Add a way to request a frozen string from to_s

Much of the time when a user calls to_s, they are just looking for a simple string representation to display or to interpolate into another string. In my brief exploration, the result of to_s is rarely mutated directly.

It seems that we could save a lot of objects by providing a way to explicitly request a frozen string.

For purposes of discussion I will call this to_frozen_string, which is a terrible name.

This would reduce string allocations dramatically when applied to many common to_s calls:

- Symbol#to_frozen_string could always return the same cached String representation. This method is heavily used by almost all Ruby code that intermingles Symbols and Strings.
- nil, true, false, and any other singleton values in the system could similarly cache and return the same String object.
- The strings coming from core types could also be in the fstring cache and deduplicated as a result.
- User-provided to_s implementations could opt-in to caching and returning the same frozen String object when the author knows that the result will always be the same.

A few ideas for what to call this:

- to_fstring or fstring reflects internal the "fstring" cache but is perhaps not obvious for most users.
- to_s(frozen: true) is clean but there will be many cases when the kwargs hash doesn't get eliminated, making matters worse.
- def to_s(frozen = false) would be mostly free but may not be compatible with existing to_s params (like Integer#to_s(radix)

This idea was inspired by schneems (Richard Schneeman)'s talk at RubyConf Thailand, where he showed significant overhead in ActiveRecord from Symbol#to_s allocation.

Related issues:
- Related to Ruby master - Feature #16153: eventually_frozen flag to gradually ...

Associated revisions

Revision 6ffc045a - 09/26/2019 08:23 AM - Eregon (Benoit Daloze)
[EXPERIMENTAL] Make Symbol#to_s return a frozen String

- Always the same frozen String for a given Symbol.
- Avoids extra allocations whenever calling Symbol#to_s.
- See [Feature #16150]

Revision 9d0866c7 - 09/26/2019 11:25 AM - byroot (Jean Boussier)
[EXPERIMENTAL] Make Module#name return a frozen String

* Always the same frozen String for a given Module or Class.
* Avoids extra allocations whenever calling Module#name.
* See [Feature #16150]

Revision eff15a26 - 09/27/2019 04:52 AM - byroot (Jean Boussier)
[EXPERIMENTAL] Make NilClass#to_s, TrueClass#to_s and FalseClass#to_s return a frozen String

* Always the same frozen String for each of these values.
* Avoids extra allocations whenever calling these 3 methods.
* See [Feature #16150]

Revision 4d3502d5 - 09/27/2019 05:01 AM - nobu (Nobuyoshi Nakada)
More tests for [Feature #16150]
Revision bea322a3 - 11/05/2019 08:30 AM - naruse (Yui NARUSE)

Revert "[EXPERIMENTAL] Make Symbol#to_s return a frozen String [Feature #16150]"

This reverts commit 6ff0c045a817fbd04a6945d3c260b55b0fa1fd1e.

History

#1 - 09/07/2019 05:22 AM - schneems (Richard Schneemann)

Thank you for writing up the proposal. This would certainly helped my optimization case. Here’s the PR I referenced in my talk where I had to work around the allocations from calling to_s on a symbol:  https://github.com/rails/rails/pull/34197

#2 - 09/07/2019 08:51 AM - headius (Charles Nutter)

Another idea: #to_z

#3 - 09/07/2019 09:29 AM - Eregon (Benoit Daloez)

Another possibility: make/let #to_s methods of core types return frozen strings.

The general contract of to_s would become: "return a String representing the object, which might be frozen. Use .to_s.dup if you need a mutable string".

That would be less compatible but would have the advantage of existing code already being able to use those savings without changes and complications.

My impression is to_s does already not guarantee to return a new String.

For example, String#to_s returns itself and mutating sth.to_s is therefore not safe if sth can be a String.

Also, to_s definitions for user classes seem likely to already return a frozen String simply because they might use a String literal (interpolated or not) for it and frozen-string-literal: true.

#4 - 09/07/2019 11:27 AM - shevegen (Robert A. Heiler)

Just a short comment - I originally had to write more, but I think my statement would be too diluted.

Also, to_s definitions for user classes seem likely to already return a frozen String simply because they might use a String literal (interpolated or not) for it and frozen-string-literal: true.

IMO, it currently can not be assumed to hold true because there may also be frozen-string-literal: false settings, and frozen strings are not yet the default nor will be in ruby 3.0, so I would not assume this the case right now. This may change in the future, but right now I think it would confuse people if to_s and to_str do NOT return a string or string-like object; and if a string is returned, it may be frozen, despite frozen-string literals set to false. The latter would qualify as a bug in my opinion.

I have no specific opinion about the suggested APIs, but the names are a bit strange; or a bit cumbersome. "fstring" reads a bit like "formatted string".

Oddly enough, I think #to_z is a bit better simply because it is shorter, but people may still ask why the "z" is there. I guess to_f was not offered as suggestion for frozen, due to it already meaning to a float representation. :) But in general, I don't really have any pro or con opinion either way. (Most of the suggestion seems to be more inspired about speed/efficiency and less about specific end-user use.)

