Ruby trunk - Feature #4151

Enumerable#categorize

12/12/2010 08:14 PM - akr (Akira Tanaka)

Status:	Rejected	
Priority:	Normal	
Assignee:	akr (Akira Tanaka)	
Target version:	2.6	

Description

=begin
Hi.

How about a method for converting enumerable to hash?

enum.categorize([opts]) {|elt| [key1, ..., val] } -> hash

categorizes the elements in enum and returns a hash.

The block is called for each elements in enum.
The block should return an array which contains
one or more keys and one value.

```
p (0..10).categorize {|e| [e % 3, e % 5] }
#=> {0=>[0, 3, 1, 4], 1=>[1, 4, 2, 0], 2=>[2, 0, 3]}
```

The keys and value are used to construct the result hash.
If two or more keys are provided
(i.e. the length of the array is longer than 2),
the result hash will be nested.

```
p (0..10).categorize {|e| [e&4, e&2, e&1, e] }
#=> {0=>{0=>{0=>[0, 8],
#           1=>[1, 9]},
#       2=>{0=>[2, 10],
#           1=>[3]}},
#    4=>{0=>{0=>[4],
#           1=>[5]},
#       2=>{0=>[6],
#           1=>[7]}}}
```

The value of innermost hash is an array which contains values for
corresponding keys.
This behavior can be customized by :seed, :op and :update option.

This method can take an option hash.
Available options are follows:

- :seed specifies seed value.
- :op specifies a procedure from seed and value to next seed.
- :update specifies a procedure from seed and block value to next seed.

:seed, :op and :update customizes how to generate
the innermost hash value.
:seed and :op behavies like Enumerable#inject.

If seed and op is specified, the result value is generated as follows.
op.call(..., op.call(op.call(seed, v0), v1), ...)

:update works as :op except the second argument is the block value itself
instead of the last value of the block value.

If :seed option is not given, the first value is used as the seed.

```
# The arguments for :op option procedure are the seed and the value.
# (i.e. the last element of the array returned from the block.)
r = [0].categorize(:seed => :s,
:op => lambda {|x,y|
p [x,y]             #=> [:s, :v]
1
}) {|e|
p e #=> 0
[:k, :v]
}
p r #=> {:k=>1}

# The arguments for :update option procedure are the seed and the array
# returned from the block.
r = [0].categorize(:seed => :s,
:update => lambda {|x,y|
p [x,y]             #=> [:s, [:k, :v]]
1
}) {|e|
p e #=> 0
[:k, :v]
}
p r #=> {:k=>1}
```

The default behavior, array construction, can be implemented as follows.

```
:seed => nil
:op => lambda {|s, v| !s ? [v] : (s << v) }
```

Note that matz doesn't find satisfact in the method name, "categorize".
[ruby-dev:42681]

Also note that matz wants another method than this method,
which the hash value is the last value, not an array of all values.
This can be implemented by enum.categorize(:op=>lambda {|x,y| y}) { ... }.
But good method name is not found yet.
[ruby-dev:42643]
--
Tanaka Akira

Attachment: enum-categorize.patch
=end

Related issues:

Related to Ruby trunk - Feature #6669: A method like Hash#map but returns hash	**Feedback**	
Related to Ruby trunk - Feature #666: Enumerable::to_hash	**Rejected**	**10/20/2008**
Related to Ruby trunk - Feature #5662: inject-accumulate, or Haskell's mapAccum*	**Rejected**	**11/23/2011**
Related to Ruby trunk - Feature #5008: Equal rights for Hash (like Array, Str...	**Rejected**	**07/10/2011**
Related to Ruby trunk - Feature #7241: Enumerable#to_h proposal	**Rejected**	**10/30/2012**
Related to Ruby trunk - Feature #7292: Enumerable#to_h	**Closed**	**11/07/2012**
Related to Ruby trunk - Feature #7793: New methods on Hash	**Assigned**	

History

#1 - 12/12/2010 09:38 PM - duerst (Martin Dürst)

=begin

On 2010/12/12 20:13, Tanaka Akira wrote:

> Hi.
>
> How about a method for converting enumerable to hash?

There are already such methods. At least group_by. Your proposal seems
to be a very special way to convert an enumerable to a hash.

> enum.categorize([opts]) {|elt| [key1, ..., val] } -> hash

> categorizes the elements in enum and returns a hash.
>
> The block is called for each elements in enum.
> The block should return an array which contains
> one or more keys and one value.
>
> p (0..10).categorize {|e| [e % 3, e % 5] }
> #=> {0=>[0, 3, 1, 4], 1=>[1, 4, 2, 0], 2=>[2, 0, 3]}

I'm trying to understand your example. Let me make a table.

original value	return_value[0]	return_value[1]
0	0	0
1	1	1
2	2	2
3	0	3
4	1	4
5	2	0
6	0	1
7	1	2
8	2	3
9	0	4
10	1	0

I think I get the idea of how this is supposed to work. But I'm not sure what the actual use cases would be. Can you give some?

For this variant, a Ruby equivalent is:

```
h = {}
(0..10).each do |e|
h[e%3] ||= []
h[e%3] << e%5
end
p h
```

Or shorter:

```
h = {}
(0..10).each { |e| (h[e%3] ||= []) << e%5 }
p h
```

I admit that it is more elegant if the initialization and the final line can be avoided, but I think that's an issue that can be dealt with separately.

