Ruby master - Feature #17016

Enumerable#scan_left

07/07/2020 05:48 PM - parker (Parker Finch)

Status:	Open	
Priority:	Normal	
Assignee:		
Target version:		

Description

Proposal

Add a #scan_left method to Enumerable.

(The name "scan_left" is based on Scala's scanLeft and Haskell's scanl. It seems like "scan_left" would be a ruby-ish name for this concept, but I'm curious if there are other thoughts on naming here!)

Background

#scan_left is similar to #inject, but it accumulates the partial results that are computed. As a comparison:

```
[1, 2, 3].inject(0, &:+) => 6
[1, 2, 3].scan_left(0, &:+) => [0, 1, 3, 6]
```

Notably, the scan_left operation can be done lazily since it doesn't require processing the entire collection before computing a value.

I recently described #scan_left, and its relationship to #inject, more thoroughly in this blog post.

Reasoning

We heavily rely on the scan operation. We use an event-sourcing pattern, which means that we are scanning over individual "events" and building up the corresponding state. We rely on the history of states and need to do this lazily (we stream events because they cannot fit in memory). Thus the scan operation is much more applicable than the inject operation.

We suspect that there are many applications that could leverage the scan operation. This question would be more easily answered by #scan_left. It is a natural fit for any application that needs to store the incrementally-computed values of an #inject, and a requirement for an application that needs to use #inject while maintaining laziness.

Implementation

There is a Ruby implementation of this functionality here and an implementation in C here.

Counterarguments

Introducing a new public method is committing to maintenance going forward and expands the size of the Ruby codebase -- it should not be done lightly. I think that providing the functionality here is worth the tradeoff, but I understand any hesitation to add yet more to Ruby!

History

#1 - 07/08/2020 04:00 AM - sawa (Tsuyoshi Sawada)

What is wrong with the following?

```
[1, 2, 3].inject([0]){|a, e| a << a.last + e} # => [0, 1, 3, 6]
[1, 2, 3].each_with_object([0]){|e, a| a << a.last + e} # => [0, 1, 3, 6]
```

#2 - 07/08/2020 02:51 PM - parker (Parker Finch)

sawa (Tsuyoshi Sawada) wrote in #note-1:

> What is wrong with the following?

> ```
> [1, 2, 3].inject([0]){|a, e| a << a.last + e} # => [0, 1, 3, 6]
> [1, 2, 3].each_with_object([0]){|e, a| a << a.last + e} # => [0, 1, 3, 6]
> ```

Good question! Using #inject or #each_with_object cannot be done lazily. We need to pass a lazy enumerator *after* applying the fold-like operation. As a toy example, instead of [1, 2, 3] we can use (1..).lazy:

```
(1..).lazy.inject([0]){|a, e| a << a.last + e} # => infinite loop
(1..).lazy.each_with_object([0]){|e, a| a << a.last + e} # => infinite loop
(1..).lazy.scan_left(0, &:+) # => Lazy enumerator
```

#3 - 07/09/2020 09:33 PM - Eregon (Benoit Daloze)

The name scan seems confusing at least, since it has nothing to do with String#scan.
And *_left has no precedent in Ruby.

It seems it's basically recording the history of each block call (+ the initial value), maybe a name expressing that would be good.

Do you have real-world usages you can show?
It seems kind of niche (seems expensive if the collection is large) and can easily be replaced by each + 2 variables (if not lazy at least).

#4 - 07/10/2020 08:57 PM - parker (Parker Finch)

- File scan_left_example.rb added

> The name scan seems confusing at least, since it has nothing to do with String#scan.
> And *_left has no precedent in Ruby.
>
> It seems it's basically recording the history of each block call (+ the initial value), maybe a name expressing that would be good.

Ah sorry, I forgot to mention that the term "scan" is what this operation is typically called in functional programming (see here for a quick overview). So that's how the name is derived. I agree that the *_left suffix doesn't sound very Ruby-ish, maybe just #scan would be better. But then we *really* have a conflict with the name of String#scan, thanks for pointing that out! Any thoughts on what this could be called to disambiguate that? Perhaps #accumulate?

> Do you have real-world usages you can show?