#5 - 09/07/2019 12:05 PM - Dan0042 (Daniel DeLorme)

Great idea.

This definitely has to be a new method because using to_s(frozen: true) would require to know if the object supports the frozen keyword for its to_s method.

As a method, the default implementation would be to_s("a","0",freeze and interpolations can automatically use that. Then it's just a matter of changing various classes (such as Symbol) so that to_frozen_string is the master implementation and to_s becomes to_frozen_string.dup

Maybe it's possible for the parser to lexically convert the expressions -expr.to_s and << expr.to_s to use to_frozen_string

Naming is hard. Short is good because we want to encourage using that method as much as possible. to_fstr, to_s!, to_fs, to_fzs
For example, String#to_s returns itself and mutating sth.to_s is therefore not safe if sth can be a String.

This is a very good point. I'd love to see all immutable core types start returning frozen string but I didn't want to push too hard. It would mean immediate benefits without any changes in existing apps and libraries.

Naming is hard. Short is good because we want to encourage using that method as much as possible. to_fstr, to_s!, to_fs, to_fzs

Oh, to_s! is pretty good.

For example, String#to_s returns itself and mutating sth.to_s is therefore not safe if sth can be a String.

String#to_s is of course a special case. But that means if a string is currently not frozen, to_frozen_string would have to do dup.freeze... doesn't that ruin the entire point of this proposal, which is about efficiency? It would mean "a #\{str\}" creates a duplicate of str because interpolation uses the supposedly efficient to_frozen_string. So maybe this proposal really should be called to_efficient_string, which doesn't guarantee a frozen string but rather attempts to minimize allocations. In that context, to_s! makes even more sense because you don't know what kind of string you'll get.

How about specifically making Symbol#to_s return a frozen String?

I tried this trivial patch:

```diff
diff --git a/string.c b/string.c
index 05ce0ed8d6..1a0fa48a6a 100644
--- a/string.c
+++ b/string.c
@@ -10866,7 +10866,9 @@ sym_inspect(VALUE sym)
     VALUE
     rb_sym_to_s(VALUE sym)
     {
-       return str_new_shared(rb_cString, rb_sym2str(sym));
+       VALUE str = str_new_shared(rb_cString, rb_sym2str(sym));
+       OBJ_FREEZE(str);
+       return str;
     }
```

And there are 0 test-spec failures and 0 test-all failures.

So, let's make Symbol#to_s frozen?

I think in general it makes a lot of sense that immutable classes return a frozen String for #to_s.

Making #to_s return a frozen String for mutable core classes like Array or Hash is likely much less interesting, as one cannot cache a single String instance but must check the object's contents everytime (and the #to_s result depends on other objects' #to_s which makes it a lot more complicated).

I also tried this version which always returns the same String for a given Symbol:

```diff
diff --git a/string.c b/string.c
index 05ce0ed8d6..f306c905bc 100644
--- a/string.c
+++ b/string.c
@@ -10866,7 +10866,7 @@ sym_inspect(VALUE sym)
     VALUE
     rb_sym_to_s(VALUE sym)
     {
-       return str_new_shared(rb_cString, rb_sym2str(sym));
+       return rb_sym2str(sym);
     }
```

And that passes both test-spec and test-all, except this test which probably needs to be adapted:
I'll make a PR.

#10 - 09/08/2019 09:58 AM - Eregon (Benoit Daloze)
PR at https://github.com/ruby/ruby/pull/2437

#11 - 09/08/2019 10:10 AM - ashmaroli (Ashwin Maroli)
Similar to Symbol#to_s, even nil.to_s allocates a new string.

```ruby
irb(main):001:0> 5.times { p nil.to_s.__id__ }
21118460
21118340
21118180
21118100
21118000
```

IMO, since nil is a singleton, nil.to_s should also return the same empty string on every call.

#12 - 09/08/2019 10:52 AM - sawa (Tsuyoshi Sawada)
For methods that convert object to a string, we already have three:

- `to_s`
- `to_str`
- `inspect`

Adding yet another method (or an option for an existing method) would make the picture too complicated. It might be a good time to reconsider the division of labor between the existing three methods.

It is suspicious whether the distinction between `to_s` vs. `to_str`, `to_h` vs. `to_hash`, `to_a` vs. `to_ary`, or `to_i` vs. `to_int` etc. is nowadays used in the way it was intended at the beginning. Note that, while the single splat `*` on an object implicitly calls the explicit `to_a`, the double splat `**` implicitly calls the implicit `to_hash`. Also note that there is no explicit counterpart for the implicit method `to_proc`, which is used as in Symbol#to_proc, Method#to_proc, Hash#to_proc, where the (missing) explicit counterpart should be used.

Since the implicit vs. explicit distinction is messed up, but we are not having a big problem with it, there is not much motivation to try to maintain such distinction.

Personally, I am not that sure why we need `to_str` just to show that an object is string-like. For example, we could have the argument of Array#join undergo `to_s` (with a fallback to `to_str`) instead of (just) `to_str`. I don't see any problem with that.