> The keys and value are used to construct the result hash.
> If two or more keys are provided
> (i.e. the length of the array is longer than 2),
> the result hash will be nested.
>
> p (0..10).categorize {|e| [e&4, e&2, e&1, e] }
> #=> {0=>{0=>{0=>[0, 8],
> # 1=>[1, 9]},
> # 2=>{0=>[2, 10],
> # 1=>[3]}},
> # 4=>{0=>{0=>[4],
> # 1=>[5]},
> # 2=>{0=>[6],
> # 1=>[7]}}}
>
> The value of innermost hash is an array which contains values for
> corresponding keys.
> This behavior can be customized by :seed, :op and :update option.
>
> This method can take an option hash.
> Available options are follows:

Who will be able to remember exactly what these options are,...? Most other methods, in particular on Enumerables, don't have that many options. I think that's for a good reason.

Regards, Martin.

- :seed specifies seed value.
- :op specifies a procedure from seed and value to next seed.
- :update specifies a procedure from seed and block value to next seed.

:seed, :op and :update customizes how to generate
the innermost hash value.
:seed and :op behavies like Enumerable#inject.

If seed and op is specified, the result value is generated as follows.
op.call(..., op.call(op.call(seed, v0), v1), ...)

:update works as :op except the second argument is the block value itself
instead of the last value of the block value.

If :seed option is not given, the first value is used as the seed.

```
# The arguments for :op option procedure are the seed and the value.
# (i.e. the last element of the array returned from the block.)
r = [0].categorize(:seed =>  :s,
:op =>  lambda {|x,y|
p [x,y]              #=>  [:s, :v]
1
}) {|e|
p e #=>  0
[:k, :v]
}
p r #=>  {:k=>1}

# The arguments for :update option procedure are the seed and the array
# returned from the block.
r = [0].categorize(:seed =>  :s,
:update =>  lambda {|x,y|
p [x,y]              #=>  [:s, [:k, :v]]
1
}) {|e|
p e #=>  0
[:k, :v]
}
p r #=>  {:k=>1}
```

The default behavior, array construction, can be implemented as follows.

```
:seed =>  nil
:op =>  lambda {|s, v| !s ? [v] : (s<<  v) }
```

Note that matz doesn't find satisfact in the method name, "categorize".
[ruby-dev:42681]

Also note that matz wants another method than this method,
which the hash value is the last value, not an array of all values.
This can be implemented by enum.categorize(:op=>lambda {|x,y| y}) { ... }.
But good method name is not found yet.
[ruby-dev:42643]

--
#-# Martin J. Dürst, Professor, Aoyama Gakuin University
#-# http://www.sw.it.aoyama.ac.jp mailto:duerst@it.aoyama.ac.jp

=end

#2 - 12/12/2010 10:04 PM - zenspider (Ryan Davis)

=begin

On Dec 12, 2010, at 04:38 , Martin J. Dürst wrote:

```
h = {}
(0..10).each do |e|
h[e%3] ||= []
h[e%3] << e%5
end
p h
```

Or shorter:

```
h = {}
(0..10).each { |e| (h[e%3] ||= []) << e%5 }
p h
```

Or shorter (and faster) still:

```
h = Hash.new { |h,k| h[k] = [] }
(0..10).each { |e| h[e%3] << e%5 }
p h
```

=end

#3 - 12/12/2010 11:43 PM - Eregon (Benoit Daloze)

=begin
On 12 December 2010 12:13, Tanaka Akira akr@fsij.org wrote:

> Hi.
>
> How about a method for converting enumerable to hash?
>
> enum.categorize([opts]) {|elt| [key1, ..., val] } -> hash
>
> categorizes the elements in enum and returns a hash.
>
> The block is called for each elements in enum.
> The block should return an array which contains
> one or more keys and one value.
>
> p (0..10).categorize {|e| [e % 3, e % 5] }
> #=> {0=>[0, 3, 1, 4], 1=>[1, 4, 2, 0], 2=>[2, 0, 3]}

I find this clearer (but categorize looks better) :

```
(0..10).group_by { |e| e%3 }.each_with_object({}) { |(k,v),h| h[k] =
v.map { |e| e%5 } }
```

(Unfortunately, the 'map' on a Hash is not very nice, so I guess you
better 'group' by hand and 'map' at the same time like Martin showed)

> Martin: I admit that it is more elegant if the initialization and the final line can be avoided, but I think that's an issue that can be dealt with separately.

Enumerable#each_with_object is handy in this case:

```
(0..10).each_with_object({}) { |e,h| (h[e%3] ||= []) << e%5 }
```

Or, with Ryan's variant:

```
(0..10).each_with_object(Hash.new { |h,k| h[k]=[] }) { |e,h| h[e%3] << e%5 }
# I wish there was a shorter way for a self-duplicating Hash's default value
```

For the rest, I agree with Martin.

Regards,
B.D.

=end

#4 - 12/14/2010 06:41 PM - akr (Akira Tanaka)

=begin
2010/12/12 "Martin J. Dürst" duerst@it.aoyama.ac.jp:

> There are already such methods. At least group_by. Your proposal seems to be
> a very special way to convert an enumerable to a hash.

Enumerable#categorize is more general than Enumerable#group_by.

Enumerable#group_by can specify only hash keys.
It cannot specify hash values.

> I think I get the idea of how this is supposed to work. But I'm not sure

> what the actual use cases would be. Can you give some?

I see.

