Ruby Enumerator

제 76회 RORLab

2014. 12. 09.

@nacyo_t

TOC

- 원시적 반복문
- 반복을 바라보는 다른 시선
- 루비의 반복 메서드
  - .each / .map / .select
- Enumerator
- Lazy Enumerator
- 결론

원시적 반복문

c의 for 반목문

```c
#include <stdio.h>
main(){
   int i;
   for(i = 0; i <= 3; i++){
      printf("%d", i)
   }
}
```

반복문이란?

- GOTO
  - 특정 시점에, 특정 조건을 만족하면 코드를 실행
  - 다시 조건을 평가하고 반복문을 빠져나가거나,
    - 맨 처음으로 되돌아감
- 반복이라는 개념이 가지는 본질적 원시성
- 충분히 추상화되어있지 않은 개념

GOTO(1)

```
10 i = 1
20 print i
30 i = i + 1
40 if i <= 3 then goto 20
```

GOTO(2)

```
i = 1
print i
i = i + 1
print i
i = i + 1
print i
```

루비의 while 반복문

```ruby
i = 1
while i <= 3 do
  puts i
  i += 1
end

# 1
# 2
# 3
# => nil
```

루비의 for..in 반복문

```ruby
for i in (1..3) do
  puts i
end

# 1
# 2
# 3
# => 1..3
```

루비의 원시적 반복 키워드

- for
- while

사용해본 적 없음

반복을 바라보는 다른 시선

c의 for 반목문

```
#include <stdio.h>
main(){
   int i;
   for(i = 1; i <= 5; i++){
      printf("%d", i ** i)
   }
}
```

R 벡터 연산

```
c(1, 2, 3, 4, 5)
# [1] 1 2 3 4 5
c(1, 2, 3, 4 ,5) ** c(1, 2, 3, 4, 5)
# [1]    1    4   27  256 3125
```

무엇이 다를까?

- 내부적으로는 반복적인 계산을 통해서 처리
- 문법적으로는 반복 개념이 보이지 않음
- 반복개념이 적절히 추상화됨

루비의 반복문

좀 더 루비 다운 접근법

```
i = 1
loop do
  puts i
  i += 1
  break if i > 3
end
```

왜 이 코드가 더 루비다울까?

- loop는 루비의 키워드가 아님
  - Kernel 클래스에 정의된 method
  - Kernel에 정의되어 있어 키워드처럼 보임
- 메서드에 블록을 결합한 호출
  - 루비에서는 매우 일반적인 메서드 호출 방법

loop 메서드와 루비 블록

```ruby
block = Proc.new do
  puts i
  i += 1
  break if i > 3
end

Kernel.loop(&block)

# 1
# 2
# 3
# LocalJumpError: break from proc-closure
# ...
```

루비의 반복문

- 컬렉션 객체들은 반복자 메서드들을 포함하고 있음
- 컬렉션에 대한 적절히 추상화된 메서드를 제공

.each

첫번째 반복자(Iterator) - each

```ruby
[1, 2, 3].each do |i|
  puts i
end
# 1
# 2
# 3
# => [1, 2, 3]
```

each가 하는 일

- 컬렉션의 요소들을 한 번씩 반복(Iterate)하면서
- each 메서드에 결합된 블록을 실행

반복자의 핵심 포인트 1 - 블록

블록은 Proc 객체

- 블록을 명시적으로 넘겨받기

```
def receive_proc(&code_block)
  puts code_block.class
  return true
end

receive_proc{ puts 'I'm Block' }

# Proc
# => true
```

Proc 객체

- 실행가능한 코드 덩어리
- Proc 객체는 익명 함수가 아님
    - 인자의 개수에 무관심함
    - return이나 break 키워드의 작동 방식이 다름

