Ruby - Feature #19744

Namespace on read

06/27/2023 02:34 AM - tagomoris (Satoshi Tagomori)

Status:	Closed	
Priority:	Normal	
Assignee:		
Target version:		

Description

What is the "Namespace on read"

This proposes a new feature to define virtual top-level namespaces in Ruby. Those namespaces can require/load libraries (either .rb or native extension) separately from the global namespace. Dependencies of required/loaded libraries are also required/loaded in the namespace.

Motivation

The "namespace on read" can solve the 2 problems below, and can make a path to solve another problem:
The details of those motivations are described in the below section ("Motivation details").

Avoiding name conflicts between libraries

Applications can require two different libraries safely which use the same module name.

Avoiding unexpected globally shared modules/objects

Applications can make an independent/unshared module instance.

(In the future) Multiple versions of gems can be required

Application developers will have fewer version conflicts between gem dependencies if rubygems/bundler will support the namespace on read.

Example code with this feature

```
# your_module.rb
module YourModule
end

# my_module.rb
require 'your_module'

module MyModule
end

# example.rb
namespace1 = NameSpace.new
namespace1.require('my_module') #=> true

namespace1::MyModule
#=> #<Module:0x00000001027ea650>::MyModule (or #<NameSpace:0x00...>::MyModule ?)
namespace1::YourModule # similar to the above

MyModule # NameError
YourModule # NameError

namespace2 = NameSpace.new      # Any number of namespaces can be defined
namespace2.require('my_module') # Different library "instance" from namespace1

require 'my_module' # require in the global namespace
```

```
MyModule.object_id != namespace1::MyModule.object_id #=> true
namespace1::MyModule.object_id != namespace2::MyModule.object_id
```

The required/loaded libraries will define different "instances" of modules/classes in those namespaces (just like the "wrapper" 2nd argument of Kernel.load). This doesn't introduce compatibility problems if all libraries use relative name resolution (without forced top-level reference like ::Name).

"On read": optional, user-driven feature

"On read" is a key thing of this feature. That means:

- No changes are required in existing/new libraries (except for limited cases, described below)
- No changes are required in applications if it doesn't need namespaces
- Users can enable/use namespaces just for limited code in the whole library/application

Users can start using this feature step by step (if they want it) without any big jumps.

Motivation details

This feature can solve multiple problems I have in writing/executing Ruby code. Those are from the 3 problems I mentioned above: name conflicts, globally shared modules, and library version conflicts between dependencies. I'll describe 4 scenarios about those problems.

Running multiple applications on a Ruby process

Modern computers have many CPU cores and large memory spaces. We sometimes want to have many separate applications (either micro-service architecture or modular monolith). Currently, running those applications require different processes. It requires additional computation costs (especially in developing those applications).

If we have isolated namespaces and can load applications in those namespaces, we'll be able to run apps on a process, with less overhead.

(I want to run many AWS Lambda applications on a process in isolated namespaces.)

Running tests in isolated namespaces

Tests that require external libraries need many hacks to:

- require a library multiple times
- require many different 3rd party libraries into isolated spaces (those may conflict with each other)

Software with plugin systems (for example, Fluentd) will get benefit from namespaces.

In addition to it, application tests can avoid unexpected side effects if tests are executed in isolated namespaces.

Safely isolated library instances

Libraries may have globally shared states. For example, Oj has a global Obj.default_options object to change the library behavior. Those options may be changed by any dependency libraries or applications, and it changes the behavior of Oj globally, unexpectedly.

For such libraries, we'll be able to instantiate a safe library instance in an isolated namespace.

Avoiding dependency hells

Modern applications use many libraries, and those libraries require much more dependencies. Those dependencies will cause version conflicts very often. In such cases, application developers should resolve those by updating each libraries, or should just wait for the new release of libraries to conflict those libraries. Sometimes, library maintainers don't release updated versions, and application developers can do nothing.

If namespaces can require/load a library multiple times, it also enables to require/load different versions of a library in a process. It requires the support of rubygems, but namespaces should be a good fundamental of it.

Expected problems

Use of top-level references

In my expectation, ::Name should refer the top-level Name in the global namespace. I expect that ::ENV should contain the environment variables. But it may cause compatibility problems if library code uses ::MyLibrary to refer themselves in their deeply nested library code.

Additional memory consumption

An extension library (dynamically linked library) may be loaded multiple times (by dlopen for temporarily copied dll files) to load isolated library "instances" if different namespaces require the same extension library. That consumes additional memory.

In my opinion, additional memory consumption is a minimum cost to realize loading extension libraries multiple times without compatibility issues.

This occurs only when programmers use namespaces. And it's only about libraries that are used in 2 or more namespaces.

The change of dlopen flag about extension libraries

To load an extension library multiple times without conflicting symbols, all extensions should stop sharing symbols globally. Libraries referring symbols from other extension libraries will have to change code & dependencies.

(About the things about extension libraries, Naruse also wrote an entry.)

Misc

The proof-of-concept branch is here: https://github.com/tagomoris/ruby/pull/1
It's still work-in-progress branch, especially for extension libraries.

Related issues:

Related to Ruby - Feature #19024: Proposal: Import Modules	**Closed**
Related to Ruby - Feature #14982: Improve namespace system in ruby to avoidin...	**Assigned**
Related to Ruby - Feature #13847: Gem activated problem for default gems	**Assigned**
Related to Ruby - Bug #19990: Could we reconsider the second argument to Kern...	**Closed**
Related to Ruby - Feature #10320: require into module	**Open**
Related to Ruby - Feature #21311: Namespace on read (revised)	**Assigned**

History

#1 - 06/28/2023 10:08 PM - janosch-x (Janosch Müller)

a very thorough proposal with an attractive modularity to it!

if i may ask a few questions:

are the described problems, apart of dependency conflicts, very common in your experience? (i've worked on several large codebases with ~1000 gems and never saw these gems trying to create the same module or edit each other's configs.)

concerning dependency conflicts:

are they perhaps the lesser evil?

maybe "evolutionary pressure" on unmaintained dependencies is actually healthy for a language ecosystem?

if there were multiple versions of the same dependency running in one process, each with different bugs, wouldn't that be harder to understand?

concerning the scenario of running whole apps in namespaces:

couldn't the issues in the shared namespace, such as dependency conflicts, also happen within each individual namespace? would i then use namespaces within namespaces to work around it?

if i want to patch a library, e.g. by prepending a module with a bugfix, would i need to remember to apply that patch to all namespaces where i use that library?

#2 - 06/29/2023 12:54 AM - tagomoris (Satoshi Tagomori)

janosch-x (Janosch Müller) wrote in #note-1:

> are the described problems, apart of dependency conflicts, very common in your experience? (i've worked on several large codebases with ~1000 gems and never saw these gems trying to create the same module or edit each other's configs.)

Yes. I saw many problems around applications that I worked for in the past. Those things were "fixed" in any way (including making a closed fork of unmaintained libraries), eventually, but I believe that it could be much easier if we had namespaces.

> **concerning dependency conflicts:**
>
> are they perhaps the lesser evil?
>
> maybe "evolutionary pressure" on unmaintained dependencies is actually healthy for a language ecosystem?

I agree. From the viewpoint of the whole ecosystem, all libraries should be maintained well, and "pressure" can work well in most cases. But on the other hand, libraries that are not updated still exist even under very strong pressure. Loading different versions of gems can solve this problem.

I agree that the healthy pressure is very important for the ecosystem. So, we have to be careful when we design the UX/API of loading multiple version gems. If it doesn't make any pressure to update dependencies, it should make another kind of library ecosystem nightmare: many old dependencies are never updated. We should avoid it.

> if there were multiple versions of the same dependency running in one process, each with different bugs, wouldn't that be harder to understand?

In my current understanding, loading multiple versions can happen from dependencies of libraries (my app uses library A and B, A uses C ver 1.x, B uses C ver 2.x), not the direct dependency of the app itself. If the library A also has a bug coming from C ver 1.x, it's also a bug of A. We should update A. Isn't it?

> **concerning the scenario of running whole apps in namespaces:**
>
> couldn't the issues in the shared namespace, such as dependency conflicts, also happen within each individual namespace? would i then use namespaces within namespaces to work around it?

Let me check my understanding of your question. Is it?: Problems around namespaces can happen in individual namespaces. Should we use namespaces to avoid those things?

(If my understanding is correct,) yes. And, I missed mentioning it in the original description, namespaces can include other namespaces (just like modules can have submodules).

In other words, when we use namespaces to require/load other libraries, we don't take care of the namespaces used in those libraries. Namespaces in required libraries may happen, but users (the caller app/lib of NameSpace#require) don't have to consider it. Users just call the public API of those libraries.

> if i want to patch a library, e.g. by prepending a module with a bugfix, would i need to remember to apply that patch to all namespaces where i use that library?

Definitely, yes. And remember, monkey-patching different libraries is powerful but evil. With great power, great responsibility.

But I guess, we'll do NameSpace#require the code including both require('a_buggy_lib') and a monkey patch on ABuggyLib. If we do require a code with require('a_buggy_lib') without such patches, that code should not be affected from the bug.

#3 - 06/29/2023 02:27 PM - janosch-x (Janosch Müller)

> Problems around namespaces can happen in individual namespaces. Should we use namespaces to avoid those things? [...] yes.

A lot of Ruby code is written in a way that relies on extending other code. E.g. a lot of rails and rspec extensions add methods to rails or rspec objects which "enrich the DSL" and which end users call to make use of these extensions. Maybe with namespaces that could look like so?

```
User = Class.new(ActiveRecord::Base)
namespace = NameSpace.new
namespace.require('paper_trail')
namespace.instance_eval { ::User.has_paper_trail }
```

However, such a call might define methods or callbacks that reference constants which are unavailable in the "main" namespace, so I'm not sure how well it would work?