Assume a table as follows.

```
ary = [
["matz", "Yukihiro Matsumoto"],
["nobu", "Nobuyoshi Nakada"],
["akr", "Tanaka Akira"],
["usa", "Usaku NAKAMURA"],
["naruse", "NARUSE, Yui"],
["ko1", "SASADA Koichi"]
]
```

Enumerable#categorize can be used to generate a hash
from the right elements to left elements.
(and left elements to right elements too.)

```
pp ary.categorize {|e| [e[1], e[0]] }
#=>
{"Yukihiro Matsumoto"=>["matz"],
"Nobuyoshi Nakada"=>["nobu"],
"Tanaka Akira"=>["akr"],
"Usaku NAKAMURA"=>["usa"],
"NARUSE, Yui"=>["naruse"],
"SASADA Koichi"=>["ko1"]}
```

Since Enumerable#group_by cannot specify hash values,
it cannot do it.

> Who will be able to remember exactly what these options are,...? Most other methods, in particular on Enumerables, don't have that many options. I think that's for a good reason.

I think option argument is adopted to ruby increasingly.
--
Tanaka Akira

=end

#5 - 12/16/2010 03:01 AM - runpaint (Run Paint Run Run)

=begin

> ```
> pp ary.categorize {|e| [e[1], e[0]] }
> #=>
> {"Yukihiro Matsumoto"=>["matz"],
> "Nobuyoshi Nakada"=>["nobu"],
> "Tanaka Akira"=>["akr"],
> "Usaku NAKAMURA"=>["usa"],
> "NARUSE, Yui"=>["naruse"],
> "SASADA Koichi"=>["ko1"]}
> ```

```
ary.map.with_object({}){|e,h| h[e[1]] = [e[0]]}
=> {"Yukihiro Matsumoto"=>["matz"],
"Nobuyoshi Nakada"=>["nobu"],
"Tanaka Akira"=>["akr"],
"Usaku NAKAMURA"=>["usa"],
"NARUSE, Yui"=>["naruse"],
"SASADA Koichi"=>["ko1"]}
```

We currently have #chunk, #group_by--so to some extent #partition--and
#slice_before for segregating an Enumerable into an Enumerable of
Enumerables. I understand their difference on paper, but still need to
experiment with them in IRB before use. This proposal would add two
more methods to this list--#categorize and matz's alternative--for
what appear to be increasingly specialised uses. I'm unconvinced.

=end

#6 - 12/18/2010 02:37 PM - akr (Akira Tanaka)

=begin
2010/12/16 Run Paint Run Run runrun@runpaint.org:

> We currently have #chunk, #group_by--so to some extent #partition--and
> #slice_before for segregating an Enumerable into an Enumerable of
> Enumerables. I understand their difference on paper, but still need to
> experiment with them in IRB before use. This proposal would add two
> more methods to this list--#categorize and matz's alternative--for
> what appear to be increasingly specialised uses. I'm unconvinced.

#chunk and #slice_before collects adjasent elements in enumerable.
#group_by and #categorize collects elements from all elements in enumerable.

#categorize is more general than #group_by.
It can specify hash values.
--
Tanaka Akira

=end

#7 - 12/20/2010 04:14 PM - duerst (Martin Dürst)

=begin
Hello Akira,

On 2010/12/14 18:41, Tanaka Akira wrote:

> 2010/12/12 "Martin J. Dürst"duerst@it.aoyama.ac.jp:
>
> > There are already such methods. At least group_by. Your proposal seems to be
> > a very special way to convert an enumerable to a hash.
>
> Enumerable#categorize is more general than Enumerable#group_by.

I know. I think it's way too general and difficult to understand.

> > I think I get the idea of how this is supposed to work. But I'm not sure
> > what the actual use cases would be. Can you give some?
>
> I see.
>
> Assume a table as follows.
>
> ary = [
> ["matz", "Yukihiro Matsumoto"],
> ["nobu", "Nobuyoshi Nakada"],
> ["akr", "Tanaka Akira"],
> ["usa", "Usaku NAKAMURA"],
> ["naruse", "NARUSE, Yui"],
> ["ko1", "SASADA Koichi"]
>]
>
> Enumerable#categorize can be used to generate a hash
> from the right elements to left elements.
> (and left elements to right elements too.)
>
> pp ary.categorize {|e| [e[1], e[0]] }
> #=>
> {"Yukihiro Matsumoto"=>["matz"],
> "Nobuyoshi Nakada"=>["nobu"],
> "Tanaka Akira"=>["akr"],
> "Usaku NAKAMURA"=>["usa"],
> "NARUSE, Yui"=>["naruse"],
> "SASADA Koichi"=>["ko1"]}
>
> Since Enumerable#group_by cannot specify hash values,
> it cannot do it.

This is one example. I don't deny that such examples exist. But I think they are not so frequent, and quite varied (i.e. there are many examples where one needs "almost something like this, but not quite exactly the same).

I think that unless we can boil down your proposal to something simple that the average advanced Ruby programmer can understand and use without having to look it up or try it out in irb all the time, we have to invest some more time to find the right method.

After all, group_by was adapted by Ruby 1.9 after a lot of field experience in Rails. And I have personally used it many times, as I think others have, too.

On the other hand, do you have that much field experience? How many others have told you that they would have used categorize a few times already if it existed? (As opposed to those who have said that it's too complicated to remember what it does, and too easy to write the equivalent by hand if needed.)

An additional thought: The example above starts with two-element arrays. Such two- or multi-element arrays are often used, but in many cases they are just an intermediate step, before creating objects. group_by seems more close to using objects (that may be why it is used a lot in Rails, where the basics of model classes are almost free). On the other hand, with multi-element arrays, I think that part of what "categorize" would do will often be handled before or after. Anyway, while we should not change Ruby so that it is too difficult to use multi-element arrays instead of objects, there is also no reason to create more methods that work better for multi-element arrays.

