Ruby master - Feature #14982

Improve namespace system in ruby to avoiding top-level names chaos

08/11/2018 04:26 AM - jjyr (Jinyang Jiang)

Status:	Open
Priority:	Normal
Assignee:	matz (Yukihiro Matsumoto)
Target version:	3.0

Description

Updated: https://bugs.ruby-lang.org/issues/14982#note-5

Why

Ruby has evaluation all class/module names in top-level context(aka TOPLEVEL_BINDING).
As a user we basically hard to know how many names in the current context, is causing chaos in some cases. For example:

case 1:

Put common used errors class in a single file, like below

```
# utils/errors.rb

class FooError
end

class BarError
end
```

In other files under 'utils' we want to use those errors, so the best practice is to use require_relative 'errors' in each file we need.

```
# utils/binary_helper.rb

# we forget require errors

module BinaryHelper
# ...
  raise BarError
# ...
end
```

But sometime we may forget to require dependencies in a file, it's hard to notice because
if RubyVM already execute the requires we still can access the name BarError,

but if user directly to require 'utils/binary_helper', he/she will got an NameError.

case 2:

Two gems use same top-level module name, so we can't use them together

The Reason of The Problem

The reason is we let module author to decision which module user can use. ('require' is basically evaluation, highly dependent on the module author's design)

But we should let users control which names to use and available in context. As many other popular languages dose(Rust, Python..)

I think the solution is basically the same philosophy compares to refinement feature.

The Design

I propose an improved namespace to Ruby, to solve the problems and still compatible with the current Ruby module system.

```
class Foo
end

# introduce Kernel#namespace
namespace :Hello do
  # avoiding namespace chaos
  # Foo -> NameError, can't access TOPLEVEL_BINDING directly

  # Kernel#import method, introduce Foo name from TOPLEVEL_BINDING
  import :Foo

  # in a namespace user can only access imported name
  Foo

  # import constant to another alias name
  # can avoid writing nested module/class names
  import :"A::B::C::D", as: :E

  # require then import, for convenient
  import :"A::B::C::D", as: :E, from: 'some_rb_file'

  # import same name from two gems
  import :"Foo", as: :Foo_A, from: 'foo_a'
  import :"Foo", as: :Foo_B, from: 'foo_b'

  # import names in batch
  import %i{"A::B::C::D", "AnotherClass"}, from: 'some_rb_file'

  # import and alias in batch
  import {:"A::B::C::D" => :E, :Foo => Foo2}, from: 'some_rb_file'

  class Bar
    def xxx
      # can access all names in namespace scope
      [Foo, Foo_A, Foo_B]
    end
  end
end

Hello.class #  -> module. namespace is just a module
Hello::Bar # so we do not broken current ruby module design

# namespace system is intent to let user to control names in context
# So user can choose use the old require way

require 'hello'

Hello::Bar


# Or user can use namespace system as we do in hello.rb

namespace :Example do
  import :"Hello::Bar", as: :Bar
  Bar # ok
  Foo # name error, cause we do not import Foo in :Example namespace
end

Foo # ok, cause Foo is loaded in TOPLEVEL_BINDING

# define nested namespace

# more clear syntax than “module Example::NestedExample”
namespace :NestedExample, under: Example do
end

namespace :Example2 do
```

```
  namespace :NestedExample do
  end
end
```

Pros:

- Completely compatible with the current module system, a gem user can completely ignore whether a gem is write in Namespace or not.
- User can completely control which names in current context/scope.
- May solve the top module name conflict issue(depends on VM implementation).
- Avoid introducing new keyword and syntax.
- Type hint or name hint can be more accuracy under namespace(not sure).

Cons:

- Need to modify Ruby VM to support the feature.

History

#1 - 08/11/2018 04:30 AM - jjyr (Jinyang Jiang)

- Description updated

#2 - 08/11/2018 04:39 AM - jjyr (Jinyang Jiang)

- Description updated

#3 - 08/11/2018 10:21 AM - jjyr (Jinyang Jiang)

- Description updated

- Subject changed from Introduce new namespace system to ruby to avoiding top-level names chaos to Improve namespace system in ruby to avoiding top-level names chaos

#4 - 08/11/2018 12:41 PM - shevegen (Robert A. Heiler)

I think this has come up before in other issue requests, at the least
in one variant or the other (if I recall correctly, Hiroshi Shibata
also suggested some variant that is a bit similar to your "import"
example, but I do not remember the full issue's content; may have
been some months ago or perhaps a few years).