Then, of course, namespaces don't have to be applicable to everything to be useful.

#4 - 06/29/2023 11:22 PM - Dan0042 (Daniel DeLorme)

This proposal seems **very** similar to #19024. Is there a difference?

Also, IIUC it seems to imply the namespace will apply transitively to every load/require/autoload inside the namespace; there was a lot of opposition

to that idea in #19024.

#5 - 06/29/2023 11:50 PM - hsbt (Hiroshi SHIBATA)

- Related to Feature #19024: Proposal: Import Modules added

#6 - 06/29/2023 11:52 PM - hsbt (Hiroshi SHIBATA)

- Related to Feature #14982: Improve namespace system in ruby to avoiding top-level names chaos added

#7 - 06/29/2023 11:56 PM - tagomoris (Satoshi Tagomori)

> This proposal seems very similar to #19024. Is there a difference?

#19024 proposes a new top-level (Kernel?) method import. The API of this proposal is much different from it. A new module for namespaces (NameSpace, but the name doesn't matter to me...), and its instance methods (require/load).

and #19024 also says:

> Only load code once. When the same file is imported again (either directly or transitively), "copy" constants from previously imported namespace to the new namespace using a registry which maps which namespace (import) was used to load which file (as shown above with activerecord/activemodel). This is necessary to ensure that different imports can "see" shared files. A similar registry is used to track autoloads so that they work correctly when used from imported code.

But in the namespace of this proposal, libraries will be re-loaded by require/load separately from the global namespace. It's very important to isolate libraries in namespaces from each other and to avoid unexpected side effects.

> Also, IIUC it seems to imply the namespace will apply transitively to every load/require/autoload inside the namespace; there was a lot of opposition to that idea in #19024.

#19024 was closed by @shioyama himself (because "I don't really feel the need for further changes to Ruby...") and has not been declined by the Ruby core team.
And I still don't have the thing that I want in Ruby. What I want is different from his one.

#8 - 07/12/2023 03:31 AM - tagomoris (Satoshi Tagomori)

Update: now the PoC branch supports require/load of native extension libraries in namespaces.

#9 - 07/12/2023 05:19 AM - kou (Kouhei Sutou)

> To load an extension library multiple times without conflicting symbols, all extensions should stop sharing symbols globally. Libraries referring symbols from other extension libraries will have to change code & dependencies.

Do you have any idea how to change for it?
https://github.com/ruby-gnome/ruby-gnome/ calls functions provided by other gems. For example, gobject-introspection gem calls functions provided by glib2 gem because underlying GObject Introspection library depends on GLib library.

#10 - 07/12/2023 09:11 AM - tagomoris (Satoshi Tagomori)

kou (Kouhei Sutou) wrote in #note-9:

> Do you have any idea how to change for it?
> https://github.com/ruby-gnome/ruby-gnome/ calls functions provided by other gems. For example, gobject-introspection gem calls functions provided by glib2 gem because underlying GObject Introspection library depends on GLib library.

I don't have clear&straight solution for now. The possible options are (independent from each other):

a. Lift up all functions to Ruby level method calls on the glib2 side, then use them on gobject-introspection
b. Split the functions from glib2 gem to a new something (directory?), then share those code by all related gems
c. Build gobject_introspection.so to depend on glib2.so in the gem glib2

I think a and b are possible, but both require huge efforts, and I'm not sure if these are acceptable or not. "a" seems to have performance issue, and "b" seems to have hard things about maintenance and gem package size.

I may have a wrong premise. My premise is, once we mark a library "x.so" depends on another library "y.so", dlopen on "x.so" recursively loads "y.so" too, and symbols in "y.so" are readable from "x.so" even when RTLD_LOCAL is specified. (Do we need to tweak LD_LIBRARY_PATH?)

If it is correct, we may solve the problem by providing any kind of feature at the native extension build step.

#11 - 07/12/2023 09:12 AM - tagomoris (Satoshi Tagomori)

tagomoris (Satoshi Tagomori) wrote in #note-10:

> I may have a wrong premise. My premise is, once we mark a library "x.so" depends on another library "y.so", dlopen on "x.so" recursively loads "y.so" too, and symbols in "y.so" are readable from "x.so" even when RTLD_LOCAL is specified. (Do we need to tweak LD_LIBRARY_PATH?)
> If it is correct, we may solve the problem by providing any kind of feature at the native extension build step.

This section is about the option "c".

#12 - 07/12/2023 02:44 PM - jeremyevans0 (Jeremy Evans)

tagomoris (Satoshi Tagomori) wrote in #note-10:

> kou (Kouhei Sutou) wrote in #note-9:
>
> > Do you have any idea how to change for it?
> > https://github.com/ruby-gnome/ruby-gnome/ calls functions provided by other gems. For example, gobject-introspection gem calls functions provided by glib2 gem because underlying GObject Introspection library depends on GLib library.
>
> I don't have clear&straight solution for now. The possible options are (independent from each other):
>
> a. Lift up all functions to Ruby level method calls on the glib2 side, then use them on gobject-introspection
> b. Split the functions from glib2 gem to a new something (directory?), then share those code by all related gems
> c. Build gobject_introspection.so to depend on glib2.so in the gem glib2

None of these are feasible for sequel_pg, which calls C functions in the pg gem extension, which is developed and maintained independently. sequel_pg should work with whatever version of pg is installed, it cannot force a specific version (so b and c option are not possibles). The C functions it calls in pg are not with VALUE arguments, and wrapping all calls would likely result in a significant slowdown (when the main purpose of sequel_pg is to improve performance). It would be unreasonable to ask pg maintainers to add Ruby methods that they don't want to expose to Ruby level to work around Namespace limitations, IMO.

If this extension issue only affects users who are using this new namespacing feature, it may be acceptable. If this is a general change that also affects users who are not using this new namespacing feature, I think the backwards compatibility costs outweigh the benefits.

In regards to the feature in general, looking at my large libraries, all of them use absolute :: references at least occasionally, and in many cases doing so is required as the relative reference would refer to a different class/module.

#13 - 07/12/2023 09:13 PM - tagomoris (Satoshi Tagomori)

jeremyevans0 (Jeremy Evans) wrote in #note-12:

> > a. Lift up all functions to Ruby level method calls on the glib2 side, then use them on gobject-introspection
> > b. Split the functions from glib2 gem to a new something (directory?), then share those code by all related gems
> > c. Build gobject_introspection.so to depend on glib2.so in the gem glib2
>
> None of these are feasible for sequel_pg, which calls C functions in the pg gem extension, which is developed and maintained independently. sequel_pg should work with whatever version of pg is installed, it cannot force a specific version (so b and c option are not possibles). The C functions it calls in pg are not with VALUE arguments, and wrapping all calls would likely result in a significant slowdown (when the main purpose of sequel_pg is to improve performance). It would be unreasonable to ask pg maintainers to add Ruby methods that they don't want to expose to Ruby level to work around Namespace limitations, IMO.

Thank you for another case. I got the details of those libraries.
Naruse suggested me another way to call the C function of different shared objects in different gems (https://twitter.com/nalsh/status/1679060154262913024). I'll take time to understand it and then comment here as a different option later. (It requires changes on the native extensions which depends on C functions of different gems too, anyway.)

> If this extension issue only affects users who are using this new namespacing feature, it may be acceptable. If this is a general change that also affects users who are not using this new namespacing feature, I think the backwards compatibility costs outweigh the benefits.

I see. Roughly speaking, there are 2 options about how to open .so libraries:

A. dlopen with RTLD_LOCAL in default. Enforce native extensions to call external C-functions in a specific way (needs upgrades)
B. dlopen with RTLD_GLOBAL in default. use RTLD_LOCAL only when a gem is marked as namespace-safe.

I know that A introduces a huge compatibility problem. I want to re-check if it's acceptable or not with the benefit of namespaces (multi-version loading, avoiding name conflicts, etc).

B is a different option. It's ideal if the marking is done automatically without any additional information (but it seems hard).
It may be feasible to request native extension users to mark their gems namespace-safe or not. In this case, for the namespace safety, native extensions which use C-functions of other gems should use a new API to call C-functions of other native extensions.

#14 - 07/17/2023 07:44 AM - tagomoris (Satoshi Tagomori)

tagomoris (Satoshi Tagomori) wrote in #note-13:

> Naruse suggested me another way to call the C function of different shared objects in different gems (https://twitter.com/nalsh/status/1679060154262913024). I'll take time to understand it and then comment here as a different option later. (It requires changes on the native extensions which depends on C functions of different gems too, anyway.)

This is the option "d" here:

d. Ruby will have APIs for native extension libraries to provide handles (the result of dlopen with RTLD_LOCAL) for the specified .so filename in its namespace. Native extension libraries can fetch symbols of C functions by the handle and a function name and call it

Just for example:

```
// in Ruby
handle = dlopen("anylib.so", RTLD_LOCAL);

// in native extensions to use C functions in anylib.so
handle = dynlib_handle(vm->current_namespace, "anylib.so");
func = dlsym(handle, "my_target_func");
func(...)

// or provide a function to return a func directly
func = resolve_symbol(vm->current_namespace, "anylib.so", "my_target_func");
```

This seems possible, and easy to implement. Native extensions which depend on other gems' C functions should be upgraded for namespace-safety, but there are no big changes or complex upgrade paths.
Native extensions which use this API should mark themselves as namespace-safe.