> > Who will be able to remember exactly what these options are,...? Most other methods, in particular on Enumerables, don't have that many options. I think that's for a good reason.
>
> I think option argument is adopted to ruby increasingly.

That may be true. But even if the average number of options for a Ruby method has slightly increased recently, your proposals still is way over average on the number of options, especially in an area (iterators on Enumerable) where options are few and far between.

Regards, Martin.

--
#-# Martin J. Dürst, Professor, Aoyama Gakuin University
#-# http://www.sw.it.aoyama.ac.jp mailto:duerst@it.aoyama.ac.jp

=end

#8 - 12/25/2010 05:45 PM - akr (Akira Tanaka)

=begin
2010/12/20 "Martin J. Dürst" duerst@it.aoyama.ac.jp:

> > Enumerable#categorize is more general than Enumerable#group_by.
>
> I know. I think it's way too general and difficult to understand.

I don't think it is too difficult.

When we create hash, it is natural that we can specify keys and values.

But we can specify only keys for Enumerable#group_by.
I feel it is restricted.

> This is one example. I don't deny that such examples exist. But I think they are not so frequent, and quite varied (i.e. there are many examples where one needs "almost something like this, but not quite exactly the same).
>
> I think that unless we can boil down your proposal to something simple that

> the average advanced Ruby programmer can understand and use without having to look it up or try it out in irb all the time, we have to invest some more time to find the right method.
>
> After all, group_by was adapted by Ruby 1.9 after a lot of field experience in Rails. And I have personally used it many times, as I think others have, too.
>
> On the other hand, do you have that much field experience? How many others have told you that they would have used categorize a few times already if it existed? (As opposed to those who have said that it's too complicated to remember what it does, and too easy to write the equivalent by hand if needed.)

I transformed CSV tables variously last several monthes in my work.
Enumerable#categorize is very useful for that.
(Unfortunately, I cannot tell you the exact examples though.)

For split a CSV table: enum.categorize {|rec| [split-key, ...] }.

For merge CSV tables: Enumerable#categorize can be used for
hash-join algorism as enum.categorize {|rec| [join-key, ...] }
http://en.wikipedia.org/wiki/Hash_join

For counting number for each category: enum.categorize(:op=>:+) {|e| [key, 1] }

Also various people asks about similar hash creation.

- [ruby-talk:351947]

{day1 => [[name_person1, room1], [name_person2, room2],
[name_person3, room2]],
day2 => [[name_person1, room1], [name_person3, room1],
[name_person2, room2]]}
to
[{room1 => [{day1 => [name_person1]},
{day2 => [name_person1, name_person3]}]},
{room2 => [{day1 => [name_person2, name_person3]},
{day2 => [name_person2]}]}]

This can be implemented as:
a = orig.map {|k, aa| aa.map {|e| [k, *e] }}.flatten(1)
pp a.categorize {|e| [e[2], e[0], e[1]] }

- [ruby-talk:347364]

[["2efa4ba470", "00000005"],
["2efa4ba470", "00000004"],
["02adecfd5c", "00000002"],
["c0784b5de101", "00000006"],
["68c4bf10539", "00000003"],
["c0784b5de101", "00000001"]]
to
{"2efa4ba470" => ["00000005", "00000004"],
"02adecfd5c" => ["00000002"],
"c0784b5de101" => ["00000006", "00000001"],
"68c4bf10539" => ["00000003"]}

This can be implemented as:
orig.categorize {|e| e }

- [ruby-talk:372481]

"A", "a", 1], ["A", "b", 2], ["B", "a", 1 to
{{"A" => {"a" => 1, "b" => 2}},
{"B" => {"a" => 1}}}

This can be implemented as:
orig.categorize(:op=>lambda {|x,y| y }) {|e| e }

- [ruby-talk:288931]

[["1", "01-02-2008", 5],
["1", "01-03-2008", 10],

["2", "12-25-2007", 5],
["1", "01-04-2008", 15]]
to
{"1" => {"01-02-2008" => 5, "01-03-2008" => 10, "01-04-2008" => 15},
"2" => {"12-25-2007" => 5}}

This can be implemented as:
orig.categorize(:op=>lambda {|x,y| y}) {|e| e }

- [ruby-talk:354519]

[["200912-829", 9],
["200912-893", 3],
["200912-893", 5],
["200912-829", 1],
["200911-818", 6],
["200911-893", 1],
["200911-827", 2]]
to
[["200912-829", 10],
["200912-893", 8],
["200911-818", 6],
["200911-893", 1],
["200911-827", 2]]

This can be implemented as:
orig.categorize(:op=>:+) {|e| e }.to_a

- [ruby-talk:344723]

a=[1,2,5,13]
b=[1,1,2,2,2,5,13,13,13]
to
[0, 0], [0, 1], [1, 2], [1, 3], [1, 4], [2, 5], [3, 6], [3, 7], [3, 8

This can be implemented as:
h = a.categorize.with_index {|e, i| [e,i] }
b.map.with_index {|e, j| h[e] ? h[e].map {|i| [i,j] } : [] }.flatten(1)

- [ruby-talk:327908]

[["377", "838"],
["377", "990"],
["377", "991"],
["377", "992"],
["378", "840"],
["378", "841"],
["378", "842"],
["378", "843"],
["378", "844"]]
to
[["377", "838 990 991 992"],
["378", "840 841 842 843 844"]]

This can be implemented as:
orig.categorize(:seed=>nil, :op=>lambda {|x,y| !x ? y.dup : (x << "
" << y) }) {|e| e }

- [ruby-talk:347700]

["a", "b", "a", "b", "b"]
to
["a", "b"] [2, 3]

This can be implemented as:
h = orig.categorize(:op=>:+) {|e| [e, 1] }
p h.keys, h.values

- [ruby-talk:343511]

[1, 2, 3, 3, 3, 3, 4, 4, 5]
to
{"3"=>4, "4"=>2}

This can be implemented as:

h = orig.categorize(:op=>:+) {|e| [e, 1] }
p h.reject {|k,v| v == 1 }

I feel many people needs hash creation.
Enumerable#categorize support them.