The most significant real-world usage that I've had it for is the one I briefly described above. The "scan" operation is a very good fit when processing streams of data changes. To really concretize it, our use case is in education data. One example is that each day we see if a student is present or absent. We use a scan over that data to count the number of absences that a student has -- a fold (i.e. #inject or #reduce) would be insufficient because the stream is lazy.

To get a small, self-contained example, I wrote up a script that uses it (and shows how other implementations of the behavior could work). That is attached, let me know if there's anything else I can do to show the usefulness here!

> It seems kind of niche (seems expensive if the collection is large) and can easily be replaced by each + 2 variables (if not lazy at least).

I don't think it's actually that niche! A lot of array algorithms become easier with a scan operation, see this post for an example. And I don't think that it's particularly expensive -- it should be similar in cost to a map.

In fact, scan-like behavior (with laziness) can be implemented with map without *too* much trouble. (This is the heart of the ruby implementation.) But assigning to a variable inside of the block passed to map doesn't feel very Ruby-ish to me:

```
val = 0
collection.map { |x| val = val + x }
```

feels less natural than something like

```
collection.scan(&:+)
```

One of my favorite aspects of Ruby is how easy it is to write in a functional programming style, and including the scan operation would expand the number of use cases covered by functional methods.

#5 - 07/11/2020 05:46 AM - nobu (Nobuyoshi Nakada)

parker (Parker Finch) wrote in #note-2:

> Good question! Using #inject or #each_with_object cannot be done lazily. We need to pass a lazy enumerator *after* applying the fold-like operation. As a toy example, instead of [1, 2, 3] we can use (1..).lazy:

```
(1..).lazy.inject([0]){|a, e| a << a.last + e} # => infinite loop
(1..).lazy.each_with_object([0]){|e, a| a << a.last + e} # => infinite loop
(1..).lazy.scan_left(0, &:+) # => Lazy enumerator
```

That sounds like Enumerator::Lazy needs #inject and #each_with_object methods to me.

#6 - 07/11/2020 05:47 PM - Eregon (Benoit Daloze)

parker (Parker Finch) wrote in #note-4:

Thanks for your reply, I think it will help to discuss this issue at the dev meeting.

Maybe prefix_sum or just prefix or something like that would work?
Having sum in it is kind of confusing though as it can be any "operation" not just +-ing numbers, but it seems an official "term" for it (https://en.wikipedia.org/wiki/Prefix_sum#Scan_higher_order_function).

> But assigning to a variable inside of the block passed to map doesn't feel very Ruby-ish to me:
>
> ```
> val = 0
> collection.map { |x| val = val + x }
> ```

It's just my opinion, but I see nothing wrong with that (details: it could be val += x).
I'd even go as far as saying each_with_object is often less readable than using a captured variable.
I think "purely functional, not a single re-assigned variable" often introduces significant extra complexity, when Ruby is a language that embraces both functional and imperative programming.
And anyway each_with_object is only useful if mutating some object (typically an Array or Hash).
Again, it's just my opinion :)

#7 - 07/11/2020 05:49 PM - Eregon (Benoit Daloze)

nobu (Nobuyoshi Nakada) wrote in #note-5:

> That sounds like Enumerator::Lazy needs #inject and #each_with_object methods to me.

A lazy #inject sounds useful.

#8 - 07/12/2020 08:52 AM - nobu (Nobuyoshi Nakada)

Eregon (Benoit Daloze) wrote in #note-7:

> nobu (Nobuyoshi Nakada) wrote in #note-5:
>
> > That sounds like Enumerator::Lazy needs #inject and #each_with_object methods to me.
>
> A lazy #inject sounds useful.

It has a backward incompatibility on the return value.
I'm afraid if it's acceptable.

#9 - 07/12/2020 11:32 AM - Eregon (Benoit Daloze)

nobu (Nobuyoshi Nakada) wrote in #note-8:

> It has a backward incompatibility on the return value.
> I'm afraid if it's acceptable.

Right, inject might return anything, not necessarily an Array and so returning a "lazy Enumerator" instead will be unwanted in at least some cases.

Using the existing lazy map seems most natural to me:

```
val = 0
p (0..).lazy.map { |x| val += x }.first(4) # => [0, 1, 3, 6]

val = 0
p [0] + (1..).lazy.map { |x| val += x }.first(3) # => [0, 1, 3, 6]
```

#10 - 07/12/2020 11:51 PM - shyouhei (Shyouhei Urabe)

It might be possible to let inject be lazy if we ignore backwards compatibility. But how do we partially evaluate that lazy enumerator then?