As an example of rearranging the division of labour, one method out of the three mentioned above could be used for creating a new string instance, and another method for referencing a cached representation.

But that is just an example. There may be other ways that can make more sense. The point is, we can perhaps make use of the existing methods rather than adding a new one.

#13 - 09/08/2019 08:16 PM - Eregon (Benoit Daloze)
ashmaroli (Ashwin Maroli) wrote:

Similar to Symbol#to_s, even nil.to_s allocates a new string.

Right, as mentioned in the description, nil, true and false#to_s could all be frozen and cached the same way as for Symbol#to_s. Although I'd think calling #to_s on those is already less frequent and performance-sensitive than Symbol#to_s.

Are there other immutable types where it would be worthwhile to cache the #to_s result?
I can't find any now, at least it doesn't seem worthwhile for numeric types, and most other core types are not immutable.

#14 - 09/09/2019 02:36 AM - Dan0042 (Daniel DeLorme)

So, let's make Symbol#to_s frozen?
I think in general it makes a lot of sense that immutable classes return a frozen String for #to_s.

That makes sense, and I agree that's the cleanest solution. There's really nothing in the contract that says to_s is supposed to return a non-frozen
But I can't agree to a backward-incompatible change without a proper deprecation period.
So I went and added Feature #16153 for a way to allow a phase-in period for frozen Symbol#to_s.

I'm unsure if such troubles are necessary, because the practical incompatibility might be extremely low, such as illustrated by in-repository specs and tests.

Do we have an example of real code breaking because of Symbol#to_s returning a frozen String?
Could someone try my PR on their app and see if it breaks anything?

I tried running ActiveSupport tests from Rails master on Ruby 2.6.2 + my PR, and found that one change is needed:

```
Error: 
ConfigurableActiveSupport#test_configuration_accessors_are_not_available_on_instance: 
FrozenError: can't modify frozen String
/home/eregon/code/rails/activesupport/lib/active_support/ordered_options.rb:43:in 'chomp!
/home/eregon/code/rails/activesupport/lib/active_support/ordered_options.rb:43:in 'method_missing'
...
```

https://github.com/rails/rails/blob/7af44f49db221c3b7f0b0d476913a74b6a1d0e4/activesupport/lib/active_support/ordered_options.rb#L42-L43

A simple fix is to use name_string = +name.to_s, or to refactor to use start_with? + name_string[0..-2] instead of chomp!
Then all ActiveSupport tests pass.

Eregon (Benoit Daloze), for context Matz recently refused a very similar change https://bugs.ruby-lang.org/issues/15836

For frozen Symbol#to_s, I see a clear benefit. But I worry a little bit for incompatibility. So how about making an experiment by the next preview(2) to see how big the incompatibility is?
String#to_s etc. is a different story. we need to discuss further.

Matz.

I get you are worried about String#to_s, but what about others that have been mentioned here, namely:

- NilClass#to_s
- TrueClass/FalseClass#to_s
- Module#name

So how about making an experiment by the next preview(2) to see how big the incompatibility is?

Thank you, I will merge https://github.com/ruby/ruby/pull/2437 and mark the change as experimental.

String#to_s etc. is a different story.

I agree String#to_s should remain as-is.
Returning a frozen String for String#to_s would just add more allocations, since the current String#to_s just returns self.

what about others that have been mentioned here

I think nil/true/false.to_s and Module#name could be frozen, much like Symbol#to_s.
There might be some incompatible usages but I would expect very rare, and the FrozenError would make it pretty clear how to deal with it

05/15/2021
I get you are worried about String#to_s, but what about others that have been mentioned here, namely:

- NilClass#to_s
- TrueClass/FalseClass#to_s
- Module#name

Ok for the experiment.

Matz.

#22 - 09/26/2019 08:23 AM - Eregon (Benoit Daloze)
- Status changed from Open to Closed

Applied in changeset git|6ffc045a817fbdf04a6945d3c260b55b0fa1fd1e.

[EXPERIMENTAL] Make Symbol#to_s return a frozen String

- Always the same frozen String for a given Symbol.
- Avoids extra allocations whenever calling Symbol#to_s.
- See [Feature #16150]

#23 - 09/26/2019 08:33 AM - Eregon (Benoit Daloze)
- Assignee set to Eregon (Benoit Daloze)
- Status changed from Closed to Assigned

Reopening, the issue was accidentally closed as I merged the Symbol#to_s change. I will make a PR for nil/true/false.to_s and Module#name to return a frozen String.

#24 - 09/26/2019 11:00 AM - byroot (Jean Boussier)
I got the Module#name one ready here: https://github.com/ruby/ruby/pull/2487

For nil/true/false it is a bit trickier because the returned string are ASCII encoded, so I'm not quite sure how to statically create ASCII fstrings with the existing APIs.

#25 - 09/26/2019 11:25 AM - byroot (Jean Boussier)
- Status changed from Assigned to Closed

Applied in changeset git|9d0866c7d7b9cbe36a851744a37806e747e0e7a8.