I also had a few ideas; e. g. to be able to attach meta-information
to class/modules (that way we can find out who is the original author,
when and what changes may have been made etc...).

I agree with this comment here a lot by the way:

```
import :"A::B::C::D", as: :E
```

Not necessarily about using the name "import", but with the ability
to re-define namespaces at require time. We can of course already
do so by including modules and removing (older) constants, but I
always thought it may be more elegant to be able to do so the moment
we require ruby code.

I am not so sure about the rest of the suggestion. I don't have any
particularly strong pro or con opinion, although I am a bit wary.

Part of the suggestions all are a bit complicated, API-wise and from
the scope. I understand that, if we want more flexibility, we may
need to be able to have a way to add code which requires more characters
and such. But one thing that is great in ruby, even if we say that
"having no namespaces is a disadvantage", is that using modules and
classes on the toplevel space, is very, very simple. People very
quickly understand that concept.

```
class Cat
  def meow
    puts 'The cat meows.'
  end
end
```

With namespaces as suggested here, we may add another layer of

complexity; and while I do agree with some stronger form of control possible over "namespaces" in ruby, I am not sure if the proposal in this form is having a good trade-off. But as I wrote, it's not that I have a big opinion either way - I think the biggest concern I have had in regards to namespaces was when ruby were to use PHP's "solution" and syntax for namespaces ... :P

As for refinements - the odd thing is that I agree behind the proposal and ideas, but the syntax and API is so weird to me and it feels ... strange to use them. I also have no alternative suggestion, so this is not good; best way would be to have both namespaces, namespace scopes and refinements in a single issue with a great, beautiful syntax. :D

(We should however had also consider whether the status quo is actually better than the proposed changes. And to some extent I'd rather use a status quo than want to transition into changes that do not seem to be as worthwhile to be had - even though I actually agree with a LOT on what is said about namespaces, refinements etc...)

As for requiring ruby code, I agree. In particular for larger projects written in ruby, it may be useful to not only have more control, but make managing that ruby code simpler. In your example, the author who wrote the code must have forgotten to require some other files; but I understand that this may be tedious if one has a large project with lots of .rb files. Then there are also circular warnings which are no fun at all.

I am confident that this may improve in the long run - matz always said that ruby is for humans rather than computers and that the core team will listen to (and prioritize on) "real problems" and painpoints people have when writing ruby code. And personally I think that a lot of these problems emerge when one writes a lot of ruby code and has lots of ruby files, too.

#5 - 08/12/2018 09:12 AM - jjyr (Jinyang Jiang)

I believe namespace can reduce the complexity of organizing codes in large projects.

Recently I write rust code in a large project. Our code base is dependent on other several complex projects. With rust use syntax(https://doc.rust-lang.org/book/second-edition/ch07-03-importing-names-with-use.html#bringing-names-into-scope-with-the-use-keyword) we can import the names which we need from other projects without chaotic the global names. It's easy to manage the complexity at the module level.

I can’t image how to handle those complexities under the current ruby requiring system.

I understand the concern, is this a “real problem” or just because we saw the feature in other languages so we want it?

I am currently working on a medium-level ruby project(https://github.com/ciri-ethereum/ciri) and I find it's easy to forget to add some unnecessary requiring.
So in this situation, a user can't directly require this file, He must handle the dependencies manually or require the whole gem.

I do not have real experiences working in millions of lines ruby project, but I believe namespace can help to control complexity, from my other languages experiences.

So I paste my proposal, let the community to discuss the problem whether is real or fake.

I realized the essence of the proposal is to allow user to manipulate Binding. Currently, Ruby only has the TOPLEVEL_BINDING, we can't isolated Binding from top-level.

So the proposal essentially requests several primitive to control Binding.

