Ruby master - Feature #11473

Immutable String literal in Ruby 3

08/21/2015 01:31 AM - ko1 (Koichi Sasada)

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

Description

Matz said "All String literals are immutable (frozen) on Ruby 3".

This ticket is place holder to discuss about that.

Related issues:

Related to Ruby master - Feature #8976: file-scope freeze_string directive	**Closed**

Associated revisions

Revision 7cf523c7 - 08/21/2015 08:47 PM - ko1 (Koichi Sasada)

- vm_opts.h, iseq.c, iseq.h: add compile option to force frozen string literals. [Feature #11473] This addition is not specification change, but to try frozen string literal world discussed on [Feature #11473]. You can try frozen string literal world using this magical line: RubyVM::InstructionSequence.compile_option = {frozen_string_literal: true} Note that this is a global compilation option, so that you need to compile another script like that: p 'foo'.frozen? #=> false RubyVM::InstructionSequence.compile_option = {frozen_string_literal: true} p 'foo'.frozen? #=> false, because this line is already compiled. p eval("'foo'.frozen?") #=> true Details:
  - String literals are deduped by rb_fstring().
  - Dynamic string literals ("...#{xyz}...") is now only frozen, not deduped. Maybe you have other ideas. Now, please do not use this option on your productions :) Of course, current specification can be changed.
- compile.c: ditto.
- test/ruby/test_iseq.rb: add a test.

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@51659 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 51659 - 08/21/2015 08:47 PM - ko1 (Koichi Sasada)

- vm_opts.h, iseq.c, iseq.h: add compile option to force frozen string literals. [Feature #11473] This addition is not specification change, but to try frozen string literal world discussed on [Feature #11473]. You can try frozen string literal world using this magical line: RubyVM::InstructionSequence.compile_option = {frozen_string_literal: true} Note that this is a global compilation option, so that you need to compile another script like that: p 'foo'.frozen? #=> false RubyVM::InstructionSequence.compile_option = {frozen_string_literal: true} p 'foo'.frozen? #=> false, because this line is already compiled. p eval("'foo'.frozen?") #=> true Details:
  - String literals are deduped by rb_fstring().
  - Dynamic string literals ("...#{xyz}...") is now only frozen, not deduped. Maybe you have other ideas. Now, please do not use this option on your productions :) Of course, current specification can be changed.
- compile.c: ditto.
- test/ruby/test_iseq.rb: add a test.

Revision 51659 - 08/21/2015 08:47 PM - ko1 (Koichi Sasada)

- vm_opts.h, iseq.c, iseq.h: add compile option to force frozen string literals. [Feature #11473] This addition is not specification change, but to try frozen string literal world discussed on [Feature #11473]. You can try frozen string literal world using this magical line: RubyVM::InstructionSequence.compile_option = {frozen_string_literal: true} Note that this is a global compilation option, so that you need to compile another script like that: p 'foo'.frozen? #=> false RubyVM::InstructionSequence.compile_option = {frozen_string_literal: true} p 'foo'.frozen? #=> false, because this line is already compiled. p eval("'foo'.frozen?") #=> true Details:
  - String literals are deduped by rb_fstring().
  - Dynamic string literals ("...#{xyz}...") is now only frozen, not deduped. Maybe you have other ideas. Now, please do not use this option on your productions :) Of course, current specification can be changed.
- compile.c: ditto.
- test/ruby/test_iseq.rb: add a test.

Revision 51659 - 08/21/2015 08:47 PM - ko1 (Koichi Sasada)

- vm_opts.h, iseq.c, iseq.h: add compile option to force frozen string literals. [Feature #11473] This addition is not specification change, but to try frozen string literal world discussed on [Feature #11473]. You can try frozen string literal world using this magical line: RubyVM::InstructionSequence.compile_option = {frozen_string_literal: true} Note that this is a global compilation option, so that you need to compile another script like that: p 'foo'.frozen? #=> false RubyVM::InstructionSequence.compile_option = {frozen_string_literal: true} p

'foo'.frozen? #=> false, because this line is already compiled. p eval("'foo'.frozen?") #=> true Details:
    - String literals are deduped by rb_fstring().
    - Dynamic string literals ("...#{xyz}...") is now only frozen, not deduped. Maybe you have other ideas. Now, please do not use this option on your productions :) Of course, current specification can be changed.
- compile.c: ditto.
- test/ruby/test_iseq.rb: add a test.

Revision 51659 - 08/21/2015 08:47 PM - ko1 (Koichi Sasada)

- vm_opts.h, iseq.c, iseq.h: add compile option to force frozen string literals. [Feature #11473] This addition is not specification change, but to try frozen string literal world discussed on [Feature #11473]. You can try frozen string literal world using this magical line: RubyVM::InstructionSequence.compile_option = {frozen_string_literal: true} Note that this is a global compilation option, so that you need to compile another script like that: p 'foo'.frozen? #=> false RubyVM::InstructionSequence.compile_option = {frozen_string_literal: true} p 'foo'.frozen? #=> false, because this line is already compiled. p eval("'foo'.frozen?") #=> true Details:
    - String literals are deduped by rb_fstring().
    - Dynamic string literals ("...#{xyz}...") is now only frozen, not deduped. Maybe you have other ideas. Now, please do not use this option on your productions :) Of course, current specification can be changed.
- compile.c: ditto.
- test/ruby/test_iseq.rb: add a test.

Revision 51659 - 08/21/2015 08:47 PM - ko1 (Koichi Sasada)

- vm_opts.h, iseq.c, iseq.h: add compile option to force frozen string literals. [Feature #11473] This addition is not specification change, but to try frozen string literal world discussed on [Feature #11473]. You can try frozen string literal world using this magical line: RubyVM::InstructionSequence.compile_option = {frozen_string_literal: true} Note that this is a global compilation option, so that you need to compile another script like that: p 'foo'.frozen? #=> false RubyVM::InstructionSequence.compile_option = {frozen_string_literal: true} p 'foo'.frozen? #=> false, because this line is already compiled. p eval("'foo'.frozen?") #=> true Details:
    - String literals are deduped by rb_fstring().
    - Dynamic string literals ("...#{xyz}...") is now only frozen, not deduped. Maybe you have other ideas. Now, please do not use this option on your productions :) Of course, current specification can be changed.
- compile.c: ditto.
- test/ruby/test_iseq.rb: add a test.

