arr = [], &proc)
  arr.each do |el|
    proc.call(el) if el.is_a? String
    el.each { |_dir, files| walk(files, &proc) } if el.is_a? Hash
  end
end
```

Теперь можно разобрать дерево

```
walk(tree) { |file| puts file }
index.rb
```

```
file01.rb
file02.rb
file03.rb
file01.md
file02.md
file03.md
index.md
arch.md
```

Releases

No releases published

Packages

No packages published

Languages

- Ruby 100.0%