Feature #13083

Regexp#{match,match?} with a nil argument are deprecated and will raise a TypeError in Ruby 3.0

Description

Just for consistency

- patch: https://github.com/ruby/ruby/pull/1506
- spec: https://github.com/ruby/spec/pull/380

Currently behaves as ( ruby --version: ruby 2.5.0dev (2016-12-28 trunk 57228) [x86_64-darwin16] )

```
'string'.__send__(:^(~:, nil) #=> nil
'string'.match(nil) #=> TypeError: wrong argument type nil (expected Regexp)
'string'.match?(nil) #=> TypeError: wrong argument type nil (expected Regexp)
:symbol.__send__(:^(~:, nil) #=> nil
:symbol.match(nil) #=> TypeError: wrong argument type nil (expected Regexp)
:symbol.match?(nil) #=> TypeError: wrong argument type nil (expected Regexp)
/regex/__send__(:^(~:, nil) #=> nil
/regex/.match(nil) #=> nil
/regex/.match?(nil) #=> false
```

Expected to

```
'string'.__send__(:^(~:, nil) #=> nil
'string'.match(nil) #=> nil
'string'.match?(nil) #=> false
:symbol.__send__(:^(~:, nil) #=> nil
:symbol.match(nil) #=> nil
:symbol.match?(nil) #=> false
/regex/__send__(:^(~:, nil) #=> nil
/regex/.match(nil) #=> nil
/regex/.match?(nil) #=> false
```

Associated revisions

Revision 4fe89e08 - 10/26/2019 08:33 PM - Eregon (Benoit Daloze)
Add entry for Feature #13083 in NEWS
  - Move Unicode changes under String / Unicode for consistency with the rest.

Revision fbacfe68 - 11/03/2019 10:14 AM - Eregon (Benoit Daloze)
Update NEWS entry for Feature #13083

Revision 8852fa87 - 12/03/2019 09:40 PM - naruse (Yui NARUSE)
Revert "Regexp#match{?} with nil raises TypeError as String, Symbol (#1506)"
This reverts commit 2a22a6b2d84e6d539e75520a7f2ecf522d50890caf.
Revert [Feature #13083]

Revision 4cc83f9a - 12/14/2019 07:05 AM - znz (Kazuhiro NISHIYAMA)
[Feature #13083] is already reverted

History

#1 - 12/29/2016 12:15 AM - nobu (Nobuyoshi Nakada)
- Description updated
I think that the way it currently exists is logical. As =~ is an Object method, it is worth while keeping the more permissible behavior whereas errors should be thrown for match and match? if nil is passed. The change that I would propose would actually to have Regexp#match and Regexp#match? both raise a type error if nil is passed as an argument. The inconsistency seems to be stronger there.

Those methods (but =~) should consistently raise exceptions.

Matz.

I heard this change breaks activerecord.

https://twitter.com/shyouhei/status/1186122901667209216
https://twitter.com/shyouhei/status/1186123025604734976
https://github.com/rails/rails/blob/01336b71af6ac798e48101b4ceeb2de00397243aa/activerecord/lib/active_record/connection_adapters/abstract_adapter.rb#L52

and sprockets

https://twitter.com/shyouhei/status/1186127103231578113
https://github.com/rails/sprockets/blob/d5aa63a73915908d062ed889a86f121b84bf9c4f/lib/sprockets/http_utils.rb#L62

There are PRs to fix those: https://github.com/rails/rails/pull/37504

I know this change is to make the code consistent but this is a backward incompatible change and I don't think it should be applied to Ruby 2.7 without deprecation. The fact that we need to change Rails and many other gems to make this work is a good indication. I thought we were avoiding making backward incompatible changes before Ruby 3.0.

I am 100% with Rafael here, this is a very risky change, Rails and Sprockets are only a couple of projects, there will be tons of stuff that will not be caught.

Why not deprecate this for now and then next release it can be removed?

The detailed situation.

Rails and other libraries and codebases rely on /regex/.match?(nil) to return false. This change is making that method to raise an error in that situation.

Changing this behavior on Ruby 2.7 means that applications will break because of this change.

This change also have dubious value. While it makes things consistent now it require the callers to do one more check.
Code that before was:

```
/foo/.match?(some_variable)
```

Need to become:

```
some_variable && /foo/.match?(some_variable)
```

or
```
/foo/.match?(some_variable) unless some_variable
```

If you ask me, I believe this change is good, but it doesn't require a breaking change. We can deprecate the old behavior and then in a next version remove it.

The codebases can change their code, of course, but so they can change for any breaking change in Ruby. If we are being careful to not introduce breaking changes in Ruby, this change should also falls in that category in my opinion. If we are ok with this breaking change, why not with others like frozen string as default? Both change will require changes to codebases and both changes have dubious value.

#13 - 11/01/2019 06:02 PM - Eregon (Benoit Daloze)
I think everyone agrees this is too breaking (just look at how many call sites need to be changed in Rails and other gems), and it's very easy to deprecate the old behavior by warning when passing nil in Ruby 2.7.

The only question for me is: who will change it to a deprecation?

#14 - 11/01/2019 07:20 PM - kachick (Kenichi Kamiya)
- Subject changed from {String|Symbol}#match{?} with nil returns falsy as Regexp#match{?} to {String|Symbol}#match{?} with nil returns falsy as Regexp#match{?} since ruby 3.0

How about https://github.com/ruby/ruby/pull/2637?

#15 - 11/03/2019 12:35 AM - shugo (Shugo Maeda)
matz (Yukihiro Matsumoto) wrote:

Thank you for the input. If you can provide a more detailed situation, it would be better.

