Ruby trunk - Feature #5663

Combined map/select method

11/23/2011 05:37 AM - wycats (Yehuda Katz)

Status:	Assigned	
Priority:	Normal	
Assignee:	matz (Yukihiro Matsumoto)	
Target version:		

Description
It is pretty common to want to map over an Enumerable, but only include the elements that match a particular filter. A common idiom is:
enum.map { |i| i + 1 if i.even? }.compact
It is of course also possible to do this with two calls:
enum.select { |i| i.even? }.map { |i| i + 1 }
Both cases are clumsy and require two iterations through the loop. I'd like to propose a combined method:
enum.map_select { |i| i + 1 if i.even? }
The only caveat is that it would be impossible to intentionally return nil here; suggestions welcome. The naming is also a strawman; feel free to propose something better.

Related issues:	
Related to Ruby trunk - Feature #13784: Add Enumerable#filter as an alias of ...	**Closed**
Related to Ruby trunk - Feature #15323: [PATCH] Proposal: Add Enumerable#filt...	**Closed**

History

#1 - 11/23/2011 05:43 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

I like the idea but not the name. Maybe something like select_non_nil would be more appropriate. English is not my mother tongue, is there any English way to say this in a shorter way?

#2 - 11/23/2011 05:45 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Or maybe just enum.filter?

#3 - 11/23/2011 06:23 AM - aprescott (Adam Prescott)

Instead of looking at it from the map + select approach for a name, what about the map + compact version? compact_map ?

#4 - 11/23/2011 06:53 AM - wycats (Yehuda Katz)

Compact implies two passes, no?

Yehuda Katz
(ph) 718.877.1325

On Tue, Nov 22, 2011 at 1:15 PM, Adam Prescott adam@aprescott.com wrote:

> Instead of looking at it from the map + select approach for a name, what
> about the map + compact version? compact_map ?

#5 - 11/23/2011 07:29 AM - shugo (Shugo Maeda)

- ruby -v changed from ruby 1.9.3p0 (2011-11-08 revision 33661) [x86_64-darwin11.2.0] to -

Hi,

2011/11/23 Yehuda Katz wycats@gmail.com:

It is pretty common to want to map over an Enumerable, but only include the elements that match a particular filter. A common idiom is:

enum.map { |i| i + 1 if i.even? }.compact

It is of course also possible to do this with two calls:

enum.select { |i| i.even? }.map { |i| i + 1 }

Both cases are clumsy and require two iterations through the loop. I'd like to propose a combined method:

enum.map_select { |i| i + 1 if i.even? }

The only caveat is that it would be impossible to intentionally return nil here; suggestions welcome. The naming is also a strawman; feel free to propose something better.

How about to add list comprehensions or Scala's for expressions instead?
For example, enum.select { |i| i.even? }.map { |i| i + 1 } can be
written as follows:

[i + 1 for i in enum if i.even?]

The syntax of list comprehensions needs more considerations.

One benefit is that nested maps can be flattened by list comprehensions.
For example, the following code:

pyths = [[x, y, z] for z in [1..Float::INFINITY].defer
x in [1..z].defer
y in [x..z].defer
if x*2 + y2 == z*2]
p pyths.take(3)

is equivalent to the following code:

pyths = (1..Float::INFINITY).defer.flat_map {|z|
(1..z).defer.flat_map {|x|
(x..z).defer.select {|y|
x*2 + y2 == z*2
}.map {|y|
[x, y, z]
}
}
}
p pyths.take(3)

Enumerable#defer is proposed in Feature #4890.

BTW, now Ruby has map and reduce as aliases of collect and inject,
but not filter as an alias of select. Why not?

--
Shugo Maeda

#6 - 11/23/2011 07:31 AM - nahi (Hiroshi Nakamura)

- Tracker changed from Bug to Feature

Bug -> Feature

#7 - 11/23/2011 08:55 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Hi Shugo, I also like the idea of supporting list comprehensions like several other languages currently do.