[EXPERIMENTAL] Make Module#name return a frozen String

* Always the same frozen String for a given Module or Class.
* Avoids extra allocations whenever calling Module#name.
* See [Feature #16150]

#26 - 09/26/2019 11:26 AM - Eregon (Benoit Daloze)
- Status changed from Closed to Assigned

byroot (Jean Boussier) Great, I merged your PR. Sorry, I should have asked first if you wanted to do it.

Do you want to do the PR for nil/true/false as well then?

#27 - 09/26/2019 11:50 AM - byroot (Jean Boussier)

Do you want to do the PR for nil/true/false as well then?
I tried but I'm quite unsure how to do it, so I don't mind if you handle it.

On a side note I'm currently running our test suite with the following monkey patch:

```ruby
module FreezeName
  def name
    name = super
    name = -name if name
  end
end
Module.singleton_class.prepend(FreezeName)
```

```ruby
module FreezeToS
  def to_s
    super
  end
end
Symbol.prepend(FreezeToS)
NilClass.prepend(FreezeToS)
TrueClass.prepend(FreezeToS)
FalseClass.prepend(FreezeToS)
```

This way I should be able to identify some breakage across a large amount of gems.

#28 - 09/26/2019 01:23 PM - nobu (Nobuyoshi Nakada)

byroot (Jean Boussier) wrote:

For nil/true/false it is a bit trickier because the returned string are ASCII encoded, so I'm not quite sure how to statically create ASCII fstrings with the existing APIs.

```ruby
static VALUE
true_to_s(VALUE obj)
{
  return rb_fstring_enc_lit("true", rb_usascii_encoding());
}
```

#29 - 09/26/2019 01:35 PM - byroot (Jean Boussier)

Ok, so I just finished updating our application to be compatible with both Symbol#to_s and Module#name returning frozen strings.

Over the 502 gems used in the repo, I found 5 impacted gems, all because of Symbol#to_s, and all in very minor ways:

- A method_missing in Pry
- Some string manipulation in the grpc gem
- Also some encoding coercion in activerecord-databasevalidations
- The memoist gem
- Another method_missing in google-cloud-core
- And of course the Rails one mentioned above but it got fixed this morning.

Inside our repo of over 1 million lines of code, I found 4 impacted methods, all were trivially fixed. Again they were all caused by Symbol#to_s, and very similar in nature to the public ones I listed above.

Overall I was able to trivially update a gigantic app to be ready for this change, all that in under an hour.

#30 - 09/26/2019 01:37 PM - byroot (Jean Boussier)

nobu (Nobuyoshi Nakada) Yes that's what I had in mind, but then from what I understand it means we have to lookup the fstring table every time rather than just return a existing pointer. So it's not really an optimisation, is it?

#31 - 09/26/2019 02:32 PM - mame (Yusuke Endoh)

byroot (Jean Boussier) wrote:

Over the 502 gems used in the repo, I found 5 impacted gems, all because of Symbol#to_s, and all in very minor ways:

A great report. For the record, I add another case I've heard:

- did_you_mean: https://github.com/yuki24/did_you_mean/pull/125/commits/ab41208165828b1df31b603e9967ea48b7fad022
Currently, seven impacted cases are found.

#32 - 09/26/2019 02:42 PM - nobu (Nobuyoshi Nakada)
Then store those objects in global variables, and rb_gc_register_mark_object.

#33 - 09/26/2019 09:39 PM - headius (Charles Nutter)
I'm glad to see some smaller parts of this are moving forward. A few updates from my end (JRuby):

- Frozen strings from Symbol etc:
  We are on board with making Symbol, nil, true, false, and Module all return frozen strings, and have an open PR to do this for Symbol right now: https://github.com/jruby/jruby/pull/5868
  - String#to_s concerns:
    Right now since it returns self, it may be frozen or unfrozen, so you are NEVER safe mutating the result of to_s directly. With the changes above, even more types will start returning frozen strings. I feel like we need a way to audit the mutation of to_s results we want to make frozen in the future, similar to the --debug:frozen-string-literal feature that prints out where a frozen string came from if you try to mutate it.
    I guess my logic boils down to this right now:
    - to_s often returns a frozen string right now.
    - to_s will return more frozen strings in the future if these PRs make it into 2.7.
    - Both now and in the future, it is not safe to mutate the results of a to_s.
    - So, let's give people a way to find these mutations as part of 2.7, so we can help eliminate such cases.

#34 - 09/26/2019 11:14 PM - byroot (Jean Boussier)
headius (Charles Nutter) I totally get your opinion on String#to_s, however on a very practical standpoint I agree with Eregon (Benoit Daloze):

Returning a frozen String for String#to_s would just add more allocations, since the current String#to_s just returns self.

If we were to freeze the return of String#to_s I wouldn't be surprised if we ended up causing much more allocation that we're saving with Symbol#to_s and Module#name.