> An additional thought: The example above starts with two-element arrays. Such two- or multi-element arrays are often used, but in many cases they are just an intermediate step, before creating objects. group_by seems more close to using objects (that may be why it is used a lot in Rails, where the basics of model classes are almost free). On the other hand, with multi-element arrays, I think that part of what "categorize" would do will often be handled before or after. Anyway, while we should not change Ruby so that it is too difficult to use multi-element arrays instead of objects, there is also no reason to create more methods that work better for multi-element arrays.

Ruby doesn't force us to create a class for programming.

I think this is a good aspect for scripting area.

> That may be true. But even if the average number of options for a Ruby method has slightly increased recently, your proposals still is way over average on the number of options, especially in an area (iterators on Enumerable) where options are few and far between.

It is natural because Enumerable is exist from old time.
I don't see any problem.
--
Tanaka Akira

=end

#9 - 12/29/2010 03:45 AM - akr (Akira Tanaka)

=begin
2010/12/27 Marc-Andre Lafortune ruby-core-mailing-list@marc-andre.ca:

> I have an alternate proposition of a modified categorize which I believe addresses the problems I see with it:
> 1) Complex interface (as was mentioned by others)

I think your 'associate' is not so simple.
Some part is more simple than 'categorize'.
Some part is more complex than 'categorize'.

> 2) By default, categorize creates a "grouped hash" (like group_by), while there is not (yet) a way to create a normal hash. I would estimate that most hash created are not of the form {key => [some list]} and I would rather have a nicer way to construct the other hashes too. This would make for a nice replacement for most "inject({}){...}" and "Hash[enum.map{...}]".

Possible.

There are 2 reasons for that I proposed a method for "grouped hash" at first.

- It doesn't lose information at key conflict.
- I (and matz) don't have a good (enough) method name for "normal hash".

I'm not sure that matz will satisfy the name 'associate'.

> My alternate suggestion is a simple method that uses a block to build the key-value pairs and an optional Proc/lambda/symbol to handle key conflicts (with the same arguments as the block of Hash#merge). I would name this simply associate, but other names could do well too (e.g. mash or graph or even to_h).

'categorize' and 'associate' differs as follows.

- 'associate' creates normal hash.

This is intentional difference.

- 'associate' doesn't create nested hash.

'associate' is simpler here.

I think 'associate' can be extended naturally that the method creates
nested hash when the block returns an array with 3 or more elements.

For the example in [ruby-talk:372481],
Your 'associate' (without above extention) solves only the nest level
but the 'categorize' solves any nest level.

```
dest == orig.categorize(:op=>lambda {|x,y| y }) {|e| e }
dest == orig.associate(:merge){|a, b, c| [a, {b=>c}]}
```

- 'associate' assumes {|v| v } if the block is not given.

This simplify some usages.
However this forbids Ruby 1.9 style enumerator creation
which returns an enumerator when block is not given.
This means we cannot write enum.associate.with_index {|v, i| ... }.

- 'associate' treates non-array block value.

This is more complex than 'categorize'.

I feel it is bit distorted specification.
Especially "(first)" in "Otherwise the value is the result of the block
and corresponding key is the (first) yielded item."

'categorize' can adopt it but I don't want.

- 'associate' doesn't use hash argument.

This may be good idea.

'categorize' needs hash argument mainly because
it must distinguish the merge function needs key or not.
(proc specified by :update needs key.
proc specified by :op don't need key.)

'associate' classify them by symbol or proc.
It can be applied for 'categorize'.

However symbol and symbol.to_proc will be different, though.

- 'associate' doesn't have a way to specify the seed.

This is simpler specification than 'categorize'
but this makes some usages more complex.

'associate' can be extended to take a second optional argument for seed.

In your 'associate' examples for [ruby-talk:347364] and
[ruby-talk:327908], array and string concatenation is O(n**2).
(n is (maximum) number of elements in a category.)

```
p dest == orig.associate(:+){|h, v| [h, [v]]}
a = [v1]
a = a + [v2]
a = a + [v3]
...

orig.associate(->(k, a, b){"#{a} #{b}"})
s = v1
s = "#{s} #{v2}"
s = "#{s} #{v3}"
...
```

To avoid this inefficiency, destructive concatenation method

can be used:

```
# or if duping the string is required (??):
orig.associate(->(k, a, b){a << " " << b}){|x, y| [x, y.dup]}
```

However the dup is required to not modify the receiver, orig.

I think seed is a simple way to avoid O(n**2) and receiver modification
without extra objects, as follows.

```
orig.categorize(:seed=>nil, :op=>lambda {|x,y| !x ? y.dup : (x <<
" " << y) }) {|e| e }
```

It could of course be argued that both associate and categorize
should be added. That may very be;

Yes.

Actually I want one more method for counting.
(I want 3 methods: grouped hash, normal hash, count hash)

I just feel that associate should
be added in priority over categorize.

matz felt similar. [ruby-dev:42643]

But we couldn't find a good name for normal hash creation method.
So the discussion is pending.

