Ruby master - Feature #18276
`Proc#bind_call(obj)` same as `obj.instance_exec(..., &proc_obj)`
10/28/2021 12:37 PM - ko1 (Koichi Sasada)

<table>
<thead>
<tr>
<th>Status:</th>
<th>Rejected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority:</td>
<td>Normal</td>
</tr>
<tr>
<td>Assignee:</td>
<td></td>
</tr>
<tr>
<td>Target version:</td>
<td></td>
</tr>
</tbody>
</table>

Description
Proc#bind_call(obj) same as obj.instance_exec(..., &proc_obj)

```
proc_obj = proc{|params...| ...}
obj.instance_exec(params..., &proc_obj)
```

is frequent pattern.

$ gem-codesearch 'instance_exec.+' | wc -l
9558

How about to introduce new method Proc#bind_call?

```
class Proc
  def bind_call obj, *args
    obj.instance_exec(*args, &self)
  end
end

pr = ->{ p self }
pr.bind_call("hello") #=> "hello"
pr.bind_call(nil) #=> nil
```

It is similar to UnboundMethod#bind_call.

My motivation;

I want to solve shareable Proc's issue https://bugs.ruby-lang.org/issues/18243 and one idea is to prohibit Proc#call for shareable Proc's, but allow obj.instance_exec(&pr). To make shortcut, I want to introduce Proc#bind_call.

UnboundProc is another idea, but I'm not sure it is good idea...

Anyway, we found that there are many usage of instance_exec(&proc_obj), so Proc#bind_call is useful not for Ractors.

Related issues:
Related to Ruby master - Bug #18243: Ractor.make_shareable does not freeze th...

Closed

History

#1 - 10/28/2021 02:50 PM - jeremyevans0 (Jeremy Evans)

I'm in favor of adding this method, and would like to see it support the following:

```
pr = ->{*a, **kw, &block} do
  # ...
  block.call(something)
end
pr.bind_call(obj, arg, kw: nil) do |something|
  # ...
end
```

This would allow you to get the equivalent of instance_exec, but with passing a block to the proc being instance exected, which is not currently possible.

#2 - 10/28/2021 03:53 PM - jhawthorn (John Hawthorn)
In Rails we use `obj.instance_exec(&proc_obj)` in a few places. One of the downsides `instance_exec` has is that it creates a singleton class for `obj`, which isn't friendly to method caches or JITs. `Proc#bind_call` would be very useful to us if it behaved similarly but did not create a singleton class.

#3 - 10/29/2021 08:24 PM - Eregon (Benoit Daloze)

jhowthorn (John Hawthorn) wrote in #note-2:

In Rails we use `obj.instance_exec(&proc_obj)` in a few places. One of the downsides `instance_exec` has is that it creates a singleton class for `obj`, which isn't friendly to method caches or JITs. `Proc#bind_call` would be very useful to us if it behaved similarly but did not create a singleton class.

I think `instance_exec` doesn't create a singleton class, only if needed by e.g. `Object.new.instance_exec { def foo; end }` (at least on TruffleRuby, maybe it depends how eagerly the cref/default definee is computed by the Ruby implementation).

I think the same applies to `Proc#bind_call` (as `-> { def foo; end }.bind_call(Object.new))`.

I'm not against such a method, but IMHO this alone is not solving #18243 in a reasonable way. If an object is made shareable, every object reachable from it should be frozen or shareable, and magically ignoring the Proc's self is conceptually ugly and I believe very confusing for many. Maybe `Ractor.make_shareable(someProc)` should return a Proc with a special receiver (e.g., `Qundefined`), which simply can't be called via `.call` on any `Ractor` and can only be called via `Proc#bind_call`.

Then at least that shareable Proc wouldn't refer to any unshared object (which would violate the docs and expected semantics of `Ractor.make_shareable`)

#4 - 10/29/2021 08:29 PM - Eregon (Benoit Daloze)

I don't like the idea to alter the semantics of Proc methods just for Ractor though (also it costs extra checks on every `Proc#call`!). The best and cleanest solution is IMHO to raise for `Ractor.make_shareable(someProc)` if the Proc self is not shareable.

See https://bugs.ruby-lang.org/issues/18243#note-5 for more details on that idea.

Then `Proc#bind_call` is simply not needed for `Ractor`. I don't mind adding `Proc#bind_call` for other purposes though. Changing the self is typically best avoided except for some cases in DSLs, as it breaks what methods the block can call in its lexical context.