#15 - 07/19/2023 06:01 AM - kou (Kouhei Sutou)

Thanks for providing your ideas.

tagomoris (Satoshi Tagomori) wrote in #note-10:

> a. Lift up all functions to Ruby level method calls on the glib2 side, then use them on gobject-introspection

It's impossible for the glib2/gobject-introspection case with the similar reason of the sequel_pg.

> b. Split the functions from glib2 gem to a new something (directory?), then share those code by all related gems

It's impossible for the glib2/gobject-introspection case because there are stateful functions. The case wants to share implementations with them.

> c. Build gobject_introspection.so to depend on glib2.so in the gem glib2

It's possible for the glib2/gobject-introspection case. It's already done (but only) for Windows build.

tagomoris (Satoshi Tagomori) wrote in #note-13:

> Naruse suggested me another way to call the C function of different shared objects in different gems (https://twitter.com/nalsh/status/1679060154262913024).
>
> Keep the handle when we execute handle=dlopen("glib2.so", RTLD_LOCAL) in the glib2 gem, pass it to gobject-introspection gem and call a function by func=dlsym(handle, "FUNCTION_NAME"); func(...).

It's not realistic for the glib2/gobject-introspection case.
(BTW, I don't know why glib2 needs to execute dlopen() by itself instead of reusing the dlopen()-ed handle by Ruby.)

In the case, I need to rewrite many codes and it also breaks API/ABI compatibilities.

> It's ideal if the marking is done automatically without any additional information (but it seems hard).

I think that it's not realistic.

> It may be feasible to request native extension users to mark their gems namespace-safe or not.

I agree with this.

tagomoris (Satoshi Tagomori) wrote in #note-14:

> Native extensions which depend on other gems' C functions should be upgraded for namespace-safety, but there are no big changes or complex upgrade paths.

I need to change at least about 300 lines for gobject-introspection. (Ruby-GNOME has more similar gems.)

```
$ grep -E '(rbg|RVAL)' ext/gobject-introspection/*.c | wc -l
305
```

(If I need to do it, I'll do it. But I think that they are big (many?) changes...)

#16 - 07/21/2023 10:28 AM - rubyFeedback (robert heiler)

Just a quick comment to Jeremy's statement:

> In regards to the feature in general, looking at my large libraries, all of them use absolute :: references at least occasionally

I use :: quite a lot in my gems as well so I can relate to this, even though none of my gems are as popular as jeremy's maintained code base. The biggest one I specifically have to use :: is for the module called "Colours", as a gem, where I then use in other gems that I maintain a submodule called "Colours" as well. So I kind of have to use "::Colours" to refer to the toplevel variant. It's not a big issue for me but one has to keep in mind to which submodule/toplevel module one refers to. I did run into issues of other gems and "namespaces" though when I had a gem called "configuration" and referred to it via ::Configuration.

Unfortunately "gem install configuration" was owned by someone else (an existing gem), so in the end I integrated my "module Configuration" into another gem, and call into that namespace. So at the least from that point of view, even if this is not directly related to tagomoris' use case, I can kind of relate to some of the rationale for the proposal. If we look at it more globally then this is a bit similar to refinements, e. g. on the one hand being able to modify ALL of ruby at "run-time" (which can be a super-great feature), but also having more fine-tuned control (such as via refinements, e. g. telling ruby that we want to refer to modified core classes and core modules only in a given project's "namespace", but not modify it outside of that).

#17 - 08/15/2023 01:27 PM - tagomoris (Satoshi Tagomori)

I'm trying to implement a feature to provide symbols from other extensions to extensions. That should satisfy the requirements like calling C-functions in an extension from other extensions.
The required changes for extensions (calls C-functions of other extensions) is,

- modify the prototypes
- call a new function (resolve_ext_symbol here) and assign the returned value

The diff of sequel_pg with this feature is here: https://github.com/jeremyevans/sequel_pg/pull/55/files
It uses 4 functions in "pg_ext", requiring 4 line changes, and 4 line additions.

What do you think about it, Jeremy, Kou and others?

#18 - 08/15/2023 01:30 PM - tagomoris (Satoshi Tagomori)

> The diff of sequel_pg with this feature is here: https://github.com/jeremyevans/sequel_pg/pull/55/files

I made a wrong pull-request. The right thing is here: https://github.com/tagomoris/sequel_pg/pull/1

#19 - 08/15/2023 02:46 PM - jeremyevans0 (Jeremy Evans)

tagomoris (Satoshi Tagomori) wrote in #note-17:

> I'm trying to implement a feature to provide symbols from other extensions to extensions. That should satisfy the requirements like calling C-functions in an extension from other extensions.
> The required changes for extensions (calls C-functions of other extensions) is,
>
> - modify the prototypes
> - call a new function (resolve_ext_symbol here) and assign the returned value
>
> The diff of sequel_pg with this feature is here: https://github.com/jeremyevans/sequel_pg/pull/55/files
> It uses 4 functions in "pg_ext", requiring 4 line changes, and 4 line additions.
>
> What do you think about it, Jeremy, Kou and others?

Considering it causes almost no code changes, and can be backwards compatible with some preprocessor macros, it looks like a good solution to me if this feature is accepted. It should probably be prefixed with rb_ if it is a symbol exported by Ruby.

#20 - 09/16/2023 01:46 PM - Eregon (Benoit Daloze)

- Description updated

#21 - 09/16/2023 02:15 PM - Eregon (Benoit Daloze)

It seems interesting.
Regarding the motivation I wanted to mention some alternatives.

> Running multiple applications on a Ruby process
> Running tests in isolated namespaces
> Safely isolated library instances

These 3 motivations can actually be solved today on TruffleRuby and I think on JRuby as well by using multiple "interpreter instances" (like the MVM project in CRuby).
For example on TruffleRuby the API is Polyglot::InnerContext.new { |ctx| ctx.eval("ruby", "...") }.
These interpreter instances are fully isolated, the only data they share is deeply immutable.
The JITed code is also shared between the different interpreter instances which is very important for warmup (with this proposal it does not seem possible to share JITed code, so it will be a lot of work for the JIT to recompile every copy of a method/gem).
At least on TruffleRuby it's also possible to pass values from one interpreter instance to another, this works by passing a proxy object, so e.g. any method call is done in the interpreter instance to which the object belongs.
That isolation is much stronger because it cannot be broken at the Ruby level, unlike ::SHARED = Object.new in this proposal.

On the native level it's a lot more tricky to isolate, for instance it's not really possible to fully fake different working directories in the same process (considering native extensions might also rely on the CWD).
Does your approach handle that, how?

And indeed as you mention native libraries are also very hard to isolate.
TruffleRuby avoids that to some extent by using GraalVM LLVM to execute their bitcode so then it can have multiple copies of the same library in a process without needing to copy the .so and without risking to share global native symbols/variables.

A big limitation of this proposal for the first 2 motivations is it cannot run these multiple applications/tests in parallel.
Unless using Ractor maybe, but not quite sure if both would work well together (and Ractor has many restrictions).

Also the CRuby GC might not scale well with multiple applications in the same process.

#22 - 09/18/2023 01:04 AM - tagomoris (Satoshi Tagomori)

Eregon (Benoit Daloze) wrote in #note-21:

> On the native level it's a lot more tricky to isolate, for instance it's not really possible to fully fake different working directories in the same process (considering native extensions might also rely on the CWD).
> Does your approach handle that, how?

No, this idea doesn't provide anything about working directories. In my idea, it's totally different from namespaces. (Possibly, there may be an idea of tools to orchestrate namespaces, working directories, and others to build isolated Ruby VM instances in CRuby.)

I also currently exclude parallel processing and CG overhead problems from this feature's scope. I'm focusing on creating a minimum feature to provide isolated library/code sets in a process to keep the distance to the goal short.

(Just an idea, Ractor + Namespace MAY be a good combination... is it?)

#23 - 09/19/2023 04:10 AM - kou (Kouhei Sutou)

Sorry. I missed #note-17.

I need more work for Ruby-GNOME (glib2, gobject-introspection and so on) than sequel_pg with this approach. Because we have many exported symbols. For example: https://github.com/ruby-gnome/ruby-gnome/blob/master/glib2/ext/glib2/glib2.def

(Note that I'll do it when we decide to use this approach. In the case, I'll add a convenience header with convenience macros to glib2 and use it from gobject-introspeciton and so on.)

FYI: I noticed that Python already has a similar feature, PyCapsule: https://docs.python.org/3/c-api/capsule.html

#24 - 09/21/2023 08:18 AM - Eregon (Benoit Daloze)

@deivid @hsbt (Hiroshi SHIBATA) and other RubyGems/Bundler maintainers:
What do you think of this proposal, especially the part related to RubyGems & Bundler?
This would enable loading different versions of a gem in the same process, together with changes in RubyGems & Bundler.

But is that what you want and is that what Ruby users want?
And is this change realistic to do in RubyGems & Bundler?

My concerns are:

- I'm not sure many Rubyists want a Bundler that feels like npm/node_modules with every dependency duplicated N times (+ of course the longer bundle times, etc).
- Will those changes be compatible enough, or would it require to be opt-in?
- Should RubyGems/Bundler try to have the minimum number of versions for each gem or stop resolving and have N copies of each gem based on how many gems depend on it?
- How can Gemfile.lock handle this? Probably the format needs to change significantly to support that?
- How can Bundler.require handle this? I think it's not possible without extra information like passing a NameSpace to gem "foo" in the Gemfile. And that feels rather hacky because how to share NameSpace objects in Gemfile and in app code? Also how to serialize such a NameSpace in Gemfile.lock? (Or maybe that last one is not necessary?)
- If this feature is used rarely it seems more likely to not work reliably for all gems, also it relies on gems with an extension depending on another extension doing changes, which might or not happen.
- How long might it take to implement all this in RubyGems & Bundler? Until then this feature is probably unusable for most Rubyists, as most use RubyGems & Bundler of course.

