Recursive freezing?

12/21/2009 12:50 PM - marcade (Marc-Andre Lafortune)

Status: Rejected
Priority: Normal
Assignee: matz (Yukihiro Matsumoto)
Target version: 2.6

Description

I like freezing my constants, config files I read, etc... I believe it is the typical use case for #freeze.

In all theses cases, what I really want to do is freeze everything. There is often no easy way to do this (e.g. for config files), or else one must explicitly call freeze a bunch of times, like:

```ruby
DEFAULT_SEND_FILE_OPTIONS = {
  :type => 'application/octet-stream'.freeze,
  :disposition => 'attachment'.freeze,
}.freeze
```

It would be very nice if there was an easy way to freeze recursively arrays, hashes, etc...

A solution would be for #freeze to accept a level argument (similar to flatten, but the default being 1), or alternatively a boolean one (recursive = false).

Should I write a patch for this feature request?

Thanks,

Marc-André

Related issues:

Related to Ruby master - Feature #17145: Ractor-aware `Object#deep_freeze` - Rejected

History

#1 - 12/21/2009 01:02 PM - shyouhei (Shyouhei Urabe)

=begin

It would be very nice if there was an easy way to freeze recursively arrays, hashes, etc...

The difficult part is those "etc". Imagine how can you freeze a lambda and its "everything"?

```ruby
irb(main):001:0> a = 0
=> 0
irb(main):002:0> p = lambda { p a }.freeze
=> #<Proc:0x00007fc681b2e310>(irb):2
irb(main):003:0> p.frozen?
=> true
irb(main):004:0> a = 1
=> 1
irb(main):005:0> p.call
1
=> nil
irb(main):006:0>
=end
```

#2 - 12/21/2009 01:16 PM - RickDeNatale (Rick DeNatale)

=begin

On Sun, Dec 20, 2009 at 11:02 PM, Shyouhei Urabe redmine@ruby-lang.org wrote:

Issue #2509 has been updated by Shyouhei Urabe.

```ruby
irb(main):001:0> a = 0
```

03/14/2022
It would be very nice if there was an easy way to freeze recursively arrays, hashes, etc...

The difficult part is those "etc". Imagine how can you freeze a lambda and its "everything"?

```ruby
irb(main):001:0> a = 0
  => 0
irb(main):002:0> p = lambda { p a }.freeze
  => #<Proc:0x00007fc681b2e310>(irb):2
irb(main):003:0> p.frozen?
  => true
irb(main):004:0> a = 1
  => 1
irb(main):005:0> p.call
  => nil
```

The other danger to consider is where does 'everything' end. This made me think of Ice Nine from Kurt Vonnegut's Cat's Cradle

http://en.wikipedia.org/wiki/Ice-nine

---

Rick DeNatale
Blog: http://talklikeaduck.denhaven2.com/
Twitter: http://twitter.com/RickDeNatale
WWR: http://www.workingwithrails.com/person/9021-rick-denatale
LinkedIn: http://www.linkedin.com/in/rickdenatale

---

#3 - 12/21/2009 03:14 PM - marcandre (Marc-Andre Lafortune)

I'm sorry for not having been more precise.

My proposal would affect directly Array, Hash, Range & Struct. Traversal would be guarded against infinite recursion, like #inspect and al. Enumerator simply forwards the recursive call to the arguments used when constructing them, as if these were stored as instance variables.

In std libraries:
- OpenStruct: nothing special to do (since sub objects are stored as instance variables)
- Delegate: To be more intuitive, they probably should not decrement the level to freeze, so that freezing a given object by 2 levels would have a similar effect than freezing its delegate by 2 levels.
- Set: Uses a hash, so should also forward the call without decrementing the level. Freezing a set would thus have similar effect to freezing an array.
- StringIO: Forwards the call to the underlying string.

I feel the the lambda example is not an issue. The local variable "a" doesn't belong to the lambda, and lambdas have no mutating methods anyways, so any lambda.freeze, any lambda.freeze(-1) have the same (trivial) effect (unless they somehow get instance variables, in which case these get frozen like any other object).

#4 - 04/02/2010 08:43 AM - znz (Kazuhiro NISHIYAMA)

- Target version changed from 1.9.2 to 2.0.0

#5 - 09/14/2010 04:09 PM - shyouhei (Shyouhei Urabe)

- Status changed from Open to Assigned

#6 - 10/25/2012 08:01 PM - yhara (Yutaka HARA)

- Description updated
Object#deep_freeze means you can freeze everything recursively, by definition. But what if the target object accidentally refers objects that are not supposed to be frozen (e.g. classes)? That would be disaster.

So, I’d rather prefer limited deep_freeze e.g.

```ruby
module DeepFreezable
  def deep_freeze
    self.each_reference do |o|
      o.deep_freeze
    end
  end
end
```

So I reject this Object#deep_freeze idea.

Matz.

Thanks for considering it.

I am a bit surprised about the reason for your rejection.

I mean, there are many ways in Ruby to shoot yourself in the foot and that is not a problem in my opinion.

I'm also sceptical as to how many objects that you would want to freeze contain a reference to a class.

In any case, if the problem really is of classes freezing, it would be easy to make Class#deep_freeze private to prevent that.

This looks to me like a strange reason to reject too. If a programmer recursively freezes an object that will freeze things that the programmer did not want to freeze, I think the program will probably crash immediately (with error messages mentioning frozen objects), and the programmer will either discover a bug, or will learn better how the program works.

P.S. Maybe the concern is that because of Duck Typing, programmers are not expected to know what their objects are...

P.P.S.(2014-06-16) It is not even necessary to have any unfrozen object: programming with everything immutable should work well too.

I have thought of the following solution: add simultaneously Object#deep_freeze and Object#insulated?. If obj.insulated? is false, obj.deep_freeze (or deep_freeze! by the way?) should raise an error.

I agree. Even if we allow a programmer to shoot themselves, they must be able to know what's going on when they do so. If a collection cannot be frozen at once, that should be notified or detected somehow.

I think we can get the effect of having an error for edge cases (deep freezing a class, for example) by simply removing Module#deep_freeze. I feel there's no need to add a new method like insulated?, simply ask instead respond_to?(:deep_freeze).