#8 - 11/23/2011 09:23 AM - wycats (Yehuda Katz)

I am nervous about list comprehensions because in almost all cases, when
you do something in Ruby, you do it by invoking a named method on an
object. It is clear by looking at a piece of code which named method will
be invoked. In Ruby 1.9, it is even trivial to learn the exact source
location of such a named method.

In contrast, list comprehensions introduce new syntax that invokes some
invisible protocol; understanding which methods are involved requires
figuring out where to look in the documentation.

Python uses protocols like this for *everything*, so list comprehensions
fit in well there. The only Ruby case I can think of that works like this
is the much maligned for/in syntax, which invokes #each under the hood.
Most people I know find this strange, I suspect because of its
inconsistency with the very strong rule that if a method is directly
invoked by some syntax, you can see it.

If you want to see where this protocol-oriented path leads us, check out
http://docs.python.org/reference/datamodel.html#special-method-names. I
much prefer Ruby's "if you want to do something, invoke a named method"
principle of uniform access[1].

[1] http://en.wikipedia.org/wiki/Uniform_access_principle

Yehuda Katz
(ph) 718.877.1325

On Tue, Nov 22, 2011 at 2:26 PM, Shugo Maeda shugo@ruby-lang.org wrote:

> Hi,
>
> 2011/11/23 Yehuda Katz wycats@gmail.com:
>
>> It is pretty common to want to map over an Enumerable, but only include
>> the elements that match a particular filter. A common idiom is:
>>
>> enum.map { |i| i + 1 if i.even? }.compact
>>
>> It is of course also possible to do this with two calls:
>>
>> enum.select { |i| i.even? }.map { |i| i + 1 }
>>
>> Both cases are clumsy and require two iterations through the loop. I'd
>> like to propose a combined method:
>>
>> enum.map_select { |i| i + 1 if i.even? }
>>
>> The only caveat is that it would be impossible to intentionally return
>> nil here; suggestions welcome. The naming is also a strawman; feel free to
>> propose something better.
>
> How about to add list comprehensions or Scala's for expressions instead?
> For example, enum.select { |i| i.even? }.map { |i| i + 1 } can be
> written as follows:
>
> [i + 1 for i in enum if i.even?]

The syntax of list comprehensions needs more considerations.

> One benefit is that nested maps can be flattened by list comprehensions.
> For example, the following code:
>
> pyths = [[x, y, z] for z in [1..Float::INFINITY].defer
> x in [1..z].defer
> y in [x..z].defer
> if x*2 + y2 == z*2]
> p pyths.take(3)
>
> is equivalent to the following code:
>
> pyths = (1..Float::INFINITY).defer.flat_map {|z|
> (1..z).defer.flat_map {|x|
> (x..z).defer.select {|y|
> x*2 + y2 == z*2
> }.map {|y|
> [x, y, z]
> }
> }
> }
> p pyths.take(3)

Enumerable#defer is proposed in Feature #4890.

BTW, now Ruby has map and reduce as aliases of collect and inject,
but not filter as an alias of select. Why not?

--
Shugo Maeda

#9 - 11/23/2011 07:33 PM - alexeymuranov (Alexey Muranov)

It seems that in full generality this method needs to accept two blocks: one for selecting and one for mapping, but this would be an unusual syntax.

So how about a lazy #selecting first, which would store a block for selecting inside enum, and to make #map check if a block for selecting is defined, and to use it? Maybe, instead of changing #map, a new variant of #map can be created that would take lazy operations into account. (For the name, i would propose #partial_map or #map_partially (http://en.wikipedia.org/wiki/Partial_function), or #map_selected.) Then the code would look like this:

enum.selecting { |i| i.even? }.map_selected { |i| i + 1 }

and only one loop will be needed.

I've read about gems defining lazy methods for Enumerable, but i do not remember if any of them is doing exactly this.
Update Here is a link to a description of Lazing project:
http://blog.gregspurrier.com/articles/lazing-lazy-enumerable-methods-for-ruby-1-9

-Alexey.