@tagomoris (Satoshi Tagomori) It'd be great to hear your thoughts on this too.

To be clear I don't think any of this should block merging this feature, but I think this feature will likely not be useful for 99+% Rubyists until that is fully implemented and supported in RubyGems & Bundler.

#25 - 09/22/2023 07:57 AM - hsbt (Hiroshi SHIBATA)

- Related to Feature #13847: Gem activated problem for default gems added

#26 - 09/22/2023 07:59 AM - sawa (Tsuyoshi Sawada)

It may be useful to allow a code block if that does not cause a trouble.

```
require "my_module"

NameSpace.new do
  self # => #<NameSpace:0x00...>
  require "my_module"
  MyModule.object_id != ::MyModule.object_id # => true
end
```

#27 - 09/22/2023 08:13 AM - hsbt (Hiroshi SHIBATA)

In short term, I'm positive to this feature because RubyGems and Bundler couldn't use or vendor C extensions of default gems now. see #13847.

And RubyGems and Bundler have few vendored ruby library like fileutils, uri and etc. We should stop it and use version provided by standard library with lock version by this feature.

In long term, I'm not sure how provide this feature to end-user by RubyGems/Bundler with a new syntax of Gemfile.

#28 - 09/22/2023 10:12 AM - Eregon (Benoit Daloze)

hsbt (Hiroshi SHIBATA) wrote in #note-27:

> And RubyGems and Bundler have few vendored ruby library like fileutils, uri and etc. We should stop it and use version provided by standard library with lock version by this feature.

Right, that'd be a nice use-case.
This would however only work for 3.3+ and for stdlib or default gems (and it seems there are fewer of these every release), it would not work for bundled gems as RubyGems itself and I guess Bundler also cannot depend on bundled gems.
So if such libraries are needed they would still need to be vendored manually.
Currently bundler/lib/bundler/vendor has connection_pool fileutils molinillo net-http-persistent thor tmpdir tsort uri.
fileutils tmpdir tsort uri are all default gems, the rest is non-stdlib/default/bundled gems so those still need to stay vendored.
fileutils tmpdir tsort uri still need to be vendored for older Rubies, so only when Bundler would drop support for 3.2 it can actually use this feature for them and other stdlib & default gems it wants/needs.

From the POV of other/starting Ruby implementations, it doesn't seem too nice that to run RubyGems one needs such a complex/intricate feature though.

#29 - 09/26/2023 05:09 PM - luke-gru (Luke Gruber)

This is an interesting feature imo, and it could be used well with ractors. For instance, if every namespace is owned by only 1 ractor then essentially some restrictions on using ractors could be loosened significantly when using the namespace.

For example, currently we can't use class variables in ractors because they're not safe when running parallel code (other ractors could mutate them).

In a namespace, though, the code is fully isolated to 1 ractor so anything can be done. As long as the namespace can't be passed around to other ractors I think it would work well. The only problem is the use of ::TopLevelConstants in a namespace, which would violate these rules and cause this to not work. I think these types of constants should be resolved to their namespace and not the actual top-level.

I think per-ractor GC is also going to be worked on at some point, so even though there would be code bloat (copies of the code) the GC load would be split across ractors.

In the short term I don't see much use of the feature for running different copies of applications in threads with how threads work currently, but running many copies of micro-apps it could be interesting with the MaNy project (light-weight ractors more like goroutines).

#30 - 09/29/2023 02:08 AM - tagomoris (Satoshi Tagomori)

Eregon (Benoit Daloze) wrote in #note-24:

> @tagomoris (Satoshi Tagomori) It'd be great to hear your thoughts on this too.

At first, the PoC code now has the version keyword argument on require even without RubyGems (with a little hacky code). So the code https://bugs.ruby-lang.org/issues/19772#note-29 works well on it. (I want to move the feature to RubyGems, eventually, though.)

My very rough ideas about Namespace and RubyGems/Bundler integration are:

- Bundler should resolve dependencies in the same way as the current one (Most libraries have just 1 version/copy under the vendor)
- Bundler should recommend users resolve conflicts by updating libraries as far as possible
- Users can configure to depend on multiple versions of a library only when conflicts cannot be resolved, through uncomfortable/un-user-friendly options/commands/etc

Having multiple versions of libraries could lead to a future with many un-maintained libraries. I think it's uncomfortable and unhealthy. So I want to design a kind of developer experience to choose updating libraries at first. Namespace should be a last resort, not the first choice.

With the idea above, Namespace is an opt-in feature. We will be able to roll out Namespace features step by step (without RubyGems/Bundler integration at first, then with them), while having a period to let gem authors support Namespace (if needed).

#31 - 10/15/2023 02:16 PM - maciej.mensfeld (Maciej Mensfeld)

If I may.

:: scope

Similar to Jeremy, my gems heavily use the top-level reference. Like others, I also have namespaces that collide with the root once. For example, Karafka::ActiveJob operates by referencing itself and its internals locally but also refers to ::ActiveJob. In some scenarios, I use it to ease with readability for developers as it is easier to track things starting from the root level. Sometimes, it is needed because of name conflicts.

User Experience

I am unsure if namespaces will be easy to debug/work with. When I debug gems, at the moment, it is fairly easy for me to understand the versioning and to be able to modify them in place when researching some bugs/monkey patches / etc. With the addition of the namespace, I can imagine this may be a bit cumbersome due to the requirement of understanding the scope in which a given piece of code operates.

Security

Code Execution

There are no explicit security risks I can imagine, but I would say that it may, as others mentioned, loosen the pressure on upgrading. I can also imagine that it may be a bit confusing to get multiple records on the same vulnerabilities per project from tools like bundler audit, as there may be scenarios that the same vulnerability will be assigned to a few versions in use.

Another question is on the complexity of things like reachability analysis. It may become a bit more complex (though not impossible - look npm).

Dependencies

Please read the Bundler section below.

Bundler

- While the plugin API is not widely used, there are some (including me) that utilize them. I'm almost certain that introduction of namespaces will cause API changes to the plugins API.

```
Bundler should resolve dependencies in the same way as the current one (Most libraries have just 1 version/cop
y under the vendor)
Bundler should recommend users resolve conflicts by updating libraries as far as possible
Users can configure to depend on multiple versions of a library only when conflicts cannot be resolved through
 uncomfortable/un-user-friendly options/commands/etc
```

I tried to find information on whether our PubGrub implementation would support such behavior but I couldn't. I only found references from Elm and Dart implementations stating that:

> Versions use the semantic versioning scheme (Major.Minor.Patch).
> Packages cannot be simultaneously present at two different versions.

Same with Dart (ref: https://dart.dev/tools/pub/versioning):

> Instead, when you depend on a package, your app only uses a single copy of that package. When you have a shared dependency, everything that depends on it has to agree on which version to use. If they don't, you get an error.

This would mean that to support any "on conflict" suggestions or resolution, we would have to replace/enhance this engine. Such changes always pose a significant risk of introducing new dependency confusion bugs. On top of that, we need to answer the question of how such multi-versions operations should behave on constraints coming from multiple sources. What if the same "name" comes from two sources, one public and one private? Since we will allow for namespacing, should such a thing be allowed? If so, we may have to update how gems are cached locally to include their full source to avoid name collisions.

Finally, if we go with this:

> Bundler should resolve dependencies in the same way as the current one

It only mitigates "complete" conflicts but does not prevent from situations where A & B depend on C and are both able to resolve to something old but acceptable. The issue of one of dependencies "limiting" things will still stay.

> How can Gemfile.lock handle this? Probably, the format needs to change significantly to support that?

Yes, though I think it can be done in a way that would be compatible as long as there are no namespaces in use.

> How long might it take to implement all this in RubyGems & Bundler? Until then, this feature is probably unusable for most Rubyists, as most use RubyGems & Bundler.

Great question to David Rodríguez - I'll ping him.

Learning from other Registries / Technologies

Before getting "full in" with a feature like this, I think it would be good to research its frequency/scale of usability. Maybe we could get anonymous data on structures of lock files from technologies that support this to analyze the frequency of such feature adoption. We could take both OSS data to understand how often this is being used in packages and get data from actual projects and some viable insights.

Summary

I'm hesitant about it.

On one side, I've been missing it a few times myself. On the other hand, I am not sure that this is a feature that will get wide adoption. Once it is in and beyond an experimental phase, it will have to get solid support from Ruby Core, RubyGems, and Bundler. For it to be considered usable, it must have a great user experience around usage and debuggability and solid documentation for users to understand.

> But on the other hand, libraries that are not updated still exist even under very strong pressure.

Absolutely, and at the same time, some of them get adopted and become maintained again. The pressure will be lowered if such a problem can be "bypassed" by namespaces.

While useful at some times, I do not feel this will get wide adoption, especially as @tagomoris (Satoshi Tagomori) himself said here:

> Bundler should resolve dependencies in the same way as the current one (Most libraries have just 1 version/copy under the vendor)
> Bundler should recommend users resolve conflicts by updating libraries as far as possible

#32 - 10/15/2023 09:46 PM - martinemde (Martin Emde)

In the original proposal you mention:

> Dependencies of required/loaded libraries are also required/loaded in the namespace.

I think this means that any require statements in the required file will also be required within the namespace. I have a question though about what happens if the file requires a different gem. Which version of the gem is loaded? How do we decide?

Currently, you will get either the version that rubygems picks or, with Bundler, you get bundled version. In the proposed solution, bundler and rubygems must not behave as they currently do.