1 isolated requiring: allows to evaluation required files under an isolated Binding(not polluting TOPLEVEL_BINDING).

```
# requiring into isolated Binding
require ‘foo’, into: :IsolatedBindingModule1
IsolatedBindingModule1.class # Module
IsolatedBindingModule1::Foo # access names

# the old way should still work
# requiring and polluting TOPLEVEL_BINDING
require ‘foo’
```

```
Foo
```

2 namespace: allows users to create an isolated Binding scope.

```
Foo
namespace do # Create an isolated binding.The name maybe not accurate, can be discussed.
    Foo # NameError
end
```

3 import: allow user import names from a Binding into another Binding.

```
Foo
namespace do
  # import name from a ruby file
  # import primitive is a convenient way to use isolated requiring
  import :Foo, from: “foo”
end
```

Then we can extend those methods to more conveniently be used, for example: https://bugs.ruby-lang.org/issues/14982#The-Design

#6 - 08/12/2018 09:36 AM - jjyr (Jinyang Jiang)

- Description updated

#7 - 08/13/2018 01:12 AM - shyouhei (Shyouhei Urabe)

I like this idea in general. I too want to have "requiring into isolated Binding".

One thing I would like to add, "namespace" shall be a keyword rather than a normal method taking a block. Blocks can be passed around:

```
namespace :Foo do
  import :Foo from: 'foo'
  def self.bar
    return lamnda do
      Foo
    end
  end
ebd

namespace(:Bar, &Foo.bar) # => Error, or ...?
```

We should forbid this kind of headache.

#8 - 08/13/2018 03:34 AM - jeremyevans0 (Jeremy Evans)

In the Why? section, case 1 is just a programming error on the library developer's or library user's part, depending on how the library is documented.

In terms of case 2 (multiple gems define overlapping constants in the top level namespace), that could be a issue in theory, but it doesn't generally present a problem in practice as library authors in most cases take care to choose non-overlapping names.

It will probably be challenging to make import work with ruby's constant lookup if the top level namespace is not actually modified. Libraries may expect that their constants names are available in the top level namespace:

```
# foo_a.rb
class Foo
  def self.foo
    ::Foo
    # or Foo
    # or Object.const_get(:Foo)
  end
end

# foo_b.rb
class Foo
  def self.foo
    ::Foo
  end
end

# main
namespace :Bar do
  import :Foo, as: :Foo_A, from: 'foo_a'
  import :Foo, as: :Foo_B, from: 'foo_b'

  Foo_A.foo
```

```
  Foo_B.foo
end
```

I suppose it is possible, but CREF handling in the VM would have to be made significantly more complex to implement it correctly (so that the code works the same both via require and import :as).

You can sort of get what you want in terms of an isolated namespace with BasicObject subclasses:

```
class Foo
end

class Hello < BasicObject
end

class Hello
  Foo # NameError
end

Hello::Foo = ::Foo

class Hello
  Foo # no NameError
end

module A
  module B
    module C
      module D
      end
    end
  end
end

Hello::E = ::A::B::C::D

def Foo.foo; 1; end
Object.send(:remove_const, :Foo)

require 'foo_a'
# in foo_a.rb
# class Foo; end
# def Foo.foo; 2; end
Hello::Foo_A = Object.send(:remove_const, :Foo)

require 'foo_b'
# in foo_b.rb
# class Foo; end
# def Foo.foo; 3; end
Hello::Foo_B = Object.send(:remove_const, :Foo)

class Hello
  class Bar
    def xxx
      [Foo, Foo_A, Foo_B].map{|s| [s, s.foo]}
    end
  end
end

Hello::Bar.new.xxx
# [[Foo, 1], [Foo, 2], [Foo, 3]]
```

Such an approach certainly has it's own issues, though. Also, it doesn't really address the issue of trying to handle overlapping top level constants in separate libraries.

I see the benefits of this proposal, as require into isolated binding is a nice to have assuming everything continues to work. However, I don't think the benefits of this proposal would exceed the implementation and maintenance cost.

#9 - 08/13/2018 06:40 AM - jjyr (Jinyang Jiang)

shyouhei (Shyouhei Urabe) wrote:

> I like this idea in general. I too want to have "requiring into isolated Binding".
>
> One thing I would like to add, "namespace" shall be a keyword rather than a normal method taking a block. Blocks can be passed around:

```
namespace :Foo do
  import :Foo from: 'foo'
  def self.bar
    return lamnda do
      Foo
    end
  end
ebd

namespace(:Bar, &Foo.bar) # => Error, or ...?
```

We should forbid this kind of headache.

Totally agreed!

#10 - 08/13/2018 07:05 AM - jjyr (Jinyang Jiang)

I can imagine how hard to implement the "requiring into isolated binding" correct in VM.

If we want to maintain the compatible we need to maintain the reference of the name cross different require/import.

