Ruby - Feature #16848

Allow callables in $LOAD_PATH

05/11/2020 11:21 AM - byroot (Jean Boussier)

Status:	Feedback
Priority:	Normal
Assignee:	
Target version:	

Description

Make it easier to implement $LOAD_PATH caching, and speed up application boot time.

I benchmarked it on Redmine's master using bootsnap with only the optimization enabled:

```
if ENV['CACHE_LOAD_PATH']
  require 'bootsnap'
  Bootsnap.setup(
    cache_dir:            'tmp/cache',
    development_mode:     false,
    load_path_cache:      true,
    autoload_paths_cache: true,
    disable_trace:        false,
    compile_cache_iseq:   true,
    compile_cache_yaml:   false,
  )
end

$ RAILS_ENV=production time bin/rails runner 'p 1'
        2.66 real         1.99 user         0.66 sys
$ RAILS_ENV=production time bin/rails runner 'p 1'
        2.71 real         1.97 user         0.66 sys
$ CACHE_LOAD_PATH=1 RAILS_ENV=production time bin/rails runner 'p 1'
        1.41 real         1.12 user         0.28 sys
$ CACHE_LOAD_PATH=1 RAILS_ENV=production time bin/rails runner 'p 1'
        1.41 real         1.12 user         0.28 sys
```

That's twice for a relatively small application. And the performance improvement is not linear; the larger the application, the larger the improvement.

How it works

require has O($LOAD_PATH.size) performance. The more gems you add to your Gemfile, the larger $LOAD_PATH becomes. require "foo.rb" will try to open the file in each of the $LOAD_PATH entries. And since more gems usually also means more require calls, loading Ruby code may take up to quadratic performance loss.

To improve this, Bootsnap pre-computes a map of all the files in your $LOAD_PATH, and uses it to convert relative paths into absolute paths so that Ruby skips the $LOAD_PATH traversal.

```
$LOAD_PATH = $w(/gems/foo/lib /gems/bar/lib)

BOOTSNAP_CACHE = {
  "bar.rb" => "/gems/bar/lib/bar.rb",
}
```

This resolves file lookup by a single hash lookup, and reduces boot performance from roughly O($LOAD_PATH.size * number_of_files_to_require) to O(number_of_files_to_require).

This optimization is also used in Gel, a Rubygems/Bundler replacement.

Trade offs

Every time $LOAD_PATH is modified, the cache must become invalidated. While this is complex to do for Bootsnap, it would be fairly easy if it is implemented inside Ruby.

More importantly, you have to invalidate the cache whenever you add or delete a file to/from one of the $LOAD_PATH members; otherwise, if you shadow or unshadow another file farther in the $LOAD_PATH, Bootsnap will load a wrong file. For instance, if require "foo.rb" initially resolves to /some/gem/foo.rb, and you create lib/foo.rb, you'll need to flush Bootsnap cache.

That latter is trickier, and Bootsnap has decided that it is rare enough to cause actual problems, and so far that holds. But that is not a trade off Ruby can make.

However that's probably a tradeoff Rubygems/Bundler can make. While it's common to edit your gems to debug something, it's really uncommon to add or remove files inside them. So in theory Rubygems/Bundler could compute a map of all files in a gem that can be required after it installs it. Then when you activate it, you merge it together with the other activated gems.

Proposal

This could be reasonably easy to implement if $LOAD_PATH accepted callables in addition to paths. Something like this:

```
$LOAD_PATH = [
  'my_app/lib',
  BundlerOrRubygems.method(:lookup),
]
```

The contract would be that BundlerOrRubygems.lookup("some_relative/path.rb") would return either an absolute path or nil. With such API, it would be easy to cache absolute paths only for gems and the stdlib, and preserve the current cache-less behavior for the application specific load paths, which are usually much less numerous. It would also allow frameworks such as Rails to implement the same caching for application paths when running in an environment
where the source files are immutable (typically production).

History

#1 - 05/11/2020 11:26 AM - byroot (Jean Boussier)

I just realized I made a mistake in my benchmark, I also both instruction sequence and load path caching enabled.

With instruction sequence disabled, the speedup is only 17% for Redmine:

```
$ RAILS_ENV=production time bin/rails runner 'p 1'
        2.64 real         1.97 user         0.66 sys
$ RAILS_ENV=production time bin/rails runner 'p 1'
        2.64 real         1.97 user         0.66 sys
$ CACHE_LOAD_PATH=1 RAILS_ENV=production time bin/rails runner 'p 1'
        2.18 real         1.83 user         0.34 sys
$ CACHE_LOAD_PATH=1 RAILS_ENV=production time bin/rails runner 'p 1'
        2.18 real         1.83 user         0.34 sys
```

However the 50% or more absolutely happens on larger applications.

#2 - 05/11/2020 01:56 PM - Eregon (Benoit Daloze)

> The other, more important difficulty, is that you also have to invalidate the cache whenever you add or delete a file in one of the $LOAD_PATH members, otherwise if you shadow or unshadow another file that is farther in the $LOAD_PATH, Bootsnap will load the wrong file.
> For instance if require "foo.rb" used to resolve to /some/gem/foo.rb, but you just created lib/foo.rb, you'll need to flush Bootsnap cache.