History

#1 - 08/21/2015 01:32 AM - ko1 (Koichi Sasada)

- Description updated

#2 - 08/21/2015 01:52 AM - ko1 (Koichi Sasada)

discussion on twitter:
https://twitter.com/yukihiro_matz/status/634386185507311616

#3 - 08/21/2015 03:16 PM - wycats (Yehuda Katz)

- Subject changed from Immutabe String literal on Ruby 3 to Immutable String literal in Ruby 3

#4 - 08/21/2015 03:28 PM - wycats (Yehuda Katz)

I would like to suggest a phased transition across several releases in Ruby 2.x (the specific version numbers and flag names are just examples):

1. In Ruby 2.3, it is possible to turn on warnings when mutating a String literal (--warn-frozen-strings).
    1. the warnings come with the place in the source where the String was originally created
2. In Ruby 2.4 (or 2.5?), the warning is on by default
    1. Once it is on by default, it probably makes sense to include the source location of the String on with a flag (--string-literal-source-information)
    2. At this point, it could be useful to have an error mode (--error-frozen-strings)

I think the magic comment solution might be helpful, but the real problems with upgrades will come from interactions between two gems. One gem might install the "frozen string literals" magic comment, and a completely other gem might try to mutate it. The second gem cannot use a magic comment to un-freeze the string, and it wouldn't be correct to submit a pull request to the first gem to remove the magic comment!

I think that a transition path across multiple releases that slowly increases the severity of the warning will help us identify how big of a problem the change will be. At the same time, it will help us transition to the new semantics a bit at a time. If the change is more of an issue than we think, it just means we need more releases of "on-by-default warning" before we can turn it into a mandatory error. Once we learn more about the kinds of problems that end up existing, there may be other things we can do to help with the transition that target those use-cases directly.

#5 - 08/21/2015 03:57 PM - enebo (Thomas Enebo)

I see no reason why 2.3 cannot also have --error-frozen-strings. As an opt-in I think it will give people more of a chance to make sure their code works.

A second possible idea would be to add a lint to possibly flag interesting idioms like String#<< where immutable strings could run into problems.

Since not all Strings will be immutable I am not sure if this would be helpful or not?

#6 - 08/21/2015 05:15 PM - mame (Yusuke Endoh)

Objection, unless a bailout is provided.

"" as a StringBuilder is acutally useful. "".dup is too tiring.

For example, how about make String#+@ an alias to String#dup?

+"" is also tiring, but better than nothing at all.

I have to insist that the bailout be one character.

--
Yusuke Endoh mame@ruby-lang.org

#7 - 08/21/2015 05:30 PM - jeremyevans0 (Jeremy Evans)

+"" is not backwards compatible, "".dup is, as is String.new. I'm definitely against +"".

One possible option would be to have "" be a mutable string, and '' be an immutable string. It doesn't make sense to have "" strings be immutable if they contain interpolation, since they can't be deduped, and '' strings can't contain interpolation.

#8 - 08/21/2015 05:45 PM - mame (Yusuke Endoh)

Jeremy Evans wrote:

> +"" is not backwards compatible, "".dup is, as is String.new. I'm definitely against +"".

It will be backwards compatible, if we introduce it to Ruby 2.3 or 2.4. I don't think that Ruby 3.0 will be released before Ruby 2.2 EOL.

> One possible option would be to have "" be a mutable string, and '' be an immutable string.

I really like your idea.

#9 - 08/21/2015 07:29 PM - spatulasnout (B Kelly)

Yusuke Endoh wrote:

> Objection, unless a bailout is provided.
>
> "" as a StringBuilder is acutally useful. "".dup is too tiring.

Wow. Thanks for mentioning string building. It occurs to me I build
strings like this somewhat regularly:

```
sql = %{SELECT #{sec_id}, pt.path, st.doc_count }
sql << %{FROM #{stats_tablename} AS st }
sql << %{JOIN #{path_tablename} AS pt ON (st.path_id = pt.id) }
```

So this will break in 3.0 ?

#10 - 08/21/2015 08:48 PM - ko1 (Koichi Sasada)

- Status changed from Open to Closed

Applied in changeset r51659.

- vm_opts.h, iseq.c, iseq.h: add compile option to force frozen string literals. [Feature #11473] This addition is not specification change, but to try frozen string literal world discussed on [Feature #11473]. You can try frozen string literal world using this magical line: RubyVM::InstructionSequence.compile_option = {frozen_string_literal: true} Note that this is a global compilation option, so that you need to compile another script like that: p 'foo'.frozen? #=> false RubyVM::InstructionSequence.compile_option = {frozen_string_literal: true} p 'foo'.frozen? #=> false, because this line is already compiled. p eval("'foo'.frozen?") #=> true Details:
  - String literals are deduped by rb_fstring().
  - Dynamic string literals ("...#{xyz}...") is now only frozen, not deduped. Maybe you have other ideas. Now, please do not use this option on your productions :) Of course, current specification can be changed.
- compile.c: ditto.
- test/ruby/test_iseq.rb: add a test.

#11 - 08/21/2015 08:50 PM - ko1 (Koichi Sasada)

- Status changed from Closed to Assigned

r51659 introduced a way to try this feature.
Have fun.

#12 - 08/21/2015 10:30 PM - akr (Akira Tanaka)

Jeremy Evans wrote:

> One possible option would be to have "" be a mutable string, and '' be an immutable string. It doesn't make sense to have "" strings be immutable if they contain interpolation, since they can't be deduped, and '' strings can't contain interpolation.

It makes that frozen string literal can not have escape sequences such as "\n".

I feel the magic comment name and command line option name,
frozen-string-literal, denotes that all string literals frozen.

#13 - 08/22/2015 05:26 PM - mame (Yusuke Endoh)

Why do matz and akr want to make String literals immutable, at first? Please explain use case.

For performance? Is it more important than usability, in the recent Ruby design?

--
Yusuke Endoh mame@ruby-lang.org

#14 - 08/22/2015 09:55 PM - wanabe (_ wanabe)

- Related to Feature #8976: file-scope freeze_string directive added

#15 - 08/23/2015 02:13 AM - akr (Akira Tanaka)

Yusuke Endoh wrote:

> Why do matz and akr want to make String literals immutable, at first? Please explain use case.