- [ruby-talk:344723]

```
a=[1,2,5,13]
b=[1,1,2,2,2,5,13,13,13]
# to
dest =
[0, 0], [0, 1], [1, 2], [1, 3], [1, 4], [2, 5], [3, 6], [3, 7], [3, 8

# This can be implemented as:
h = a.categorize.with_index {|e, i| [e,i] }
b.map.with_index {|e, j| h[e] ? h[e].map {|i| [i,j] } : [] }.flatten(1)
# or
h = a.each_with_index.associate
b.map.with_index{|e, i| [h[e], i] }
```

Your solution depends on 'a' has no duplicated elements.
Since [ruby-talk:344723] asks about INNER JOINING,
I think 'a' may have duplicated elements.

```
a=[1,1]
b=[1,1]
# to
dest = [0, 0], [1, 0], [0, 1], [1, 1
h = a.categorize.with_index {|e, i| [e,i] }
p dest == b.map.with_index {|e, j| h[e] ? h[e].map {|i| [i,j] } : []
}.flatten(1)
#=> true
h = a.each_with_index.associate
p dest == b.map.with_index{|e, i| [h[e], i] }
#=> false
--
Tanaka Akira

=end
```

#10 - 03/18/2012 06:48 PM - akr (Akira Tanaka)

- Description updated

- Assignee set to akr (Akira Tanaka)

#11 - 03/18/2012 11:01 PM - trans (Thomas Sawyer)

I'm sorry, but #categorize has big code smell. Seems like too much functionality is being shoved into one method. It is still not clear to me after reading over explanation many times how #categorize works. So either it is relatively straight forward but the explanation is overly complexed, or the method itself is too complex. The later seems more likely given the number of options, especially lambda options, the method takes.

Let's look at a given example:

- [ruby-talk:288931]

[["1", "01-02-2008", 5],
["1", "01-03-2008", 10],
["2", "12-25-2007", 5],
["1", "01-04-2008", 15]]
to
{"1" => {"01-02-2008" => 5, "01-03-2008" => 10, "01-04-2008" => 15},
"2" => {"12-25-2007" => 5}}

Implemented as:

orig.categorize(:op=>lambda {|x,y| y}) {|e| e }

Traditionally, the above would be something like:

hash = Hash.new{|h,k| h[k]={}}
list.each do |group, date, size|
hash[group][date] ||= 0
hash[group][date] += size
end

This may be longer but it is very readable. Actually, if Ruby would *finally* offer some convenience method for the first line, e.g. Hash.auto{{}} instead of Hash.new{|h,k| h[k]={}}, then

hash = Hash.auto{Hash.auto(0)}
list.each{ |group, date, size| hash[group][date] += size }

I suspect almost every case for using #categorize will be able to be treated in much the same manner.

This is not to say however that I don't think some form(s) of Enumerable -> Hash would not be useful. I think it would, in fact I sometimes use Enumerable#mash (alias #graph).

```
[1,2,3].mash{ |v| [v.to_s, v] }  #=> {'1'=>1, '2'=>2, '3'=>3}
```

But I would rather see a few methods that cover basic use cases that can work together and with other methods to build up more complex solutions then to create a single method that tries to cover every complex possibility in one go.

#12 - 03/31/2012 11:09 AM - mame (Yusuke Endoh)

- Status changed from Open to Assigned

#13 - 06/24/2012 01:03 AM - marcandre (Marc-Andre Lafortune)

- Target version set to 2.0.0

Is anyone going to submit a slide-show about this?

I'm starting to think these could be split into 3 methods:

Enumerable#index_by, #associate and #categorize.

index_by uses the result of the block as key and the enumeration as values
associate uses the result of the block as value and the enumeration as keys
categorize expects an array of keys with a value.

All methods return a hash, or an enumerator if no block is passed.
All methods accept an optional merge argument to deal with eventual key conflicts, similar to Hash#merge.
It is either a proc/lambda acting like the block of Hash#merge and taking 3 arguments (key, old_val, new_val),
or a symbol, that acts like an operator or method between two values to be merge, similarly to Enumerable#inject.

Hash would specialize associate (| index_by) so that the key (| value) used would be only the key (| value), not the [key, value] pair:

{:hello => 42}.index_by{|k, v| k.to_s} # => {'hello' => 42}, not {'hello' => [:hello, 42]}
{:hello => 42}.associate{|k, v| v.to_s} # => {:hello => "42"}, not {[:hello, 42] => "42"}

Note: Rails' ActiveSupport already defines index_by; this index_by is just a refinement of it.

#14 - 06/24/2012 02:40 AM - trans (Thomas Sawyer)

Could you give an example of #categorize?

Also, I thought this method(s) purpose is to convert Enumerable to Hash not just Hash to Hash. Examples with Arrays would be good too.

Glancing at ActiveSupport's #index_by, I am not sure about implementation. Can it even handle |key,value| pairs?

https://github.com/rails/rails/blob/d696f8de92ceb3cb38b11ec3b200b082b9578742/activesupport/lib/active_support/core_ext/enumerable.rb#L94

For Hash, I have always liked Facets #rekey which can take |key| or |key,value| in block. It can also take a conversion hash, e.g.

{:a=>1}.rekey(:a=>:b) #=> {:b=>1}

The #index_by name is okay for Enumerable, although a little confusing with Array#index --by name it is easy to think it reorders Enumerable into an Array based on an index number returned from the block.

