The description explains the necessity for an eager method for lazy enumerators in Ruby. The code snippet provided demonstrates the intended behavior of the `eager` method:

```ruby
/*
 * call-seq:
 * _lzy_.eager -> enum
 *
 * Returns a non-lazy Enumerator converted from the lazy enumerator.
 *
 * This is useful where a normal Enumerable object needs to be
 * generated while lazy operation is still desired to avoid creating
 * intermediate arrays.
 *
 * enum = huge_collection.lazy.flat_map(&:children).reject(&:disabled?).eager
 * enum.map {_x| ...}  # an array is returned
 */
```

The associated revision adds the `eager` method to the `Lazy` module, allowing users to convert lazy enumerators to eager ones when needed, without changing the intended lazy behavior of other parts of the code.

The history log includes comments on the necessity and implementation of the method, as well as feedback on the approach taken in the revision.

My understanding is that Lazy is kind of a low-level tool for implementing efficient pipelines and not normally part of a method API, so being able to convert a lazy Enumerator to an eager Enumerator helps users write efficient code while not requiring library developers to support Lazy in every method that takes an Enumerable.
knu (Akinori MUSHA) wrote:

Suppose I want to pass the enum to another method that takes an Enumerable object where the passed object is expected to act eagerly. One of the most significant use cases is when the callee uses take_while on a given object expecting to receive the result as an array, where a normal Enumerable object that generates a list lazily is ideal.

I see, thanks for explaining.
One argument would be the callee should do take_while {}.to_a if they really want an Array and not just an Enumerable (which has most of Array's methods).
But I understand it's nice to be able to use an unmodified library and not needing the library to know about Enumerator::Lazy.

#6 - 06/06/2019 01:55 PM - knu (Akinori MUSHA)

One argument would be the callee should do take_while {}.to_a if they really want an Array and not just an Enumerable (which has most of Array's methods).
But I understand it's nice to be able to use an unmodified library and not needing the library to know about Enumerator::Lazy.

Yeah, that would be the way completely against duck-typing. In my opinion, Enumerator::Lazy objects should only be used locally not to be passed around, because it claims to be Enumerable but does not work exactly the same as other Enumerable objects, requiring others to take special care of them.

With eager added, you could consider Lazy as a DSL for generating a memory efficient enumerator that is compatible with any Enumerable friendly method with this idiom: enum.lazy.method[].chain().eager.

#7 - 06/06/2019 05:00 PM - Eregon (Benoit Daloze)

knu (Akinori MUSHA) wrote:

Yeah, that would be the way completely against duck-typing.

Right, to clarify I meant the library would need the extra .to_a where it expects an Enumerator method to return an Array and not just some Enumerable.
So basically it would need to know that a Enumerator::Lazy could be passed, and so the extra .to_a is needed.
If the library treat the result just like an Enumerable, no modification would be needed.
Anyway, +1: from me.

#8 - 06/12/2019 08:22 AM - mame (Yusuke Endoh)

+1 for the feature. I'm not a fan for the notation .lazy. ... .eager, though.

How about this style?

```
[0, 1, 2].lazy{|e| e.map{|n| n + 1}.map{|n| n.to_s} }
```

#9 - 06/13/2019 07:05 AM - knu (Akinori MUSHA)

mame (Yusuke Endoh) wrote:

How about this style?

```
[0, 1, 2].lazy{|e| e.map{|n| n + 1}.map{|n| n.to_s} }
```

Is it like extending lazy() as below?

```ruby
def lazy(&block)
  if block
    return block.call(lazy).eager
  end
  Enumerator::Lazy.new(self)
end
```

It may be an Interesting addition. I'm not sure what it should do if the block does not return a lazy enumerator, though.

#10 - 06/13/2019 07:14 AM - zverok (Victor Shepelev)
How about this style?

```ruby
[0, 1, 2].lazy{|e| e.map{|n| n + 1}.map{|n| n.to_s}}
```

#=> an Enumerator containing "1", "2", and "3"

I believe that it defies the whole idea of lazy enumerators. Instead of chaining laziness (and having regular polymorphic enumerator, that can be passed into other methods, or returned from some methods to "external" clients, and so on), e.g. instead of lazy enums being "just enums", it requires wrapping everything in a lazy {} block, effectively limiting applicability of the idea.

This approach ("everything inside the block is lazy-computed") may be an optimization technique; but lazy enumerators currently is an idiom, even if underappreciated (especially ActiveSupport loves to switch from "generic enumerables" to "just arrays"), but very prominent and interesting.

I'd rather keep this issue simple. We can talk about it later in another issue.

I don't see how the suggestion defies anything, changes the current lazy method (without block) and or "requires" anything. It simply extends it with a block form that would automatically return an eager enumerator.

My issue with lazy{} is that it forces the user to return a lazy enumerator from the block (otherwise it raises?, e.g. `[1].lazy{ 42 } # => RuntimeError?). The fact that the result could be completely unrelated is for me a design smell, e.g. `[1].lazy { [1, 2, 3].lazy }).

I'm personally +1 for eager. I'd also alias Enumerator#eager to #itself.

Could you elaborate on why? Are you thinking of situations where lazy and eager enumerators are mixed?

Yes, I was thinking of a method accepting a potentially lazy enumerator. It would be to avoid having to write enum = enum.eager if enum.respond_to?(:eager) and simply calling enum.eager no matter if the argument is eager or not.

It's in the same way as Enumerator::Lazy#lazy is an alias to itself.

I don't expect it to be very useful though, lazy enumerators are probably rare and not often mixed with eager enumerators, so it might be premature. I just think the extra cognitive load is basically zero.
#15 - 09/03/2019 08:02 AM - knu (Akinori MUSHA)
Thanks, I see your point. It could be handy, but it seems to me that adding Enumerable#eager is like encouraging users to consider calling enum.eager on every given enumerable object and supporting Lazy objects to be passed around; that's opposite of what I was trying to achieve with this proposal.

Rather than spreading the practice of doing enum.eager.map {...} or enum.map {...}.to_a, I would encourage users to keep Lazy in implementation details and always use eager enumerators in API and data interchange.

#16 - 09/04/2019 07:17 AM - Anonymous
- Status changed from Open to Closed

Applied in changeset git|1d4bd229b898671328c2a942b0408065c640c28.

Implement Enumerator::Lazy#eager [Feature #15901]

Files

| 0001-Implement-Enumerator-Lazy-eager.patch | 2.32 KB | 06/05/2019 | knu (Akinori MUSHA) |