Ruby trunk - Feature #14781

Enumerator.generate

05/22/2018 10:23 AM - zverok (Victor Shepelev)

Status:	Feedback
Priority:	Normal
Assignee:	
Target version:	

Description

This is alternative proposal to Object#enumerate (#14423), which was considered by many as a good idea, but with unsure naming and too radical (Object extension). This one is less radical, and, at the same time, more powerful.

Synopsys:

- Enumerator.generate(initial, &block): produces infinite sequence where each next element is calculated by applying block to previous; initial is first sequence element;
- Enumerator.generate(&block): the same; first element of sequence is a result of calling the block with no args.

This method allows to produce enumerators replacing a lot of common while and loop cycles in the same way #each replaces for.

Examples:

With initial value

```
# Infinite sequence
p Enumerator.generate(1, &:succ).take(5)
# => [1, 2, 3, 4, 5]

# Easy Fibonacci
p Enumerator.generate([0, 1]) { |f0, f1| [f1, f0 + f1] }.take(10).map(&:first)
#=> [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

require 'date'

# Find next Tuesday
p Enumerator.generate(Date.today, &:succ).detect { |d| d.wday == 2 }
# => #<Date: 2018-05-22 ((2458261j,0s,0n),+0s,2299161j)>

# Tree navigation
# ---------------
require 'nokogiri'
require 'open-uri'

# Find some element on page, then make list of all parents
p Nokogiri::HTML(open('https://www.ruby-lang.org/en/'))
  .at('a:contains("Ruby 2.2.10 Released")')
  .yield_self { |a| Enumerator.generate(a, &:parent) }
  .take_while { |node| node.respond_to?(:parent)  }
  .map(&:name)
# => ["a", "h3", "div", "div", "div", "div", "div", "div", "body", "html"]

# Pagination
# ----------
require 'octokit'

Octokit.stargazers('rails/rails')
# ^ this method returned just an array, but have set `.last_response` to full response, with data
# and pagination. So now we can do this:
p Enumerator.generate(Octokit.last_response) { |response|
    response.rels[:next].get                          # pagination: `get` fetches next Response
  }
  .first(3)                                           # take just 3 pages of stargazers
  .flat_map(&:data)
```

```
# `data` is parsed response content (stargazers themselves)
  .map { |h| h[:login] }
# => ["wycats", "brynary", "macournoyer", "topfunky", "tomtt", "jamesgolick", ...
```

Without initial value

```
# Random search
target = 7
p Enumerator.generate { rand(10) }.take_while { |i| i != target }.to_a
# => [0, 6, 3, 5,....]

# External while condition
require 'strscan'
scanner = StringScanner.new('7+38/6')
p Enumerator.generate { scanner.scan(%r{\d+|[-+*/]}) }.slice_after { scanner.eos? }.first
# => ["7", "+", "38", "/", "6"]

# Potential message loop system:
Enumerator.generate { Message.receive }.take_while { |msg| msg != :exit }
```

Reference implementation: https://github.com/zverok/enumerator_generate

I want to **thank** all peers that participated in the discussion here, on Twitter and Reddit.

History

#1 - 05/22/2018 09:24 PM - shevegen (Robert A. Heiler)

I agree with the proposal and name.

I would like to recommend and suggest you to add it to the next ruby developer meeting for matz' to have a look at and decide. (I think most people already commented on the other linked suggestion, so I assume that the issue here will remain fairly small.)

By the way props on the examples given; it's a very clean proposal, much cleaner than any proposals I ever did myself :D, and it can be taken almost as-is for the official documentation, IMO. :)

Now of course we have to wait and see what matz and the other core devs have to say about it.

Here is the link to the latest developer meeting:

https://bugs.ruby-lang.org/issues/14769

#2 - 06/21/2018 08:02 AM - knu (Akinori MUSHA)

What about adding support for ending an iteration from a given block itself by raising StopIteration, rather than having to chain it with take_while?

#3 - 06/21/2018 08:17 AM - knu (Akinori MUSHA)

In today's developer meeting, we kind of loved the functionality, but haven't reached a conclusion about the name.

Some candidates:

- Enumerator.iterate(initial = nil) { |x| ... }
  Haskell has a similar function named iterate.

- Enumerator.from(initial) { |x| ... }
  This would sound natural when the initial value is mandatory.

#4 - 06/21/2018 08:26 AM - matz (Yukihiro Matsumoto)

- Status changed from Open to Feedback

I am not fully satisfied with the name generate since the word does not always imply sequence generation. If someone has better name proposal, I welcome.

Matz.

#5 - 06/21/2018 08:59 AM - sawa (Tsuyoshi Sawada)

I propose the following:

- Enumerator.sequence
- Enumerator.recur

#6 - 06/21/2018 10:06 AM - zverok (Victor Shepelev)

I like #sequence, too.