My first intent is explained at:
https://bugs.ruby-lang.org/issues/8976
This is refered from this issue as a related ticket.

At the meeting (DevelopersMeeting20150820Japan), matz decided by Matsuda-san's argument.
Matsuda-san explains his motivation in
https://bugs.ruby-lang.org/issues/8976#note-30

In short, current situation frustrate developers, especially library developers.
The problem is caused by an ambivalent between clean code and fast code.

> For performance? Is it more important than usability, in the recent Ruby design?

This issue tries to match performance (fast code) and usability (clean code).

I agree more code (dup, String.new or something new) is required to distinguish
string modifiable from string never modified.
It decline usability.
However it prevents unintentional string modification.
This can improve usability.
So, total usability changes are not trivial.

However if code bloat is small enough, latter (usability improvement) should win.

In my experiment, modification is about one per 700 lines.
I feel the usability decline is acceptable.

My experiment is here:
https://github.com/akr/ruby/tree/frozen-string

```
% git diff -U0 trunk lib/**/*.rb|grep '^+'|grep -v '^+++'|wc
    256    1273   10887
% wc lib/**/*.rb|tail -1
 175569  537606 4639935 total
```

So, the library modification rate is one per 175569/256=685.8 line.

#16 - 08/23/2015 05:25 PM - mame (Yusuke Endoh)

Thank you for the explanation and references.

I'm neutral to the magic comment. It would be better if it could specify finer-grained range than a whole file, but I have no strong objection. But "immutable by default" is completely another topic.

> However it prevents unintentional string modification.

I thought of that. In fact, I can somewhat understand this motivation. However:

- Is this type of bug so frequent? I cannot remember the last time when I experienced such a bug.
- To prevent this type of bug, why is it sufficient to freeze only literals? String#+ also must return a frozen String, I think.
- I believe that the same logic applies to Array, ultimately. Will all Array literals be also frozen?

I don't think that "immutable by default" is a proper way.

> In my experiment, modification is about one per 700 lines.

Thank you for the quantitative information. How long does it take to fix the 256 places?

IMO, the fact that you need so many changes that can be measured quantitatively, shows that this approach is too radical to accept ;-) What do any others think?

--
Yusuke Endoh mame@ruby-lang.org

#17 - 08/24/2015 12:33 AM - akr (Akira Tanaka)

Yusuke Endoh wrote:

> I'm neutral to the magic comment. It would be better if it could specify finer-grained range than a whole file, but I have no strong objection. But "immutable by default" is completely another topic.

We have the most fine-grain notation already: "foo".freeze.
Unfortunately, this notation causes developer's ambivalent: fast code or clean code.

matz don't like the magic comment.
So the issue is not accepted long time and
finally accepted as a migration path to "immutable by default".

"immutable by default" is a way to solve the ambivalent with matz's preference.

> > However it prevents unintentional string modification.
>
> I thought of that. In fact, I can somewhat understand this motivation. However:

I think the major factor why matz accept "immutable by default" is not this effect.
This effect is not discussed at the meeting.

Even if this effect is small, it doesn't change the conclusion.

> Thank you for the quantitative information. How long does it take to fix the 256 places?

Few days, as far as I remember.

#18 - 08/24/2015 08:18 AM - sawa (Tsuyoshi Sawada)

As for a "finer-grained range than a whole file" directive, it reminds me of the using method. Their scope look similar to me. Can we have built-in modules named as FrozenString and UnfrozenString, which give special interpretations when used with using?

```
using FrozenString

  # string literals here are interpreted as frozen

using UnfrozenString
```

```
# string literals here are not interpreted as frozen
```

And when a whole file needs to be directed in one or the other way, we can just put one of these lines at the beginning of the file.

To me, it also looks less ugly than a magic comment.

#19 - 08/24/2015 11:56 AM - mame (Yusuke Endoh)

> "immutable by default" is a way to solve the ambivalent with matz's preference.

So, you mean:

- because the magic comment is not accepted as-is,
- we suppose "immutable by default" as a future goal, and
- the magic comment can be now accepted as a migration path.

The logic seems weird to me. What is the true goal? If "immutable by default" is the true goal, we should discuss the advantage/disadvantage of itself. The magic comment is irrelevant.

> I think the major factor why matz accept "immutable by default" is not this effect.

The advantages are:

- improves the performance
- prevents users from shooting themselves in the foot (this effect is limited)

The disadvantages are:

- breaks the compatibility significantly
- damages the usability

IMO, the long-term design preference of Ruby should be: usability > compatibility > performance.

--
Yusuke Endoh mame@ruby-lang.org

#20 - 08/24/2015 12:08 PM - akr (Akira Tanaka)

Yusuke Endoh wrote:

> > "immutable by default" is a way to solve the ambivalent with matz's preference.
>
> So, you mean:
>
> - because the magic comment is not accepted as-is,
> - we suppose "immutable by default" as a future goal, and
> - the magic comment can be now accepted as a migration path.
>
> The logic seems weird to me. What is the true goal? If "immutable by default" is the true goal, we should discuss the advantage/disadvantage of itself. The magic comment is irrelevant.

I think it is not a problem.
"immutable by default" solves the ambivalent.
It makes developers happy.
This can be considered better usability.

#21 - 08/24/2015 01:38 PM - mame (Yusuke Endoh)

Akira Tanaka wrote:

> This can be considered better usability.

I agree with a useful feature for easily and elegantly improving performance, as long as it does not hurts any other usability and compatibility. Even if it hurts to some extent, it is worth considering if it is really useful for performance improvement.

In this case, it significantly damages usability and compatibility. Also, I'm afraid if it is not so useful in terms of performance improvement.

It is said that 90% of the execution time of a computer program is spent executing 10% of the code. Making 100% of the code immutable resembles

"premature optimization is the root of all evil".

--
Yusuke Endoh mame@ruby-lang.org

#22 - 08/24/2015 03:16 PM - akr (Akira Tanaka)

Yusuke Endoh wrote:

> In this case, it significantly damages usability and compatibility. Also, I'm afraid if it is not so useful in terms of performance improvement.

This issue tries to solve more than performance.
As Matsuda-san explained, it solves a social problem.