Proc 객체의 특징 - 정의된 인자 개수 무시

- each에서는 하나의 값만을 넘긴다
- 하지만 블록 인자 개수는 몇 개라도 작동

```
[1,2,3].each{|a|}                  # => [1,2,3]
[1,2,3].each{|a,b|}                # => [1,2,3]
[1,2,3].each{|a,b,c,d,e,f,g|} # => [1,2,3]
```

Proc 객체의 특징 - retrun의 작동 방식

- 둘러싼 함수의 return으로 작동함

```
def proc_test_proc
  loop do
    [1, 2, 3].each { return 0 }
  end
end

proc_test # => 0
```

진짜 익명 함수 lambda

루비의 익명 함수 - lambda (1)

```ruby
[1,2,3].each(&(->(i){puts i}))
# 1
# 2
# 3
# => [1,2,3]
```

```ruby
[1,2,3].each(&(->(i, j){puts i}))
ArgumentError: wrong number of arguments (1 for 2)
# 인자 갯수가 맞지 않아서 에러
```

루비의 익명 함수 - lambda (2)

```
def proc_test_lambda
  loop do
    [1, 2, 3].each(&->(i){ return 0 })
  end
end

prot_test_lambda
# 익명 함수 내에서만 return이 되서
# 무한 루프에 빠짐
```

반복자의 핵심 포인트 2 - 반환값

반복자는 반환값을 가진다

```ruby
[1, 2, 3].each do |i|
  puts i
end
# 1
# 2
# 3
# => [1, 2, 3]
```

출력을 하지 않는 예제

```
[1, 2, 3].each do
end
# => [1, 2, 3]
```

반환되는 [1, 2, 3]의 정체?

each는 메서드

- 메서드는 반환값을 가진다
- 반환값은 필요에 따라서 재사용 가능
    - 열거자 메서드 체인이 가능한 이유

.map

반복문으로 배열 조작하기

- 반복문으로 가장 많이 하는 일 중 하나

each로 새로운 배열 만들기

```
arr = [1, 2, 3, 4, 5]
new_arr = []

arr.each {|i| new_arr.push(i**i) }
new_arr
# => [1, 4, 27, 256, 3125]
```

두번째 반복자(Iterator) - map

```ruby
[1, 2, 3, 4, 5].map{|i| i**i }
# => [1, 4, 27, 256, 3125]

# 단순한 대입문
arr = [1, 2, 3, 4, 5]
new_arr = arr.map{|i| i**i }
```

map이 하는 일

- map은 각 요소를 차례로 블록에 넘겨
- 블록의 평가 결과(반환값)로 새로운 배열을 만든다

주의사항 - 블록의 반환값과 map의 반환값

- 블록의 반환값
  - 4가 `{|i| i**i}` 블록에 넘겨지면
  - 마지막 표현식인 `i**i`가 블록의 평가 결과(반환값)
    - 여기선 256
- map의 반환값
  - 블록의 반환값들로 구성된 배열
  - 여기선 `[1, 4, 27, 256, 3125]`

.select

세번째 반복자(Iterator) - select

```
# 1에서 10까지 숫자에서 3의 배수 찾아내기
(1..10).select{|i| i % 3 == 0}
# => [3, 6, 9]
```

select가 하는 일

- 각 요소에 대해 블록을 평가하고,
- 반환값이 true인 요소들로만 구성된 배열을 반환

좀 더 자세히 들여다보기

```
(1..10).map{|i| [i, i % 3 == 0] }
# => [[1, false]
#     [2, false]
#     [3, true]
#     [4, false]
#     [5, false]
#     [6, true]
#     [7, false]
#     [8, false]
#     [9, true]
#     [10, false]]
```

select의 반대 표현 - reject

```ruby
# 1에서 10까지 숫자에서 3의 배수가 아닌 수 찾아내기
(1..10).reject{|i| i % 3 == 0}
# => [1, 2, 4, 5, 7, 8, 10]
```

더 많은 반복자 메서드

- 루비의 꽃, 열거자 Enumerable 모듈

Enumerator

내부 반복자(Internal Iterator)