#7 - 06/21/2018 10:07 AM - zverok (Victor Shepelev)

Though, I should add that Enumerator.generate (seen this way, not just .generate alone) seems to clearly state "generate enumerator" :)

#8 - 06/21/2018 10:21 AM - mame (Yusuke Endoh)

zverok (Victor Shepelev) wrote:

> Though, I should add that Enumerator.generate (seen this way, not just .generate alone) seems to clearly state "generate enumerator" :)

"generate" seems too general. It looks the most typical or primitive way to create an enumerator, but it is not.

Haskell provides "iterate" function for this feature, but it resembles an iterator in Ruby.

#9 - 06/21/2018 11:14 AM - Eregon (Benoit Daloze)

I like Enumerator.generate, since it's really to generate a lazy sequence, to generate an Enumerator, from a block.

In the end it is basically as powerful as Enumerator.new, so I see no problem to have a factory/constructor-like name.

Enumerator.sequence sounds like it could be an eager sequence, and doesn't tell me the block is generating the next value, so I don't like it.

Enumerator.iterate sounds like we would iterate something, but we don't, we generate a sequence lazily.
The iteration itself is done by #each, not "Enumerator.iterate".
I think it only works well in Haskell due to their first-class functions, but even then "iterating" (repeated applications of) a function doesn't sound clear to me or map well to Ruby.

#10 - 06/21/2018 01:49 PM - matz (Yukihiro Matsumoto)

I don't like recur. Probably it came from recurrence but programmers usually think of recursive because they see recursive more often. FYI, the word recur is used in Clojure for the recursive purpose. I don't like iterate either, as Eregon (Benoit Daloze) stated.

Enumerator.generate may work because it **generates Enumerator** in a fashion different from Enumerator.new.

Matz.

#11 - 06/21/2018 02:04 PM - mame (Yusuke Endoh)

Ah, I meant iterate is not a good name for ruby. Sorry for the confusion.

#12 - 06/22/2018 02:35 AM - knu (Akinori MUSHA)

I'm not very fond of generate because it's not the only way to generate an Enumerator. There could be more to come.

#13 - 10/07/2018 12:26 PM - zverok (Victor Shepelev)

- Description updated

#14 - 10/10/2018 06:44 AM - akr (Akira Tanaka)

How about "recurrence" as method name?
It is noun, though.

#15 - 10/18/2018 02:24 PM - knu (Akinori MUSHA)

- File enumerator_from.rb added

I've been thinking about this, and I have some ideas I want to share:

- To recursively traverse ancestors of a node is one of the most typical use cases, so that should be made easy to do.
- When and how to end a sequence may vary, so there should be some flexibility in defining an end. For example, nil is not always the dead end. It could mean something; you might even want to end a sequence with an explicit nil as sentinel. Rescuing an exception and treating it as an end might not be a good option because that would make debugging hard, but StopIteration should be a good fit as a signal for an end.

- Sometimes you'd need to look back two or more steps to generate a new value (not to mention Fibonacci series), so the constructor should preferably take multiple seeds.
- Sometimes seeds are not subject of yielding; it would be handy if you could specify how many leading seeds to skip.

In the original proposal, there are some tricks needed to define an end of a sequence or to look back multiple preceding terms, so I've come up with an alternative API that builds them in as keyword options:

```
Enumerator.from(seeds, drop: 0, allow_nil: false) { |*preceding_terms| next_term }

  seeds: Array of objects to be used as seeds (required, but can be an empty array)
  drop: How many leading terms to skip
  allow_nil: True if nil should not end the enumerator
```

I wrote an experimental implementation and test cases in the attached file.

I'll be working on it further in this weekends' hackathon, so any input is appreciated!

#16 - 10/18/2018 06:30 PM - zverok (Victor Shepelev)

knu (Akinori MUSHA)
The ultimate goal for my proposal is, in fact, promoting Enumerator as a "Ruby way" for doing all-the-things with loops; not just "new useful feature".

That's why I feel really uneasy about your changes to the proposal.

drop

```
# from: `drop: 2` is part of Enumerator.from API
Enumerator.from([node], drop: 2, &:parent).map(&:name)
# generate: `drop(2)` is part of standard Enumerator API
Enumerator.generate(node, &:parent).take(6).map(&:name).drop(2)
```

allow_nil (by default false: nil stops enumeration)

```
# from:
# implicit "stop on nil" is part of Enumerator.from convention that code reader should be aware of
Enumerator.from([node], &:parent).map(&:name)
# don't stop on nil is explicit part of the API
Enumerator.from([node], allow_nil: true) { |n|
      raise StopIteration if n.nil?
      n.parent
    }.map { |n| n&.name }

# generate: "stop on nil" is explicit and obvious
Enumerator.generate(node, &:parent).take_while(&:itself).map(&:name)
# no mentioning of unnecessary "we don't need to stop on nil", no additional thinking
p Enumerator.generate(node) { |n|
      raise StopIteration if n.nil?
      n.parent
    }.map { |n| n&.name }
```

start with array (I believe 1 and 0 initial values are the MOST used cases)

```
# from: we should start from empty array, expression nothing but Enumerator.from API limitation
Enumerator.from([]) { 0 }.take(10)
# generate: no start value
Enumerator.generate { 0 }.take(10)

# from: work with one value requires not forgetting to arrayify it
Enumerator.from([1], &:succ).take(10)
# generate: just use the value
Enumerator.generate(1, &:succ).take(10)

# from: "we pass as much of previous values as initial array had" convention
Enumerator.from([0, 1]) { |i, j| i + j }.take(10)
# generate: regular value enumeration, next block receives exactly what previous returns
Enumerator.generate([0, 1]) { |i, j| [j, i + j] }.take(10).map(&:last)
# ^ yes, it will require additional trick to include 0 in final result, but I believe this is worthy sacrifice
```