#5 - 10/29/2021 08:30 PM - Eregon (Benoit Daloze)

- Related to Bug #18243: `Ractor.make_shareable` does not freeze the receiver of a Proc but allows accessing ivars of it added

#6 - 11/01/2021 02:44 AM - Dan0042 (Daniel DeLorme)

Why `proc.bind(obj).call` ? It seems a more "proper" API, more composable. You can bind once and then call multiple times. Maybe `Ractor.make_shareable(proc.bind(nil))`. I understand the performance benefit of `bind_call` but in #15955, UnboundMethod#bind_call was introduced as an optimization for hot spots, to be used by "only some fundamental libraries".

#7 - 11/03/2021 11:54 PM - Eregon (Benoit Daloze)

Dan0042 (Daniel DeLorme) wrote in #note-6:

Why `proc.bind(obj).call` ? It seems a more "proper" API, more composable. You can bind once and then call multiple times. Maybe `Ractor.make_shareable(proc.bind(nil))`. I understand the performance benefit of `bind_call` but in #15955, UnboundMethod#bind_call was introduced as an optimization for hot spots, to be used by "only some fundamental libraries".

:+1: I think that's useful. I thought to the name `Proc#with_self(nil)` in #18243 but `#bind` is much better. `Ractor.make_shareable(proc.bind(nil))` is a clean solution, I like it.

I think we don't even need `Proc#bind_call` then, or only as a replacement for `instance_exec(&proc)`.

#8 - 11/04/2021 11:39 AM - byroot (Jean Boussier)

  only as a replacement for `instance_exec(&proc)`.

Assuming I correctly understand how it would work, then yes it would be great to have it to replace lots of costly `instance_exec`.

#9 - 11/04/2021 01:13 PM - Eregon (Benoit Daloze)

byroot (Jean Boussier) wrote in #note-8:

Assuming I correctly understand how it would work, then yes it would be great to have it to replace lots of costly `instance_exec`.

I don't think it would change anything performance-wise. The only thing is it's possible to pass a block to the called proc that way (
#10 - 11/04/2021 01:19 PM - Eregon (Benoit Daloze)
Ah, maybe Koichi meant that bind_call doesn't change the default definee like instance_exec does? i.e., does

class C
  -> {
    def foo
      end
  }.bind_call(Object.new)
end

define foo on that object's singleton class, or as an instance method of class C?
I'd assume on that object's singleton class like instance_exec, but I guess it's not the only possibility.

#11 - 11/08/2021 03:45 PM - Dan0042 (Daniel DeLorme)
Eregon (Benoit Daloze) wrote in #note-10:

I would assume on that object's singleton class like instance_exec, but I guess it's not the only possibility.

But it's interesting how instance_eval/instance_exec automatically creates a singleton_class, I wasn't aware of that before.

p ObjectSpace.each_object(Class) .count #=> 363
Object.new.instance_eval{ }
p ObjectSpace.each_object(Class) .count #=> 364
Object.new.instance_exec{ }
p ObjectSpace.each_object(Class) .count #=> 365

It looks like the singleton_class is eagerly created just in case the eval'd block contains a def. But it shouldn't be too hard to fix that to lazily create the singleton_class when needed.

#12 - 11/16/2021 06:22 AM - ko1 (Koichi Sasada)
Thank you for discussion.
My assumption is not same as instance_exec/eval, only replacing the self.
So the description was wrong.

#13 - 11/16/2021 06:23 AM - ko1 (Koichi Sasada)
Dan0042 (Daniel DeLorme) wrote in #note-6:

Why not proc.bind(obj).call ? It seems a more "proper" API, more composable. You can bind once and then call multiple times. Maybe Ractor.make_shareable(proc.bind(nil)), I understand the performance benefit of bind_call but in #15955, UnboundMethod#bind_call was introduced as an optimization for hot spots, to be used by "only some fundamental libraries".

Proc#bind(obj) returns new Proc or modify the Proc (mutate the Proc)?

#14 - 11/16/2021 12:31 PM - Eregon (Benoit Daloze)
ko1 (Koichi Sasada) wrote in #note-13:

Proc#bind(obj) returns new Proc or modify the Proc (mutate the Proc)?

Returns a new Proc, mutation would be very bad (similar to changing from proc to lambda semantics).
My assumption is not same as instance_exec/eval, only replacing the self.

I think we should optimize #instance_exec in CRuby so it only creates the singleton class lazily, like on TruffleRuby. It seems several people care about that.

Not changing the default definee seems confusing. I think .bind.call/bind_call should behave like instance_exec in that regard (instance_exec can already be surprising, let's not make an extra variant of it with subtle changes).

#15 - 12/03/2021 04:04 AM - ko1 (Koichi Sasada)
- Status changed from Open to Rejected

Ok, I close this ticket.