

Ruby master - Feature #17145

Ractor-aware `Object#deep_freeze`

09/03/2020 06:42 PM - marcandre (Marc-Andre Lafortune)

Status:	Rejected	
Priority:	Normal	
Assignee:		
Target version:		
Description		
I'd like to propose Object#deep_freeze:		
Freezes recursively the contents of the receiver (by calling deep_freeze) and then the receiver itself (by calling freeze).		
Values that are shareable via Ractor (e.g. classes) are never frozen this way.		
<pre># freezes recursively: ast = [:hash, [:pair, [:str, 'hello'], [:sym, :world]]].deep_freeze ast.dig(1, 1) # => [:str, 'hello'] ast.dig(1, 1).compact! # => FrozenError # does not freeze classes: [[String]].deep_freeze String.frozen? # => false # calls `freeze`: class Foo def freeze build_cache! puts "Ready for freeze" super end # ... end [[[Foo.new]].deep_freeze # => Outputs "Ready for freeze"</pre>		
I think a variant deep_freeze! that raises an exception if the result isn't Ractor-shareable would be useful too:		
<pre>class Fire def freeze # do not call super end end x = [Fire.new] x.deep_freeze! # => "Could not be deeply-frozen: #<Fire:0x00007ff151994748>"</pre>		
Related issues:		
Related to Ruby master - Feature #2509: Recursive freezing?	Rejected	12/21/2009
Related to Ruby master - Feature #17100: Ractor: a proposal for a new concurr...	Closed	
Related to Ruby master - Feature #17274: Ractor.make_shareable(obj)	Closed	
Related to Ruby master - Feature #17273: shareable_constant_value pragma	Closed	

History

#1 - 09/03/2020 06:42 PM - marcandre (Marc-Andre Lafortune)

- Related to Feature #2509: Recursive freezing? added

#2 - 09/03/2020 07:00 PM - Eregon (Benoit Daloze)

- Related to Feature #17100: Ractor: a proposal for a new concurrent abstraction without thread-safety issues added

#3 - 09/03/2020 07:06 PM - Eregon (Benoit Daloze)

A dynamic call to freeze causes extra calls, and needs checks that it was indeed frozen.
So for efficiency I think it would be better to mark as frozen internally without a call to freeze on every value.

Also a leaf freeze call could technically mutate the object referencing it and e.g., add a new @ivar, which would make it complicated to ensure everything is frozen
(would need to mark as shallow-frozen first to prevent that, and only as deep-frozen once all contained values are deeply-frozen).

Is there a compelling reason to call a user-defined freeze for every value?

#4 - 09/03/2020 07:20 PM - ko1 (Koichi Sasada)

One concern about the name "freeze" is, what happens on shareable objects on Ractors.
For example, Ractor objects are shareable and they don't need to freeze to send beyond Ractor boundary.

I also want to introduce Mutable but shareable objects using STM (or something similar) writing protocol (shareable Hash). What happens on deep_freeze?

#5 - 09/03/2020 07:27 PM - Eregon (Benoit Daloze)

Maybe we should have a method that ensure an object graph is shareable?
Not sure what a good name for that would be (shareable is hard to spell).
So that would noop for special shareable objects like you say, and freeze the rest.
Note that a shareable Hash or so would need to only allow shareable key/value/elements whenever writing to it.

#6 - 09/03/2020 07:34 PM - marcandre (Marc-Andre Lafortune)

Eregon (Benoit Daloze) wrote in [#note-3](#):

A dynamic call to freeze causes extra calls

Yes

and needs checks that it was indeed frozen.

That won't add any noticeable overhead

Is there a compelling reason to call a user-defined freeze for every value?

Yes. Some objects may need to calculate lazy operations (@cache ||= potentially_long_calculation)

#7 - 09/03/2020 07:39 PM - marcandre (Marc-Andre Lafortune)

ko1 (Koichi Sasada) wrote in [#note-4](#):

One concern about the name "freeze" is, what happens on shareable objects on Ractors.
For example, Ractor objects are shareable and they don't need to freeze to send beyond Ractor boundary.

I also want to introduce Mutable but shareable objects using STM (or something similar) writing protocol (shareable Hash). What happens on deep_freeze?

I think these objects should stop the propagation. The name make_shareable_via_ractor_by_deep_freezing_what_is_necessary would be more accurate but too long ☐☐ Maybe ractorable?