For Enumerable-to-Hash, Facets defines every simple variety: splat, flat, multi and assoc. There is also the default auto type which looks at the block and the receiver to select one of these four.

https://github.com/rubyworks/facets/blob/master/lib/core/facets/to_hash.rb

I realize the method names like #to_h_splat" aren't as pretty but they self document better then having to learn what is meant by less specific terms like "associate" and "categorize". However, this implementation is not the best approach either b/c its original code was written before Enumerator and Lazy came along. It does help flesh ideas/approaches, but I think it can be done better now.

Perhaps the best approach --one that would simplify the interface, still provide all the capabilities and have clear semantically named methods, would be via intermediate methods to get the enumerable in a prepared state, followed by a simple call of #to_h. For example, flatten is simplest, enum.flatten.to_h (although there is issue of odd number elements to consider). Something likewise might be possible for the rest. Is that doable?

#15 - 06/24/2012 04:20 AM - marcandre (Marc-Andre Lafortune)

Hi,

trans (Thomas Sawyer) wrote:

> Could you give an example of #categorize?

There are many already in the discussion... A simple one:

```
[:a, "foo", :a, :a].categorize(:+){|i| ["occurences of #{i}", 1]} # => { "occurences of a" => 3,  "occurences of foo" => 1}
```

> Also, I thought this method(s) purpose is to convert Enumerable to Hash not just Hash to Hash. Examples with Arrays would be good too.

They do convert Enumerable to Hash. Hash just has specialized versions, like Hash#select is a specialized version of Enumerable#select.

> Glancing at ActiveSupport's #index_by, I am not sure about implementation. Can it even handle |key,value| pairs?

Not sure what your question is, but that implementation is not relevant to the discussion. Here's a Ruby implementation:

```
module Enumerable
  def index_by(merge = nil)
    return to_enum(merge) unless block_given?
    if merge.is_a?(Symbol)
      method = merge
      merge = ->(_, before, after) { before.send(method, after) }
    end
    h = {}
    each do |val|
      key = yield val
      h[key] = if h.has_key?(key) && merge
                 merge[key, h[key], val]
               else
                 val
               end
    end
    h
  end
end

class Hash
```

```
  def index_by(merge = nil)
    # same...
    each do |key, val|
      key = yield key, val
    # same ...
  end
end
```

> For Hash, I have always liked Facets #rekey which can take |key| or |key,value| in block.

I don't know of a single core method that behaves differently depending of the arity of the block.

> Perhaps the best approach (...) would be via intermediate methods to get the enumerable in a prepared state, followed by a simple call of #to_h.

Matz is not positive about Array#to_h. I wish I could convince him that Hash[ary] is so much uglier than ary.to_h.

#16 - 06/24/2012 12:54 PM - trans (Thomas Sawyer)

marcandre (Marc-Andre Lafortune)

> There are many already in the discussion... A simple one:

So you are referring then the original #categorize method you proposed? That being so, I remain convinced it is far too complicated.

> I don't know of a single core method that behaves differently depending of the arity of the block.

Actually it might be able to work via a splat, no arity needed. Even so, I don't see how that's an issue, it's implementation detail and there's no good reason to require two different methods to do the same thing b/c of this. Besides, method definitions often handle variations in arity.

> Matz is not positive about Array#to_h.

Matz has accepted #to_h.

Looking at your implementation of #index_by, it needs some improvement, but that's just detail. The main thing is the merge functionality. Speaking of uncommon behavior, taking a lambda as an non-block argument is generally not considered a good approach. I think it is code smell, and typically means the method needs to be broken down into two method calls.

#17 - 07/01/2012 01:04 AM - Eregon (Benoit Daloze)

Reading this again (and what I could understand from the ruby-dev part), I think this is a very important feature, and I also believe one that has much been asked.

I think the need for the Hash part of Array#map is the clearer. I was initially thinking to have one generic method to avoid n specialized methods, but I think the "map" kind Hash primitives should be added first.
These are (these names are just to make the purpose clear): map_value, map_key and map_pair. map_pair might be very much replaced by categorize/associate though, so it might not be worth to add it.

There is the alternative to use #each_with_object, but I think it is not even close to be as clear and readable:
h.each_with_object({}) { |(k,v),h| h[k] = v * 2 }
h.map_value { |v| v * 2 }

I think map_value and map_key are worth as is, because building an Array for changing only the keys or the values is not showing well what the user wants to achieve, nor very readable. And I think checking the block return value to see if it is an Array is not a good specification, as you can not have easily an Array as value anymore.

About categorize and associate, I agree the default should rather be "last value" than an Array, even though losing information is a bad default.

I think categorize interface could be simplified a little, by having an interface closer to #inject. That is categorize(init = nil, sym_or_proc). I believe only one of :op and :update should be kept, to simplify the interface (categorize's merge proc should not know about the keys I think, otherwise, one might just do { |init, (*keys, value)| } but it's not very nice).

marcandre wrote:

> Is anyone going to submit a slide-show about this?

I'm wishing to do so. I don't expect associate or categorize to be decided at the meeting, but I think it would be a nice occasion to go on with this and find out what should be added.

#18 - 07/01/2012 04:01 AM - marcandre (Marc-Andre Lafortune)

Eregon (Benoit Daloze) wrote:

> Reading this again (and what I could understand from the ruby-dev part), I think this is a very important feature, and I also believe one that has much been asked.
>
> I think the need for the Hash part of Array#map is the clearer. I was initially thinking to have one generic method to avoid n specialized methods, but I think the "map" kind Hash primitives should be added first.

I agree about the importance and its order.

> These are (these names are just to make the purpose clear): map_value, map_key and map_pair. map_pair might be very much replaced by categorize/associate though, so it might not be worth to add it.

Only problem I see with map is that it produces arrays (also flat_map).