But granted that this now cause this somewhat weird situation where to_s might or might not return a frozen string, that being said it's already kinda the case:

```ruby
>> "foo".freeze.to_s.frozen?
=> true
```

#35 - 09/26/2019 11:49 PM - headius (Charles Nutter)

But granted that this now cause this somewhat weird situation where to_s might or might not return a frozen string, that being said it's already kinda the case

Yeah, this is precisely the problem. It is made even worse by the large number of libraries that already set frozen-string-literal.

One thing we should all be able to agree on at this point: it is not safe to blindly mutate the result of String#to_s, since more and more cases will produce frozen strings. With the experimental changes accepted above, an additional set of to_s results will also be frozen. So, by extension, it is not now and will never be safe to blindly mutate the result of calling any to_s.

I'd argue it's actually NEVER safe to modify the result of calling to_s, because for mutable source strings you'd be mutating the original contents! This is probably not desired behavior in the large majority of existing code. With that condition in mind, it would actually be much safer for us to start returning frozen strings from String#to_s as soon as possible.

I know this is a difficult pill to swallow. The original purpose of this issue was to consider adding a way to request frozen strings from to_s, since that would at least be an opt-in. Hence the suggestions of a to_z or similar new mechanism.

In the 2.7 timeframe, I'd like to see some combination of the following:

- A way to explicitly request a frozen string from any object, such as to_z, so code could start migrating toward frozen strings today.

This would be an extension of existing "str".freeze and "-str" logic for explicitly requesting a single frozen string literal, but would allow making this request for any to_s result.

- A debug mode that would warn if code attempts to mutate the result of String#to_s (e.g. --debug:frozen-string-to_s), since based on the above conditions this is almost never advisable.
This will help us audit existing code and start “fixing” it right now without a hard break. I’d like to see the above warning all the time, but that would require tracking source file and line for all literal strings (as in the current --debug:frozen-string-literal).

- A pragma to explicitly set a file as having mutable-string-literal, as a local escape hatch for future default frozen-string-literal.
- A command-line flag to force mutable-string-literal when no pragma exists, as a global escape hatch for future default frozen-string-literal.

These are a simple way to guarantee a soft landing if (when?) we default to frozen-string-literal globally in 3.0. All cases that would break with default frozen-string-literal could at least be made to work with the flag.

#36 - 09/26/2019 11:53 PM - headius (Charles Nutter)
Oh I should point out another problem with the current situation...

```
some_str.to_s.freeze
```

The above code is also unsafe right now, because since to_s returns self, we're now freezing the original string!

In order to safely get a frozen string from an object, you must ALWAYS do some_str.to_s.dup.freeze. In order to safely get a mutable string from an object, you must ALWAYS do some_str.to_s.dup.

So what are we saving by leaving the current unsafe behavior in place?

#37 - 09/27/2019 04:53 AM - byroot (Jean Boussier)
- Status changed from Assigned to Closed

Applied in changeset git|eff15a269fde37d2b09cf1dfe8c1b1b6e377a32.

[EXPERIMENTAL] Make NilClass#to_s, TrueClass#to_s and FalseClass#to_s return a frozen String

* Always the same frozen String for each of these values.
* Avoids extra allocations whenever calling these 3 methods.
* See [Feature #16150]

#38 - 09/27/2019 08:12 AM - byroot (Jean Boussier)

In order to safely get a frozen string from an object, you must ALWAYS do some_str.to_s.dup.freeze. In order to safely get a mutable string from an object, you must ALWAYS do some_str.to_s.dup.

Not exactly, -@ and +@ makes this much simpler.

-@ will return self if the string is already frozen.

+@ will return self if the string is already mutable.

So these two allows you to safely get either a frozen or mutable string with the minimum amount of allocations.

#39 - 09/28/2019 04:33 AM - headius (Charles Nutter)

Not exactly, -@ and +@ makes this much simpler

I do like the unary operators, but they also have some precedence oddities:

```
>> "foo".size
=> -3
>> (%"foo").size
=> 3
```

And it doesn't work at all if you're chaining method calls:

```
>> ary.to_s.frozen?
NoMethodError: undefined method `+0' for false:FalseClass
    from (irb):8
    from /usr/bin/irb:11:in `<main>'
```

But you are right, instead of the explicit dup with possible freeze you could use - or + on the result of to_s. However it's still not safe to modify it since it would modify the original string too.
Hi,

In message "Re: [ruby-core:95142] [Ruby master Feature#16150] Add a way to request a frozen string from to_s" on Sat, 28 Sep 2019 04:33:32 +0000 (UTC), headius@headius.com writes:

Issue #16150 has been updated by headius (Charles Nutter).

Not exactly, -@ and +@ makes this much simpler

I do like the unary operators, but they also have some precedence oddities:

```ruby
>> -"foo".size
 => -3
>> (-"foo").size
 => 3
```

And it doesn't work at all if you're chaining method calls:

```
How about making String#+ and #- without argument behave like #+@ and #@ respectively, so that we can write:

"foo"-.size
ary.+.to_s.frozen?
```

matz.