Actually, due to the caching of $LOADED_FEATURES, it won't ever load lib/foo.rb for require "foo".
The $LOADED_FEATURES cache keeps a map of feature to $LOAD_PATH index/indices.

#3 - 05/11/2020 01:58 PM - Eregon (Benoit Daloze)

There is a now a spec for that as we replicated that caching in TruffleRuby:
https://github.com/oracle/truffleruby/commit/30024ed8c8f0900cf03bdc7fdf3fa7b4776837ac

#4 - 05/11/2020 02:50 PM - byroot (Jean Boussier)

@Eregon (Benoit Daloze) only if you did actually require it. My explanation is probably a bit confusing.

```
$LOAD_PATH = [
  "stdlib/"
]
Bootsnap.setup
# Bootsnap see `delegate.rb` in the stdlib, so cache as `{"delegate.rb" => "/stdlib/delegate.rb"}`
```

```
$LOAD_PATH.unshift("./lib/")
require "delegate.rb" # if ./lib/ also has a `delegate.rb` it should win
```

And that behavior as well had to be reimplemented in Ruby to preserve compatibility. That's a lot of complex and not so fast code that could entirely be eliminated if Ruby provided the proper hooks.

#5 - 05/11/2020 06:27 PM - Eregon (Benoit Daloze)

Right.

I always thought RubyGems should be smarter and when requiring foo/bar.rb it could look for a gem named foo first, and not look in all gems if the file exist.
This feature could help achieve some of that, instead of going through a long $LOAD_PATH it could use the gem name (or some index) to find where the file lives without searching multiple directories.

#6 - 05/11/2020 06:42 PM - byroot (Jean Boussier)

Also alternatively the callable could be responsible for actually loading the code rather than return an absolute path.

This would allow for more flexibility in implementing alternative code loaders, e.g. read the source from an archive, or any other thing I'm not thinking about just yet.

#7 - 05/12/2020 02:52 AM - nobu (Nobuyoshi Nakada)

- Description updated

byroot (Jean Boussier) wrote:

> ## Proposal
>
> This could be reasonably easy to implement if $LOAD_PATH accepted callables in addition to paths. Something like this:
>
> ```
> $LOAD_PATH = [
>   'my_app/lib',
>   BundlerOrRubygems.method(:lookup),
> ]
> ```

This seems similar to my proposal in several years ago, to allow loading from archived libraries.
It was not callable but calling a particular method, I don't remember accurately now though.

#8 - 05/12/2020 10:06 AM - byroot (Jean Boussier)

@nobu (Nobuyoshi Nakada) indeed. I initially only focused on solving the performance problem of having a large $LOAD_PATH, but maybe it's the occasion to also allow for alternative storage mechanisms.

In such case it can be interesting to look at how Python does it: https://www.python.org/dev/peps/pep-0302/, In short they have two extension interfaces, find_module and load_module.

I suppose Kernel.require can check for two distinct methods as well.

#9 - 05/12/2020 02:15 PM - mame (Yusuke Endoh)

As far as my understanding, this is a self-contained simple example to demonstrate the proposal (to make dev-meeting discussion easy)

```
gem_path_resolver = -> feature do
  case feature
  when "bar" then "/gems/bar/bar.rb"
  when "baz" then "/gems/baz/baz.rb"
  else
    nil
  end
end

$LOAD_PATH = [
  "my_app/",  # there is my_app/foo.rb
  gem_path_resolver,
  "/stdlib/", # there is /stdlib/qux.rb
]

require "foo" # load my_app/foo.rb
require "bar" # load /gems/bar/bar.rb
require "baz" # load /gems/baz/baz.rb
require "qux" # load /stdlib/qux.rb
```

In practical, gem_path_resolver should respect Gemfile or something.

Let me know if I am wrong.

My comment: it seems to make sense, but I'm interested in opinion from rubygems/bundler developers.

#10 - 05/12/2020 06:23 PM - byroot (Jean Boussier)

@mame (Yusuke Endoh) yes, your example fits my proposal.

But note that I'm also open to other solutions, as long as it allows to resolve paths of the $LOAD_PATH.

#11 - 05/12/2020 06:30 PM - ko1 (Koichi Sasada)

Just curious:
How to implement the gem_path_resolver? gem_path_resolver should know the set of existing files in the gem install directories.
What happens if a new file is created in a gem directory after installation? I understand we can ignore such a corner case.

#12 - 05/12/2020 06:46 PM - byroot (Jean Boussier)

> How to implement the gem_path_resolver?

By simply walking down the directories in the loadpath.

In the context of Bundler/Rubygems it could directly be done post install, in the case of Bootsnap it's done on the fly on cache miss.

> What happens if a new file is created in a gem directory after installation? I understand we can ignore such a corner case.