> marcandre wrote:
>
> > Is anyone going to submit a slide-show about this?
>
> I'm wishing to do so. I don't expect associate or categorize to be decided at the meeting, but I think it would be a nice occasion to go on with this and find out what should be added.

I'll summarize my proposal of [ruby-core:45809] in a slide and we'll see.

trans (Thomas Sawyer) wrote:

> > Matz is not positive about Array#to_h.
>
> Matz has accepted #to_h.

Check again. There is now nil.to_h, Hash#to_h, etc..., but no Array#to_h.

> Speaking of uncommon behavior, taking a lambda as an non-block argument is generally not considered a good approach.

If you need two blocks, there's no other choice. See Enumerable#find.

#19 - 07/01/2012 04:48 AM - marcandre (Marc-Andre Lafortune)

- File associate.pdf added

Attaching one-minute slide-show

#20 - 07/01/2012 04:57 AM - Eregon (Benoit Daloze)

- File feature4151.pdf added

marcandre (Marc-Andre Lafortune) wrote:

> Attaching one-minute slide-show

Attaching mine.
I hope both can be shown if it makes sense, after all there are many features about this.

#21 - 07/02/2012 02:05 AM - mame (Yusuke Endoh)

Marc-Andre and Eregon, your slides (!) are received. Thank you!

--
Yusuke Endoh mame@tsg.ne.jp

#22 - 07/08/2012 09:18 PM - trans (Thomas Sawyer)

=begin
I have given this subject matter considerable thought, and I believe the best definition is as follows:

Apply each element of an enumerable to a hash

```
# by iterating over each element and yielding
# the hash and element.
#
#   [1,2,3].categorize{|h,e| h[e] = e+1 }
#   #=> {1=>2, 2=>3, 3=>4}
#
def categorize(init={})
h = init
each{ |v| yield(h,v) }
h
end
```

This achieves the feature --to create a hash from an enumerable, while still allowing control of the merge procedure (and without any need for an extra :op procedure argument).

Yes, I know my docs for the method suck. Feel free to explain it better.

On a side note, ActiveSupport's #index_by method wouldn't be a bad addition to core either, although a more percise name might be #key_by (b/c "index" commonly refers to array's numerical indexes, as opposed to hash's "key").
=end

#23 - 07/08/2012 09:28 PM - trans (Thomas Sawyer)

One further point... whether #categorize is still the best name for this given the definition seems a reasonable question. An alternative might be #hinge, in the sense that it hangs consecutive elements on to a hash.

#24 - 07/08/2012 10:56 PM - Eregon (Benoit Daloze)

trans (Thomas Sawyer) wrote:

> I have given this subject matter considerable thought, and I believe the best definition is as follows:
>
> ```
> # Apply each element of an enumerable to a hash
> # by iterating over each element and yielding
> # the hash and element.
> #
> #   [1,2,3].categorize{|h,e| h[e] = e+1 }
> #   #=> {1=>2, 2=>3, 3=>4}
> #
> def categorize(init={})
> h = init
> each{ |v| yield(h,v) }
> h
> end
> ```

That's Enumerable#each_with_object with
arguments reversed and a default {} for obj:

```
[1,2,3].categorize { |h,e| h[e] = e+1 }

[1,2,3].each_with_object({}) { |e,h| h[e] = e+1 }
```

It's shorter, mostly because of #each_with_object length (#6687)

#index_by is #group_by with a "keep last" merging strategy.
Categorize can do all of this, and much more. That's why I think it's interesting.
But I agree we need simpler methods for most use cases.

#25 - 07/08/2012 11:45 PM - trans (Thomas Sawyer)

After a nap I awoke realizing the exact same thing. Which leads me to think likewise, we need such a method --a new "each_with_object" method with very concise name and an the initial object defaulting to {}. Probably also with the block parameters reversed to be like #inject, though that is not as important as the former two points.

#26 - 07/10/2012 12:30 AM - alexeymuranov (Alexey Muranov)

=begin
I would add a new method to initialize or fill (({Hash})): (({Hash::new(enum) { |k| value_for_key(k) }}))

Also, how about:

Object#tap_for_each(enum) { |o, e| do something to o using e }

For example:

hash = {}.tap_for_each([1, 2, 3]) { |h, e| h[e] = e + 10 }

The implementation is the following:

class Object
def tap_for_each(enum)
enum.each { |e| yield(o, e) }; self
end
end

This is not shorter than each_with_object, but reads differently.

Also the following can be implemented:

Object#tap_for_each(enum1, enum2, ...) { |o, e1, e2, ...| do something to o using e1, e2,... }

Updated 2012-07-13.
=end

#27 - 07/23/2012 11:47 PM - mame (Yusuke Endoh)

We discussed your slide at the developer meeting (7/21), but
cannot reach agreement. What is proposed in both Benoit and
MarcAndre slides, was not concrete enough.

One comment: in general, a collection of small methods that do
one thing is better than one big method that can do anything.

Akr, who is the original proposer, will revise the proposal
according to matz's comment.

--
Yusuke Endoh mame@tsg.ne.jp

#28 - 10/28/2012 11:11 PM - akr (Akira Tanaka)

- Target version changed from 2.0.0 to 2.6

#29 - 10/22/2017 01:57 AM - akr (Akira Tanaka)

- Status changed from Assigned to Rejected

I think this is too complex proposal, now.

Files

associate.pdf	145 KB	07/01/2012	marcandre (Marc-Andre Lafortune)
feature4151.pdf	78.7 KB	07/01/2012	Eregon (Benoit Daloze)