Update   If the goal is not to solve this in full generality, but to optimize for one use case, then what about #map_and_compact ? I think it is ok to have relatively long names for methods that solve relatively restricted problems.

#10 - 11/24/2011 12:21 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Alexey, what about thread safety in this case? Would the selecting return a new object?

#11 - 11/24/2011 01:53 AM - alexeymuranov (Alexey Muranov)

Rodrigo Rosenfeld Rosas wrote:

> Alexey, what about thread safety in this case? Would the selecting return a new object?

I do not know much about threads. I think #selecting can return a new object, and #selecting! can add lazy selection to the object on which it is called. Those more knowledgable please comment with something wiser.

#12 - 11/24/2011 09:54 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

I was just suggesting that maybe avoiding laziness for this case would be less of a trouble since we wouldn't need to worry about thread safety...

#13 - 11/24/2011 04:12 PM - nobu (Nobuyoshi Nakada)

- Category set to lib

- Target version set to 2.0.0

```
=begin
What about:
(1..10).grep(->(i){i.even?}){|i|i+1}
or
(1..10).grep(:even?.to_proc){|i|i+1}
=end
```

#14 - 11/24/2011 05:49 PM - trans (Thomas Sawyer)

Related methods:

- filter (http://rubyworks.github.com/facets/doc/api/core/Enumerable.html#method-i-filter)
- purge (http://rubyworks.github.com/facets/doc/api/core/Enumerable.html#method-i-purge)

#15 - 11/24/2011 08:41 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Nobuyoshi, wouldn't &:even? be equivalent to :even?.to_proc? I just find that the example reads better this way ;)

(1..10).grep(&:even?){|i|i+1}

But anyway, I don't find that "grep" is self-explanatory, although I like the idea of passing two procs.

#16 - 11/24/2011 08:43 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

How can I find the syntax for highlighting the code in this Redmine instance?

#17 - 11/25/2011 12:00 PM - nobu (Nobuyoshi Nakada)

Rodrigo Rosenfeld Rosas wrote:

> Nobuyoshi, wouldn't &:even? be equivalent to :even?.to_proc? I just find that the example reads better this way ;)

They are different.
&expr calls #to_proc method on the result of expr, to achieve a Proc object.

> (1..10).grep(&:even?){|i|i+1}

It's a syntax error.

#18 - 11/28/2011 10:23 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Em 25-11-2011 01:00, Nobuyoshi Nakada escreveu:

> Issue [#5663](#) has been updated by Nobuyoshi Nakada.
>
> Rodrigo Rosenfeld Rosas wrote:
>
> > Nobuyoshi, wouldn't&:even? be equivalent to :even?.to_proc? I just find that the example reads better this way ;)
> > They are different.
> > &expr calls #to_proc method on the result of expr, to achieve a Proc object.
> >
> > (1..10).grep(&:even?){|i|i+1}
> > It's a syntax error.

Wow! Thanks! This is unexpected to me. I didn't know about the difference.

Anyway, I still didn't understand why (1..10).grep(&:even?) works, but
(1..10).grep(&:even?){|i| i+1} not.

Actually, I didn't understand your explanation. You said that &expr
calls #to_proc on the result of expr. What is the expr on each example?
Is there any place I could further read about those differences?

Thank you, Rodrigo.

#19 - 11/28/2011 11:23 AM - nobu (Nobuyoshi Nakada)

Hi,

(11/11/28 10:05), Rodrigo Rosenfeld Rosas wrote:

> &expr calls #to_proc method on the result of expr, to achieve a Proc object.

achieve a Proc object, and convert it into a block.

> Anyway, I still didn't understand why (1..10).grep(&:even?) works, but (1..10).grep(&:even?){|i| i+1} not.

The former is equivalent to

(1..10).grep {|i| i.even?}

You can't pass two or more blocks to one method call.

