Ruby master - Bug #15428

Refactor Proc#>> and #<<

12/17/2018 09:56 PM - zverok (Victor Shepelev)

Status:	Open		
Priority:	Normal		
Assignee:			
Target version:			
ruby -v:		**Backport:**	2.4: UNKNOWN, 2.5: UNKNOWN, 2.6: UNKNOWN

Description

#6284 introduced Proc#>> and Proc#<<, but the requirements to the argument is totally inconsistent with ANY other place in Ruby.

Currently, it is the **only** place in Ruby where coercing argument to Proc is done with #call method. Everywhere else it is done with #to_proc, and #call method never had any special significance except for .() sugar. I believe there are two possible actions:

1. change #>> and #<< to use #to_proc (which will give Symbols composability for free), **or, alternatively**
2. state that #call from now on has a special meaning in Ruby and probably decide on other APIs that should respect it (for example, auto-define #to_proc on any object that has #call)

Either is OK, the current situation is not.

PS: One more problem (that probably should be discussed separately) is that check for #call existence is performed pretty late, which can lead to this kind of errors:

```
# At code loading time:

# I erroneously thought this is correct. It is not, but the line would perform without
# any error.
PROCESSOR = JSON.method(:parse) >> :symbolize_keys

# Later, in runtime:
'{"foo": "bar"}'.then(&PROCESSOR)
# NoMethodError (undefined method `call' for :symbolize_keys:Symbol)
```

UPD 2018-12-29: As this ticket was ignored prior to 2.6 release, I rewrote it in an "actionable" instead of "question" manner.

Related issues:

Related to Ruby master - Feature #15483: Proc or Method combination with Symbol	**Rejected**

Associated revisions

Revision 33a75edd - 01/10/2019 06:01 AM - nobu (Nobuyoshi Nakada)

proc.c: check if callable

- proc.c: check the argument at composition, expect a Proc, Method, or callable object. [ruby-core:90591] [Bug #15428]

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@66769 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 66769 - 01/10/2019 06:01 AM - nobu (Nobuyoshi Nakada)

proc.c: check if callable

- proc.c: check the argument at composition, expect a Proc, Method, or callable object. [ruby-core:90591] [Bug #15428]

History

#1 - 12/17/2018 10:00 PM - zverok (Victor Shepelev)

BTW, having >> and << convert objects via to_proc would have a nice (?) addition of chaining symbols too:

```
PROCESS = ->(url) { get(url) } >> JSON.method(:parse) >> :symbolize_keys >> :invert
```

#2 - 12/18/2018 10:33 PM - shevegen (Robert A. Heiler)

Are you sure that .call had the same meaning as to_proc?

I vaguely remember the old pickaxe having mentioned .call a
lot, but I can not recall it having mentioned to_proc much
at all. It has been quite some years since I last had a look
at the pickaxe though; obviously since ruby changes that also
means that the way we use ruby changes.

As for the lambda syntax and << - I think it would be better
to keep the proposals different if possible since at the
least to me, ->(x) { y } << z looks ... very, very weird.

It would be better to keep the issue here about call versus
to_proc, in my opinion.

#3 - 12/29/2018 12:43 PM - zverok (Victor Shepelev)

- Description updated

- Subject changed from Proc composition: what can quack like Proc? to Refactor Proc#>> and #<<

#4 - 12/29/2018 12:45 PM - zverok (Victor Shepelev)

- Backport set to 2.4: UNKNOWN, 2.5: UNKNOWN, 2.6: UNKNOWN

- Tracker changed from Misc to Bug

#5 - 01/10/2019 05:08 AM - mrkn (Kenta Murata)

- Related to Feature #15483: Proc or Method combination with Symbol added

#6 - 01/10/2019 06:01 AM - nobu (Nobuyoshi Nakada)

- Status changed from Open to Closed

Applied in changeset trunk|r66769.

---

proc.c: check if callable

- proc.c: check the argument at composition, expect a Proc, Method, or callable object. [ruby-core:90591] [Bug #15428]

#7 - 01/12/2019 09:36 AM - zverok (Victor Shepelev)

- Status changed from Closed to Open

Sorry for my arrogance, but I am kinda surprised (to say the least) with how this ticket is treated.