#11 - 07/14/2020 05:50 PM - parker (Parker Finch)

shyouhei (Shyouhei Urabe) wrote in #note-10:

> It might be possible to let inject be lazy if we ignore backwards compatibility. But how do we partially evaluate that lazy enumerator then?

I think this is the crux of the issue. Because #inject *needs* to evaluate every element in order to return a meaningful value, it can't be partially evaluated. The "scan" operation allows for partial evaluation.

#12 - 07/14/2020 06:20 PM - parker (Parker Finch)

Eregon (Benoit Daloze) wrote in #note-6:

> Maybe prefix_sum or just prefix or something like that would work?
> Having sum in it is kind of confusing though as it can be any "operation" not just +-ing numbers, but it seems an official "term" for it (https://en.wikipedia.org/wiki/Prefix_sum#Scan_higher_order_function).

I agree that including "sum" in the name is confusing. I think that "prefix_sum" is just used to describe the sum operation, and if it is generalized to other operations then it is typically called "scan".

Eregon (Benoit Daloze) wrote in #note-9:

> Using the existing lazy map seems most natural to me:
>
> ```
> val = 0
> p (0..).lazy.map { |x| val += x }.first(4) # => [0, 1, 3, 6]
>
> val = 0
> p [0] + (1..).lazy.map { |x| val += x }.first(3) # => [0, 1, 3, 6]
> ```

It's definitely possible to use map! I think it is simpler to use a scan, especially if the first element needs to be included in the lazy enumerator. For example:

```
(1..).lazy.scan(0, &:+)
```

would need to be:

```
val = 0
[val].chain((1..).lazy.map { |x| val += x })
```

because the + method doesn't work with lazy enumerators.

#13 - 07/15/2020 02:32 PM - Eregon (Benoit Daloze)

For the name I think just scan would be best then.
And accept the fact it's completely unrelated to String#scan.

#14 - 07/16/2020 05:17 AM - mame (Yusuke Endoh)

Is this what you want?

```
irb(main):001:0> (1..).lazy.enum_for(:inject, 0).map {|a, b| a + b }.take(10).force
=> [1, 3, 6, 10, 15, 21, 28, 36, 45, 55]
```

#15 - 07/17/2020 02:29 PM - parker (Parker Finch)

mame (Yusuke Endoh) wrote in #note-14:

> Is this what you want?
>
> ```
> irb(main):001:0> (1..).lazy.enum_for(:inject, 0).map {|a, b| a + b }.take(10).force
> => [1, 3, 6, 10, 15, 21, 28, 36, 45, 55]
> ```

Oh interesting, I hadn't considered that approach! That is very close to the behavior of the scan operation, and might be a good way to implement it. (The only difference in behavior I see is that the first element (0) is not included in the resulting enumerator with this enum_for approach.)

Are you suggesting that we should use that approach instead of implementing a built-in scan method? Or is the example to clarify what the behavior of the scan method would be?

#16 - 07/18/2020 03:23 AM - nobu (Nobuyoshi Nakada)

parker (Parker Finch) wrote in #note-15:

> Are you suggesting that we should use that approach instead of implementing a built-in scan method? Or is the example to clarify what the behavior of the scan method would be?

As it came from the lazy inject approach, the former.

#17 - 07/20/2020 06:02 AM - matz (Yukihiro Matsumoto)

I don't see any real-world use-case for scan_left. Is there any?
In addition, the term scan does not seem to describe the behavior of keeping the past sequence of accumulation.
Although I know the name from the Haskell community, I don't agree with the name.

Matz.

#18 - 07/22/2020 05:23 PM - parker (Parker Finch)

matz (Yukihiro Matsumoto) wrote in #note-17:

> I don't see any real-world use-case for scan_left. Is there any?

I think there are real-world use cases!

Would you consider converting a history of transactions into a history of account balances a valid use-case? That can be done easily with a scan. For example, if you have transactions = [100, -200, 200] then you can find the history of account balances with transactions.scan_left(0, &:+) # => [0, 100, -100, 100].