> Actually, I didn't understand your explanation. You said that &expr calls #to_proc on the result of expr. What is the expr on each example? Is
> there any place I could further read about those differences?

expr is :even? symbol literal here.

--
Nobu Nakada

#20 - 11/28/2011 09:53 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Em 27-11-2011 23:59, Nobuyoshi Nakada escreveu:

> ...
>
>> Anyway, I still didn't understand why (1..10).grep(&:even?) works, but (1..10).grep(&:even?){|i| i+1} not.
>> The former is equivalent to
>
> (1..10).grep {|i| i.even?}
>
> You can't pass two or more blocks to one method call.

Ah, ok, thanks. So this is actually the explanation why the other
example won't work.

Maybe if Ruby accepted blocks as parameters, like Groovy does and as
José Valim has proposed in a talk at RubyConf Brazil, that could be
achievable as

(1..10).grep&:even?, {|i| i+1}

Has this feature already been discussed?

#21 - 11/29/2011 08:30 AM - alexeymuranov (Alexey Muranov)

Rodrigo Rosenfeld Rosas wrote:

> Em 25-11-2011 01:00, Nobuyoshi Nakada escreveu:
>
>> Issue #5663 has been updated by Nobuyoshi Nakada.
>>
>> Rodrigo Rosenfeld Rosas wrote:
>>
>>> Nobuyoshi, wouldn't&:even? be equivalent to :even?.to_proc? I just find that the example reads better this way ;)
>>> They are different.
>>> &expr calls #to_proc method on the result of expr, to achieve a Proc object.
>>>
>>> (1..10).grep(&:even?){|i|i+1}
>>> It's a syntax error.
>
> Wow! Thanks! This is unexpected to me. I didn't know about the difference.
>
> Anyway, I still didn't understand why (1..10).grep(&:even?) works, but
> (1..10).grep(&:even?){|i| i+1} not.
>
> Actually, I didn't understand your explanation. You said that &expr
> calls #to_proc on the result of expr. What is the expr on each example?
> Is there any place I could further read about those differences?
>
> Thank you, Rodrigo.

I am not a specialist, but it seems that you can only use ampersand in a def with the *last* parameter, and in such case this last parameter becomes
the name of a block passed to the method. Similarly, when you call a method, you can either supply a block, or pass a proc as a block by adding it as
the last argument prefixed with ampersand, but not both, because you cannot pass two blocks to a method (but you can pass multiple procs). So
basically only one ampersand is allowed, and only if you do not supply a block.

Here is a link about blocks and procs, but it does not explain this detail:
http://www.robertsosinski.com/2008/12/21/understanding-ruby-blocks-procs-and-lambdas/

#22 - 11/29/2011 08:32 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Hi Alexey,

Yes, I know about this. What José Valim suggested (well, he actually wondered, since he doesn't believe this could be changed in Ruby due to the
large existent code base) was the removal of the block concept. From what I've understood, the block syntax would actually create a proc (or a
lambda) that would be passed as a parameter. And I also find this less confusing, with the similar callable approach taken by C, JavaScript, Python
and most languages out there I guess.

#23 - 12/01/2011 05:29 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Em 30-11-2011 05:41, Ondřej Bílka escreveu:

> On Tue, Nov 29, 2011 at 08:32:01PM +0900, Rodrigo Rosenfeld Rosas wrote:
>> ... What José Valim suggested (well, he actually wondered, since he doesn't believe this could be changed in Ruby due to the large existent code base) was the removal of the block concept. From what I've understood, the block syntax would actually create a proc (or a lambda) that would be passed as a parameter. And I also find this less confusing, with the similar callable approach taken by C, JavaScript, Python and most languages out there I guess.
>> It already is. For example
>> def foo(&p)
>> p.call
>> end

This is not what I'm talking about. The other callable languages
wouldn't have special treatments for blocks like block_given? or yield,
for instance.

I'm talking about "method(some_param){some_block 'here'}" being exactly
equal to "method some_param, lambda {some_block 'here'}" (or using a
proc instead of a lambda, I'm not sure about which one would be a better
fit).

