Ruby master - Feature #7297
map_to alias for each_with_object
11/07/2012 05:30 AM - nathan.f77 (Nathan Broadbent)

Status: Rejected
Priority: Normal
Assignee:
Target version: 2.0.0

Description
I would love to have a shorter alias for 'each_with_object', and would like to propose 'map_to'. Here are my arguments:

- It reads logically and clearly:

```ruby
[1, 2, 3].map_to({}) { |i, hash| hash[i] = i ** 2 }
#=> {1 => 1, 2 => 4, 3 => 9}
```

- Rubyists are already using 'map' to build and return an array, so it should be obvious that 'map_to(object)' can be used to build and return an object.

- Given that 'each' and 'each_with_index' return the original array, I feel that the 'each_with_object' method name is slightly counterintuitive. 'map_to' might not be 100% semantically correct, but it's obvious that it will return something other than the original array.

- Many people (myself included) were using inject([]) { |hash, el| ... ; hash } instead of 'each_with_object', partly because of ignorance, but also because 'each_with_object' is so long. 'map_to' is the same length as inject, and means that you don't have to return the object at the end of the block.

- Only a single line of code is needed to implement the alias.

Best,
Nathan

Related issues:
Related to Ruby master - Feature #7340: 'each_with' or 'into' alias for 'each... Open

History
#1 - 11/07/2012 08:38 AM - matz (Yukihiro Matsumoto)
- Status changed from Open to Rejected

I feel the name #map_to does not make sense for the behavior.
If the behavior were like this,

```ruby
def map_to(o)
  self.each {|x|
    o << x
  }
  o
end
```

at least makes sense. I know that's not what you want, so I reject this proposal.

Matz.

#2 - 11/07/2012 09:52 PM - rosenfeld (Rodrigo Rosenfeld Rosas)
I just searched each_with_object in GitHub and aside from some specs testing each_with_object all real-use cases I could see after navigating through several pages of the results were ".each_with_object([])...":

https://github.com/search?langOverride=&language=Ruby&q=each_with_object&repo=&start_value=15&type=Code

That is why I think we should have a .hash_map method instead of another generic one:

```ruby
double = numbers.hash_map{|h, n| h[n] = n ** 2 }
```
Better to read than

double = numbers.each_with_object([]){ |n, h| h[n] = n * 2 }

But other than that, I don't really think map_to is a bad name for the generic case (I'm replacing the arguments order of the block since it makes more sense to me instead of using it as an alias to each_with_object):

double = numbers.map_to([]){ |h, n| h[n] = n * 2 }

I read this as "map numbers to hash" which makes lots of sense to me. Much more than each_with_object. Which only seems to be used with hashes by the way...

#3 - 11/08/2012 12:09 AM - trans (Thomas Sawyer)

=begin
But the crux of the problem is simply that #each_with_object has a name that it is too long, which greatly deters usage. I know I am loathe to use it even when it would be useful for this simple reason. And that has also has a lot to do with the fact that the last word of the method's name, "_object", is complete redundant. Of course it is a object! Ruby is an OOPL! So just shortening it to #each_with alone would make a big difference.

Beyond that, one might argue the block parameters should be in the opposite order to align with #inject, but that isn't at all important. More significant would be to default the argument to an empty hash, since that would be the most common case. I think that's a good idea so the common case can be more concise. But in doing that, it makes more sense to call the method #map_with and have it return the object.

So the doubles example would be:

doubles = numbers.map_with{ |n, h| h[n] = n * 2 }

And one could also do things like:

doubles = numbers.map_with({}).{ |n, a| a << n * 2 }

It's a very versatile method and the name makes sense.
=end

#4 - 11/08/2012 12:32 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Hi Thomas,

I agree that _object is a redundant suffix and that map_with is probably a better description than each_with or each_with_object.

But it makes sense to me that each_with_object should yield(item, object) because of the name of the method. I do agree that the order of the arguments isn't very important but I would prefer to have the returning value to become first as it seems more natural to me and it also aligns well with inject like you said.

Now, what I'm not comfortable with is using map_with with {} as the default argument. I agree this is the commonest case, this is not the issue. The problem is that this doesn't read well for someone trying to understand the code:

doubles = numbers.map_with{ |n, h| h[n] = n * 2 }

This reads much better:

doubles = numbers.map_with({}).{ |n, a| a << n * 2 } # although I'd reorder the arguments of the closure.

So, if it was my decision to design the language I would probably deprecate each_with_object in favor of map_with, I'd replace the order of the arguments in the block and would also create another method:

def hash_map(&block)
  map_with({}, &block)
end

Instead of hash_map I'd also support the idea of using to_h as suggested by someone a while back:

doubles = numbers.to_h{[h, n] h[n] = n * 2 }

#5 - 11/08/2012 03:09 AM - bitsweat (Jeremy Daer)

The common thread here is that people want a hash conversion in an enumerable chain, where it feels fluent and natural, rather than wrapping the result with Hash[] which makes the code read backward. #each_with_object is wonderful, but verbose for this use.