- Ticket is closed silently (are you aware that it is an event when Redmine doesn't send the notification, so unless I'll check manually, I am not aware that ticket is closed?)
- Developer meeting discussion log seems to completely miss the point of the ticket. Quoting from there:

  Mame: Use a block. You then want syntactic sugar for passing an argument, and then for partial application, blah blah blah.
  knu: We should not add fancy new features around procs and symbols if the real problem is that block syntax is not good enough, like |x| is always necessary. (See above for the default block parameter syntax)

I understand that typical problem with my "large" tickets is I am trying to consider too much things (believing it would describe the problem better), and sometimes it is too hard to tell what's my main point. So I should try better.

Main point of this ticket: Special treatment of #call in Proc#>> and Proc#<< is **inconsistent** with the rest of the language.

There are **two possible solutions** for this problem:

1. **change #>> and #<< to use #to_proc**, or, alternatively
2. **codify that #call from now on has a special meaning** in Ruby and probably decide on other APIs that should respect it

This ticket is NOT about "I want to chain symbols, it is nice". What I want is **language consistency**, not **fancy new features**. In fact, I want exactly the opposite as "fancy new features (that ignore consistency)", which current implementation of #>>/#<< apparently is.

#8 - 01/12/2019 11:35 AM - nobu (Nobuyoshi Nakada)

Sorry, I was going to reopen this after r66769, but have forgotten.

> Currently, it is the **only** place in Ruby where coercing argument to Proc is done with #call method.

These methods do not coerce argument, but just expect a method on it.

> 1. change #>> and #<< to use #to_proc

#to_proc is not an implicit conversion method, as it needs the specific syntax, &.

> 1. state that #call from now on has a special meaning in Ruby

It is a usual case that a method may expect its argument to have particular method(s),
e.g., String#+ expects #to_str on non-string argument.

#9 - 01/12/2019 12:08 PM - zverok (Victor Shepelev)

> #to_proc is not an implicit conversion method, as it needs the specific syntax, &.

But ideologically, it is the way to designate "this object quacks as a Proc", e.g. could be used instead of a Proc object.

> It is a usual case that a method may expect its argument to have particular method(s),
> e.g., String#+ expects #to_str on non-string argument.

Well, I believe that #to_str is an argument towards my point :)

```
foo = "test"
foo + bar # foo is a String, the operator requires bar to have #to_str, coercing it to a compatible type
foo = -> { puts "test" }
foo << bar # foo is a Proc, the operator requires bar to have #to_proc, coercing it to a compatible type
```

I believe that it is not just a "requirement for particular method", but consistent language rule.

Currently, we have (not, in fact, very well-documented, but still existing) convention, that some methods are used for coercion of other objects to compatible class. Typically, they are discussed in class the method coerces to (frequently in .try_convert method), or in Object. That's true for to_s/to_str, to_a/to_ary, to_enum, to_io, and creates pretty strong intuition of "if we want it to be compatible with class X, there are some to_<x> or alike methods to be implemented".

I am aware the #call convention is somewhat embraced in some **third-party** projects. But currently, **inside** language, there is no "implied intuition" about #call method. We have a small syntactic sugar feature of .(), which is rarely used in language docs, and no core classes except for Proc and Method have a #call method (but they also both have #to_proc); other objects "quacking like callable" use #to_proc to designate this fact. The discrepancy between two is hard to explain and document, and I believe that there would be a lot of confusion with this feature.

PS: I just wonder, why #call convention was chosen over #to_proc, initially?..

#10 - 01/12/2019 12:31 PM - nobu (Nobuyoshi Nakada)

> > #to_proc is not an implicit conversion method, as it needs the specific syntax, &.
>
> But ideologically, it is the way to designate "this object quacks as a Proc", e.g. could be used instead of a Proc object.

The point is that is done **explicitly**.

> > It is a usual case that a method may expect its argument to have particular method(s),
> > e.g., String#+ expects #to_str on non-string argument.
>
> Well, I believe that #to_str is an argument towards my point :)

No, #to_proc is not similar to #to_str, but #to_s.

> Currently, we have (not, in fact, very well-documented, but still existing) convention, that some methods are used for coercion of other objects to compatible class. Typically, they are discussed in class the method coerces to (frequently in .try_convert method), or in Object. That's true for to_s/to_str, to_a/to_ary, to_enum, to_io, and creates pretty strong intuition of "if we want it to be compatible with class X, there are some to_<x> or alike methods to be implemented".

#to_str and #to_s differ in explicitness, and #to_proc is in the #to_s side.

> I am aware the #call convention is somewhat embraced in some **third-party** projects. But currently, **inside** language, there is no "implied intuition" about #call method. We have a small syntactic sugar feature of .(), which is rarely used in language docs, and no core classes except for Proc and Method have a #call method (but they also both have #to_proc); other objects "quacking like callable" use #to_proc to designate this fact. The discrepancy between two is hard to explain and document, and I believe that there would be a lot of confusion with this feature.