Think of a situation:

```
# foo.rb
class Foo
  @count = 0
  def self.count
    @count += 1
  end
end

# another_file.rb
namespace do
  import :Foo, from: 'foo'
  Foo.count # should return 1
end

require 'foo', into: :FooSpace
FooSpace::Foo.count # should return 2

# require 'foo'

Foo.count # should return 3
```

The name Foo from path 'foo' should always be the same reference whether how the name imported.
And imaging in case we have path 'foo/bar' and 'foo' require 'foo/bar', the name from 'foo/bar' should also be the same reference across files.

It's mean each time we require or require into a file, we put it in a Binding associated with the file path. (so we can think the requiring in ruby VM is like to find a file and Binding by absolute path)

In VM level, we may also need to implement a nested Binding structure to support this feature: find a name from each required bindings. (like examples above, RubyVm howto lookup the name Foo?)

So it is hard to support this feature, and maybe cause VM performance issue on name lookup.

#11 - 08/16/2018 09:02 PM - jjyr (Jinyang Jiang)

Propose new syntax for https://bugs.ruby-lang.org/issues/14982#note-5 (The core idea is not changing)

```
# add 'isolate' keyword to describe an isolated binding scope
isolate
  # .....
  # can't access unimported names from isolate binding
  import :Foo, from: 'foo'
end

# isolated module

isolate module A
# ....
end

# equivalent to
```

```
isolate
  module A
  end
end

# isolated class

isolate class A
# ....
end

# equivalent to
isolate
  class A
  end
end

# require and import is not changed since previous describe
require 'foo', into: :IsolatedModuleFoo
import :Foo, as :SecondFoo, from 'foo2'
```

#12 - 08/23/2018 07:25 AM - ko1 (Koichi Sasada)

- Target version set to 3.0

- Assignee set to matz (Yukihiro Matsumoto)

#13 - 09/12/2018 07:48 AM - Iloeki (Loic Nageleisen)

Please allow me to humbly present this design and implementation of import semantics aiming to solve precisely the current issue, taking inspiration from Go and Python.

Basically it implements a Package class inheriting Module and leveraging the Kernel#load ability to wrap with its second argument, together with module_eval.
With it you can currently do:

```
import('foo')                    # import package file as `foo`
import('foo/baz')                # import nested package file as `baz`
import('foo', to: :method)       # make available as method `foo` using a side effect (default)
import('foo', to: :const)        # make available as constant `Foo` using a side effect
import('foo', to: :local)
# (attempt to, see below) make available as local variable `foo` using a side effect
import('foo', as: :bar)          # rename the target set as a side effect to `bar`, can be combined with `to:`
f = import('foo', to: :value)    # no side effect, explicit assignment to a local var
Foo = import('foo', to: :value)  # no side effect, explicit assignment to a local const
def foo; import('foo', to: :value); end  # you get the idea
```

More examples are available in the README and in test.rb along with the test fixture tree (those are not unit tests though, apologies for the bad naming on my part).

The various possibilities of to: (nil, :method, :const, :local) are to explore the possible ways to make the module available to the caller.

The implementation works on current Ruby, and would work even better save for a limitation of bind_local_variable_set. It requires the binding_of_caller gem in some situations but it is not a strict requirement (not needed with to: :value).

The package file does not contains a declaration of the Package instance (as class and module do), as it will be deduced from the file name, the goal being to box and isolate automatically the file contents, and not requiring new (IMHO awkward) keywords like isolate. Maybe using a file extension such as .rbp to distinguish .rb files written to be required from those to be imported could be useful, but it is definitely not mandatory. Maybe a package keyword (like in Go) at the start could be useful to guard against the file being required. but again, this is not mandatory. Code leveraging this design also has the advantage to eliminate indentation, repetition, and boilerplate that is typically present in deeply nested ruby files when conventionally matching the file and directory names with the module and class names (please look at the test directory tree in the linked repo at the top to see what I mean). Another advantage is that code reloading becomes quite trivial.

What do you think?

#14 - 11/22/2018 03:04 PM - ciconia (Sharon Rosner)

I'd like to show something I've been working on for the last few months. It's called Modulation, a small (less than 300 LOC) gem providing an alternative way to manage dependencies in Ruby applications. Modulation provides complete isolation of each module (i.e. source file), and enforces *explicit* exporting and importing of constants and methods. Any implementation details may be completely hidden by each module.