I have described our current use case of scan_left here. We use an event-sourced architecture where we scan over a lazy stream of events, building up the state as we go. It is important for our use case that we are able to access past states in addition to the final state.

This post shows how it can make repeated subarray operations more efficient -- the example there is that if you have an array of values representing changes over time periods, then you can easily aggregate those into changes over different time periods using a scan. This post does not use scan, but instead has a workaround because scan doesn't exist:

```
sums = [0]
(1..gains.length).each do |i|
  sums[i] = sums[i - 1] + gains[i - 1]
end
```

could, if scan was introduced, be replaced with:

```
sums = gains.scan_left(0, &:+)
```

Do those use cases seem sufficient?

> In addition, the term scan does not seem to describe the behavior of keeping the past sequence of accumulation.
> Although I know the name from the Haskell community, I don't agree with the name.

I agree the name is not ideal. It is easy to get it confused with the idea of a string scanner. As you mentioned, scan is the name used by Haskell, but it is also used by Scala, Rust, and C++. So it is a widely-used term, even if it's not the best one, which makes me think that it might be good to use it.

Alternatively, what do you think about the name accumulate, which Wolfram uses? I think it gets the idea across better than scan.

Are there other names you think should be considered?

#19 - 07/22/2020 06:12 PM - RubyBugs (A Nonymous)

parker (Parker Finch) wrote in #note-18:

> Are there other names you think should be considered?

In keeping with the Ruby-ish collection methods that end with "-ect", how about

- **reflect** -- the idea is to contrast with inject, this "reflects" all intermediate states
- **project** -- the idea is that the original Enumerable is "projected" in a mathematical sense into the plane defined by the stateful function that is passed in

(just my 2 cents...)

#20 - 07/23/2020 03:12 AM - nobu (Nobuyoshi Nakada)

With https://github.com/ruby/ruby/pull/3337, you can

```
(1..).lazy.inject(0, :+).first(10) #=> [0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
```

#21 - 07/23/2020 03:39 PM - parker (Parker Finch)

nobu (Nobuyoshi Nakada) wrote in #note-20:

> With https://github.com/ruby/ruby/pull/3337, you can
>
> ```
> (1..).lazy.inject(0, :+).first(10) #=> [0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
> ```

Thank you for writing that nobu! That is exactly the behavior we need.

I'm worried about the backwards compatibility of changing the behavior of #inject on lazy enumerables. Since right now [1,2,3].lazy.inject(:+) # => 6 I think it would be breaking to change that behavior to be [1,2,3].lazy.inject(:+) # => #<Enumerator::Lazy: #<Enumerator::Lazy: [1, 2, 3]>:inject>.

I'm also worried about #inject having different behavior for strict and lazy enumerables. I don't think it would be good to have [1,2,3].inject(:+) be different than [1,2,3].lazy.inject(:+).force.

Do you think your implementation could be used for a new method, whether we call it scan or accumulate or project or something else? I think it would be good to have consistent behavior between strict and lazy enumerables.

NOTE: For anyone who is newer to this thread and might have missed it -- you can find this behavior, with some examples, implemented in Ruby here.

#22 - 07/23/2020 09:44 PM - y.annikov (Yakov Annikov)

I've created a PR with the implementation of scan and lazy.scan methods https://github.com/ruby/ruby/pull/3358
Feel free to reject it but any comments are appreciated. I got a lot of fun writing this code and looking at Ruby source code.

#23 - 07/24/2020 02:30 AM - nobu (Nobuyoshi Nakada)

RubyBugs (A Nonymous) wrote in #note-19:

> In keeping with the Ruby-ish collection methods that end with "-ect", how about
>
> - **reflect** -- the idea is to contrast with inject, this "reflects" all intermediate states
> - **project** -- the idea is that the original Enumerable is "projected" in a mathematical sense into the plane defined by the stateful function that is passed in

I'd like that “this "reflects" all intermediate states” part.