I disagree the usability problem.
I think it is acceptable.

I agree the compatibility problem.
So we need a migration path: magic comment.

#23 - 08/24/2015 03:42 PM - mame (Yusuke Endoh)

Akira Tanaka wrote:

> This issue tries to solve more than performance.
> As Matsuda-san explained, it solves a social problem.

Adding .freeze blindly is stupid. It should be added only when it is really needed, i.e., only when a string literal in a core loop was proved to be the bottleneck. The core loop is <10% of the code. Do not bring the burden to the innocent 90% of the code.

--
Yusuke Endoh mame@ruby-lang.org

#24 - 08/24/2015 03:57 PM - mame (Yusuke Endoh)

I often write [a, b].max. This idiom is easy to read and write. But it is slow, especially in the core loop, because it creates an Array object every times.

For the reason, do you replace all [a, b].max with a < b ? b : a, not only in the core loop but also in an initialization code that is executed once? I believe that is stupid.

--
Yusuke Endoh mame@ruby-lang.org

#25 - 08/24/2015 04:22 PM - akr (Akira Tanaka)

Yusuke Endoh wrote:

> Akira Tanaka wrote:
>
> > This issue tries to solve more than performance.
> > As Matsuda-san explained, it solves a social problem.
>
> Adding .freeze blindly is stupid. It should be added only when it is really needed, i.e., only when a string literal in a core loop was proved to be the bottleneck. The core loop is <10% of the code. Do not bring the burden to the innocent 90% of the code.

The social problem currently happen seems that not everyone thinks as you.
Your cleverness doesn't solve the social problem, unfortunately.

I think changing Ruby is easier than educating people.

#26 - 08/24/2015 05:13 PM - mame (Yusuke Endoh)

- Status changed from Assigned to Open

Akira Tanaka wrote:

> I think changing Ruby is easier than educating people.

Oh, PHP-like attitude. I've always trusted Ruby so far. Is Ruby dying?

--
Yusuke Endoh mame@ruby-lang.org

#27 - 08/24/2015 06:09 PM - akr (Akira Tanaka)

Yusuke Endoh wrote:

> Oh, PHP-like attitude. I've always trusted Ruby so far. Is Ruby dying?

I'm not sure you mean.

#28 - 08/24/2015 06:30 PM - shreeve (Steve Shreeve)

As a ruby user since 2001, I'd agree with Yusuke Endoh, when he suggests the long-term design preference of Ruby should be:

usability > compatibility > performance

There are plenty of languages to select if one is truly focused on performance, static typing, different syntax, etc.

Seems like there should be a quick way to flag strings as immutable (perhaps just prefixing with a sigil such as '@'?) and that this approach would be used in libraries and other performance centric code. But, if this change means that the following won't work (not my code, just copied from above):

```
sql = %{SELECT #{sec_id}, pt.path, st.doc_count }
sql << %{FROM #{stats_tablename} AS st }
sql << %{JOIN #{path_tablename} AS pt ON (st.path_id = pt.id) }
```

then, it seems like we're losing some of what makes ruby so fun and easy to use.

#29 - 08/25/2015 12:23 AM - akr (Akira Tanaka)

Steve Shreeve wrote:

> As a ruby user since 2001, I'd agree with Yusuke Endoh, when he suggests the long-term design preference of Ruby should be:
>
> usability > compatibility > performance

I think the usability can be improved.
We don't need to choose "foo" and "foo".freeze.
We will need to choose "foo" and "foo".dup instead.
The latter decision is easier than the former because the former needs benchmark and what is core loop.
I think the social problem is caused by the former decision is difficult, or different between people.
After this issue, developers can write clean and fast code with less brain power and the social problem is solved.
This effect can win the small code bloat with dup (or String.new or something new).

I agree the compatibility problem.
I hope the migration path, the magic comment and command line option,
mitigate the problem enough.
Current code can work for Ruby 3.0 with trivial one line addition:
"# frozen-string-literal: false".
If the performance improvement can be discarded, even the addition is unnecessary.
The only necessity is the command line option, --disable-frozen-string-literal.

The performance will be better.

So, this issue doesn't violate Endoh-san and your order.

> Seems like there should be a quick way to flag strings as immutable (perhaps just prefixing with a sigil such as '@'?) and that this approach would be used in libraries and other performance centric code. But, if this change means that the following won't work (not my code, just copied from above):

It can, if we find a good one and succeed to persuade matz.
For example, Endoh-san have an idea: +"foo". (See comment #6)
Maybe, !"foo" is another option.

I feel "foo".dup is good enough, though.

#30 - 08/25/2015 03:35 AM - mame (Yusuke Endoh)

Please consider the difference of the development style: hobby development and enterprise development.

I'm on hobby development. I mainly use Ruby to write a short program, including one-liner. I don't use a framework like Rails.
In this style, .dup, command-line argument (including setting RUBYOPT), and magic comment are all big pain.

On the other hand, in general the setting is not a pain for the enterprise development like Rails. Each user needs to think nothing if the framework does the setting instead of users. Considering this use case, magic comment and command-line argument may not a reasonable mean even for enterprise style. I guess it would be good if the "immutable by default" setting should be done via method call (e.g., require "app.rb", string_immutable: true) or global setting (e.g., RubyVM::StringImmutable = true), and so on.

So, please keep "mutable by default", and reconsider if the setting means (magic comment and command-line argument) are really suitable for the enterprise development style.

--
Yusuke Endoh mame@ruby-lang.org

#31 - 08/25/2015 03:58 AM - shevegen (Robert A. Heiler)

The magic comment is no problem, at least not to me, if I understood it correctly.

So the old behaviour can be retained at least as long as the magic comment will
remain, one just has to set the # frozen-string-literal: false" part, probably
on the next line after the # Encoding line.

So, I could remain lazy too and retain the old string behaviour of ruby, which would
actually be easier for me to do than having to manually find all occurances where
I would require to have a dynamic string in any of my several thousand .rb files,
and manually have to apply .dup there, which is quite a lot of work. (I never
thought about the problem between mutable or immutable string before because I
never had to think about it; speed never was a problem for me either. This change
would force me to have to review all my code and think about whether I require
a mutable string or whether I don't.)