#24 - 12/29/2011 12:03 PM - kernigh (George Koehler)

=begin
Nobuyoshi Nakada wrote:

> What about:
> (1..10).grep(->(i){i.even?}){|i|i+1}
> or
> (1..10).grep(:even?.to_proc){|i|i+1}

This looks smart. One can also write it as
(1..10).grep((proc &:even?), &:succ)
but that might be too ugly.

I forgot about #grep, so I would have used #flat_map:
(1..10).flat_map{|i| i.even?? [i + 1] : []}

I like #flat_map, but it might have a disadvantage: it must allocate several new zero-element and one-element arrays.
=end

#25 - 02/01/2012 02:53 AM - Anonymous

> It is pretty common to want to map over an Enumerable, but only include the elements that match a particular filter. A common idiom is:
>
> enum.map { |i| i + 1 if i.even? }.compact
>
> It is of course also possible to do this with two calls:
>
> enum.select { |i| i.even? }.map { |i| i + 1 }
>
> Both cases are clumsy and require two iterations through the loop. I'd like to propose a combined method:

+1 for a combined map and select. I find myself reaching for it every
so often and wishing it were there.

#26 - 02/01/2012 11:18 AM - trans (Thomas Sawyer)

In Facets it's called #compact_map, but that's only b/c I thought better of monkey patching #compact itself.

I think it would be a slick feature for #compact to take a block.

#27 - 02/01/2012 11:23 AM - trans (Thomas Sawyer)

In Facets it's called #compact_map, but that's only b/c I though better of
monkey patching #compact itself.

I think it would be a slick addition for #compact to take a block.

#28 - 02/01/2012 12:53 PM - nobu (Nobuyoshi Nakada)

Hi,

(12/02/01 11:18), Thomas Sawyer wrote:

> In Facets it's called #compact_map, but that's only b/c I thought better of monkey patching #compact itself.

> I think it would be a slick feature for #compact to take a block.

Mere "compact" doesn't feel implying "map".

--
Nobu Nakada

#29 - 02/01/2012 01:10 PM - funny_falcon (Yura Sokolov)

I often wish to have methods, which likes to inject but do use return value of block for next iteration:

```
class Enumerable
  def accum(container)
    each{|args| yield container, args}
    container
  end
end
```

Then I could use it in following ways:
instead of

```
enum.inject({}){|h, k| h[k] = true; h}
enum.map{|ar| ar.select{|i| i.usefull?}}.flatten(1)
enum.map{|i| i + 1 if i.even?}.compact
```

I could

```
enum.accum({}){|h,k| h[k] = true}
enum.accum([]){|res, ar| ar.each{|i| res << i if i.usefull?}}
enum.accum([]){|res, i| res << i + 1 if i.even?}
```

Well, it is shorter only in case of inject, but I still will prefer such method, cause I don't stress GC with many short living arrays.
And maybe two common container-s could have separate methods:

```
class Enumerable
  def accum_hash(&block)
    accum({}, &block)
  end
  def accum_ar(&block)
    accum([], &block)
  end
end
```

```
enum.accum_hash{|h,k| h[k] = true}
enum.accum_array{|res, ar| ar.each{|i| res << i if i.usefull?}}
enum.accum_array{|res, i| res << i + 1 if i.even?}
```

#30 - 02/01/2012 02:11 PM - nobu (Nobuyoshi Nakada)

Yura Sokolov wrote:

> I often wish to have methods, which likes to inject but do use return value of block for next iteration:

Try each_with_object.

#31 - 02/01/2012 08:41 PM - trans (Thomas Sawyer)

@Nakada Why not?

"Returns a copy of self with all nil elements removed. If block is given, returns the yield of each member sans those that yield nil."

[1,2,3,4].compact{ |x| x.even? ? x*2 : nil } #=> [4,16]

Seems clear enough to me. Besides, I think we'd be quite hard pressed to find some other term that implies both "map" and "compact".