I see 2 solutions for how requires and dependency resolution are handled within namespaces:

1. Dependency resolution within a namespace is fully assumed by the creator of the namespace. Nothing is inherited. Rubygems does not exist in the new namespace unless required.
2. Rubygems and Bundler must fully support namespaces.

The first case does not provide much value. The idea of libraries having dependencies is a rubygems idea. Without a rubygems you must control your own LOAD_PATH and decide which versions to load. Finding a compatible version, as given by the library's gemspec, is something done at install or by bundler during resolution, so the gem and it's dependencies will need to have the correct paths added before requiring the gem.

A simplistic implementation might add bundler/inline to each new namespace, declaring the gems and version used in the namespace at the top of the namespace. There are obvious reasons that this is not desirable from the perspective of running a production application. Bundler would need to be fully isolated in this case just like the rest of the namespace. If we follow my option 1 to this conclusion, we are arrive at something like a ruby virtual machine that allows passing messages between the isolated namespace "processes". There are languages like this.

The second case is where we see benefit. Implementing a wrapper around the basic namespace primitive seems like an absolute requirement. Therefore the value of this proposal is directly dependent on support in rubygems and bundler.

I believe implementing support will be very time consuming. I imagine this future of multi-namespace dependency resolution as a "multiverse" of dependencies, where there are "alternate universes" with branching consequences for each multi-version gem. Bundler must resolve dependencies for each namespace and so bundler must know which gems will be used in each namespace.

This dictates that Bundler is the owner of application namespaces. They will rarely be created within typical application code and primarily exist as dependency resolution namespaces.

The proposed primitive is only the start. Significant work would need to be done to bundler and rubygems to make this useful to almost anyone at all. I don't doubt that there is significant value to those encountering the problems discussed in this thread, but realizing that value will require significant work in rubygems and bundler. I encourage us to consider the work involved in rubygems and bundler as part of the implementation of this proposal and a requirement for making it more than a curious but mostly unused feature.

#33 - 10/16/2023 08:50 AM - maciej.mensfeld (Maciej Mensfeld)

Here are a few more thoughts from me that may have an impact on this, and I hope David will speak up as well:

Bundler and RubyGems (RG&B) should be considered in deciding on the acceptance or not of this feature. Why? Because the end-user expectation of the Ruby ecosystem will be for them to support that. Without this, we will have a half-baked functionality that, due to backward compatibility, will have to be maintained. So, to start with, we will have to solve many problems in RG&B related to the implementation and user experience, and then we will have to support it for years or forever. Once that is done, putting aside core errors, I can expect that it is going to be RG&B that will have to deal with end-user complaints and problems, similar to how it is RG&B that gets the support requests about the platform and its operations + bundler errors, etc. This requires work and resources.

Even if RG&B decides not to do it initially, I can expect that there will be pressure on bringing this up because "why it is a feature in Ruby but not in RG&B that are part of Ruby ecosystem". I've seen no propositions of any DSL or API for Bundler to operate and multi-version algorithms for resolution are also not easy to create/implement and make secure.

The more I read this feature proposal, the more I see that it was thought from Ruby language perspective but not fully scoped from the Ruby ecosystem perspective. IMHO, it lacks proper scope assessment looking from RG&B perspective.

#34 - 10/16/2023 10:10 AM - byroot (Jean Boussier)

> Bundler and RubyGems (RG&B) should be considered in deciding on the acceptance or not of this feature.

I'm just as doubtful as you on the desirability of being able to load two distinct versions of a dependency, but it is only one of the possible use case of this feature.

Even if bundler/rubygems keep preventing to install/bundle two versions of a package, some users may still find that feature useful to better enforce boundaries inside their large applications.

#35 - 10/16/2023 10:15 AM - maciej.mensfeld (Maciej Mensfeld)

byroot (Jean Boussier) wrote in #note-34:

> > Bundler and RubyGems (RG&B) should be considered in deciding on the acceptance or not of this feature.
>
> I'm just as doubtful as you on the desirability of being able to load two distinct versions of a dependency, but it is only one of the possible use case of this feature.
>
> Even if bundler/rubygems keep preventing to install/bundle two versions of a package, some users may still find that feature useful to better

> enforce boundaries inside their large applications.

You are absolutely right and maybe my previous statement of:

> Bundler and RubyGems (RG&B) should be considered in deciding on the acceptance or not of this feature.

Should be revisited to say:

"RG&B should be considered in designing/preparing the scope of this feature" rather than its acceptance. My point is not to prevent this feature but to define the expected result of this feature within the whole ecosystem and not just the core. Maybe there are way in which this could coop with RG&B as they are but from my perspective, there are still many unknowns and we may end up with a half-baked (from the ecosystem perspective) feature.

#36 - 10/16/2023 06:49 PM - deivid (David Rodríguez)

From my side, I'm happy if this feature is added because as mentioned by @hsbt (Hiroshi SHIBATA) it'd be useful for us internally.

But I'm pretty strongly against providing any fuctionality that allows resolving and loading multiple versions of the same gem In Bundler/RubyGems, even if this feature could make that potentially easier. It'd still be a lot of work to even provide a sane specification of the feature (let alone implement it), and over the years there's been little interest in something like this. I don't believe Ruby developers miss the ability of NPM package managers to do this, rather the opposite.

#37 - 10/16/2023 08:46 PM - martinemde (Martin Emde)

Besides my position on integrating with RG&B, I wanted to add my thoughts about the feature itself separately and how it might end up being used in gems.

Usage in gems

I immediately imagine this feature being picked up by gems (again, please ignore whether or not bundler supports multiple versions of gems).

Gem Namespaces

Most gems want to isolate their own namespace and not pollute the namespaces into which they are required. Right now it is *customary* but not required that a gem defines a top level module that matches the gem name, following a naming convention. These modules are "weak" namespaces compared to the proposed namespace because I can't contain anything I require within the namespace as well.

If namespaces had always been available, modules and namespaces would be the same thing. We would just consider any require/load statement within a module to load nested into that module. require on Kernel is global, require within a module is namespaced. You might even expect places that currently require a file within a method or module to instead write Kernel.require.

Gems may, with this feature, become more "strongly" namespaced, like the following:

```
# lib/my_gem.rb
namespace MyGem
  require "oj"

  autoload :Feature, "my_gem/feature.rb"
  # maybe this feature uses json.
end
```

Now this gem doesn't need to worry about which json library is already loaded or if this gem has polluted the global namespace by requiring a certain json library.

Gems like this become self contained "cells", that provide an interface and functionality but don't alter any code outside of their own namespace, even when they require external libraries.

Patching/plugins

If my_gem wanted to add features to something outside of itself, if it was a plugin for another gem, then it needs to interact with an external namespace. In that case I might do something like this:

```
require "rack"

# same code as above in lib/my_gem.rb

Rack.include(MyGem::Feature) # `::Rack` should also work the same here
```

Just looks like normal code, but I bring it up because in means I should be able to include a module from one namespace into a module from another namespace. That module may even have access to a specific set of libraries that aren't in the destination module. Is support for that planned?

Breaking patches out of a namespace

If I require a namespaced library using a namespace, I could control what the top level name of the lib, as proposed, but then I can't still use the patch to Rack that I wrote above.

Assuming MyGem is a name conflict for me, I could do the following.

```
namespace Libs
  # How do I add the Rack patch again? Like this? I don't have a good idea for how this would work.
  Rack == ::Rack # but how does the external require come into this namespace.

  require("my_gem")
  # Maybe
end

Rack = Libs::Rack # This feels more natural, but still weird and wouldn't work for multiple patches.
```

Top level ::MyGem

Not being able to use ::MyGem seems like a big problem. Without a solution for accessing "the top level of my namespace" and "the top level Kernel namespace", I expect there will be some unfixable module references. There should be a syntax for how this can be done.

Within MyGem, I would expect ::MyGem to access the gem's namespace declared in the code above. If MyGem was required into another namespace by an external require, that should be invisible to this gem and ::MyGem should not change how it works.

When you mix the patching of rack with a nested outside_namespace.require("my_gem"), then we must expect that rack will also be required into that namespace and patched inside the namespace.

Conclusion

Appreciate your time to read my exploration of this feature. Unless we do something to prevent it, I think gems will do this because it makes sense to do so. Is this going to work? Will it break everything?

Thank you!

#38 - 10/17/2023 07:25 PM - fxn (Xavier Noria)

What happens with the top-level constants defined by the interpreter? In particular, is Object.object_id != ns1::Object.object_id != ns2::Object.object_id?

#39 - 10/18/2023 05:02 PM - Eregon (Benoit Daloze)

@fxn This proposal does not define multiple Object classes.

I think this proposal has limited value because if gems use this functionality they will multiply their memory footprint because each copy of a gem (with the same version) is pure footprint overhead.
And there is no real isolation either, it is easy to break it, e.g. with ::Foo.

#40 - 10/18/2023 06:28 PM - fxn (Xavier Noria)

Thanks @Eregon (Benoit Daloze).

My main concern is that the gem author is not in control. You write your code, assume constants are stored in certain places, assume the nesting you are seeing with your eyes [*], and assume what constant resolution algorithms are going to find. In particular, *your* top-level constants belong to Object.

However, with this, that control is moved to client code, the gem author has lost it. That doesn't sound right to me, as a gem author your code has to be deterministic.

The case of Oj should not justify a workaround in the language, in my opinion, the design simply did not take into account the gem can be a dependency of several. That design is the one to be fixed, I think.

[*] That is why I am also not in favor of a second argument to load.

#41 - 10/19/2023 12:34 AM - Dan0042 (Daniel DeLorme)

In general I'm against npm-style multiple loading of dependencies, but I think I found a use case for this that I could get behind.