The problem with "API complication" is inconsistency. Like, a newcomer may ask: Why Enumerator.from has "this handy drop: 2 initial arg", and each don't? Use cases could exist, too!

#17 - 10/19/2018 03:30 AM - knu (Akinori MUSHA)

zverok (Victor Shepelev) wrote:

knu (Akinori MUSHA)
The ultimate goal for my proposal is, in fact, promoting Enumerator as a "Ruby way" for doing all-the-things with loops; not just "new useful feature".

That's why I feel really uneasy about your changes to the proposal.

Thanks for your quick feedback, and for bringing up this issue.

drop

```
# from: `drop: 2` is part of Enumerator.from API
Enumerator.from([node], drop: 2, &:parent).map(&:name)
# generate: `drop(2)` is part of standard Enumerator API
Enumerator.generate(node, &:parent).take(6).map(&:name).drop(2)
```

I presume .take(6) is inserted by mistake, but with it or not the following map and drop methods belong to Enumerable, and are Array based operations that create an intermediate array per call. So, I consider them as Array/Enumerable API rather than Enumerator API. Creating intermediate arrays is not only a waste of memory but also against the key concept of Enumerator: to deal with an object as a stream, which may be infinite.

Adding .lazy before .drop(2) can be a cure, but then the value you get is a lazy enumerator that is incompatible with an non-lazy enumerator. For instance, Lazy#map, Lazy#select etc. return Lazy objects, so you can't always pass one to methods that expect a normal Enumerable object.

I've always thought that Lazy#eager that turns a lazy enumerator back to a non-lazy enumerator would be nice, but .lazy.map{}.eager would look messy anyway.

```
# implicit "stop on nil" is part of Enumerator.from convention that code reader should be aware of
```

I think it's good and reasonable default behavior to treat nil as an end. Taking your Octokit example, the block could be { |response| response.rels[:next]&.get } to make it go through all pages and automatically stop if nil were treated as an end. You omitted a .take_while in the example, but you'd get an error if there were less than 3 pages. You'd almost always need to either explicitly raise StopIteration in the initial block or chain .take_while/.take if there were no default end, and the choice between them is not obvious.

start with array (I believe 1 and 0 initial values are the MOST used cases)

```
# from: we should start from empty array, expression nothing but Enumerator.from API limitation
Enumerator.from([]) { 0 }.take(10)
# generate: no start value
Enumerator.generate { 0 }.take(10)
```

The limitation only came from what the word from sounds like. I picked the name from and Enumerator.from {} just didn't sound right to me, so I made the argument mandatory. You can just default the first argument to [] if it reads and writes better, possibly with a different name than from which I won't insist on.

```
# from: work with one value requires not forgetting to arrayify it
Enumerator.from([1], &:succ).take(10)
# generate: just use the value
Enumerator.generate(1, &:succ).take(10)
```

Yeah, due to our keyword arguments being pseudo ones, you can't use variable length arguments for a list of objects that might end with a hash. We'll hopefully be getting it right by Ruby 3.0.

There's much room for consideration of the name and method signature. Perhaps multiple factory methods could work better.

```
# from: "we pass as much of previous values as initial array had" convention
Enumerator.from([0, 1]) { |i, j| i + j }.take(10)
# generate: regular value enumeration, next block receives exactly what previous returns
Enumerator.generate([0, 1]) { |i, j| [j, i + j] }.take(10).map(&:last)
# ^ yes, it will require additional trick to include 0 in final result, but I believe this is worthy sacri
fice
```

The former directly generates an infinite Fibonacci sequence and that's a major difference. Taking a first few elements with .take is just for testing (assertion) purposes and not part of the use case. When solving a problem like "Find the least n such that $\sum_{k=1}^{n}$ fib(k) >= 1000", take wouldn't work optimally.

The problem with "API complication" is inconsistency. Like, a newcomer may ask: Why Enumerator.from has "this handy drop: 2 initial arg", and each don't? Use cases could exist, too!

I understand that sentiment, but there's no surprise that a factory/constructor method of a dedicated class often takes many tunables while individual instance methods do not. If people all said they need it as a generic feature, it wouldn't be a bad idea to me to consider adding something like Enumerable#skip(n) that would return an offset enumerator.

#18 - 03/22/2019 07:25 AM - knu (Akinori MUSHA)

- Subject changed from Enumerator#generate to Enumerator.generate

Files

enumerator_from.rb	3.16 KB	10/18/2018	knu (Akinori MUSHA)