#41 - 09/29/2019 12:29 PM - byroot (Jean Boussier)

However it's still not safe to modify it since it would modify the original string too.

IMHO that's two different use cases. Either:

- You want to be sure not to mutate the string, then you use .dup
- You don't care about mutating the original string, you just want to minimize allocations, then you use +@.

It's similar with freeze and -@. Either:

- You simply want to prevent the string from being mutated, you use .freeze
- You know you're going to hold on that string and want to optimize memory retention, you use -@.

How about making String#+ and #- without argument behave like #+@ and #@ respectively, so that we can write:

```
IMHO .- and .+ is not very elegant. Proper method names explaining the intent would be preferable.

- @ could be dedup, or deduplicate.
- +@ could be mutable or mut.
```

#42 - 09/29/2019 07:29 PM - jonathanhefner (Jonathan Hefner)

- @ could be dedup, or deduplicate.

dedup is an interesting choice because it implies the opposite of dup.

Although you might run into naming collisions if/ when you extend this functionality to other types. For example, I would expect Array#dedup to have a very different meaning, entirely unrelated to freezing.

Another alternative might be something like intern_s, since String#intern already exists. (Although this might also be awkward to extend to other types. For example, should it be Array#intern_a for consistency or simply Array#intern?)

#43 - 09/29/2019 10:08 PM - Eregon (Benoit Daloze)

I'll get to the point because I think we're starting to discuss unrelated things in this issue.

headius (Charles Nutter) wrote:

I'd argue it's actually NEVER safe to modify the result of calling to_s [...]

05/15/2021
Yes, and it has never been. String#to_s has always returned itself, including for frozen strings, so of course to_s means "convert to a String if you're not already" and gives no guarantee to return a new String ready to be mutated. It is the same for all other to_* methods, isn't it?

I think this is already well understood by Rubyists, and the changes for Symbol#to_s, Module#name, true/false/nil.to_s just make an existing behavior a bit more common. So, I think there is nothing to change about String#to_s, except maybe improving the documentation of Kernel#to_s in general.

I know this is a difficult pill to swallow. The original purpose of this issue was to consider adding a way to request frozen strings from to_s, since that would at least be an opt-in. Hence the suggestions of a to_z or similar new mechanism.

I think there is no need for yet another "convert to String" method. We changed the methods that used to allocate on every call instead of returning an internal frozen String (I don't think there are others, are there?). Making String#to_s not return self is just worse for allocations, and would be inconsistent with other to_* methods. Is there any advantage to make String#to_s not return self?

- A way to explicitly request a frozen string from any object, such as to_z, so code could start migrating toward frozen strings today.

Why are String#-@ and str.dup.freeze not enough?

- A pragma to explicitly set a file as having mutable-string-literal, as a local escape hatch for future default frozen-string-literal.
- A command-line flag to force mutable-string-literal when no pragma exists, as a global escape hatch for future default frozen-string-literal.

That's about frozen string literals, #11473, please comment there, I don't think we should mix both issues.

In fact, I think this issue is done and is already correctly closed. We addressed the cases that were allocating too much, without the need for a new method for converting to strings.

#44 - 10/21/2019 01:09 PM - byroot (Jean Boussier)
Quick update on compatibility with this change.

I opened PRs on the 5 affected gems I found:

- https://github.com/googleapis/google-cloud-ruby/pull/4109
- https://github.com/matthewrudy/memoist/pull/82
- https://github.com/grpc/grpc/pull/20417
- https://github.com/pry/pry/pull/2084
- https://github.com/wvanbergen/activerecord-databasevalidations/pull/23

So far 2 have been merged and released, and 3 are awaiting a reaction from their respective maintainers.

#45 - 11/01/2019 12:55 AM - hsbt (Hiroshi SHIBATA)
This change must be reverted.

I got the issue report from Heroku users.

https://github.com/heroku/heroku-buildpack-ruby/issues/833#issuecomment-548592514

When Rails 6.0.0 and old version users specified Ruby 2.7, Their application is not working with Ruby 2.7.

#46 - 11/01/2019 08:54 AM - matz (Yukihiro Matsumoto)
OK, frozen Symbol#to_s should be reverted. Maybe we can challenge in the future (with the deprecation process first).

Matz.

#47 - 11/01/2019 05:43 PM - Eregon (Benoit Daloze)
hsbt (Hiroshi SHIBATA) wrote:

This change must be reverted.

I don't think it's fair to decide that on your own, from apparently just one issue.

Could you detail the situation, i.e., in which place the problem occurs and why can't we make a release of that gem before 2.7.0 is out? The user seems to say it's not caused by Rails.
rafaelfranca (Rafael França) told me Rails 6.0.1 is scheduled for November 5, way before the Ruby 2.7 release (see https://weblog.rubyonrails.org/2019/10/31/Rails-6-0-1-rc1-released/).
So if that issue is caused by Rails, it should be fixed before any stable release of Ruby 2.7 comes out.