#8 - 09/03/2020 08:44 PM - ko1 (Koichi Sasada)

I think these objects should stop the propagation. The name make_shareable_via_ractor_by_deep_freezing_what_is_necessary would be more accurate but too long ☐☐
Maybe ractorable?

For Ractor, it is very clear. Another candidate is to_shareable?

However, leave from Ractor's perspective, it is strange name, like:

```
CONST = [...].to_shareable
```

Maybe the author don't want to care about Ractor.
The author want to declare "I don't touch it". So "deep_freeze" is better.

hmmm.

#9 - 09/04/2020 02:55 AM - duerst (Martin Dürst)

marcandre (Marc-Andre Lafortune) wrote in [#note-7](#):

I think these objects should stop the propagation. The name `make_shareable_via_ractor_by_deep_freezing_what_is_necessary` would be more accurate but too long ☹☹ Maybe `ractorable`?

What about `ractor_freeze`?

#10 - 09/20/2020 05:57 PM - Eregon (Benoit Daloze)

Maybe `obj.shareable`? And that would then return as-is if already shareable, and make it shareable if not (deep freezing until it reaches already-shareable objects)

I don't like anything with "ractor" in the name, that becomes not descriptive of what it does and IMHO looks weird for e.g. gems not specifically caring about Ractor.

How about first having `deep_freeze` that just freezes everything (except an object's class)?

And then maybe an extra method to deal with already-shareable (if needed), which is probably much less frequently needed. (e.g. it seems rather uncommon that a module constant contains a Ractor reference)

The intention of `deep_freeze` is I think clear: make this object and whatever it refers to immutable, so it can be shared freely via reference without any copying.

And it applies to many other things besides just Ractor, so it seems a good functionality to have in general.

For instance, https://github.com/dkubb/ice_nine already exists, can define `#deep_freeze` and is [widely used](#).

#11 - 09/21/2020 01:50 PM - marcandre (Marc-Andre Lafortune)

ko1 (Koichi Sasada) wrote in [#note-8](#):

For Ractor, it is very clear. Another candidate is `to_shareable`

Methods `to_...` should be reserved for conversion methods, i.e. methods that may return a different object than the receiver. What I am proposing would always return the receiver. `to_shareable` would only be an acceptable name if it returned a deeply frozen copy, but I don't think that's what we need most.

Eregon (Benoit Daloze) wrote in [#note-10](#):

Maybe `obj.shareable`?

`shareable` is good. It's more accurate than `deep_freeze` if we think of ractor sharable structures.

Another alternative would be `freeze(shareable: true)`. It has the advantage of no possible conflict. It could also raise if it fails to produce a shareable object, which should be rare enough to warrant an exception. Something like `shareable: :try` could provide a non-raising alternative.

#12 - 10/02/2020 08:44 AM - kirs (Kir Shatrov)

I'd really like to support this change for reasons I've describe in <https://bugs.ruby-lang.org/issues/17180>

We can take something as simple as `URI::parse("http://example.com")` as an example. Right now that can't be used from a Ractor because `URI::parse` refers to a few internal constants like `URI::RFC3986_PARSER` that are not frozen (and thus not sharable).

If we went with an explicit `deep_freeze` we might need to update all those constants like `URI::RFC3986_PARSER` in the standard library to be also call `deep_freeze`. That doesn't have to be a bad thing but something to be aware of.

#13 - 10/02/2020 04:10 PM - marcandre (Marc-Andre Lafortune)

Maybe we can start with `Ractor.shareable(obj)`?

The major reason why we need it is that there is no way to create a Ractor sharable recursive structure currently.

```
child = {type: :foo}
node = {type: bar, children: [child].freeze}.freeze
child[:parent] = node
child.freeze
```

At this point, everything is frozen, but neither `child` nor `node` are Ractor sharable.

`Ractor.shareable(node)` would check for this, and set both `node` and `child`'s "shareable" flag; after that `Ractor.shareable?(node)` would return true.

API-wise, `Ractor.shareable(node)` might be a good start, or `node.freeze(shareable: true)`, or `node.deep_freeze`

#14 - 10/19/2020 08:48 PM - ko1 (Koichi Sasada)

I implemented `Object#deep_freeze(skip_shareable: false)` for trial.

https://github.com/ko1/ruby/pull/new/deep_freeze

- It doesn't call `#freeze`, but only set frozen flag.
- It set frozen bit for any objects such as `Binding`, `Thread`, `Queue` and so on, but no effect except adding ivars, etc.
- Two pass freezing method
 - (1) collect reachable objects
 - (2) freeze collected objects

This two pass strategy allows errors during collection phase (1). For example, if we introduce unable to "freeze" objects, we can stop collection and raise some error without any side-effect. However, if we call `#freeze`, we can't cancel intermediate state (some objects are frozen, some are not). I have no idea how to solve it, if we need to call `freeze` for each instance.