Ruby has the idea of an association already: a key and value paired together. It's used by Array#assoc to look up a value from a list of pairs and by Hash#assoc to return a key/value pair. Building up a mapping of key/value pairs is associating keys with values. So consider Enumerable#associate which builds a mapping by associating keys with values:

module Enumerable
  # Associates keys with values and returns a Hash.
end

08/07/2021 2/6
If you have an enumerable of keys and want to associate them with values, pass a block that returns a value for the key:

```
[1, 2, 3].associate { |i| i ** 2 }
# => { 1 => 1, 2 => 4, 3 => 9 }
```

```ruby
%w(tender love).associate &:capitalize
# => {"tender"=>"Tender", "love"=>"Love"}
```

If you have an enumerable key/value pairs and want to associate them, omit the block and you'll get a hash in return:

```
1, 2], [3, 4].associate
# => { 1 => 2, 3 => 4 }
```

```ruby
def associate(mapping = {})
  if block_given?
    each_with_object(mapping) do |key, object|
      object[key] = yield(key)
    end
  else
    each_with_object(mapping) do |(key, value), object|
      object[key] = value
    end
  end
end
```

## #6 - 11/08/2012 03:25 AM - john_firebaugh (John Firebaugh)

I reviewed my own use of Hash[], and in the majority of cases, I'm transforming both the key and value. So I would prefer a more flexible definition:

```ruby
module Enumerable
  # Associates keys with values and returns a Hash.
  #
  # If you have an enumerable of keys and want to associate them with values,
  # pass a block that returns a value for the key:
  #
  # %w(tender love).associate &:capitalize
  # => {"tender"=>"Tender", "love"=>"Love"}
  #
  # [1, 2, 3].associate { |i| i * 2, i ** 2 }
  # => { 2 => 1, 4 => 4, 6 => 9 }
  #
  # If you have an enumerable key/value pairs and want to associate them,
  # omit the block and you'll get a hash in return:
  #
  # 1, 2], [3, 4].associate
  # => { 1 => 2, 3 => 4 }
  
  def associate(mapping = {})
    if block_given?
      each_with_object(mapping) do |key, object|
        key, value = yield(key)
        object[key] = value
      end
    else
      each_with_object(mapping) do |(key, value), object|
        object[key] = value
      end
    end
  end
end
```

## #7 - 11/08/2012 03:25 AM - john_firebaugh (John Firebaugh)

john_firebaugh (John Firebaugh) wrote:

> I reviewed my own use of Hash[], and in the majority of cases, I'm transforming both the key and value. So I would prefer a more flexible definition:

```ruby
module Enumerable
  # Associates keys with values and returns a Hash.
  #
  # If you have an enumerable of keys and want to associate them with values,
  # pass a block that returns a value for the key:
  #
  # %w(tender love).associate &:capitalize
  # => {"tender"=>"Tender", "love"=>"Love"}
  #
  # [1, 2, 3].associate { |i| i * 2, i ** 2 }
  # => { 2 => 1, 4 => 4, 6 => 9 }
  #
  # If you have an enumerable key/value pairs and want to associate them,
  # omit the block and you'll get a hash in return:
  #
  # 1, 2], [3, 4].associate
  # => { 1 => 2, 3 => 4 }
  
  def associate(mapping = {})
    if block_given?
      each_with_object(mapping) do |key, object|
        key, value = yield(key)
        object[key] = value
      end
    else
      each_with_object(mapping) do |(key, value), object|
        object[key] = value
      end
    end
  end
end
```
If you have an enumerable key/value pairs and want to associate them,
# omit the block and you'll get a hash in return:
#
# 1, 2], [3, 4].associate
#  #=> { 1 => 2, 3 => 4 }
def associate(mapping = {})
  if block_given?
    each_with_object(mapping) do |key, object|
      key, value = yield(key)
      object[key] = value
    end
  else
    each_with_object(mapping) do |(key, value), object|
      object[key] = value
    end
  end
end

---

#8 - 11/08/2012 03:26 AM - john_firebaugh (John Firebaugh)

john_firebaugh (John Firebaugh) wrote:

john_firebaugh (John Firebaugh) wrote:

I reviewed my own use of Hash[], and in the majority of cases, I'm transforming both the key and value. So I would prefer a more flexible definition:

module Enumerable
  # Associates keys with values and returns a Hash.
  #
  # If you have an enumerable of keys and want to associate them with values,
  # pass a block that returns a tuple for the key and value:
  #
  #  %w( tender love ).associate { |n| [n, n.capitalize] }
  #  #=> {"tender"=>"Tender", "love"=>"Love"}
  #
  #  [1, 2, 3].associate { |i| [i * 2, i ** 2] }
  #  #=> { 1 => 2, 4 => 6, 8 => 9 }
  #
  # If you have an enumerable key/value pairs and want to associate them,
  # omit the block and you'll get a hash in return:
  #
  #  1, 2], [3, 4].associate
  #  #=> { 1 => 2, 3 => 4 }
  def associate(mapping = {})
    if block_given?
      each_with_object(mapping) do |key, object|
        key, value = yield(key)
        object[key] = value
      end
    else
      each_with_object(mapping) do |(key, value), object|
        object[key] = value
      end
    end
  end
end

---

#9 - 11/08/2012 03:29 AM - john_firebaugh (John Firebaugh)

Ugh, sorry for the spam. I didn't realize "editing" a comment actually posts a new one.
John, I have a bit of usage like that. Consider that collection.map { |element| [key, value] }.associate handles that cleanly.