"Deprecation" for String#to_sym to return a frozen String seems unfortunately very complicated. 
#16153 exposes one way to do it but that didn't get much agreement, "half-frozen Strings" don't make much sense to me and I would think are not intuitive to many.

So, given that this change has AFAIK a very limited impact and well-evaluated by byroot (Jean Boussier), I think it is fine to keep the change.
#13083 OTOH is a far more breaking change and that one can easily go through deprecation first.

---

hsbt (Hiroshi SHIBATA)

So if that issue is caused by Rails, it should be fixed before any stable release of Ruby 2.7 comes out.

This change affects many developers, company and applications with upgrading work. How about Rails 5.2 and 5.1 users? Do we abandon them?
You should think about the cost of upgrading the real-world application. The upgrading Rails is different from the one library.

---

rafaelfranca (Rafael França)

This issue is also fixed on Rails 5.2 and will be released in the next version of Rails.

Users of any version of Rails below 5.2 are already abandoned. The Rails core team don't support those versions anymore. So, they already need to upgrade their applications, with this change or not.

We could still say that users would still need to upgrade to Rails 6.0.1 because of this change but in my opinion that is already required. Users that stay on Rails 6.0.0 would not receive security upgrades without having to upgrade to 6.0.1 (or the latest version before the security patch) first.

---

Eregon (Benoit Daloze)

The plan is Rails 6 users can update to Rails 6.0.1 and Rails 5.2 users can update to Rails 5.2.4 (assuming 5.2.4 will be released before 2.7, we need to confirm with Rails devs).
That should be easy and recommended for security and other bug fixes anyway.

I agree it's not ideal, but I also feel it's really not the biggest problem for Ruby 2.7 in production, and we already have a good plan.
The fact that people trying Ruby 2.7 will get many keyword argument warnings (which also affects performance) seems far more problematic to me (https://bugs.ruby-lang.org/issues/16289#note-5).
And keyword arguments warnings fixes are currently not backported (I guess because they require so many changes).

---

Eregon (Benoit Daloze) wrote:

I agree it's not ideal, but I also feel it's really not the biggest problem for Ruby 2.7 in production, and we already have a good plan.
The fact that people trying Ruby 2.7 will get many keyword argument warnings (which also affects performance) seems far more problematic to me (https://bugs.ruby-lang.org/issues/16289#note-5).
And keyword arguments warnings fixes are currently not backported (I guess because they require so many changes).

Yes, keyword argument is the most difficult challenge for Ruby 2.7.
And it is also very important toward Ruby 3.0.
As a release manager I don't want to add more trouble like Symbol#to_s.
As already Matz stated, I reverted this at bea322a3.

byroot (Jean Boussier)

If that change is reverted, could we make it a deprecation instead?
Any idea how such deprecation warning would be implemented?

---

Eregon (Benoit Daloze)

As already Matz stated, I reverted this at bea322a3.
So you reverted without a single case that was actually problematic due to the change? I'm very disappointed, deprecation without a proper explanation seems inappropriate.

I expect a clear explanation of why this is problematic. It is not problematic for the case of Rails, as clearly explained in the comments above.

matz (Yukihiro Matsumoto) On what did you base your decision? I hope something else than a single unclear bug report.

#55 - 11/05/2019 10:24 PM - Eregon (Benoit Daloze)
I'm sorry if the above is rude.

I find it frustrating that the change is reverted without much else than "a possible incompatibility reported by one user" and for which we already have a plan to address it easily (just use the latest Rails of that release series, which is likely already needed for other fixes).

#56 - 11/06/2019 12:29 AM - duerst (Martin Dürst)
Just to cross-check:

Eregon (Benoit Daloze) wrote:

I didn't find the text "a possible incompatibility reported by one user" anywhere in this issue, Hiroshi (Shibata) writes "I got the issue report from Heroku users.", which indicates it's more than a single user.

and for which we already have a plan to address it easily (just use the latest Rails of that release series, which is likely already needed for other fixes).

I think it's not a bad idea to have a bit of slack for version combinations. If we put too many restrictions on what version of something can be combined with what version of something else, we easily get into situations where there are no viable combinations anymore.

#57 - 11/06/2019 12:34 AM - hsbt (Hiroshi SHIBATA)
This discussion started with "how big the incompatibility is?" at https://bugs.ruby-lang.org/issues/16150#note-18

The old version of Rails is incompatible. And I also got the issue of middleman depends on memoist. So, my company blog built by middleman is broken now with Ruby 2.7.0-preview2.

We found the big incompatibility now. We should restart to discuss this feature.

#58 - 11/06/2019 10:27 AM - Eregon (Benoit Daloze)
duerst (Martin Dürst) wrote:

I didn't find the text "a possible incompatibility reported by one user" anywhere in this issue.

It's my interpretation and the only piece of information I had so far from https://github.com/heroku/heroku-buildpack-ruby/issues/833#issuecomment-548592514

That bug report is rather unclear, apparently even using latest Rails didn't solve the issue, and previous comments on that issue are completely unrelated.