`Object#deep_freeze(skip_shareable: true)` skips shareable objects to freeze. Maybe it is `ractor_freeze`, `shareable` and so on. I don't have strong opinion about this naming, but I try to unify with `Object#deep_freeze`.

#15 - 10/19/2020 08:53 PM - ko1 (Koichi Sasada)

Maybe we need to introduce new protocol for `T_DATA`.

For example...:

- Time should be frozen.
- `Thread`, `Fiber` should not be frozen. At least they can't become shareable objects.
- `Proc` and `Binding` objects are also not frozen (error on lvar assignment for frozen?)
- ... maybe many others ...

To make conservative, all of them are unable to freeze, or allow freezing but not shareable objects.

#16 - 10/19/2020 09:24 PM - marcandre (Marc-Andre Lafortune)

ko1 (Koichi Sasada) wrote in [#note-14](#):

I implemented `Object#deep_freeze(skip_shareable: false)` for trial.

https://github.com/ko1/ruby/pull/new/deep_freeze

- It doesn't call `#freeze`, but only set frozen flag.
- It set frozen bit for any objects such as `Binding`, `Thread`, `Queue` and so on, but no effect except adding ivars, etc.
- Two pass freezing method
 - (1) collect reachable objects
 - (2) freeze collected objects

This two pass strategy allows errors during collection phase (1). For example, if we introduce unable to "freeze" objects, we can stop collection and raise some error without any side-effect. However, if we call `#freeze`, we can't cancel intermediate state (some objects are frozen, some are not). I have no idea how to solve it, if we need to call `freeze` for each instance.

`Object#deep_freeze(skip_shareable: true)` skips shareable objects to freeze. Maybe it is `ractor_freeze`, `shareable` and so on. I don't have strong opinion about this naming, but I try to unify with `Object#deep_freeze`.

This is great! ☺☺

Two major issues remain:

1) This does not work for recursive structures, in the sense that they will not be marked as `Ractor shareable`. This is capital, particularly since it is the (only) part of this method that can not be done in pure Ruby.

2) There is no callback mechanism for user classes (since this does not call `#freeze`). Typical use case would be for expensive calculations that are done lazily. A class may want to pre-build them before caching. There should be some callback mechanism. My opinion is that we should worry about things working well for well-programmed cases *before* we worry about exceptions (e.g. cases that fail to deeply freeze).

I think that freezing a class "behind its back" by not calling `#freeze` breaks the contract and could be a source of incompatibility.

I see no incompatibility with the two pass system and calling `#freeze` on the list of reachable objects. I have no issue with a structure being half frozen if things fail; the developer was ready to deep freeze all of it, so having only half frozen, even though it clearly was not the intention, does not feel like a big issue.

#17 - 10/20/2020 12:43 AM - ko1 (Koichi Sasada)

marcandre (Marc-Andre Lafortune) wrote in [#note-16](#):

1) This does not work for recursive structures, in the sense that they will not be marked as `Ractor shareable`. This is capital, particularly since it is the (only) part of this method that can not be done in pure Ruby.

Current implementation doesn't mark as shareable, but we can make it. The problem is what happens on half-frozen case... Half-shareable is not acceptable. If half-frozen is acceptable, freezing with traversing, and mark as shareable at second phase is possible.

2) There is no callback mechanism for user classes (since this does not call #freeze). Typical use case would be for expensive calculations that are done lazily. A class may want to pre-build them before caching. There should be some callback mechanism. My opinion is that we should worry about things working well for well-programmed cases *before* we worry about exceptions (e.g. cases that fail to deeply freeze). I think that freezing a class "behind its back" by not calling #freeze breaks the contract and could be a source of incompatibility. I see no incompatibility with the two pass system and calling #freeze on the list of reachable objects. I have no issue with a structure being half frozen if things fail; the developer was ready to deep freeze all of it, so having only half frozen, even though it clearly was not the intention, does not feel like a big issue.