#call method is called by Enumerable, Enumerator, finalizer and eval.

> PS: I just wonder, why #call convention was chosen over #to_proc, initially?..

As stated already, #to_proc is an explicit conversion, so it would not called implicitly.

#11 - 01/12/2019 12:35 PM - nobu (Nobuyoshi Nakada)

nobu (Nobuyoshi Nakada) wrote:

> #call method is called by Enumerable, Enumerator, finalizer and eval.

Sorry, the last is Signal, not eval.

#12 - 01/12/2019 01:24 PM - zverok (Victor Shepelev)

> #call method is called by Enumerable, Enumerator, finalizer and Signal.

That's interesting. I just looked through the docs and understood I was not very aware of this fact:

- Enumerable#find (is the only method using #call, right?) says just "calls ifnone and returns its result", not specifying how exactly it "calls";
- Enumerator::new says "callable object" without further explanation;
- Signal#trap says "Otherwise, the given command or block will be run".

OK, that' the problem of docs, I'll handle it. I also never have seen either used in the wild (in a form of "callable object passing"), but maybe it is just me.

Now, from what I can understand, you followed this rule of thumb:

- if something is passed with &, it is converted with to_proc (it is something that have some method of coercion to a Proc)
- if something is passed as a regular argument, to call later, it is used with call (it is something that think of itself as a kind of Proc)

So, in fact, it is my "option 2": "codify that #call from now on has a special meaning in Ruby", and considering your answer, it is kinda already done since the beginning of time, just not very obvious :)

OK, I can spot 3 problems here:

1. It is not very obvious from documentation and general language structure, as I mentioned above
2. Unlike any other implicit/explicit pairs, "argument construction operator" uses explicit conversion method (* uses #to_ary and ** uses #to_hash)
3. There is no means/helpers/practices to draw explicit method from implicit (though implicit is easier to define)

I believe that (2) is what we'll just need to live with, but both (1) and (3) could be solved with introducing core module Callable (akin to Enumerable and Comparable), defined, let's say, like this:

```
module Callable
  def to_proc
    proc { |*a, &b| call(*a, &b) }
  end
end
```

That's not so much "non-trivial code", as "atomic declaration of statement": "this thing is callable; Proc and Method are callable, but you can define your own; proc composition composes any callable objects".

Usage in some client code:

```
class SetStatus < BusinessAction
  def self.call(task, status)
    new(task, status).validate.call # ...or something
  end

  extend Callable
```

```
end

tasks_and_statuses.each(&SetStatus)
```

WDYT?

#13 - 01/13/2019 05:56 AM - nobu (Nobuyoshi Nakada)

zverok (Victor Shepelev) wrote:

> Now, from what I can understand, you followed this rule of thumb:
>
> - if something is passed with &, it is converted with to_proc (it is something that have some method of coercion to a Proc)
> - if something is passed as a regular argument, to call later, it is used with call (it is something that think of itself as a kind of Proc)

Right.

> So, in fact, it is my "option 2": "codify that #call from now on has a special meaning in Ruby", and considering your answer, it is kinda already done since the beginning of time, just not very obvious :)

More precisely, a special meaning to the Proc class, and other classes expect the role.

> OK, I can spot 3 problems here:
>
> 1. It is not very obvious from documentation and general language structure, as I mentioned above
> 2. Unlike any other implicit/explicit pairs, "argument construction operator" uses explicit conversion method (* uses #to_ary and ** uses #to_hash)
> 3. There is no means/helpers/practices to draw explicit method from implicit (though implicit is easier to define)

As for (2), * uses #to_a whereas ** uses #to_hash, I agree that there is an inconsistency.

> I believe that (2) is what we'll just need to live with, but both (1) and (3) could be solved with introducing core module Callable (akin to Enumerable and Comparable), defined, let's say, like this:
>
> ```
> module Callable
>   def to_proc
>     proc { |*a, &b| call(*a, &b) }
>   end
> end
> ```
>
> That's not so much "non-trivial code", as "atomic declaration of statement": "this thing is callable; Proc and Method are callable, but you can define your own; proc composition composes any callable objects".

I'm neutral about it, callable gem seems existing.

#14 - 01/20/2019 08:04 PM - Eregon (Benoit Daloze)

nobu (Nobuyoshi Nakada) I think we should backport r66769 to Ruby 2.6, could you do it or file an issue for it?