#32 - 02/13/2012 08:12 PM - mame (Yusuke Endoh)

- Status changed from Open to Rejected

In #708, Matz approved a related ticket #4890.
Let's discuss there.

--
Yusuke Endoh mame@tsg.ne.jp

#33 - 02/13/2012 08:43 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

I don't think #4890 removes the need for this ticket. This is about API, not implementation. Even if we can use lazy maps/collects to implement this API, I still think it should exist.

#34 - 02/13/2012 08:51 PM - mame (Yusuke Endoh)

- Status changed from Rejected to Assigned

- Assignee set to matz (Yukihiro Matsumoto)

Okay reopened.

Do you mean we need map_select as a shorthand?

--
Yusuke Endoh mame@tsg.ne.jp

#35 - 02/13/2012 09:09 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Yes, Yusuke, exactly. Thanks for reopening.

#36 - 02/14/2012 12:30 AM - trans (Thomas Sawyer)

One of the problems with this method is that in becomes impossible to get nil or false as a valid selection/non-compaction.

Couple thoughts on handling this. For starters maybe #compact should be able to take an argument, e.g.

[1,2,3,1].compact(1) => [2,3]

(This in itself seems pretty useful, regardless.)

Then a #compact_map could do the same. If we need to be able to get nil/false results we could then use an alternative.

foo = Object.new

[0,1,-1].compact_map(foo){ |e| e.zero? ? foo : (e > 0 ? true : false) } #=> [true,false]

(Here's a case where some sort of global null instance might be useful, in place of foo.)

But then again, a delayed enumeration is about as concise, and maybe a bit easier to read.

[0,1,-1].denum.select{ |e| !e.zero? }.map{ |e| e > 0 ? true : false }.to_a #=> [true,false]

Something to consider.

#37 - 02/14/2012 04:45 AM - jballanc (Joshua Ballanco)

What about using "next" for this purpose? Currently:

```
(1..10).to_a.map { |i| i % 3 == 0 ? next : i**2 }
=> [1, 4, nil, 16, 25, nil, 49, 64, nil, 100]
```

Is it so common to use "next" with a return value inside of an Enumerable#map call? Even so, could we special case "next" so that instead of returning nil, it skips the element entirely?

#38 - 02/14/2012 05:17 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Thomas, I didn't find your example for justifying a "null" instance good enough. Why not just using "nil" instead of "foo"?

But I liked your suggestion on "compact" allowing an argument defaulting to nil.

Joshua, I like the idea of next inside a map doing the compact thing, but I guess this will be hard to get approval for from Ruby Core team.

It would be a surprising feature in Ruby. Ruby is usually a pretty concise language with very little surprises... I'm not sure if they'll want to introduce such different "next" behavior only for the map/collect method.

#39 - 02/14/2012 05:48 AM - trans (Thomas Sawyer)

@rodrigo

Your right, I could have used nil, so my example wasn't a particularly good one for what I was trying to demonstrate. Here's a better demonstration:

```
[0,1,-1,2].compact_map(foo) do |e|
case e
when  0 then nil
when  1 then true
when -1 then false
else       foo
end
end
#=> [nil,true,false]
```

#40 - 02/14/2012 05:52 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

I supposed that was your point, but I waited for your new example, before talking about this.

The point is that usually we want some examples from the real world for considering some feature.

So, I don't find this example likely to actually happen on real software. Could you please provide some real scenario where having nils in the result would be better than any other values?

#41 - 03/05/2012 11:47 AM - funny_falcon (Yura Sokolov)

2012/2/1 Nobuyoshi Nakada nobu@ruby-lang.org

> Issue #5663 has been updated by Nobuyoshi Nakada.

> Yura Sokolov wrote:

>> I often wish to have methods, which likes to inject but do use return
>> value of block for next iteration:

> Try each_with_object.

Yeah, it is good :) though it have a long name.