Would it be possible to use this features to load a Ractor-incompatible library inside a NameSpace inside a Ractor? Let's say:

```
Ractor.new do
  ns = NameSpace.new  #might be better to call this "Sandbox" ?
  ns.require('nokogiri')
  ns::Nokogiri.HTML5(some_html)
end
```

Normally, Nokogiri cannot be used inside a Ractor, but using this NameSpace feature it might be possible to have Nokogiri fully sandboxed/scoped to a single Ractor, so it would behave like it was in the main Ractor. Possible?

#42 - 10/20/2023 01:36 AM - tagomoris (Satoshi Tagomori)

martinemde (Martin Emde) wrote in #note-37:

> ## Patching/plugins
>
> If my_gem wanted to add features to something outside of itself, if it was a plugin for another gem, then it needs to interact with an external namespace. In that case I might do something like this:
>
> ```
> require "rack"
>
> # same code as above in lib/my_gem.rb
>
> Rack.include(MyGem::Feature) # `::Rack` should also work the same here
> ```
>
> Just looks like normal code, but I bring it up because in means I should be able to include a module from one namespace into a module from another namespace. That module may even have access to a specific set of libraries that aren't in the destination module. Is support for that planned?

@martinemde Let me check my understanding of your question: "Does Rack (or ::Rack) work with Oj loaded separately in the namespace for MyGem after the include?"

If so, yes. It's planned (and it's working on my PoC branch, without the "Gem Namespaces" feature).

#43 - 10/20/2023 01:43 AM - tagomoris (Satoshi Tagomori)

fxn (Xavier Noria) wrote in #note-38:

> What happens with the top-level constants defined by the interpreter? In particular, is Object.object_id != ns1::Object.object_id != ns2::Object.object_id?

@fxn This proposal says: "a new feature to define virtual top-level namespaces in Ruby", and we can define methods at the top level. So, Object in a namespace should be different from each Object in other namespaces (and the global one).

Yes, Object.object_id != ns1::Object.object_id and ns1::Object.object_id != ns2::Object.object_id should be true. It's required to have different sets of top-level methods in each namespace.

#44 - 10/20/2023 01:51 AM - tagomoris (Satoshi Tagomori)

Dan0042 (Daniel DeLorme) wrote in #note-41:

> Would it be possible to use this features to load a Ractor-incompatible library inside a NameSpace inside a Ractor? Let's say:
>
> ```
> Ractor.new do
>   ns = NameSpace.new  #might be better to call this "Sandbox" ?
>   ns.require('nokogiri')
>   ns::Nokogiri.HTML5(some_html)
> end
> ```
>
> Normally, Nokogiri cannot be used inside a Ractor, but using this NameSpace feature it might be possible to have Nokogiri fully sandboxed/scoped to a single Ractor, so it would behave like it was in the main Ractor. Possible?

@Dan0042 Currently, in this proposal, it's impossible. Now I don't have any features/specifications/mechanisms related to Ractor in this proposal because it makes the feature more complex.
Precisely speaking, it's not been considered. I think the feature is very useful&helpful if it can load Ractor-unready libraries only in a Ractor. But it's not planned for now. All modules in every namespace are visible from all Ractors.

#45 - 10/20/2023 02:43 AM - martinemde (Martin Emde)

> If so, yes. It's planned (and it's working on my PoC branch, without the "Gem Namespaces" feature).

Nice. Thanks for taking the time to answer and clarify.

Also, I think that "gem namespaces" are something that are enabled implicitly by this feature. I brought up like "is this what we want? because that's how you get this." So unless there's a way to force a flat namespace during a require, then I suspect namespaced gems will be published.

#46 - 10/20/2023 03:06 PM - fxn (Xavier Noria)

@tagomoris (Satoshi Tagomori) got it, thanks!

So, if I have

```
class C
end
```

and client code loads that file under a namespace ns, then ns::Object.constants includes :C?

And ::C in my required code is locally resolved to Object::C, which externally is ns::Object::C or simply ns::C?

#47 - 10/20/2023 03:57 PM - jeremyevans0 (Jeremy Evans)

If Object.object_id != ns1::Object.object_id, I assume BasicObject.object_id != ns1::BasicObject.object_id, but Ruby currently does not allow for copying the root class:

```
ruby -e BasicObject.dup
-e:1:in `initialize_copy': can't copy the root class (TypeError)
        from -e:1:in `initialize_dup'
        from -e:1:in `dup'
        from -e:1:in `<main>'
```

How is this situation resolved? I'm assuming that ns::Object = Object.dup, but maybe that isn't how it works.

Additionally, for:

```
class A
end

class B < A
end
```

then if a namespace is created afterward, ns::B.superclass is ns::A, correct? If that is the case, does this copy the entire class hierarchy and modify all super pointers?

#48 - 10/20/2023 05:38 PM - Eregon (Benoit Daloze)

> Yes, Object.object_id != ns1::Object.object_id

OK. I did not expect that and the PoC does not do anything like that.
What about class C; end when loaded in a namespace, from which Object does it inherit then?
What about instance & class variables of Object, would you deep copy them?
I think this approach cannot provide full isolation, e.g. ::Object.define_method would anyway be global, so it seems not so valuable to try to isolate more (and it also it increases the complexity of the feature a lot).

The copy of Object (assuming it's via Object.dup or so) seems pretty bad BTW because if someone defines a method in Object then it won't be visible in ns1::Object, unless the method definition occurs before the namespace is created.
That seems a recipe for subtle bugs.
Having multiple snapshot copies of Object is I think quite confusing.
For instance ActiveSupport core extensions to Object would not work in namespaces, unless all core classes are duplicated which seems undesirable and starts to sound like multiple interpreters (or processes) is a much cleaner and clearer approach then (BTW BasicObject, Class and Module all cannot meaningfully be copied).

What about String, would you copy that too?
I would assume not, because it seems pretty bad to duplicate every String literal per namespace and have N copies of each core class (String interning would only work per namespace).
Plus it would be quite inefficient as it would not be possible to just embed the String instance in the bytecode then (with frozen_string_literal: true).

Even Ractor does not try to isolate modules & classes, they are shared between ractors.

My understanding is this proposal only namespaces new modules/classes defined while a namespace is active when loading/requiring files (using Kernel#load(file, wrap=module)).
And it can load each file multiple times, i.e. (at most) once per namespace.
That part seems simple enough, the rest seems incredibly complex and confusing semantics-wise.

#49 - 10/22/2023 02:32 PM - fxn (Xavier Noria)

@Eregon (Benoit Daloze) I understand the patch is a partial implementation of the proposal (also, the author of the ticket said Object and ns::Object would be different).

It cannot be otherwise, because if Object and ns::Object where the same object and we have

```
X = 1
```

then a reference to X would resolve to Object::X, which is ---under that hypothesis--- the same as ns::Object::X, thus not providing the desired isolation.

On the other hand, if all objects and all pointers for all instances are adjusted, we are seeing other debatable consequences.

I feel like there is a pain point around isolation in the community, but the current design of Ruby simply cannot address it, because all is global, and private or private_constant are insufficient.

I don't have a technical answer to that, but I feel like you should have a formal definition of "library" (nowadays, the abstraction is "loading individual files with require", a gem is not a formal entity in the language), and a definition of "library-level visibility", not available to client code.

#50 - 10/22/2023 04:03 PM - fxn (Xavier Noria)

Let me clarify the example using X, because the integer value and the wording may confuse, perhaps.

If foo.rb has

```
X = 1
```

I expect as a Ruby programmer that in the next line Object.constants includes :X. If client code loads foo.rb "under a virtual namespace", then the idea would be that you don't see X in your top-level. But if Object and ns::Object are the same, the resolution algorithm will find X just fine (assuming the constants table is not per "virtual namespace").

That is why I believe Object.equal?(ns::Object) has to be false to accomplish the wanted isolation.

#51 - 10/23/2023 06:40 AM - tagomoris (Satoshi Tagomori)

After my last comment and comments from others, I should say I could make a wrong comment about ns::Object. I didn't think about it enough. Sorry, guys.

My main concern about Object was the owner of top-level methods. My comment ("Yes, Object.object_id != ns1::Object.object_id ... should be true.") has been derived from the idea that "top-level methods will be defined as the singleton method of Object" and a too easy idea that "If a namespace has its own top-level methods, its Object should be different from the global one."
But it was too easy and simplified, without concern about the object hierarchy, etc.

Now, there are two options for the top-level method owners.

A: Namespaces have their own Object, but it will be defined as a subclass of the global Object

It's just like class ns::Object < Object; end. The definition will be empty. So all top-level method calls in the namespace will be resolved by the global Object in default. Once new top-level methods are defined in the namespace, they'll be defined on the ns::Object.

Pros) It does not change the current Ruby's way: "The main object will be backed by Object, and top-level methods will be defined on Object."
Cons) It breaks the hierarchy of objects. Once a String object is created in a namespace, its inheritance tree will be String < ::Object, not String < ns::Object < Object. This could be confusing.

B: Namespaces do not have their own Object, and all top-level methods and constants will be defined on the Namespace object itself

In this option, all top-level methods in a namespace will be defined as singleton methods of the namespace object. The main object will be backed by the namespace object too. All top-level module/class names in a namespace will be owned by the namespace (ns.constants.contains(:X) #=> true).

This idea is almost as-is of my PoC implementation, except for the "main" and its behavior in namespaces.

Pros) Object hierarchy will be clean and less confusing. Implementation will be much simpler than the option A.
Cons) The "virtual" top-level object will not be Object anymore. It changes Ruby's manner and may be confusing.

=====

Now I prefer option "B" (yes, it's different from my last comment), but I want to hear feedback and comments from others. Thank you for the discussion @fxn and @Eregon (Benoit Daloze).

