Ruby trunk - Feature #15483

Proc or Method combination with Symbol

12/29/2018 10:40 AM - aycabta (aycabta .)

Status:	Rejected	
Priority:	Normal	
Assignee:		
Target version:		

Description

In [Feature #6284], Matz said

> We need more discussion if we would add combination methods to the Symbol class.

Right, let's get started to discuss.

For your information, recent a few months I'm discussing this with osyo (manga osyo) .

This is a discussion of "design"

I understand that all features of this issue have both merits and demerits, but I guess that language design is most important. All features of this issue related to each other.

Abstract

At present, you can use Proc#>> or Proc#<< with Symbol#to_proc.

```
%w{72 101 108 108 111}.map(&(:to_i.to_proc >> :chr.to_proc))
# => ["H", "e", "l", "l", "o"]
```

This is convenient but methods that take block can take a proc with & syntax sugar instead of #to_proc by right, like [1, 2, 3].map(&:to_s). So Symbol#to_proc looks like too long for Proc#>> or Proc#<<. Therefore, you need new syntax sugar.

Receiver

Symbol#>> and Symbol#<<

Symbol#>> and Symbol#<< will be considered, but this means that Symbol is treated as Proc partially. The [1, 2, 3].map(&:to_s) treats Symbol as Proc partially too, but it's with pre-positioned &.

```
%w{72 101 108 108 111}.map(&(:to_i >> :chr.to_proc))
# => ["H", "e", "l", "l", "o"]
```

I can't come up with other ideas for the Symbol receiver.

New &:symbol_name syntax sugar for :symbol_name.to_proc

```
%w{72 101 108 108 111}.map(&(&:to_i >> :chr.to_proc)))
# => ["H", "e", "l", "l", "o"]
```

Argument

Calls #to_proc by Proc#>> or Proc#<< internally as a duck typing

```
%w{72 101 108 108 111}.map(&(:to_i.to_proc >> :chr))
# => ["H", "e", "l", "l", "o"]
```

In this case, Proc#>>(:to_i.to_proc >>) calls Symbol#to_proc(for :chr) inside.

This is useful to use with Hash#to_proc:

```
h = { Alice: 30, Bob: 60, Cris: 90 }
%w{Alice Bob Cris}.map(&(:to_sym.to_proc >> h))
# => [30, 60, 90]
```

Proc#>> and Proc#<< take block as an argument

```
%w{72 101 108 108 111}.map(&(:to_i.to_proc >> &:chr))
```

Combination of receiver and argument

Symbol#>> and calling #to_proc internally:

```
%w{72 101 108 108 111}.map(&(:to_i >> :chr))
# => ["H", "e", "l", "l", "o"]
```

&:symbol_name syntax sugar for :symbol_name.to_proc and Symbol#>> and taking block:

```
%w{72 101 108 108 111}.map(&(&:to_i >> &:chr))
# => ["H", "e", "l", "l", "o"]
```

Related issues:

Related to Ruby trunk - Bug #15428: Refactor Proc#>> and #<< **Open**

History

#1 - 12/29/2018 10:40 AM - aycabta (aycabta .)

- Backport deleted (2.4: UNKNOWN, 2.5: UNKNOWN, 2.6: UNKNOWN)

- Tracker changed from Bug to Feature

#2 - 12/29/2018 11:22 AM - aycabta (aycabta .)

- Description updated

#3 - 12/29/2018 12:19 PM - shevegen (Robert A. Heiler)

I am biased so I do not want to digress from this thread too much while explaining my bias. However had,
I still want to state a few things:

- In regards to Symbol, this is a language design decision, how Symbols are to be used. I think we can have valid arguments for both main variants, e. g. to keep Symbols simple, or to allow more flexibility. Personally I'd rather prefer them simple, largely because I don't feel most proposals for change make them better and most definitely not prettier; but I have no real problem either way here.

Still, in regards to proposals allowing for more flexibility of Symbols, this leads me to:

- **Syntax consideration**. To me personally the proposed syntax is not very elegant.