- 지금까지의 이야기
- 디자인 패턴에서는 이렇게도 부름
- 실질적으로 lisp 계열 언어에서 받아들인 개념

Enumerator(열거자)

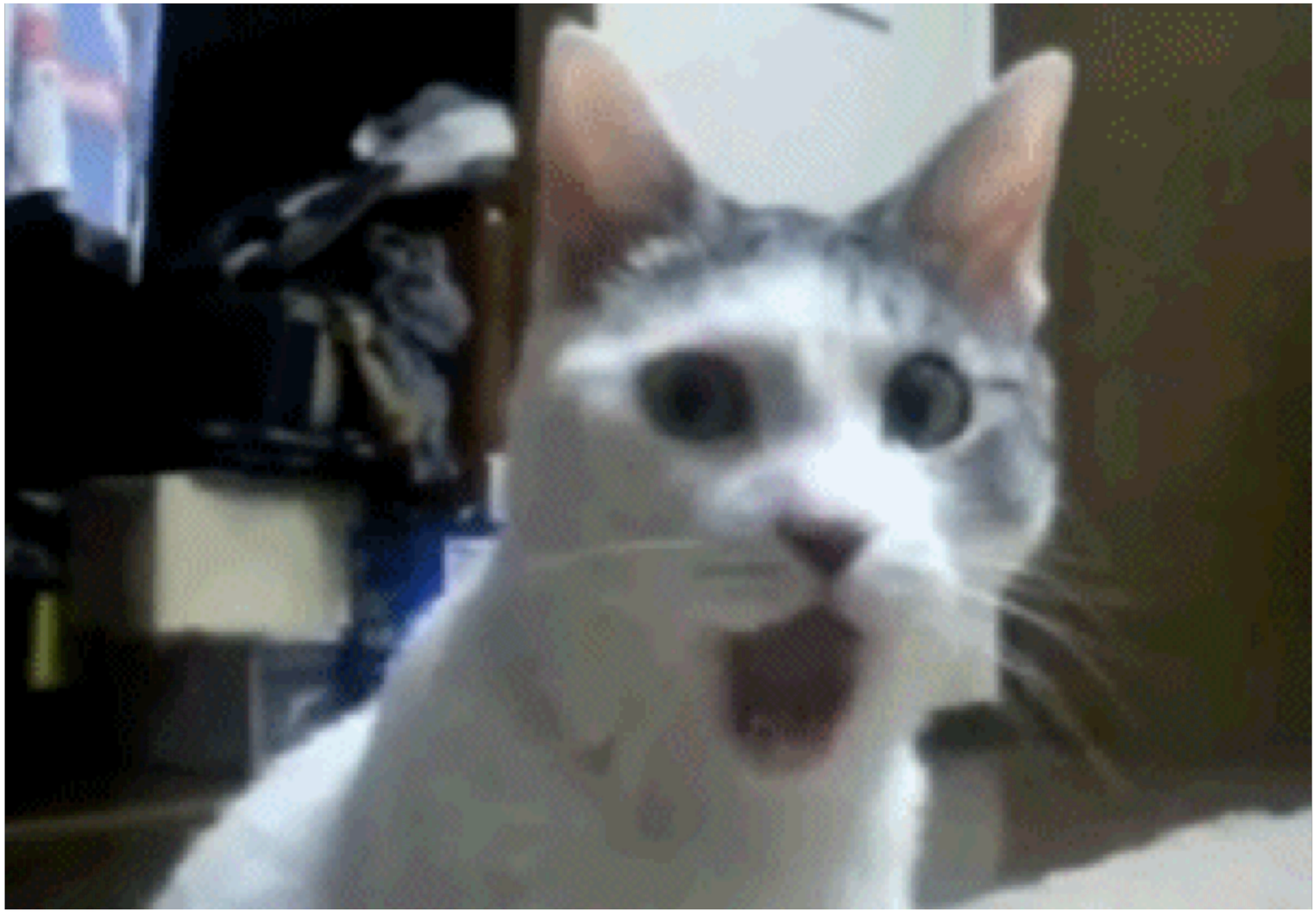
- 루비는 어떻게 내부 반복자를 구현하는가?