My situation is the same as Rafael's, and I fixed my code using the safe navigation operator:

https://github.com/shugo/textbringer/commit/c859d13

Code like `/regex/ =~ str` has often been used when `str` may be `nil`, and such code has been rewritten using `Regexp#match?` after Ruby 2.4. This change will break all such code.

#16 - 11/03/2019 10:13 AM - Eregon (Benoit Daloze)
- Subject changed from {String|Symbol}#match{?} with nil returns falsy as Regexp#match{?} since ruby 3.0 to Regexp#{match,match?} with a nil argument are deprecated and will raise a TypeError in Ruby 3.0

#17 - 11/03/2019 10:18 AM - Eregon (Benoit Daloze)
kachick (Kenichi Kamiya) wrote:

How about https://github.com/ruby/ruby/pull/2637?

I merged that PR, thank you. `Regexp#{match,match?}` with a `nil` argument now just warns.

The plan is then to make them raise `TypeError` in Ruby 3.0.

#18 - 11/03/2019 10:31 AM - Eregon (Benoit Daloze)
shugo (Shugo Maeda) wrote:

```
Code like `/regex/ =~ str` has often been used when `str` may be `nil`, and such code has been rewritten using `Regexp#match?` after Ruby 2.4. This change will break all such code.
```

That's a good point. We can still discuss on this ticket whether we should change behavior at all, but I think so far everyone was fine with deprecation (warning) and then removal later on, so I merged the PR
Is there a benefit to the change apart from consistency? A concrete benefit I mean.

- Eregon (Benoit Daloze) wrote:
  
  Code like `/regex/ =~ str` has often been used when `str` may be `nil`, and such code has been rewritten using `Regexp#match?` after Ruby 2.4. This change will break all such code.

  That's a good point.
  We can still discuss on this ticket whether we should change behavior at all,
  but I think so far everyone was fine with deprecation (warning) and then removal later on, so I merged the PR

  Thank you.
  I reopen this issue for further discussion.

- shugo (Shugo Maeda) wrote:
  
  Note: warning message should be improved if this is deprecated before 2.7.0.

- Dan0042 (Daniel DeLorme) wrote:

  Is there a benefit to the change apart from consistency? A concrete benefit I mean.

  I think consistency is good here, and also it seems to me `regexp.match?(s)` with `s` being `nil` could actually catch bugs where `s` is not expected to be `nil`. It seems rather odd to me to validate some input and even accept `nil` as an "OK" value to match against a `Regexp`. `nil` means typically "missing value" in such a case, and often deserves its own behavior (e.g., could be empty string is not provided, could be an error, could be special behavior just for `nil`, could be some non-empty default `String` which would help readability).

  In the Rails PR I noticed a case that doesn't but is fairly close to end up in `nil.to_i`, which would likely be a bug/ unintended: https://github.com/rails/rails/pull/37504/files#diff-c226a4680f86689c3c170d4bc5911e96

- shevegen (Robert A. Heiler) wrote:

  This is only one point of view.

  Sometimes there are "wrong" use cases or use cases that were not anticipated by the core team. In my opinion, whenever possible, it is best to encourage consistency and clarity, as close as possible to matz' original design consideration (because I think that this way people will not be confused as much; remember the confusion from people thinking that there is a singular "principle of least surprise", but then everyone using their own definition for that - that can not work).

  To the suggestion itself: I have no particular pro or con opinion either way. None of the code that I use would be affected by the change, should it come. Ruby 3.0 is not so far away so there may be lots of changes coming to ruby past 3.0 anyway. :)

- knu (Akinori MUSHA) wrote:

  Can't we reconsider this? Or we'll be doomed to back out all those changes we believed to improve performance.
knu (Akinori MUSHA) wrote:

I'm a bit surprised and confused. The change that has been made is not what this issue was originally about, and actually the opposite, I think.

Per matz's decision: https://bugs.ruby-lang.org/issues/13083#note-3
I renamed the issue in https://bugs.ruby-lang.org/issues/13083#note-16 to match the current change.
Now it's just a deprecation warning.

I believe many of us have got used to the original behavior, that is, methods of a Regexp object work permissively and accept nil, and we know we've migrated many pieces of code from /re/ =~ nilable / /re/ === nilable to /re/.match?(nilable) for the sake of performance and readability just as shugo said above.

Can't we reconsider this? Or we'll be doomed to back out all those changes we believed to improve performance.

nilable&.match?(/re/) would be an easy way to rewrite those cases, no?

matz (Yukihiro Matsumoto) and others: what do you think, should we make Regexp#match,match? consistent and not accept nil or accept them as special-case to match the =~ and === operators?

Eregon (Benoit Daloze) wrote:

I believe many of us have got used to the original behavior, that is, methods of a Regexp object work permissively and accept nil, and we know we've migrated many pieces of code from /re/ =~ nilable / /re/ === nilable to /re/.match?(nilable) for the sake of performance and readability just as shugo said above.

Can't we reconsider this? Or we'll be doomed to back out all those changes we believed to improve performance.

nilable&.match?(/re/) would be an easy way to rewrite those cases, no?

matz (Yukihiro Matsumoto) and others: what do you think, should we make Regexp#match,match? consistent and not accept nil or accept them as special-case to match the =~ and === operators?

I also received the same feedback from Rails people.

nilable&.match?(/re/) sounds a reasonable option, but why do we provide a such pitfall?

It's not my intention to suffer users. Let's cancel the change.

Matz.

Applied in changeset git|8852fa876039ed177fd5e867f36177d8aa9ff411c.

Revert "Regexp#match(?) with nil raises TypeError as String, Symbol (#1506)"

This reverts commit 2a22a6b2d8465934e75520a71dcf522d50890caf.
Revert [Feature #13083]