#24 - 07/25/2020 06:20 AM - duerst (Martin Dürst)

parker (Parker Finch) wrote in #note-21:

> nobu (Nobuyoshi Nakada) wrote in #note-20:
>
> > ```
> > (1..).lazy.inject(0, :+).first(10) #=> [0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
> > ```
>
> I'm worried about the backwards compatibility of changing the behavior of #inject on lazy enumerables. Since right now [1,2,3].lazy.inject(:+) # => 6 I think it would be breaking to change that behavior to be [1,2,3].lazy.inject(:+) # => #<Enumerator::Lazy: #<Enumerator::Lazy: [1, 2, 3]>:inject>.
>
> I'm also worried about #inject having different behavior for strict and lazy enumerables. I don't think it would be good to have [1,2,3].inject(:+) be different than [1,2,3].lazy.inject(:+).force.

I fully agree. Lazy variants of methods should only differ with respect to laziness, not anything else.

#25 - 07/25/2020 07:49 AM - duerst (Martin Dürst)

nobu (Nobuyoshi Nakada) wrote in #note-23:

> RubyBugs (A Nonymous) wrote in #note-19:
>
> > In keeping with the Ruby-ish collection methods that end with "-ect", how about
> >
> > - **reflect** -- the idea is to contrast with inject, this "reflects" all intermediate states
> > - **project** -- the idea is that the original Enumerable is "projected" in a mathematical sense into the plane defined by the stateful function

> > that is passed in
>
> I'd like that “this "reflects" all intermediate states” part.

I think this is way too generic. In the same vein, we could call it "return", because it returns all intermediate states.
Also, I think "project" isn't appropriate, because project is usually associated with a dimension reduction.

The most specific word in the above explanation is "intermediate". I suggest we search more along these lines. An example would be something like "inject_with_intermediates".

BTW, I also checked APL, where '\' is used for what we are discussing here, and is called scan. The fact that this is included in APL shows that this is in some sense a core operation. It doesn't appear e.g. in a 1972 manual for IBM APL\360 (http://www.softwarepreservation.org/projects/apl/Manuals/APL360UsersManuals), which means it may not have been there from the start. But I found some hints in a newer document (1982, http://www.softwarepreservation.org/projects/apl/Manuals/SharpAPLManualCorrections). My guess is that the name did not come from APL (which is one of the oldest functional programming languages).

#26 - 07/25/2020 10:12 AM - y.annikov (Yakov Annikov)

y.annikov (Yakov Annikov) wrote in #note-22:

> I've created a PR with the implementation of scan and lazy.scan methods https://github.com/ruby/ruby/pull/3358
> Feel free to reject it but any comments are appreciated. I got a lot of fun writing this code and looking at Ruby source code.

I renamed it to reflect for a while though I like scan more.
Here are examples of the behaviour of reflect method that I've implemented:

```
# reflect
(0...0).reflect(:+) => []
(5..10).reflect(:+) => [5, 11, 18, 26, 35, 45]
(5..10).reflect { |sum, n| sum + n } => [5, 11, 18, 26, 35, 45]
(5..10).reflect(1, :*) => [1, 5, 30, 210, 1680, 15120, 151200]
(5..10).reflect(1) { |product, n| product * n } => [1, 5, 30, 210, 1680, 15120, 151200]

# lazy.reflect
(0..Float::INFINITY).lazy.reflect(:+).first(4) => [0, 1, 3, 6]
(0..Float::INFINITY).lazy.reflect(:+).first(4) => [1, 1, 2, 4]
enum = (5..10).lazy.reflect(:+) => #<Enumerator::Lazy: #<Enumerator::Lazy: 5..10>:reflect(:+)>
6.times.map { enum.next } => [5, 11, 18, 26, 35, 45]
enum = (5..10).lazy.reflect(1, :*) => #<Enumerator::Lazy: #<Enumerator::Lazy: 5..10>:reflect(1, :*)>
7.times.map { enum.next } => [1, 5, 30, 210, 1680, 15120, 151200]
```

UPD: Some fixes according to duerst (Martin Dürst) comments.

#27 - 07/28/2020 05:42 PM - RubyBugs (A Nonymous)

To return to naming for a moment. Did I understand an interest may exist in naming that reflects a connection or duality with inject?

If so, might the **scan** operation find a better Ruby name in **interject**?

The idea would be to say to new learners, with zero prior context: “the interject method has the same form as inject, but returns the stream of **intermediate** values”?