In particular:

```
.map(&(&:to_i >> &:chr))
```

Is really not pretty. We use '& three' times there; and the new >>. It does not really feel consistent
with other parts of ruby in my opinion, syntax-wise alone. I have less of a problem with a single & but
I also dislike that I have to look carefully, e. g to distinguish between a** .map(&:)** versus a **.map(&)**
variant. Do we really want to have to look for & now carefully and a : or no :, on top of it? The second
variant also packs a lot more information into the method-call, which makes it a bit hard to see what
is going on to me, e. g. **.map(&(&:to_i >> :chr.to_proc))**. And the >> which I am also not a big fan of,
but as said in the beginning, I am biased already, so my comments will be biased as well.

- Another issue I have, and this is more general, that I do not really see the massive benefit. This is not solely confined to the proposal here, and is obviously subject to personal opinion/evaluation and how you use ruby ("more than one way to use ruby", too), but more generally about some other related proposals too, where I am not really sure if the change is needed or provides a lot of really useful things that we need.

I understand it if the goal is more flexibility in what we can do; for example, I think I also stated before
that I am in agreement with proposals to allow arguments to methods given rather than solely be able
to use e. g. .map(&:method SOME WAY FOR ARGUMENTS HERE). The major problem I have with most
of these proposals I have seen so far is syntax-wise. We do not have that many characters while staying
in ASCII land, but the core of ruby is very elegant and quite simple, syntax-wise (for me). Several of the
proposals in the last ~3 years or so, are, to me, syntax-wise, not really elegant. Syntax is not everything
but if I have to stare at code a lot then I'd rather look at good syntax than bad one.

Anyway, I'll close my comment here.

#4 - 01/08/2019 02:47 PM - osyo (manga osyo)

I am thinking like this.

NOTE: Here we define it as follows.

- functional object
  - defined #call (and #<< #>>) object
  - e.g. Proc Method
- blockable object
  - defined #to_proc object
  - e.g. Symbol Hash

Current

- Proc#<< and Proc#>> arguments is functional object call #call.
- Proc#<< and Proc#>> is not call #to_proc
- Proc#<< and Proc#>> is not accept block argument

Composite function in Ruby

- Composite function is functional object and functional object
- functional object >> functional object # => OK
- functional object >> other object # => NG
- other object >> functional object # => NG

Symbol is functional object

- Symbol is blockable object
- Symbol is not functional object
- Handling Symbol with compositing functions is incorrect
- What about other blockable objects?
  - e.g. Hash
  - Hash is functional object?

Proc#<< is call #to_proc ?

- It should be explicitly converted to Proc (functional object) with# to_proc
  - proc << :hoge => NG: :hoge is not Proc
  - proc << :hoge.to_proc => OK : Explicitly convert :hoge to Proc
- Same as not handling "42" as an Integer
  - 1 + "42" => NG : "42" is not an Integer
  - 1 + "42".to_i => OK : Explicitly convert "42" to a Proc

Proposal1 : Symbol to functional object

- define Symbol#>> Symbol#<< Symbol#call
- What about other blockable objects?
  - Hash is functional object?
- Is it really necessary for Symbol ?
- Is Symbol really a "functinal object" ?

```
# Symbol to functional object
class Symbol
    def call(*args, &block)
        to_proc.call(*args, &block)
    end

    def <<(other)
        to_proc << other
    end

    def >>(other)
        to_proc >> other
    end
end

p %w{72 101 108 108 111}.map(&(:to_i >> :chr))
# => ["H", "e", "l", "l", "o"]
```

Proposal2 : Symbol to functional object

- Proc#<<(other) to Proc#<<(other, &block)
- Prioritize other ?

```
class Proc
    prepend Module.new {
        def <<(other = nil, &block)
            # other or block?
            super(other || block)
        end

        def >>(other = nil, &block)
            # other or block?
            super(other || block)
        end
    }
end

# :to_i convert to Proc
# must be `.>>`
p %w{72 101 108 108 111}.map(&(:to_i.to_proc.>> &:chr))
# => ["H", "e", "l", "l", "o"]
```