\$ gem-codesearch 'def freeze' | gist -p => <https://gist.github.com/ko1/63b8d43218884249e782d63c2f27b927>

I never realized that so many freeze redefinition are used. Checking the code, some of them freeze attribute objects, which they are frozen with deep_freeze. I can find some cases to calculate lazy (?).

mmm. Maybe this method is used with literals, so there is a chance to optimize for such cases.

I still not sure we can remain the half-frozen state, so I want to ask other comments.

#18 - 10/20/2020 10:00 AM - marcandre (Marc-Andre Lafortune)

ko1 (Koichi Sasada) wrote in [#note-17](#):

I never realized that so many freeze redefinition are used. Checking the code, some of them freeze attribute objects, which they are frozen with deep_freeze. I can find some cases to calculate lazy (?).

Indeed, I imagine that the majority are used to do a deep-freeze, so calling #freeze or not will not make a difference.

An example of lazy computation I can remember writing is a [generic memoization module in DeepCover](#) which is then used [in different places](#). Yes, I'm lazy enough that I factorized the @cache ||= pattern ☐☐

I still not sure we can remain the half-frozen state, so I want to ask other comments.

How about 3 pass?

- 1) Collect reachable objects; abort on non-freezable.
- 2) Call #freeze on reachable objects. Exception due to custom method failing => half frozen state (just write better code ☐☐)
- 3) On success, mark all objects as Ractor shareable.

#19 - 10/20/2020 08:23 PM - Eregon (Benoit Daloze)

ko1 (Koichi Sasada) wrote in [#note-14](#):

I implemented Object#deep_freeze(skip_shareable: false) for trial.

This sounds good to me.

IMHO if something fails in the middle, it's OK for part of it to be frozen.

The intention was to freeze the whole object graph anyway.

The code using the object graph after deep_freeze assumes it was frozen, there shouldn't be any issue if some of it is frozen, the code expected that. (also such code would only run if the exception is caught)

In some languages with unification (I know of Oz), it's also accepted that if it fails in the middle, then the partial state is still there and not magically undone.

I don't think we should call user methods, that will require a lot more scanning (before & after calling user freeze), and if some userfreeze would create new objects and attach them to the object graph, deep_freeze could never end.

Is there a concrete case where it is problematic to not call user's freeze?

I think the design should allow to do everything in a single pass for efficiency, and notably use the "deep frozen/shareable" bit as a "already visited object" marker.

This is also how I implemented "Deep Sharing" during my PhD in <https://eregon.me/blog/assets/research/thread-safe-objects.pdf> section 5.3, and I think we should design deep_freeze to allow for the same optimization to be used because it's a large performance gain compared to a generic routine.

#20 - 10/20/2020 08:42 PM - Eregon (Benoit Daloze)

A draft to make this in a single pass (but it's late, I might have missed something):

- if the object is already deeply-frozen/shareable, return, otherwise:

- mark an object as frozen if not already (so the reachable objects from it don't change)
- iterate children
- mark the object as deeply frozen

but this doesn't handle recursion.

So we could mark as deeply frozen first, and remember to undo that if we cannot freeze some object. However, is there any object that cannot be frozen? I would think not.

So:

- if the object is already deeply-frozen/shareable, return, otherwise:
- mark an object as deeply frozen
- iterate children and recurse

Since that wouldn't call user methods, there is no need to worry about something observing an object marked as deeply frozen but not actually deeply frozen yet.

However, a different Thread could see that, so it can be a problem without a GIL.

So, I think we could use an extra bit for "visited by deep_freeze":

- if the object is already deeply-frozen/shareable or deep_freeze-visited, return, otherwise:
- mark an object as deep_freeze-visited and as frozen if not already (so the reachable objects from it don't change)
- iterate children and recurse
- mark an object as deeply frozen

#21 - 10/20/2020 09:42 PM - marcandre (Marc-Andre Lafortune)

Eregon (Benoit Daloz) wrote in [#note-19](#):

Is there a concrete case where it is problematic to not call user's freeze?

I thought I [already answered that](#), but here's a short example:

```
def Stats
  # ...

  def freeze
    extended_results # build cache if need be
    super
  end

  def extended_results
    @extended_results_cache ||= calc_extended_results
  end

  private
  def calc_extended_results
    # lots of calculations
  end
end
```

There is currently no way for #frozen? to be true (and @instance ||= to fail) without going through the #freeze method. I think we should not break that, especially if there is no callback mechanism to circumvent this.