But from a syntactic point of view the two variants are not the same:

```
_ = ''
_ << 'Hello '
_ << 'World!'
```

feels cleaner to me than this variant:

```
_ = ''.dup
_ << 'Hello '
_ << 'World!'
```

The feeling of using ruby there would be different between 3.x on the one
hand, and 2.x and 1.x on the other hand if one would have to change all
'' invocations to ''.dup to retain a mutable string.

While I assume that matz designed ruby to incorporate and extend from
many ideas among many other programming languages too, and beauty would
not necessarily be a primary design goal of ruby - I still think that
ruby is the most beautiful among the programming languages out there.

I love that the code that I can write in it is beautiful as well, at least
to me.

When I compare it to my old PHP code, ruby is much prettier - the prettiest
PHP code I could write would never be as close to the prettiest ruby code I
could write. Ruby can be terse and pretty at the same time; the above
change would seem to require slightly more verbosity than before.

What if rather than adding an extra sigil or requiring of ruby hackers
to manually add a method call if they want to retain the old string
object behaviour, we could use "" and '' to differ instead?

"" is used for interpolation already, which is sort of dynamic, so
"" could default to a dynamic string, whereas '' could be used to
denote an immutable string. This of course may still require changes
in existing ruby code, but people would not have to manually apply
a method call on string objects in all their ruby code, and the
strings would be either mutable or immutable.

I understand that this is not exactly referring to the social problem
that was described above, and I also understand the other arguments
such as the speed argument or thread safety perhaps, but that social
problem at the same time also means that the old behaviour would have
to be changed in the transition phase. I am not entirely sure I have
understood why it seems necessary even though I am aware that other
languages such as python and crystal also have immutable strings - I

also understand that this is perfectly fine for any language designer
to make as a decision, that is also perfectly ok.

To me so far, if I understood it correctly, the main gain seems to be
a speed improvement. Well, speed improvements are always nice to have,
but the above one also will come as a permanent trade-off in syntax.

It would be great if we could rather have two variants of ruby - one
that is uglier, but lightning fast, and one that is beautiful poetry
but at the cost and price of slower execution. I understand that this
is not easily possible, but I myself never picked ruby over the other
languages due to any problems of speed, mind you; what convinced me
was the old interview about the philosophy of ruby and happiness
while programming:

http://www.artima.com/intv/ruby.html

From a syntactic point of view, I personally would favour the old way
without having to manually do an extra .dup to make a string mutable.

It probably also won't affect me for a very long time since I can retain
the magic comment anyway, but I worry like ~5 years in the future if
I will have to do the switch.

Thank you.

#32 - 08/25/2015 12:38 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

I'm a supporter of the idea of making strings frozen by default. Maybe some syntax for unfrozen strings like 'unfrozen'u might be a good idea rather than using dup, but I'm not really worried about this.

Performance should indeed be a concern in my opinion. Just because there are other faster languages it doesn't mean performance should not be a priority for Ruby. Ruby has many strong features but that doesn't mean we shouldn't try our best to make it the faster we can. Of course we have to evaluate the trade-offs since usability is also one of the main strengths in Ruby. But in this case, I don't think compatibility should be the main concern for Ruby 3 when this change would clearly improve the overall speed.

For the builder pattern discussed in this thread, it would be a good chance to change it to use an array and a join: builder = []; builder << 'string'...; builder.join.

I prefer it being immutable by default as it even makes it easier for someone reading a code to understand whether the string is meant to be changed or not... In other words, I also think it improves code reading, so usability, besides the performance improvement. I'd love to see immutable string literals by default.

Also, to be honest, I don't really like the idea of magic comments. Maybe if it would be a temporary hack until Ruby 4, but I'd rather prefer the old gem codes to be properly fixed instead.

Here's another idea with compatibility in mind. MRI could record whether a frozen string was explicitly frozen by the code. When it wasn't, in case someone tries to perform a mutable operation on it, if the interpreter was run with the option '--allow-strings-to-mutate' it could then detect the case and unfreeze the string, rather than raising, and emit a warning and let the program continue as expected. This would give some warnings until all cases are fixed and the switch could be then turned off for applications which fixed all cases...

#33 - 08/26/2015 09:29 AM - duerst (Martin Dürst)

Akira Tanaka wrote:

> This issue tries to solve more than performance.
> As Matsuda-san explained, it solves a social problem.

Can you please provide a pointer to the "social problem" explained by Akira Matsuda?

The only reference I found in this issuue is to https://bugs.ruby-lang.org/issues/8976#note-30. I don't see any social problems mentioned in that comment. It mentions mainly performance, and code ugliness.

For me, a "social problem" would be e.g. avoiding useless discussions and fights between programmers.

#34 - 09/15/2015 03:07 AM - duerst (Martin Dürst)

Robert A. Heiler wrote:

```
_ = ''
_ << 'Hello '
_ << 'World!'
```

> feels cleaner to me than this variant:

```
_ = ''.dup
_ << 'Hello '
_ << 'World!'
```

One way to write this would be

```
_ = ''
_ += 'Hello '
_ << 'World!'
```

However, in a more complicated context, it may not be easy to know when the first change to the empty string occurs (assuming that using += in all cases is less efficient).

#35 - 09/18/2015 05:38 AM - avit (Andrew Vit)

Rodrigo Rosenfeld Rosas wrote:

> if the interpreter was run with the option '--allow-strings-to-mutate' it could then detect the case and unfreeze the string, rather than raising, and emit a warning and let the program continue as expected.

Unfreezing & mutating would change every other place that string was referenced. This should deserve more than a warning, it would be an error for sure!

Automatically duping it could be just as bad. It's impossible to know when other references to that string are expecting it to be mutated, or if a new copy is fine.

#36 - 09/18/2015 02:40 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Hi Andrew, why mutating a string would change every other place the string is referenced? Wouldn't the string reference remain the same?