다시 each로...

each를 블록없이 호출해 보신 적이 있나요?

```
[1, 2, 3].each
# => #<Enumerator: ...>
```

Enumerator가 반환됐다?!



Ruby 문서에도 그렇게 적혀있음

each { |elm| block } → obj

click to toggle source

each → enum

each(*appending_args) { |elm| block } → obj

each(*appending_args) → an_enumerator

Iterates over the block according to how this Enumerator was constructed. If no block and no arguments are given, returns self.

each 재사용하기

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

Enumerator 객체로 구현하는 외부 반복자

```
enum = [1, 2, 3].each
enum.next # => 1
enum.next # => 2
enum.next # => 3
enum.next # => 4
enum.next # => 5
...
```

농담입니다

Enumerator 객체로 구현하는 외부 반복자

```
enum = [1, 2, 3].each
enum.size # => 3
enum.next # => 1
enum.next # => 2
enum.next # => 3
enum.next
# StopIteration: iteration reached an end
```

Loop로 사용하는 외부 반복자

```
enum = [1, 2, 3].each
loop do
  puts enum.next
end

# 1
# 2
# 3
# => nil
# StopIteration이 발생하면 loop가 종료됨
```

외부 반복자의 장점

- 반복 작업을 명시적으로 컨트롤 할 수 있음
- 다수의 컬렉션들을 복합적으로 제어할 수 있음
- 더 자세한 활용은 상상에 맡깁니다

Lazy Enumerator

더 이상 반복자는 배열이 아니다

- 이제 Collection을 Iterate한다는 말과는 안녕
- 반복자는 Enumerator이다.

잠깐 중학교 수학 이야기...

- 원소 나열법
  - {1, 2, 3, 4, 5}
  - {1, 2, … , 100}
- 조건 제시법
  - {x| x는 2의 배수}
  - {x| x는 3으로 나눠지면서 5의 배수}

컬렉션은 원소 나열법으로 표현

```
# {1, 2, 3, 4, 5}
[1, 2, 3, 4, 5]
# {1, 2, ... , 100}
(1..100).to_a
```

조건 제시법 on Ruby

```
even = (1..Float::INFINITY).
  lazy.
  select{|i| i % 2 == 0}
# => <Enumerator::Lazy: ...>
```

Lazy Enumerator

```ruby
even = (1..Float::INFINITY).
  lazy.
  select{|i| i % 2 == 0}
even.size       # => nil
even.next       # => 2
even.next       # => 4
even.first(10) # => [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
even.next       # => 6
even.rewind     # => <Enumerator::Lazy: ...>
even.next       # => 2
```

컬렉션을 반복한다(x)

Enumerator는 Fiber로 구현

- Fiber(Coroutine)
- 다수의 진입점과 반환점을 가지는 서브루틴

Fiber 예제 (1) 무작위 숫자

```
# 무한히 0~99 사이의 숫자를 생성하는 Fiber
r = Fiber.new{ loop {Fiber.yield(rand) } }
r.resume # => 24
r.resume # => 68
r.resume # => 51
...
```

Fiber 예제 (2) 피보나치 수열

```
fib = Fiber.new do
  a, b = 0, 1
  loop do
    a, b = b, a + b
    Fiber.yield(a)
  end
end

fib.resume # => 1
fib.resume # => 1
fib.resume # => 2
fib.resume # => 3
fib.resume # => 5
...
```

Ruby 반복문의 정체

열거자를 반복한다

Enumerator를 Iterate한다

감사합니다 ;)

@nacyo_t