Ruby master - Feature #10320

require into module

10/02/2014 10:44 PM - sowieso (So Wieso)

Status: Open
Priority: Normal
Assignee:
Target version:

Description

When requiring a library, global namespace always gets polluted, at least with one module name. So when requiring a gem with many dependencies, at least one constant enters global namespace per dependency, which can easily get out of hand (especially when gems are not enclosed in a module).

Would it be possible to extend require (and load, require_relative) to put all content into a custom module and not into global namespace?

Syntax ideas:

```ruby
require 'libfile', into: :Lib  # keyword-argument
require 'libfile' in Lib
# with keyword, also defining a module Lib at current binding (unless defined? Lib)
require_qualified 'libfile', :Lib
```

This would also make including code into libraries much easier, as it is well scoped.

```ruby
module MyGem
  require 'needed' in Need

  def do_something
    Need::important.process!
  end
end
end
# library user is never concerned over needed's content
```

Some problems to discuss:

- requiring into two different modules means loading the file twice?
- monkeypatching libraries should only affect the module → auto refinements?
- maybe also allow a binding as argument, not only a module?
- privately require, so that required constants and methods are not accessible from the outside of a module (seems to difficult)
- what about $global constants, read them from global scope but copy-write them only to local scope?

Similar issue:

https://bugs.ruby-lang.org/issues/5643

Related issues:

- Related to Ruby master - Feature #5643: require/load options and binding option
- Related to Ruby master - Feature #13847: Gem activated problem for default gems

History

#1 - 10/03/2014 04:48 PM - nobu (Nobuyoshi Nakada)
- Description updated

So Wieso wrote:

```ruby
require 'libfile', into: :Lib  # keyword-argument
require 'libfile' in Lib
# with keyword, also defining a module Lib at current binding (unless defined? Lib)
require_qualified 'libfile', :Lib
```

Why the first and the last use a symbol?
The second will be difficult as require is not a reserved word but a mere method call.

- maybe also allow a binding as argument, not only a module?

Does it make sense?

#2 - 10/03/2014 04:48 PM - nobu (Nobuyoshi Nakada)
- Related to Feature #5643: require/load options and binding option added

#3 - 10/06/2014 12:49 PM - sowieso (So Wieso)
I chose the symbol :Lib, as I thought Ruby would complain if the constant Lib would not exist at this time. The keyword in would define it, if it would not exist. I would prefer if we could solve it without using symbols, but writing module Lib; end before the first require doesn't look nice.

Sorry, I didn't consider that require is a method, so I guess the keyword option (in) doesn't fit. (Alternatively we could define suffix in as enclosing the given module:

```ruby
require 'file'
in Lib
```

#is equivalent
module Lib
  require 'file'
end

but then require has to check for its nesting.
}

#4 - 10/06/2014 01:46 PM - mikegee (Michael Gee)
So Wieso wrote:

```ruby
require 'file' in Lib
# is equivalent
module Lib
  require 'file'
end
```

I don't like changing require with one argument to mean something else. It would break too much legacy code.

If you want to require a feature into your current namespace how about this:

```ruby
module Lib
  require 'lib', in: self
end
```

#5 - 10/06/2014 02:59 PM - nobu (Nobuyoshi Nakada)
So Wieso wrote:

I chose the symbol :Lib, as I thought Ruby would complain if the constant Lib would not exist at this time. The keyword in would define it, if it would not exist. I would prefer if we could solve it without using symbols, but writing module Lib; end before the first require doesn't look nice.

It's ambiguous if Lib is a module or a class, when only the name is provided.

#6 - 10/07/2014 12:27 PM - sowieso (So Wieso)
Michael Gee wrote:

I don't like changing require with one argument to mean something else. It would break too much legacy code.

You are definitely right here, we should not do that.

Nobuyoshi Nakada wrote:

It's ambiguous if Lib is a module or a class, when only the name is provided.

Is it? If the constant is already defined, take it (class or module). If not create a new module by this name.

#7 - 10/07/2014 01:50 PM - nobu (Nobuyoshi Nakada)
So Wieso wrote:

Nobuyoshi Nakada wrote:

It's ambiguous if Lib is a module or a class, when only the name is provided.

Is it? If the constant is already defined, take it (class or module). If not create a new module by this name.

If you don't want to create a module by that name, you don't need to use the name. Should not introduce implicit conversion between a name and a module. You can use anonymous module too if is is a module object.

I think this feature should be an instance method of Module, similar to load rather than require.

#8 - 01/20/2016 05:28 PM - jwmittag (Jörg W Mittag)

Nobuyoshi Nakada wrote:

I think this feature should be an instance method of Module, similar to load rather than require.

Yes, I believe having Module#load and possibly Module#require and Module#require_relative would be the most logical:

```ruby
# not namespaced:
require 'foo'
module Bar; end

# namespaced:
module Bar
  require 'foo'
end
```

Re-using those names would be a backwards-incompatible change, though. I have seen people use the second form sometimes.

This is a replacement for the hard-to-use wrap optional argument to Kernel#load:

```ruby
load 'foo', true

# is almost equivalent to
Module.new.load 'foo'
```

This can be naturally extended to Binding#load, Binding#require, and Binding#require_relative.

As with the wrap argument to Kernel#load, the question is, what should references like ::String in the loaded script refer to? I think it would be nice if it offered complete isolation by default, with an option to revert to the current behavior of load, e.g. with an API like this:

```ruby
class Module
  def load(path, pollute_global: false)
  end
end
```

#9 - 08/29/2017 08:57 AM - hsbt (Hiroshi SHIBATA)

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

#10 - 06/10/2021 05:51 AM - jaesharp (J Lynn)

I'd like to note that there exists a gem called modules (https://rubygems.org/gems/modules) which uses Kernel#load with the wrap=true option in order to implement a module import/export resolution system similar in nature to the one described here and to node.js's commonjs module system semantics. Perhaps this is sufficient to meet people's needs, if brought in?

#11 - 06/10/2021 06:18 AM - texpert (Aurel Branzeanu)

jaesharp (J Lynn) wrote in #note-10:

I'd like to note that there exists a gem called modules (https://rubygems.org/gems/modules) which uses Kernel#load with the wrap=true option in order to implement a module import/export resolution system similar in nature to the one described here and to node.js's commonjs module system semantics. Perhaps this is sufficient to meet people's needs, if brought in?

And another similar gem is modulation (https://rubygems.org/gems/modulation/versions/0.25)
texpert (Aurel Branzeanu) wrote in #note-11:

And another similar gem is modulation (https://rubygems.org/gems/modulation/versions/0.25)

It looks much more developed and actively maintained! Thank you! :)