#37 - 09/19/2015 06:40 AM - avit (Andrew Vit)

Rodrigo Rosenfeld Rosas wrote:

> Hi Andrew, why mutating a string would change every other place the string is referenced? Wouldn't the string reference remain the same?

```
a = "asdf".freeze
b = "asdf".freeze
a.object_id == b.object_id
```

Frozen strings for the same value are the same object. If you "unfreeze" one, it would change every other one. Two copies of a string can have the same value for different reasons; it would be very wrong if both changed. On the other hand, if we automatically dup it then it's just like doing this, which won't do what you expect:

```
builder = "".freeze
(builder.dup) << "asdf"
builder == "asdf" # false
```

#38 - 09/23/2015 05:01 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

I never suggested to dup it, but to actually "unfreeze" it:

```
a = b = 'asdf'.freeze

a.unfreeze # this is currently not possible, but thould be performed automatically
           # internally by my proposal without changing the object_id

a.object_id == b.object_id # this should be true for what I proposed.
```

#39 - 09/23/2015 06:53 PM - avit (Andrew Vit)

Hi Rodrigo, I think you need to look into the implications of what you are proposing here.

For example: In one place you use the string literal "name" e.g. @db_columns = ["name"]. In a different context you perform "name" << ",email" for a different purpose. If these become unfrozen instances of the same object, then both would be mutated unintentionally. This is why "unfreeze" does not exist.

#40 - 09/24/2015 08:28 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

I understand that but unfreeze would only be called for strings which haven't been explicitly frozen by the users by calling "name".freeze. For those cases there won't be any unfreezing. Actually "unfreeze" won't be exposed publicly, but would only be used internally by Ruby when it detects the String was frozen by default and there's an attempt to change it, just to keep backwards compatibility transparently. After all, this is how it currently

works if you try to do ["name"].first << ',email'.

Implementing what I suggested won't introduce any subtle bugs, it will only warrant backward compatibility while allowing strings to be frozen by default until the first time there's an attempt to mutate them.

#41 - 09/24/2015 08:46 PM - jeremyevans0 (Jeremy Evans)

Rodrigo Rosenfeld Rosas wrote:

> Implementing what I suggested won't introduce any subtle bugs, it will only warrant backward compatibility while allowing strings to be frozen by default until the first time there's an attempt to mutate them.

Rodrigo, how do you propose ruby handle the following case:

```
10.times do |i|
  a = b = ''
  p [i, a.frozen?, b.frozen?, a.object_id, b.object_id]
  a << '1'
  p [i, a.frozen?, b.frozen?, a.object_id, b.object_id]
end
```

If you mutate a, what happens to b, and would their object_ids change? What would the value of a and b be in the nth iteration?

#42 - 09/28/2015 12:43 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

a.frozen? Should always return false unless explicitly frozen. That's the only way to keep backwards compatibility, right? Ruby might introduce some other method to get the information whether the string is still in the default initial frozen state though... Maybe something like String#changed?...

a.object_id and b.object_id should always remain the same, after all they point to the same object. If you change a you're automatically changing b as well. For your specific example, it should print the following all times, assuming 123 would be their object_id (actually obviously the first element will vary from 0 to 9):

[0, false, false, 123, 123]

#43 - 09/28/2015 02:47 PM - jeremyevans0 (Jeremy Evans)

Rodrigo Rosenfeld Rosas wrote:

> a.frozen? Should always return false unless explicitly frozen. That's the only way to keep backwards compatibility, right? Ruby might introduce some other method to get the information whether the string is still in the default initial frozen state though... Maybe something like String#changed?...
>
> a.object_id and b.object_id should always remain the same, after all they point to the same object. If you change a you're automatically changing b as well. For your specific example, it should print the following all times, assuming 123 would be their object_id (actually obviously the first element will vary from 0 to 9):
>
> [0, false, false, 123, 123]

If the object ids remain the same across iterations, then all of the strings must share the same buffer. Since you are appending to the buffer in each iteration, that means that a and b are no longer initialized to '' in subsequent loop iterations. Basically, you can't reuse the same string object for literal strings unless it remains frozen, otherwise the literal string value can change and if changed would not reflect the ruby code you've written.

#44 - 09/28/2015 07:20 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

I don't follow your concerns. I've suggested something in an attempt to get the performance improvements when possible while keeping backwards compatibility. That means it should work like it currently does:

```
a = b = ''
a.object_id == b.object_id # true
a << '1'
a.object_id == b.object_id # true
a == '1' && b == '1' # true
```

No surprises at all. It just means that a and b won't get any performance improvements in this case because they will be unfrozen after (a << '1'). But it won't cause any backwards incompatibilities either. The automatic unfreezing that I suggested will indeed unfreeze the string, which means the shared buffer will change, as expected. Why is this a problem if this is indeed the expected behavior to keep backwards compatibility with existing Ruby applications?

#45 - 09/28/2015 07:59 PM - jeremyevans0 (Jeremy Evans)

Rodrigo Rosenfeld Rosas wrote:

> No surprises at all. It just means that a and b won't get any performance improvements in this case because they will be unfrozen after (a << '1').

> But it won't cause any backwards incompatibilities either. The automatic unfreezing that I suggested will indeed unfreeze the string, which means the shared buffer will change, as expected. Why is this a problem if this is indeed the expected behavior to keep backwards compatibility with existing Ruby applications?

Because with existing ruby applications, each literal string created has a different object_id. With immutable strings, all literal strings with the same value have the same object_id, because they are the same object.

```
(0...10).map{''.object_id}.uniq.length
# 1 if literal strings are frozen
# 10 if not (unless GC during map)
```

The whole point of freezing literal strings is so you can save allocations. If literal strings don't give you the same object, you could unfreeze/modify, but then you aren't saving allocations. If literal strings give you the same object, allowing you to save allocations, you can't unfreeze/modify them safely. If we allowed unfreeze/modify on frozen strings:

```
b = proc{''}
b.call #=> ''
a = (0...10).map(&b)
a #=> ['', '', '' ...]
a[0] << 'b'
a #=> ['b', 'b', 'b' ...]
b.call #=> 'b'
```

Is that the behavior you want? If not, what is your proposal for making unfreeze/modify work for literal strings while at the same time saving allocations?

#46 - 09/28/2015 08:47 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Oh, now I got it, sorry for the confusion and thanks for explaining. I didn't pay attention that the performance gains were due to saving allocations by sharing the same buffer everytime the same string is built. In that case, sorry, but I have no idea on how to keep backwards compatibility while still taking advantage of the performance benefits.

On the other hand, the technique I described could be used in a working application with some callbacks whenever the string is automatically unfrozen so that someone could keep it running in production and record a log of all places that would generate an exception once the automatic frozen strings are in effect. This way we could have an idea beforehand on what we should change in our application before we enable the automatic frozen feature in it.