#22 - 10/21/2020 05:23 AM - marcandre (Marc-Andre Lafortune)

Looking at def freeze in the top ~400 gems, I found 64 in sequel gem alone, and 28 definitions in the rest ☐☐.

Excluding sequel, half do deep freeze. The other use cases:

Cache prebuilding:

https://github.com/rails/rails/blob/master/activerecord/lib/active_support/time_with_zone.rb#L503-L507

<https://github.com/jeremyevans/sequel/blob/master/lib/sequel/adapters/mysql.rb#L150>
<https://github.com/mongodb/mongooid/blob/master/lib/mongooid/criteria.rb#L239>
<https://github.com/sporkmonger/addressable/blob/master/lib/addressable/uri.rb#L846-L858>
<https://github.com/sporkmonger/addressable/blob/master/lib/addressable/template.rb#247-L250>

Instance variable dupping + freeze:

https://github.com/rails/rails/blob/master/activemodel/lib/active_model/attributes.rb#L118
https://github.com/rails/rails/blob/master/activerecord/lib/active_record/core.rb#L494

Another usecase is lazy defined methods (I presume like OpenStruct before the recent changes):

<https://github.com/jeremyevans/sequel/blob/master/lib/sequel/plugins/finder.rb#L167>
https://github.com/solnic/virtus/blob/master/lib/virtus/instance_methods.rb#L148

Cases not actually freezing the receiver (?):

<https://github.com/mongodb/mongooid/blob/master/lib/mongooid/document.rb#L55-L69>
https://github.com/datamapper/dm_core/blob/master/lib/dm-core/support/lazy_array.rb#L300-L313
https://github.com/dtao/safe_yaml/blob/master/lib/safe_yaml/transform/transformation_map.rb#L24
<https://github.com/paulelliott/fabrication/blob/master/lib/fabrication/schematic/manager.rb#L23>

I've been wanting for a long while to propose an API for caching methods, and that could be made Ractor compatible and would resolve most of these cases, but maybe it's too early to consider it?

I was thinking `cache_method :foo` could "magically" cache the result of `foo` on first call (per Ractor), store it and return that in the future, with no possibility to invalidate that cache or guarantee that the method `foo` couldn't be called a few times in case of race conditions.

#23 - 10/21/2020 01:28 PM - Dan0042 (Daniel DeLorme)

marcandre (Marc-Andre Lafortune) wrote in [#note-22](#):

I've been wanting for a long while to propose an API for caching methods, and that could be made Ractor compatible and would resolve most of these cases

+1

The `@var ||= expr` idiom works well for most cases of memoization, but breaks down for nil values and frozen objects. It's easy enough to handle these cases with a small memoization library (and in fact I do so in my code). But since frozen objects are now going to be everywhere because of ractor, I think it's time to introduce this in core.

#24 - 10/21/2020 06:46 PM - Eregon (Benoit Daloze)

- Related to Feature #17274: `Ractor.make_shareable(obj)` added

#25 - 10/21/2020 07:03 PM - Eregon (Benoit Daloze)

[marcandre \(Marc-Andre Lafortune\)](#) Thanks, I didn't get the use-case from the DeepCover above (I just missed the freeze definition).

I think the last variant of #20 works too if we call `user freeze` instead of `internal freeze`. As long we call it before iterating `ivars/reachable_objects` of that object, we should be fine.

#26 - 10/23/2020 08:00 PM - ko1 (Koichi Sasada)

Sorry I misunderstood this proposal.
`Ractor.make_shareable(obj)` proposal ([#17274](#)) satisfies the functionality proposed here.
(I thought `deep_freeze(skip_shareable: false)` was the proposal. sorry I skipped to read).

We discussed about the name "deep_freeze", and Matz said `deep_freeze` should be only for freezing, not related to Ractor. So classes/module should be frozen if `[C].deep_freeze`. This is why I proposed a `Object#deep_freeze(skip_shareable: true)` and `Ractor.make_shareable(obj)`.

Anyway maybe the functionality should be okay (calling freeze by `make_shareable` proposal).

So naming issue is remained?

- `Object#deep_freeze` (matz doesn't like it)
- `Object#deep_freeze(skip_shareable: true)` (I don't know how Matz feel. And it is difficult to define `Class/Module/...` on `skip_shareable: false`)
- `Ractor.make_shareable(obj)` (clear for me, but it is a bit long)
- `Ractor.shareable!(obj)` (shorter. is it clear?)
- `Object#shareable!` (is it acceptable?)
- ... other ideas?