I think it's not a bad idea to have a bit of slack for version combinations. If we put too many restrictions on what version of something can be combined with what version of something else, we easily get into situations where there are no viable combinations anymore.

I would think it's always advisable to update Rails before Ruby.
The other way around seems a recipe for more warnings and more issues.

#59 - 11/06/2019 10:37 AM - Eregon (Benoit Daloze)
hsbt (Hiroshi SHIBATA) wrote:

This discussion started with "how big the incompatibility is?" at https://bugs.ruby-lang.org/issues/16150#note-18

The old version of Rails is incompatible. And I also got the issue of middleman depends on memoist. So, my company blog built by middleman is broken now with Ruby 2.7.0-preview2.

We found the big incompatibility now. We should restart to discuss this feature.
Thank you for the more detailed explanation.

IMHO, requiring to update (not upgrade) Rails before Ruby is good practice anyway, so the fact that 6.0.0 has an issue is unfortunate but not a deal breaker.

FWIW, the PR to memoist has just been merged (not yet released): https://github.com/matthewrudy/memoist/pull/82

I understand having to update multiple dependencies to use Ruby 2.7 is not great.

With this change reverted, are both cases working?

So I guess we should try to deprecate mutable String#to_sym via #16153 now. On the upside, it seems useful to have such a deprecation facility for making more Strings frozen.

#60 - 12/23/2019 03:29 PM - Eregon (Benoit Daloze)
- Assignee deleted (Eregon (Benoit Daloze))

#61 - 12/27/2019 12:13 AM - zunda (zunda an)

With help by mame-san, I found nil.to_s returning a frozen empty string (""") broke the http gem on ruby-2.7.0. I wonder how other gems maybe affected.

The code raises FrozenError when trying to force_encoding after reading a nil from the stream:
https://github.com/httprb/http/blob/v3.3.0/lib/http/response/body.rb#L31
https://github.com/httprb/http/blob/v3.3.0/lib/http/response/body.rb#L52

clearing it also records a warning:

For now, this can be worked around with making sure to have the readpartial method return an unfrozen empty string:
https://github.com/zunda/http/compare/v3.3.0...870d374#diff-b31c2c9884d039633a34d10a344d68b

matz (Yukihiro Matsumoto) wrote:

byroot (Jean Boussier) wrote:

I get you are worried about String#to_s, but what about others that have been mentioned here, namely:

- NilClass#to_s
- TrueClass/FalseClass#to_s
- Module#name

Ok for the experiment.

Matz.

#62 - 12/27/2019 11:37 AM - Eregon (Benoit Daloze)

zunda (zunda an) wrote:

For now, this can be worked around with making sure to have the readpartial method return an unfrozen empty string:

Yes, that looks like a good fix to me. I would indeed expect usages of read and readpartial to assume it to return a newly-allocated mutable String. Using nil#to_s to generate an empty mutable String was arguably rather hacky. I'd write it like chunk || +"" or chunk || "".dup.

#63 - 02/10/2020 07:40 PM - Eregon (Benoit Daloze)

For making Symbol#to_s frozen, see #16153.

#64 - 02/10/2020 07:40 PM - Eregon (Benoit Daloze)

- Related to Feature #16153: eventually_frozen flag to gradually phase-in frozen strings added

#65 - 04/27/2020 10:58 AM - byroot (Jean Boussier)

Could we consider Symbol#to_s retuning frozen strings again?
We've been running with https://github.com/Shopify/symbol-fstring for about 5 months now, and I think the backward incompatibility problem is much less important now.

It's true that 6 months ago it was breaking a handful of very popular gems, but since then they've all been updated except grpc.

#66 - 04/29/2020 11:24 PM - sam.saffron (Sam Saffron)
I can confirm that Discourse bench and Discourse works with symbol-fstring today.

Old versions of pry were broken, but stuff seems fine now.

#67 - 07/08/2020 07:42 PM - headius (Charles Nutter)
I'd like to revisit this since it seems there's anecdotal evidence that returning frozen strings from Symbol#to_s is not as big a compatibility issue as thought.

Doing it for the next major release would make sense.

#68 - 09/04/2020 07:33 AM - matz (Yukihiro Matsumoto)
I admit Symbol#name that returns a frozen string from a symbol. This can be a building block of the proposal.
As a direct solution for this issue, we have to face the naming problem (as always).

Matz.

#69 - 09/10/2020 10:24 AM - tom-lord (Tom Lord)
matz (Yukihiro Matsumoto) wrote in #note-68:

I admit Symbol#name that returns a frozen string from a symbol. This can be a building block of the proposal.
As a direct solution for this issue, we have to face the naming problem (as always).

Matz.

For what it's worth, I agree with the above suggestion to try making a "breaking" change to Symbol#to_s.

Introducing a Symbol#name (https://github.com/ruby/ruby/pull/3514) should be "Plan B", only if there's significant backlash from a release candidate.

#70 - 09/10/2020 01:52 PM - byroot (Jean Boussier)
Honestly now that Symbol#name is exposed, it's very easy to simply replace to_s with Symbol.alias_method(:to_s, :name).