Here's a simple example:

greeter.rb:

```
export :greet
```

```
GREETING = 'Hello'

def greet(name)
  puts "#{GREETING}, #{name}!"
end
```

app.rb:

```
require 'modulation'
Greeter = import('./greeter')

Greeter.greet('world')

puts Greeter::GREETING #=> will raise NameError, since GREETING was not exported
puts GREETING #=> will raise NameError, since GREETING was defined inside the Greeter module
```

(Yes, this is a very basic example, there are more examples here: https://github.com/ciconia/modulation/tree/master/examples)

The idea is to load each file in the context of a new Module instance, and expose only the definitions that were explicitly exported. In addition, there's support for default exports, reloading of modules at run-time, and mocking of dependencies for testing purposes.

There's a bunch of patterns and techniques that Modulation makes much easier to implement: singletons, functional code, inversion of control. Hopefully this contributes to the present discussion and maybe others would find it useful.

Source code here: https://github.com/ciconia/modulation

#15 - 11/23/2018 01:38 PM - vo.x (Vit Ondruch)

ciconia (Sharon Rosner) wrote:

> I'd like to show something I've been working on for the last few months. It's called Modulation

Wow, that sounds super useful. This is one of the few things I'd love to see included in StdLib.

Would you mind to show us, how it could help to improve situation with things like Molinillo bundled twice in Ruby, once in RubyGems, second time in Bundler?

#16 - 11/23/2018 02:26 PM - ciconia (Sharon Rosner)

> Would you mind to show us, how it could help to improve situation with things like Molinillo bundled twice in Ruby, once in RubyGems, second time in Bundler?

This is not what Modulation is meant to solve. External dependencies (in the form of gems) are beyond the scope of what Modulation is really about. Although Modulation can be used in gems, and does support importing of gems that use Modulation, it is not a replacement for either Rubygems or Bundler.

I'd like to spend a few moments and explain a bit more about how Modulation works and why it's a good solution for managing dependencies. Modulation defines the Kernel#import method for loading dependencies. When import is called, Modulation creates an *anonymous* module and loads the given source file using #instance_eval. Once the source file has been loaded, the same associated module is returned to any code that imports the same source file. Thus, all files referring to the same dependency are in effect using the same Module instance for accessing that dependency. This allows some interesting uses such as reloading of modules at run-time, mocking of dependencies, or dependency injection.

#17 - 11/24/2018 05:04 PM - vo.x (Vit Ondruch)

ciconia (Sharon Rosner) wrote:

> > Would you mind to show us, how it could help to improve situation with things like Molinillo bundled twice in Ruby, once in RubyGems, second time in Bundler?
>
> This is not what Modulation is meant to solve. External dependencies (in the form of gems) are beyond the scope of what Modulation is really about.

Molinillo is not external dependency:

https://github.com/ruby/ruby/tree/trunk/lib/bundler/vendor/molianillo/lib/molinillo
https://github.com/ruby/ruby/tree/trunk/lib/rubygems/resolver/molinillo/lib/molinillo

As you can see, it is bundled twice. The problem is that although there should be one copy of Molinillo, there are two copies, which artificially differs in

namespace, to avoid collision if there was Molinillo gem installed.

#18 - 11/24/2018 08:35 PM - Eregon (Benoit Daloze)

IMHO, the case of Molinillo is the job of RubyGems/Bundler developers to solve: settle on one version, and reuse RubyGems' vendored copy for Bundler.
Or is there any hidden problem in there?

#19 - 11/24/2018 09:05 PM - vo.x (Vit Ondruch)

Eregon (Benoit Daloze) wrote:

> Or is there any hidden problem in there?

Yes, support of older RubyGems I suppose.

#20 - 11/24/2018 09:09 PM - vo.x (Vit Ondruch)

Not mentioning that Bundler bundles other libraries, such as net-http-persistent and Thor. It would make update of this libraries easier, e.g. they could be submodule in code base, because they would not need modified namespace.

#21 - 11/24/2018 09:40 PM - ciconia (Sharon Rosner)

> As you can see, it is bundled twice. The problem is that although there should be one copy of Molinillo, there are two copies, which artificially differs in namespace, to avoid collision if there was Molinillo gem installed.

In that case, yes, a tool like Modulation could be used to completely isolate private copies of Molinillo, without contaminating the global namespace.

#22 - 03/23/2019 01:10 AM - chocolateboy (Chocolate Boy)

I like this idea (and the other suggestions and implementations). I proposed something similar for Crystal a while back here.