#27 - 10/25/2020 01:39 PM - Eregon (Benoit Daloze)

- Related to Feature #17273: `shareable_constant_value pragma` added

#28 - 10/25/2020 01:45 PM - Eregon (Benoit Daloze)

ko1 (Koichi Sasada) wrote in [#note-26](#):

So classes/module should be frozen if [C].deep_freeze.

I think it's very rare somebody would want that.

Would that freeze the Array class too because the class is reachable from the object?

And since the superclass and the constants are reachable from a module, every single named module and constant would be frozen?

It could be a useful thing, but IMHO that is never the intention of obj.deep_freeze.

So stopping at the class seems natural (and also never freezing modules).

(we could also have skip_modules: defaults to true)

Also shareable-but-not-immutable objects seem to have little point to freeze them (Ractor/Thread::TVar).

So I think skip_shareable could default to true for deep_freeze:

<https://bugs.ruby-lang.org/issues/17273#note-9>

I don't think any name besides deep_freeze is going to be as intuitive.

#29 - 10/25/2020 07:33 PM - marcandre (Marc-Andre Lafortune)

Not only should the default be skip_shareable: true, I don't really see the use-case for skip_shareable: false. Unless someone comes up with one, the option should probably be removed altogether, or maybe we could start without the option and see.

If Object#make_shareable is preferred, that's ok too, but I still think deep_freeze is clearer.

#30 - 10/25/2020 09:30 PM - Dan0042 (Daniel DeLorme)

Let's say that in the future Array becomes Ractor-safe (i.e. shareable). Would that mean then that [].deep_freeze would no longer freeze the array?

#31 - 10/26/2020 05:28 AM - marcandre (Marc-Andre Lafortune)

Dan0042 (Daniel DeLorme) wrote in [#note-30](#):

Let's say that in the future Array becomes Ractor-safe (i.e. shareable). Would that mean then that [].deep_freeze would no longer freeze the array?

My understanding is that will never be the case. You can't have 1) fast and 2) concurrent access to mutable data. There might very well be a SharedArray class, but I don't see how Array could ever become shareable.

#32 - 10/26/2020 07:46 AM - Eregon (Benoit Daloze)

marcandre (Marc-Andre Lafortune) wrote in [#note-31](#):

My understanding is that will never be the case. You can't have 1) fast and 2) concurrent access to mutable data.

Digressing a bit, I would say it's possible ([thread-safe Array](#)), but not with Ractor's model which requires copying or moving objects.

So yes, I think a mutable Array will never become Ractor-shareable because it would be too incompatible (due to copying/moving the elements or only accepting shareable values).

Also if we have a type for which it becomes problematic, we could always add some keyword argument to handle it specially.

I think freezing all named modules and their constants could possibly be something useful for Ractor, but that IMHO should be its own method, like Ractor.freeze_all_modules or so. I guess in practice it doesn't work well at all, as it becomes very likely with more code, that some code needs mutable modules (e.g., @ivars on modules, autoload & lazy method definitions, ...).

Maybe Ractor.freeze_all_constants is more useful. But again with more code there is a high chance something expects a mutable constant, and that would work fine if non-main Ractors don't use it.

Anyway, they are special cases, and I think for #deep_freeze, it's just not interesting to freeze modules.

BTW, https://github.com/dkubb/ice_nine does not freeze modules either.

(ruby -rice_nine -e 'IceNine.deep_freeze(Enumerable); p Object.frozen?' => false)

#33 - 10/26/2020 07:58 AM - matz (Yukihiro Matsumoto)

- Status changed from Open to Rejected

This looks very interesting, but it would introduce quite big incompatibility. I don't want to break existing code.

Matz.

#34 - 10/26/2020 09:02 AM - matz (Yukihiro Matsumoto)

I made a mistake in the previous message (the reply was for another issue). Sorry for confusion.

As far as I understand, the behavior of freeze and make_shareable are different. So making deep_freeze Ractor-aware does not make sense.

Matz.

#35 - 10/26/2020 01:36 PM - Dan0042 (Daniel DeLorme)

marcandre (Marc-Andre Lafortune) wrote in [#note-31](#):

There might very well be a SharedArray class, but I don't see how Array could ever become shareable.

Sorry, I think my point was not very clear. Replace "Array" with any class name. If deep_freeze does not freeze the object (because it's shareable), that makes its semantics kinda confusing. But it seems Matz already thinks so too.