#47 - 10/04/2015 05:45 PM - araipiyo (Shunichi Arai)

Akira Tanaka wrote:

> I think the usability can be improved.
> We don't need to choose "foo" and "foo".freeze.
> We will need to choose "foo" and "foo".dup instead.
> The latter decision is easier than the former because the former needs benchmark and what is core loop.

That argument doesn't make sense for me. The argument only explains "usability for performance", so it's not usability per se, it's rather about performance.

> I agree the compatibility problem.
> I hope the migration path, the magic comment and command line option,
> mitigate the problem enough.
> Current code can work for Ruby 3.0 with trivial one line addition:
> "# frozen-string-literal: false".
> If the performance improvement can be discarded, even the addition is unnecessary.
> The only necessity is the command line option, --disable-frozen-string-literal.

I worry those options will be disappeared in future versions. I need to make sure that the option will be guaranteed to remain exist in future versions.

As a person who is responsible managing for many Ruby-based enterprise products, I really care about the compatibility. It was a nightmare to migrate our 1.8.7 codes to 1.9, because of many small incompatiblity in the release. Even small incompatibilities, it's a real burden for users if there are many...

#48 - 11/13/2015 02:26 AM - mwpastore (Mike Pastore)

Has anyone suggested introducing new quote-like operators to define mutable strings in Ruby 2.3 (with immutable string pragma) and 3.0? I was thinking something like:

```
# immutable without interpolation
'foo'
%q{foo}
```

```
# immutable with interpolation
"bar"
%Q{bar}

# mutable without interpolation
String.new('foo')
%y{foo} # proposed

# mutable with interpolation
String.new("bar")
%Y{bar} # proposed
```

Also, what about mutable vs. immutable heredoc?

#49 - 11/13/2015 07:12 AM - duerst (Martin Dürst)

Mike Pastore wrote:

> Has anyone suggested introducing new quote-like operators to define mutable strings in Ruby 2.3 (with immutable string pragma) and 3.0? I was thinking something like:
>
> # mutable without interpolation
>
> String.new('foo')
> %y{foo} # proposed
>
> # mutable with interpolation
>
> String.new("bar")
> %Y{bar} # proposed

This was mentioned as part of the general discussion on making string literals immutable, but not in much detail. I suggest that you propose this as a new feature, with some good use cases, if you really are interested.

#50 - 01/02/2018 10:59 AM - perlun (Per Lundberg)

FWIW, Rubocop suggests now in recent versions that you use the "unary plus" as the "one-character" approach to thaw a frozen string: http://www.rubydoc.info/gems/rubocop/RuboCop/Cop/Performance/UnfreezeString

More details in the PR that introduced this cop: https://github.com/bbatsov/rubocop/pull/4586

#51 - 03/13/2018 02:33 AM - hsbt (Hiroshi SHIBATA)

- Status changed from Open to Assigned

#52 - 03/13/2018 05:05 PM - shevegen (Robert A. Heiler)

Since this will soon be discussed in the upcoming ruby developer meeting, I only wanted to add two small things:

First, Yusuke Endoh wrote back then:

> "" as a StringBuilder is acutally useful. "".dup is too tiring.

I myself use frozen Strings a lot these days. I am fine with frozen Strings in Ruby.

But I also agree that "" is prettier, IMO, than is "".dup.

The **second thing**, I can understand that stdlib/core of ruby wants to use frozen Strings when possible due to various reasons, in particular (assumingly) speed or perhaps "economics of usage" (perhaps may use slightly less memory on the average computer as well in typical ruby programs?).

I have no problem with that at all. I myself, though, would like to **retain the possibility** to **not** use frozen Strings if possible, **WHEN** I would prefer not to. The **current way** via a frozen-string comment at the ~top of a .rb file **works well for me** - I would like to see this way retained. Right now I just toggle "true" or "false" there and can have the desired behaviour.

That way, ruby can change to default to frozen strings by default,
all of the time, but ruby hackers can also continue to specify
that they may want to not use frozen strings. (I use frozen Strings
almost everywhere nowadays but I still want to retain the option
to not have to use it; I use .dup explicitely though since I
do not like the unary - operator/thingy even though it is
much shorter).

#53 - 01/10/2019 05:49 AM - matz (Yukihiro Matsumoto)

- Status changed from Assigned to Closed

I consider this for years. I REALLY like the idea but I am sure introducing this could cause HUGE compatibility issue, even bigger than Ruby 1.9. So I officially abandon making frozen-string-literals default (for Ruby3).

This does not mean we are going to remove the frozen-string-literal feature that can be specified by magic comments.

Matz.

#54 - 09/17/2019 04:28 PM - headius (Charles Nutter)

I regret missing this "final" decision, but I believe we should reconsider.

> I am sure introducing this could cause HUGE compatibility issue

Truly? We have had the syntactic hacks (.freeze etc), frozen-string-literal pragma, and the command-line global flag for many years now. A very large corpus of existing code already opts into frozen string literals today, including most of Rails and its dependencies. RuboCop (the most popular linter by far) and other tools have been recommending users include the frozen-string-literal pragma as well.

And I have a path forward that would easily address any breakage: we add a mutable-string-literal pragma.

1. Ruby 2.7 should include a mutable-string-literal pragma that allows users to opt into mutable strings on a file-by-file basis. It would be ignored when the frozen-string-literal feature is disabled.
2. We all start testing our code on Ruby 2.7 with --enable:frozen-string-literal and fix code to opt into mutable strings using either "".dup or the file-wide mutable-string-literal pragma.
3. By the time Ruby 3.0 is released in late 2020, all we should need to do is enable frozen-string-literal by default.

I really don't think this migration path would be difficult, and I suspect that the vast majority of Ruby code out there would work out of the box or require only minor changes to be compatible.

#55 - 09/17/2019 09:50 PM - shevegen (Robert A. Heiler)

> I suspect that the vast majority of Ruby code out there would work out of the box or
> require only minor changes to be compatible.

This depends on the ruby code. Some projects will be semi-dormant due to various
reasons.

Semi-old gems for example.

If matz wants people to upgrade to ruby 3.0, then it actually makes a lot of sense
to want to make the transition period simple and painless.