#28 - 07/28/2020 11:34 PM - duerst (Martin Dürst)

RubyBugs (A Nonymous) wrote in #note-27:

> To return to naming for a moment. Did I understand an interest may exist in naming that reflects a connection or duality with inject?

I guess that wouldn't be a bad idea.

> If so, might the **scan** operation find a better Ruby name in **interject**?
>
> The idea would be to say to new learners, with zero prior context: “the interject method has the same form as inject, but returns the stream of **intermediate** values”?

My most immediate impression of interject is that it would add something between Array elements (e.g. [5..8].interject(0) #→ [5, 0, 6, 0, 7, 0, 8] or so).
But the parallel to inject is appealing. And inject itself isn't really that much of a descriptive name, either.

We would have to check whether interject is used in other languages, and for what.

#29 - 07/30/2020 07:16 PM - parker (Parker Finch)

I'd like to sum up where we're at with this discussion. Let me know if you disagree with my interpretation here!

1. There is some support for the idea of adding this method.
2. We should avoid changing the behavior of #inject on lazy enumerables, since it would be confusing for the lazy version of a method to have different behavior than the strict version.
3. There is concern around the name of this method. Possibilities that have been discussed are:
    - #scan: This name is not very intuitive for the operation and also has other meanings. However, it is used in many other programming languages.
    - #accumulate: This would have my vote at the moment. It suggests that something is accumulated throughout the iteration of the collection. However, it is still a very generic term.
    - #reflect, #project, #interject: These end in -ect as many Ruby/Smalltalk methods on collections do. However, they have little meaning related to the operation at hand. I agree that it would be good to show the duality between #inject and this operation, but I'm concerned about using an unrelated word.

I think the next steps here are to get confirmation from matz (Yukihiro Matsumoto) as to whether or not he thinks this would be a good method to add to Ruby (I described some use cases in #note-18) and to decide on a name. Does that make sense?

#30 - 07/31/2020 05:47 AM - nobu (Nobuyoshi Nakada)

I found a word traject, means to transport, transmit, or transpose.
It may (or may not) imply trajectory...

#31 - 08/10/2020 04:47 PM - parker (Parker Finch)

nobu (Nobuyoshi Nakada) wrote in #note-30:

> I found a word [traject], means to transport, transmit, or transpose.
> It may (or may not) imply [trajectory]...
>
> [traject]: https://www.dictionary.com/browse/traject
> [trajectory]: https://www.dictionary.com/browse/trajectory

That's an interesting word that I don't know. It looks like it's archaic, so I don't think that it has much meaning anymore. That's kind of nice, since there's not a conflicting definition that people will have in their heads. However, it also doesn't have a meaning that describes what the method does.

It does have a nice symmetry with inject though! Curious if others have thoughts?

#32 - 08/21/2020 02:22 PM - parker (Parker Finch)

I'd like to propose that we name this method #accumulate. matz (Yukihiro Matsumoto) do you think that is an acceptable name?

#33 - 08/29/2020 11:41 AM - Dan0042 (Daniel DeLorme)

#accumulate is good, but since this question linked in the OP was asking for the "cumulative sum" ... what about #cumulative ?

```
[1,2,3].cumulative.sum            #=> [1,3,6]
[1,2,3].cumulative.inject(5, :*)  #=> [[5], [5,10], [5,10,30]]
[1,2,3].cumulative.select(&:odd?) #=> [[1], [1,3]]
[1,2,3].cumulative.map{ _1 * 3 }  #=> [[3], [3,6], [3,6,9]]
```

#34 - 09/03/2020 02:57 PM - parker (Parker Finch)

Dan0042 (Daniel DeLorme) wrote in #note-33:

> ... what about #cumulative ?

Oh that's interesting! I had leapt straight to verbs since that tends to be the pattern for methods that transform enumerables, but the examples of [1,2,3].cumulative.some_other_method are compelling and make sense in a "it returns a cumulative enumerator" way.

I don't think it fits quite as well when a block is passed though; [1,2,3].cumulative(0, &:+) doesn't read as naturally to me as the imperative #accumulate.

I think either option is good -- although since there's a pattern of enumerable methods being named imperatively (e.g. #map, #select, #inject, #drop) I still slightly lean toward the #accumulate option.

Files

scan_left_example.rb	2.93 KB	07/10/2020	parker (Parker Finch)