In my code survey, nearly all my uses of Hash[] and #inject are covered by #associate - associating an collection of keys with calculated values.

For more complex scenarios, using more verbose, powerful API like #inject, #each_with_object, or #map + #associate feels appropriate.

John, I have a bit of usage like that. Consider that collection.map { |element| [key, value] }.associate handles that cleanly.

In my code survey, nearly all my uses of Hash[] and #inject are covered by #associate - associating an collection of keys with calculated values.

For more complex scenarios, using more verbose, powerful API like #inject, #each_with_object, or #map + #associate feels appropriate.

The common thread here is that people want a hash conversion in an enumerable chain, where it feels fluent and natural, rather than wrapping the result with Hash[] which makes the code read backward. #each_with_object is wonderful, but verbose for this use.

Ruby has the idea of an association already: a key and value paired together. It's used by Array#assoc to look up a value from a list of pairs and by Hash#assoc to return a key/value pair. Building up a mapping of key/value pairs is associating keys with values. So consider Enumerable#associate which builds a mapping by associating keys with values:

Exactly.

Here's the secret plan I propose:
1) Let's convince Matz about Enumerable#to_h by adding your +1 to #7292.
2) When that's done, and only when that's done, we can convince Matz that 'associate' is the right name for a nifty method that associates keys with values. See my 1 min slide in #4151. I believe my proposal agrees with yours and it also handles collisions and the special case of Hash#assoc. Note that we should start with Enumerable#to_h. If we have it, then we can see it's probably best if [] .associate (with no block) returns an enumerator, not a hash.
3) After that, we can see if we can come up with a general 'categorize' that could replace map[ [k, v] ].to_h as well as more complex nested cases.

I'd also propose you move the discussion of these features from here (a rejected request about an alias of map_with_object!) to the right issues.

I like Enumerable#associate. It would cover many of the cases where each_with_object ([]) is currently used.

However, I think '1,2,13,4 .to_h' would be better for when an Array is already in the 'associated' form. See #7292 for that discussion: https://bugs.ruby-lang.org/issues/7292 .

If the #to_h method was also available, then #associate could require a block, and it would be the inverse of Active Support's 'index_by' method (it would yields values instead of keys): http://api.rubyonrails.org/classes/Enumerable.html#method-i-index_by

I'm still wondering if there should be a method that allows you to return both key and value, but 'map [e | e * 2, e ** 2].to_h' would probably be sufficient.

(sorry for bringing #7292 into the discussion, but I feel like it's a related issue.)

Some background:
#4151 proposes an Enumerable#categorize API, but it's complex and hard to understand its behavior at a glance.
#7292 proposes an Enumerable#to_h == Hash[...API, but I don't think of association/pairing as explicit coercion, so #to_h feels misfit.

Associate is a simple verb with unsurprising results. It doesn't introduce ambiguous "map" naming. You associate an enumerable of keys with yielded values.

Some before/after examples:

Before: Has[h filenames.map | file | [filename, download_url(filename)] |]
After: filenames.associate | [filename] download_url filename |
Before: alphabet.each_with_index.each_with_object({}) { |(letter, index), hash| hash[letter] = index }
After: alphabet.each_with_index.associate

Before: keys.each_with_object({}) { |k, hash| hash[k] = self[k] } # a simple Hash#slice
After: keys.associate { [key] self[key] }

Apologies for hijacking this rejected ticket with a different proposal. I will open a new feature request if anyone is positive on this usage.

#14 - 11/08/2012 05:53 AM - nathan.f77 (Nathan Broadbent)

Associate is a simple verb with unsurprising results. It doesn't introduce ambiguous "map" naming. You associate an enumerable of keys with yielded values.

... Apologies for hijacking this rejected ticket with a different proposal. I will open a new feature request if anyone is positive on this usage.

Yep, I'll add my +1 to this feature request. I think #associate is a great name and would be a very useful method.

#15 - 11/08/2012 06:04 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

I like the "associate" name but I prefer the implementation to be like the hash_map described above. I have lots of situations where the element of the Enumerable is not the key of the resulting hash. Your suggestions sounds inefficient to me for this case:

collection.map { |v| [v.attribute_id, v.some_value] }.associate

I guess this will always perform worse than:

collection.associate{|h, v| h[v.attribute_id] = v.some_value }

#16 - 11/08/2012 08:48 AM - matz (Yukihiro Matsumoto)

In the discussion, you guys came up with several new ideas, and they are welcome. Submit those ideas as issues.

* each_with alias to each_with_object
* Enumerable#associate

Matz.

#17 - 11/14/2012 09:07 AM - bitsweat (Jeremy Daer)

Thank you. Enumerable#associate proposed: #7341

#18 - 05/10/2014 03:22 PM - akr (Akira Tanaka)

- Related to Feature #7340: 'each_with' or 'into' alias for 'each_with_object' added