What I also don't fully understand is how people want to pinpoint this to mean
that ruby will never default to frozen strings. 2020 is not the end of the world
right? So if you were to have frozen strings in 2021, that would be about 2
years from now on. That's not a long time.

On the other hand, if you want to add incompatibilities then transitioning into
ruby will become harder, too. There are always trade-offs.

In my opinion, if matz's objective is to make the transition to ruby 3.0 simple,
then it actually makes a lot of sense to postpone frozen strings by default.

(I should also add that I do not really have a strong opinion either way. I
use frozen strings in most of my ruby projects, most of them set to true via
the toplevel comment, so either way, it would not affect me. But you have to
look at projects that are only semi-active. Although in theory perhaps for
gem-based projects this could be solved, e. g. older projects could automatically
default to non-frozen strings, whereas newer projects might be assumed to
want to use frozen strings; with the toplevel comment overriding this anyway.

That way old, semi-maintained gems could be kept at non-frozen strings, as
it used to be, so they would not break due to that.)

#56 - 09/18/2019 01:00 AM - headius (Charles Nutter)

> This depends on the ruby code. Some projects will be semi-dormant due to various reasons.

That's for us to address as a community. Are we going to let a single decade-old gem prevent us from moving Ruby forward? What's the threshold? There's libraries out there that don't work on Ruby 1.9. We left them behind or replaced them. And are people depending on a gem that's unmaintained *really* going to be the ones to jump on Ruby 3.0 the day after Christmas 2020?

This is also still supposition. Name some gems that are unmaintained and in wide use. We can fix them! We have the technology!

> In my opinion, if matz's objective is to make the transition to ruby 3.0 simple, then it actually makes a lot of sense to postpone frozen strings by default.

Postpone until when? 3.1? So then 3.1 will be the hard break?

They've been discussed for what, ten years now? How long is long enough?

We've added many ways for people to start transitioning to immutable literal strings, and people are using those mechanisms widely. We've pushed this transition a long time, and we still have another year until 3.0 is out and longer than that until people will *need* to make a move. What is the threshold for being "ready" to make this change?

Unless we're planning to wait until Ruby 4.0 in 2030 to do this, I think we should do it now.

> I use frozen strings in most of my ruby projects, most of them set to true via the toplevel comment, so either way, it would not affect me.

Exactly. Most people already do use frozen string literals. And adding a pragma means we can transition troublesome code to the new way with a single line per affected file. Heck, we can even add --enable:mutable-literal-string for people that are stuck with some of that old unmaintained code, allowing them to have a soft landing.

#57 - 09/18/2019 12:47 PM - Dan0042 (Daniel DeLorme)

> And I have a path forward that would easily address any breakage: we add a mutable-string-literal pragma.

I don't understand, why a new pragma? Is there a reason why the existing frozen-string-literal: false is not good enough?

Side note to this discussion: I've made proposal #16153 for a way to gradually phase-in frozen strings. IMHO that would really help with a transition to frozen-string-literal: true by default. But then again Katz made roughly the same proposal 4 years ago and it was ignored?

#58 - 09/19/2019 03:52 AM - headius (Charles Nutter)

> Is there a reason why the existing frozen-string-literal: false is not good enough?

My understanding of frozen-string-literal is that if true it makes string literals in the same file mutable. If false, it does not do that.

If we make the global default to freeze string literals, would frozen-string-literal: false for a given file make them mutable?

I guess the interaction between the "false" state and the current runtime default is what has me confused. I see "true" and "false" here more like "on" and "off", and if frozen-string-literal is off, to me that means it does nothing at all and whatever defaults are in place take effect.

I could be convinced otherwise. I'd be very surprised if we find any corpus of code setting frozen-string-literal: false, so maybe we are free to define what it means if the default becomes to freeze string literals. All I want is a pragma that guarantees a file will have pre-3.0 frozen string literal behavior, since that will be a one-line escape valve until the code can be adapted.

> I've made proposal #16153 for a way to gradually phase-in frozen strings.

I agree! I'd love to see some sort of warning happen in 2.7...perhaps a flag you can pass or just a new warning that recommends alternative code or the appropriate pragma.

I believe it will be a missed opportunity if we don't make the switch in 3.0.

#59 - 09/21/2019 05:13 PM - mame (Yusuke Endoh)

I'm still against frozen-string-literal by default. It is arguable if the string creation limits performance so much in real-world programs. We need to first measure how much Ruby can be faster by frozen-string-literal. If it is not significant, Ruby should prefer dynamics and flexibility.

I'd like to share one interesting example. RDoc has used frozen-string-literal: true since Ruby 2.5.0. At the release, NEWS said "This (frozen-string-literal) reduces document generation time by 5%." However, it turned out to be wrong.
There were originally some code fragments in RDoc assuming that a string is mutable. To enable frozen-string-literal, they were changed, which brought performance degradation. After that, enabling frozen-string-literal made RDoc a bit faster, but the net result was worse than the original.

- original rdoc: 79.5 sec
- code-changed rdoc: 81.3 sec
- frozen-string-literal rdoc: 80.0 sec

aycabta (aycabta .) performed a great postmortem examination. See https://gist.github.com/aycabta/abdfaa75ea8a6877eeb734e942e73800 in detail.

By the way, RDoc's generation time is now about half by some algorithmic improvements. In RDoc case, it is not significant whether string literals are frozen or not.

#60 - 09/26/2019 07:49 PM - Eregon (Benoit Daloze)

headius (Charles Nutter) wrote:

> If we make the global default to freeze string literals, would frozen-string-literal: false for a given file make them mutable?

Yes, in fact shortly after --enable-frozen-string-literal was introduced, most stdlib files got a frozen-string-literal: false so they would still work with that flag enabled:
https://github.com/ruby/ruby/commit/3e92b635fb5422207b7bbdc924e292e51e21f040

There are still 906 usages of frozen_string_literal: false in the MRI codebase as of today BTW (312 in stdlib files, the rest in tests).

I agree frozen String literals by default would be nice.
I'm not sure how much would be the compatibility impact.

Maybe we should gather some statistics in e.g. popular gems and all gems for how much files use the frozen-string-literal pragma.
Another way would be to run the test suites of a set of gems and see how much breaks with --enable-frozen-string-literal, including gems not using the pragma.