Proposal3 : Define syntax sugar for #to_proc

- For example, define #to_proc to ~@.
  - or other Unary operator
  - +@ -@ ! & ?
- Do not change current specifications
- I think this is good

```
# Add ~@
class Object
    # ~ is to_proc
    # ~ or other unary operator?
    def ~@
        to_proc
    end
end

# Use Symbol#to_proc
p %w{72 101 108 108 111}.map(&(:to_i.to_proc >> :chr.to_proc))

# alias ~ is to_proc
p %w{72 101 108 108 111}.map(&~:to_i >> ~:chr)
```

Thank you :)

Japanese

#5 - 01/09/2019 05:40 AM - nobu (Nobuyoshi Nakada)

Why not using refinements?

```
# symbol/functionalized.rb
module Symbol::Functionalized
  refine(Symbol) do
    def call(*args, &block)
      to_proc.call(*args, &block)
    end

    def <<(other = (b = true), &block)
      to_proc << (b ? block : other.to_proc)
    end

    def >>(other = (b = true), &block)
      to_proc >> (b ? block : other.to_proc)
    end
  end
end

require 'symbol/functionalized'
using Symbol::Functionalized

p %w{72 101 108 108 111}.map(&:to_i >> :chr) #=> ["H", "e", "l", "l", "o"]
```

#6 - 01/09/2019 06:48 AM - osyo (manga osyo)

hi, nobu :)

> ▯▯
> Why not using refinements?

It is example code.
Also, Symbol#call is not called in Proc#<<.

```
# Error: undefined method `call' for :chr:Symbol (NoMethodError)
p %w{72 101 108 108 111}.map(&proc { |s| s.to_i } >> :chr) #=> ["H", "e", "l", "l", "o"]
```

#7 - 01/09/2019 07:18 AM - nobu (Nobuyoshi Nakada)

```
# symbol/functionalized.rb
module Symbol::Functionalized
  refine(Symbol) do
    def call(*args, &block)
      to_proc.call(*args, &block)
    end

    def <<(other = (b = true), &block)
      to_proc << (b ? block : other.to_proc)
    end

    def >>(other = (b = true), &block)
      to_proc >> (b ? block : other.to_proc)
    end
  end

  refine(Proc) do
    def <<(other)
      super(other.to_proc)
    end

    def >>(other)
      super(other.to_proc)
    end
  end
end
```

#8 - 01/09/2019 10:27 AM - nobu (Nobuyoshi Nakada)

I made function-composite gem, as a PoC.

#9 - 01/09/2019 12:37 PM - osyo (manga osyo)

I think it will not work in the following cases.

```
# NG: Error undefined method `call' for :chr:Symbol (NoMethodError)
p (30.method(:+) >> :chr).call 42


h = { Alice: 30, Bob: 60, Cris: 90 }

# OK
p (:to_sym >> h).call "Alice"
# => 30

# NG
p (h << :to_sym).call "Bob"
```

Would you like to add Method#>> and Hash#>>, or other object #>> definitions?
I do not think that is good.
I think it is necessary to clearly separate "functional object"(e.g. Proc, Method) and "blockable object"(e.g. Symbol, Hash).
I think that it should handle only functional object in the composite function.

#10 - 01/10/2019 05:08 AM - mrkn (Kenta Murata)

- Related to Bug #15428: Refactor Proc#>> and #<< added

#11 - 01/10/2019 05:37 AM - matz (Yukihiro Matsumoto)

- Status changed from Open to Rejected

I feel the expression ary.map(&(:to_i << :chr)) is far less readable than ary.map{|x|x.to_i.chr}.
And the latter is faster and can take arguments NOW e.g. ary.map{|x|x.to_i(16).chr}.

Given these superiorities, this proposal does not sound attractive.

Matz.

p.s.
And this can lead to the default block parameter like it.