Yura

#42 - 04/01/2012 02:39 AM - matz (Yukihiro Matsumoto)

- Status changed from Assigned to Feedback

I am OK with the original map_select behavior, but I don't like the name #map_select.
It is combination of -ect family name with map.

I prefer the name #filter, but it might be confused by simple alias of select.
Any idea?

Matz.

#43 - 04/01/2012 11:18 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

I actually prefer map_select as it is self-explanatory. There isn't a single word that is able to clearly provide the description for what it does.

#44 - 04/01/2012 11:53 AM - nobu (Nobuyoshi Nakada)

Hi,

(12/02/01 18:14), Юрий Соколов wrote:

> Try each_with_object.

>> Yeah, it is good :) though it have a long name. It's pity that 1.9.3 hadn't
>> it.

It's long indeed, but 1.9.3 had it.

$ ./ruby -v -e 'p((1..3).each_with_object({}){|x,a|a[x]=x+1})'
ruby 1.9.3p0 (2011-10-30 revision 33569) [x86_64-darwin11.3.0]
{1=>2, 2=>3, 3=>4}

--
Nobu Nakada

#45 - 04/02/2012 02:40 AM - trans (Thomas Sawyer)

Maybe #select_yield, since that what it is doing.

Which reminds me, is there an equivalent #find method to this (e.g. #find_yield), that returns the yield result instead of the element?

#46 - 04/13/2012 11:21 AM - shugo (Shugo Maeda)

matz (Yukihiro Matsumoto) wrote:

> I am OK with the original map_select behavior, but I don't like the name #map_select.
> It is combination of -ect family name with map.
>
> I prefer the name #filter, but it might be confused by simple alias of select.
> Any idea?

I think filter should be a simple alias of select.

Otherwise, we can't complain about Scala's collect:p

How about filter_map?

I've found that Scheme has filter-map.
From SRFI-1:

filter-map f clist1 clist2 ... -> list
Like map, but only true values are saved.
(filter-map (lambda (x) (and (number? x) (* x x))) '(a 1 b 3 c 7))
=> (1 9 49)
The dynamic order in which the various applications of f are made is not specified.
At least one of the list arguments must be finite.

#47 - 04/13/2012 08:51 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

+1

#48 - 05/31/2012 07:48 AM - alexeymuranov (Alexey Muranov)

=begin
By the way, is anybody using the (({nil})) returned by (({#compact!})) ?

enum.map { |i| i + 1 if i.even? }.compact

makes un unnecessary copy, but i cannot use

enum.map { |i| i + 1 if i.even? }.compact!

because it will return (({nil})) if no changes were made.

Wouldn't it be better if (({#compact})) and (({#compact!})) returned the same thing?
=end

#49 - 06/20/2012 07:23 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Em 19-06-2012 17:32, Roger Pack escreveu:

> How about #map_some as the name?
> #map_if ?

I loved this one :)

#50 - 08/14/2012 06:24 PM - drKreso (Kresimir Bojcic)

+1 I also like filter_map

(My other favourite would be map_selected which I think is more expressive for non-lisp-brainwashed)

#51 - 08/14/2012 08:25 PM - trans (Thomas Sawyer)

Instead of thinking of it as a special type of #map, I suggest thinking about it as a special type of #select. The reason is that we could also use the same type of behavior for #find. Which is why I suggest #select_yield and #find_yield (or #yield_select and #yield_find). The names of both of these need to be considered together.

#52 - 08/14/2012 09:53 PM - Anonymous

On Tue, Aug 14, 2012 at 1:25 PM, trans (Thomas Sawyer)
transfire@gmail.com wrote:

> Instead of thinking of it as a special type of #map, I suggest thinking about it as a special type of #select. The reason is that we could also use the same type of behavior for #find. Which is why I suggest #select_yield and #find_yield (or #yield_select and #yield_find). The names of both of these need to be considered together.

To me it looks like a not-so-special type of inject.

How would map_select differ from inject, please?