I'm sorry again for my confusing last comment. I thought @fxn's discussion point was about the owner of singleton methods and constants, and I wanted to explain those are different between namespaces, and between a namespace and the global. But my answer was simply incomplete and wrong.

#52 - 10/23/2023 06:52 AM - tagomoris (Satoshi Tagomori)

jeremyevans0 (Jeremy Evans) wrote in #note-47:

> If Object.object_id != ns1::Object.object_id, I assume BasicObject.object_id != ns1::BasicObject.object_id, but Ruby currently does not allow for copying the root class:
> (snip)...
> How is this situation resolved? I'm assuming that ns::Object = Object.dup, but maybe that isn't how it works.

@jeremyevens0 I had never had an idea of the different BasicObject in namespace from the global one. And I cannot suppose the reason why a namespace should have its own BasicObject when it has its own Object. Could you explain the reason why you assumed it?

(And my idea of per-namespace Object is not by .dup but defining a subclass of ::Object to inherit the global top-level methods.)

> Additionally, for:
>
> ```
> class A
> end
>
> class B < A
> end
> ```
>
> then if a namespace is created afterward, ns::B.superclass is ns::A, correct? If that is the case, does this copy the entire class hierarchy and modify all super pointers?

If a namespace is created and then it loads the Ruby script with the definition above, it newly defines A and B as ns::A and ns::B from the global viewpoint, and yes, ns::B.superclass is ns::A. It doesn't copy anything from the global namespace.

If a namespace is just created afterward (without loading the script), there are no definitions of A and B in the namespace. ns::A just causes NameError.

Does it answer your question?

#53 - 10/23/2023 01:15 PM - Dan0042 (Daniel DeLorme)

> That is why I believe Object.equal?(ns::Object) has to be false to accomplish the wanted isolation.

Not necessarily. Object.constants could return a different set of values depending on which Namespace you're in, similar to how Thread.current returns a different object depending on which Thread is currently running.

#54 - 10/23/2023 01:44 PM - fxn (Xavier Noria)

@Dan0042 The previous paragraph ends with

> assuming the constants table is not per "virtual namespace"

Did not elaborate on this because it is not on the table, and because it opens its own rabbit hole.

#55 - 10/23/2023 02:08 PM - Dan0042 (Daniel DeLorme)

Sorry I missed that. But I think this idea needs to be "on the table" because duplicating Object opens a pretty big rabbit hole of its own.

#56 - 10/23/2023 02:38 PM - jeremyevans0 (Jeremy Evans)

tagomoris (Satoshi Tagomori) wrote in #note-52:

> jeremyevans0 (Jeremy Evans) wrote in #note-47:
>
> > If Object.object_id != ns1::Object.object_id, I assume BasicObject.object_id != ns1::BasicObject.object_id, but Ruby currently does not allow for copying the root class:
> > (snip)...
> > How is this situation resolved? I'm assuming that ns::Object = Object.dup, but maybe that isn't how it works.
>
> @jeremyevens0 I had never had an idea of the different BasicObject in namespace from the global one. And I cannot suppose the reason why a namespace should have its own BasicObject when it has its own Object. Could you explain the reason why you assumed it?

I assumed it because you stated Object.object_id != ns1::Object.object_id. Seems odd that Object.superclass == ::Object in a namespace, but that is certainly better than duplicating the entire class hierarchy. However, it's going to be strange to have:

```
# Opens namespace Object
class Object
  def foo; end
end

# Opens global String?
class String
```

```
  def foo; end
end
```

> Does it answer your question?

Yes, thank you.

#57 - 10/23/2023 02:59 PM - fxn (Xavier Noria)

@tagomoris (Satoshi Tagomori) I feel that for the proposal to be better understood, it needs to explain where are constants defined, and how constant resolution algorithms change.

Please, remember than when you write

```
class C
end
```

there is a constant lookup, because Ruby needs to know if it has to create or reopen a class object.

#58 - 10/24/2023 02:31 AM - tagomoris (Satoshi Tagomori)

@jeremyevans0 (Jeremy Evans) That's the point. I agree that it makes things strange if namespaces have their own Object.

And, @fxn also pointed out the discussion point - what should class C; end in a namespace do when it's already defined in the global namespace?
If it reopens C for monkey patching, it means the code in namespaces cannot redefine its own C separately from the global namespace. So, this behavior is completely against the main feature of this proposal.

So, the behavior about C should be:

- the constant reference C should be resolved in the namespace at first, then (if there is no C in the namespace) be resolved in the global namespace
- the statement class C; end should define a new class C in the namespace because C is not defined in the namespace

The big problem: in namespaces, existing monkey patching code will not work. With the PoC implementation, monkey patches should be like class ::String; ...; end (I know people don't write such code currently).
I still do not have a clear solution. We discussed about this point in the DevMeeting@2023-10 but didn't get any solution/conclusion at that time.

#59 - 10/24/2023 07:32 AM - fxn (Xavier Noria)

> the statement class C; end should define a new class C in the namespace because C is not defined in the namespace

@tagomoris (Satoshi Tagomori) which would be the superclass of that C if loaded under a virtual namespace?

#60 - 10/24/2023 08:36 AM - tagomoris (Satoshi Tagomori)

@fxn It depends on whether namespaces have their own Object or not (https://bugs.ruby-lang.org/issues/19744#note-51).
If they have (on option A), the superclass is ns::Object.
If they don't (on option B), the superclass is ::Object.

But now an idea popped up to me that the superclass of C can be ::Object even when namespaces have their own Object. In that case, Object is just an object to contain constants and singleton methods.
That should be, of course, super confusing. Having Object per namespace sounds very bad idea to me now...

#61 - 10/24/2023 09:12 AM - fxn (Xavier Noria)

@tagomoris (Satoshi Tagomori) One tricky part is constant resolution, I believe. For example, consider:

```
class String # (1)
end

class C # (2)
  String
end

module M # (3)
  String
end
```

If String becomes a new class object in (1), because the constant does not exist in the namespace, then in (2) the constant reference points to the top-level String. Reason is, the resolution algorithm finds the constant in the ancestor chain of C.

Similarly, in (3) the current resolution algorithm finds String in Object because it is manually checked for modules once the nesting and the ancestor chain are exhausted.

However, besides stuff like that, the main problem is that the author of this code

```
class String
end
```

is no longer sure they are reopening the class. It depends on the caller, they lost determinism. This is my main concern.

Another question, what happens with dynamic require calls? Consider for example

```
class Adapter
  def self.for(name)
    require "adapter/#{name.underscore}"
    Adapter.const_get(name, false).new
  end
end
```

Is that dynamic require able to pass the namespace down?

#62 - 10/24/2023 12:57 PM - tagomoris (Satoshi Tagomori)

fxn (Xavier Noria) wrote in #note-61:

> @tagomoris (Satoshi Tagomori) One tricky part is constant resolution, I believe. For example, consider:
> (snip)

@fxn Ah, yes. Thank you for pointing it out. That's another problem of having per-namespace Object.
In the case without per-namespace Object, top-level constants belong to the namespace and it'll be resolved prior to the global top-level constants.
So, this problem can be one of the reasons to choose the design without per-namespace Object.

> However, besides stuff like that, the main problem is that the author of this code
>
> ```
> class String
> end
> ```
>
> is no longer sure they are reopening the class. It depends on the caller, they lost determinism. This is my main concern.

Yes. It depends on the caller, in my opinion, even without namespaces. Imagine (your sample code is in "callee.rb"):

```
class C < BasicObject
  def methods; []; end
end
Object.const_set(:String, C)
require './callee'
```

I know it's an extreme example, but I want to say with this example that namespaces make code less deterministic, but the original code is also not completely deterministic. It depends on the caller even for now. Isn't it?
The part of my proposal title "on read" means: it depends on the caller.
(Of course, I also want to find a way to not break compatibility of existing numerous gems around monkey patches, though.)

> Another question, what happens with dynamic require calls? Consider for example
>
> ```
> class Adapter
>   def self.for(name)
>     require "adapter/#{name.underscore}"
>     Adapter.const_get(name, false).new
>   end
> end
> ```
>
> Is that dynamic require able to pass the namespace down?

If the caller of Adapter.self exists in the namespace, the require is called in the namespace. I think it should be expected behavior for all users.

The problem is the case when it is called as ns::Adapter.for from out of the namespace. My opinion is that it should be called as ns.require because require is to load a library in that namespace. But the current PoC doesn't do it, and now I don't have clear idea how to implement it. I'll try to find it.

#63 - 10/24/2023 01:12 PM - fxn (Xavier Noria)

@tagomoris (Satoshi Tagomori) Regarding the BasicObject example, yes, Ruby allows you to re-assign to builtin constants, but it is a different level of lack of guarantee in practice, the way I see it. Perhaps there are *levels* of determinism?

Nowadays, under normal circumstances you expect

```
class String
  alias blank? empty?
end
```

to reopen the class object stored originally in String. If that depends on how you are loaded, your code won't work as you programmed it.

#64 - 10/24/2023 01:24 PM - Dan0042 (Daniel DeLorme)

@tagomoris (Satoshi Tagomori), have you considered the idea that ::Foo should refer to the Foo constant within the namespace instead of the top-level? That would mean the namespace is fully isolated and prevented from reaching outside of itself, which makes sense to me. The namespace can be seeded with the same constant names and objects as the core ruby environment so that ::Object and others are available by default. In the ticket I haven't managed to find any explanation of why we need to reach outside of the namespace, what's the use case?