Ciao,
Sheldon.

#53 - 10/25/2012 08:29 PM - yhara (Yutaka HARA)

- Target version changed from 2.0.0 to 2.6

#54 - 03/03/2014 12:04 PM - shugo (Shugo Maeda)

- Status changed from Feedback to Assigned

Yukihiro Matsumoto wrote:

> I am OK with the original map_select behavior, but I don't like the name #map_select.
> It is combination of -ect family name with map.
>
> I prefer the name #filter, but it might be confused by simple alias of select.
> Any idea?

What do you think of the following counter proposals?

- select_yield
- filter_map
- map_if

#55 - 03/03/2014 01:55 PM - sawa (Tsuyoshi Sawada)

I would like to propose:

- partial_map

#56 - 03/04/2014 01:52 PM - sawa (Tsuyoshi Sawada)

Also, regarding Yehuda Katz's concern:

> The only caveat is that it would be impossible to intentionally return nil here; suggestions welcome.

I would like to propose that the method takes an optional argument that determines what element is to be removed. By default, this is nil.

```
[1, 2, 3, 4].partial_map{|i| i + 4 if i.even?} # => [6, 8]

s = "abc"
[0, 1, 2, 3, 4].partial_map(:ignore){|i| i.even? ? s[i] : :ignore} # => ["a", "c", nil]
```

#57 - 03/18/2014 06:27 PM - sawa (Tsuyoshi Sawada)

I came to think that, if this feature is to be added, then are other methods to be considered: flat_map and to_h in addition to map. I think it should be

either the case that the corresponding counterpart to all of these methods should be added, or none.

#58 - 04/09/2014 02:41 PM - epidemian (Demian Ferreiro)

I found this issue looking for a single-pass alternative to .map{...}.compact.

The former example uses { |i| i + 1 if i.even? }, which reads almost like a comprehension and has a clear separation between the condition and the transformation. But sometimes it's not so easy to achieve such separation:

```
# Generate an array of integers from untrusted input.
['', '42', 'nope', :not_even_int_convertible].compact_map { |x| Integer(x) rescue nil } # => [42]
```

BTW, i think that compact_map sounds quite natural for what this method does, but it could also be an extension to compact as others have mentioned:

```
['', '42', 'nope', :not_even_int_convertible].compact { |x| Integer(x) rescue nil } # => [42]
```

As long as this is considered an alternative to map + concat, i think it makes sense not to worry about the edge case of wanting to preserve the nils, as the purpose of compact is to wipe them out. For those cases a combination of select and map can be used, or grep with a proc as its "pattern" argument to avoid having two iterations.

#59 - 02/10/2015 10:30 PM - rokob (Andrew Ledvina)

```
The only caveat is that it would be impossible to intentionally return nil here
```

I don't see why you need that, just have the block return a pair (keep_if_true, mapped_value). I also would second the name filter_map. Ignoring all edge-cases you could just do something like:

```
module Array
  def filter_map
    result = []
    self.each do |elem|
      keep, value = yield elem
      result << value if keep
    end
    result
  end
end
```

```
> [1,32,9,33,2,13].filter_map{|v| [v < 20, v%2==1 ? v*2 : nil]]}
 => [2, 18, nil, 26]
```

Although to be fair, filter_map is just reduce with a more complicated block

```
> [1,32,9,33,2,13].reduce([]){|acc, v| v<20 ? v%2==1 ? acc << v*2 : acc << nil : nil; acc}
 => [2, 18, nil, 26]
```

#60 - 08/31/2017 06:38 AM - shugo (Shugo Maeda)

- Related to Feature #13784: Add Enumerable#filter as an alias of Enumerable#select added

#61 - 12/25/2017 06:15 PM - naruse (Yui NARUSE)

- Target version deleted (2.6)

#62 - 02/13/2019 02:35 AM - shugo (Shugo Maeda)

- Related to Feature #15323: [PATCH] Proposal: Add Enumerable#filter_map added