#65 - 10/24/2023 02:12 PM - tagomoris (Satoshi Tagomori)

@fxn Yes, I know my example was too extreme. I just wanted to say it's not enough reasonable to say "It's not deterministic."
I will propose any feature/limitation/behavior about reopening classes (whitelists? declarations? or something else?).
And I wish if Ruby has any keywords or syntax only to reopen classes.

```
class extension String
end
```

(The idea above is just an idea, not my serious proposal :P)

#66 - 10/24/2023 02:35 PM - tagomoris (Satoshi Tagomori)

Dan0042 (Daniel DeLorme) wrote in #note-64:

> @tagomoris (Satoshi Tagomori), have you considered the idea that ::Foo should refer to the Foo constant within the namespace instead of the top-level?

Yes, but I concluded that it's both almost impossible and unrealistic.

If code in a namespace cannot refer the global namespaces, it means everything in a Ruby process should be copied into the namespace. Otherwise, code in namespaces cannot refer :String, ::Kernel, and everything else. It's an unbelievably high-cost operation, and also unsafe because Ruby's core classes aren't designed with such copy-safety.

At the same time, if a namespace copies definitions of the global namespace, libraries cannot define their own classes/modules in the namespace. For example, if oj is already required in the global namespace, it causes a conflict when another require "oj" (for a different version) is called in the namespace. It just doesn't work.

#67 - 10/24/2023 02:51 PM - Eregon (Benoit Daloze)

Since this proposal is not providing full isolation, I think it should be kept as simple and minimal as possible.
If class String; end would define ns::String then "abc".class == String would not hold, that seems pretty bad.
And that ns::String would just be an unusable empty shell.

So mostly I think we should rely on the semantics of Kernel#load(file, wrap: wrap_module).
For example class C; end does, if there is an existing C return it, and otherwise defines wrap_module::C.

This means loading ActiveSupport twice (e.g. in 2 namespaces) would produce some "method redefined" warnings, since ActiveSupport monkey-patches Object, Array, String, etc.
I don't think that is solvable with a proposal like this one, it would require making copies of such core classes which is a deep rabbit hole.
At that point, it's much closer to multiple interpreters in one process or fork and might as well do/use that as it's much easier to understand and the isolation there is near complete.
OTOH these warnings may be fairly harmless as long as it's the same definition for the same monkey-patched method, but that is not always the case (e.g. if redefining a method based the previous definition and not just defining a new method)

These limitations does make me think the practical value of the proposal is lower than people might expect.
Probably it cannot be used effectively with several gems which use monkey-patching.

#68 - 10/24/2023 07:30 PM - fxn (Xavier Noria)

I see it as @Eregon (Benoit Daloze).

Let me share my point of view more broadly.

First of all, I believe solutions should be given to library authors, not to client code. Bottom up. If client code needs to workaround limitation of the language, you have to revise the language, not hack around it. Isolation may be cool provided it is transparent to the code being isolated. Like, for your app living in a container is quite transparent, as @Eregon (Benoit Daloze) said, forking is quite transparent.

C++, Java, Python, and many others have namespaces and import mechanisms. Ruby does not have them in the same manner, but modules and

conventions de facto act as separators among libraries. So we already have the tool to avoid conflicts.

As I said, Oj is the one to revise its API, in my view.

We do not have a tool to *formally* prevent access to internal lib stuff from client code, granted, but that is a different problem to the one addressed here (according to the motivations in the description). And, at the end of the day, in a so remarkably dynamic language like Ruby, at some point discipline from client code is to be expected.

#69 - 10/25/2023 12:30 AM - tagomoris (Satoshi Tagomori)

Eregon (Benoit Daloze) wrote in #note-67:

> So mostly I think we should rely on the semantics of Kernel#load(file, wrap: wrap_module).
> For example class C; end does, if there is an existing C return it, and otherwise defines wrap_module::C.

Kernel#load(file, wrap) doesn't work as you wrote. class C defines a new class C even when the caller script already defined C.

```
# main.rb
class C
  def yay
    "c"
  end
end
m = Module.new
load('./sub.rb', m)

# sub.rb
class C
  def foo
    yay
  end
end

p C.new.foo

# result
sub.rb:3:in `foo': undefined local variable or method `yay
' for #<#<Module:0x00000001006d3980>::C:0x0000000100974c48> (NameError)

    yay
    ^^^
 from /Users/tagomoris/sub.rb:7:in `<top (required)>'
```

#70 - 10/25/2023 07:13 AM - fxn (Xavier Noria)

load file, mod pushes mod to the nesting. The constant lookup issued by the class/module keywords checks the first element of the nesting (Object if the nesting is empty).

This is a feature I don't personally like very much (said that above), because client code has the ability to change your execution context in a non-transparent way (that C in the example is no longer ::C). You could accomplish that before this feature was added too with the string variant of evals, but that was more obscure.

#71 - 10/25/2023 02:08 PM - Eregon (Benoit Daloze)

tagomoris (Satoshi Tagomori) wrote in #note-69:

> Kernel#load(file, wrap) doesn't work as you wrote. class C defines a new class C even when the caller script already defined C.

Interesting. So that means with the current POC it doesn't work to monkey-patch classes like e.g. activesupport does? (i.e., warning or errors when loading activesupport twice, conflicts if different versions)
Unless they use C.class_exec do ... end but that seems much less common than class C; end.

I guess the semantics of class C; end are like that is they allow things like module Foo; class String; end; end and that defines a new String class.

Maybe we could change the semantics of class C; end to lookup in Object after mod with load(file, mod)?
But that wouldn't work e.g. if Oj is loaded in the global namespace and then in a child namespace.
So it seems impossible for class C; end to handle both isolation and monkey patching (unless all Object constants have copies under each namespace which has its own problems and tons of complexity).

It does feel like Ruby does not provide the proper tools for this kind of namespace isolation though, and that it cannot easily be added either.
Maybe this is the wrong way to approach things, MVM (multiple interpreters in one process) or fork are well known and have clear isolation boundaries and semantics.

#72 - 10/29/2023 01:52 AM - retro (Josef Šimánek)

Eregon (Benoit Daloze) wrote in #note-24:

> @deivid @hsbt (Hiroshi SHIBATA) and other RubyGems/Bundler maintainers:

Hello, Josef (https://github.com/simi) from RubyGems team here.

> What do you think of this proposal, especially the part related to RubyGems & Bundler?

Even I do like the basic idea of the proposal (and I do see benefit mainly for RubyGems/Bundler internals), I would prefer to keep possible RubyGems/Bundler user changes in separate issue and keep this one as simple as possible.

> This would enable loading different versions of a gem in the same process, together with changes in RubyGems & Bundler.
> But is that what you want and is that what Ruby users want?

I'm strongly against this, since this promotes few anti-patterns.

Per my experience RubyGems.org is currently very healthy ecosystem and one of the main reason is fact opening PR and kindly ask for release of gem on GitHub (or any other platform) is usually the easiest way to fix the dependency problems and this is nicely in line with whole MINASWAN concept. Every RubyGems.org user benefits from this.

> And is this change realistic to do in RubyGems & Bundler?

IMHO not currently, this will probably result in need of extending/replacing resolver.

> To be clear I don't think any of this should block merging this feature, but I think this feature will likely not be useful for 99+% Rubyists until that is fully implemented and supported in RubyGems & Bundler.

I see no reason to block this, but as mentioned, it would be great to keep RubyGems/Bundler expectations outside of this issue.

#73 - 11/14/2023 07:46 PM - Eregon (Benoit Daloze)

- Related to Bug #19990: Could we reconsider the second argument to Kernel#load? added

#74 - 11/14/2023 07:59 PM - Eregon (Benoit Daloze)

@tagomoris (Satoshi Tagomori) Given this feature is not possible to provide correctly and I don't think it's fixable either, I think it would be best to close/reject this issue, WDYT?

One big problem is monkey patching via class String; ...; end e.g. like in activesupport cannot possibly work in isolated namespaces.
That defines a new broken/almost-empty String class which is bad, as then any references to String in code loaded in that namespace would be wrong (e.g. "".is_a?(String) # => false).
And class X; end cannot reopen an existing class under ::Object otherwise class Foo in a namespace would be shared with the global namespace and defeat the whole point of namespaces.

And there are also the various problems raised by @fxn here and in #19990.

If we want isolation in Ruby I think it cannot be half-way, I think it must be multi-interpreter/VM (or existing multi-processes) with clear semantics and full isolation.
That is a well-understood model and any sharing (e.g. of some immutable data structures, bytecode, etc) there is transparent and safe, it's just an optimization.
Anything in between is incredibly fragile and complicated to understand + it would break for many existing gems, so it would be unsafe to use (reminds me a bit of $SAFE).

Regarding RubyGems/Bundler using something like this to vendor its dependencies without imposing a given version on the application:

- It's probably best for these to avoid depending on native extensions as much as possible anyway.
- The current approach by manually vendoring Ruby code under some module is actually as good as this could be and a lot clearer, simpler and less magical.

#75 - 04/02/2024 06:34 AM - hsbt (Hiroshi SHIBATA)

- Related to Feature #18376: Version comparison API added

#76 - 04/02/2024 06:35 AM - hsbt (Hiroshi SHIBATA)

- Related to deleted (Feature #18376: Version comparison API)

#77 - 04/02/2024 06:35 AM - hsbt (Hiroshi SHIBATA)

- Related to Feature #10320: require into module added

#78 - 05/06/2025 05:19 AM - tagomoris (Satoshi Tagomori)

This is replaced by #21311.
Could someone close this?

#79 - 05/06/2025 05:23 AM - jeremyevans0 (Jeremy Evans)

- Status changed from Open to Closed

#80 - 05/06/2025 12:12 PM - Eregon (Benoit Daloze)

- Related to Feature #21311: Namespace on read (revised) added