Yes, Bootsnap/Bootscale/Gel all run with the assumptions that installed gems are immutable (or more exactly that no path is added or removed).

And while it's common to edit a source file from an installed gem to put a breakpoint or something, it's extremely uncommon to add or remove a file from and installed gem. After 5 years of using that technique, I never heard of it causing trouble for someone, at least not inside gems.

However that is a concern for the paths owned by the application itself since they're much more frequently edited. But because of the APIs currently offered by Ruby, Bootsnap & co have no choice but to cache the integrality of the load path. The new API I propose would allow Bootsnap/Gel/Rubygems/Bundler to only care about gems, definitely eliminating this issue.

And if really necessary, a command akin to gem pristine could be provided to rebuild the gem index.

#13 - 05/19/2020 03:40 PM - Dan0042 (Daniel DeLorme)

I really love this idea; it's simple yet powerful. I do have a few questions though.

1. How would this API handle .rb vs .so loading? IIRC there was some additional logic to ensure that a .rb file later in the $LOAD_PATH would win over an earlier .so file. If resolver.call("foo.rb") returns "/path/to/foo.rb", should resolver.call("foo") do the same thing or return nil? In other words should the "omitted extension" handling be done inside or outside of the resolver?

2. How would this work with the polyglot gem? Let's say you have a gem that registers the "rgl" extension and also includes several .rgl files in its lib directory; you may want resolver.call("bar") to find /path/to/bar.rgl, or at the very least you'd want resolver.call("bar.rgl") to find the file without trying to find bar.rgl.{rb,so} instead. This implies the resolver must index all files, not just the known extensions. Or provide a mechanism for adding file extensions to index.

#14 - 05/19/2020 06:56 PM - sawa (Tsuyoshi Sawada)

- Description updated

#15 - 05/19/2020 08:15 PM - byroot (Jean Boussier)

> In other words should the "omitted extension" handling be done inside or outside of the resolver?

In my opinion it should be done inside, because otherwise it would hinder performance. But yes it would potentially change the precedence in some very rare cases.

But note that the idea being to use that new api in Bundler/Rubygems, and ideally the stdlib as well. So the callables would be last which would render the behavior change extremely unlikely.

> How would this work with the polyglot gem?

If that's a use case we want supported, then what we discussed @nobu (Nobuyoshi Nakada) is probably preferable. e.g. check for two distinct methods rather than a single call. Or even consider than .call should load the code by contract, but it's trickier as you need to do the right thing to avoid double loading etc.

#16 - 05/21/2020 04:16 PM - ko1 (Koichi Sasada)

One concern we found is some scripts expect $: contains String objects which represent a path.
Typically some scripts (gems) find a file in a $: ($:.each{|path| ...}). It will be a compatibility issue.
Maybe we need a trick to keep compatibility.

For example, accessing $: returns an array of String, and searcher (BundlerOrRubygems.method(:lookup) for example) expands to String objects.
In this case, searcher is not only callable object, but an object which implements some methods like to_a (returns expanded path array) and find(feature).

#17 - 05/26/2020 07:58 AM - ko1 (Koichi Sasada)

- Status changed from Open to Feedback

As I described, the current approach has a compatibility issue.
Could you propose another proposal to care compatibility?

In discussion at devmeeting, there is possibility to introduce a feature to help $LOAD_PATH observing which bootsnap does.

Also Matz said he prefers to introduce Python's loader/finder mechanism.
To introduce them, maybe deep discussion is needed.

#18 - 05/26/2020 09:55 AM - byroot (Jean Boussier)

Sorry I missed your answer from 5 days ago. I wish I'd answered before your second meeting (Redmine no longer send notifications...).

> Could you propose another proposal to care compatibility?

To be honest I kinda expected this concern, but hoped it wouldn't be one.

What Python went through

That exact same problem was posed to Python when they did this https://www.python.org/dev/peps/pep-0302/#rationale (the whole section is worth a read)

It was proposed to make these Loaders quacks like strings to offer backward compatibility, but they decided to go with a fairly complex combination of sys.path_hooks, sys.meta_path and sys.path_importer_cache alongside sys.path, which I have to admit I don't really like.
It is hard to understand the flow, requires to keep yet another cache of which loader can load what, and it is hard to tightly control the load order between custom loaders and regular paths.

String duck-typing

So my personal preference would be to make these Finders/Loader delegate to a string on method missing, with maybe a deprecation warning.
That's very easy to implement and would keep backward compatibility in the vast majority of the cases.

It also keep the main benefit which is easy and simple control over the load order.

$LOAD_PATH as a "view"

But if it's considered too dirty, then a more complex proposal could be to make $LOAD_PATH a simple "view" of $REAL_LOAD_PATH (name to be defined, ideally it would no longer be a global).

Its value when accessed would be $LOAD_PATH.select(String). It would also continue to support mutations, e.g. $LOAD_PATH.unshift('/some/path') can easily be forwarded to $REAL_LOAD_PATH.

Looking at $LOAD_PATH.public_methods I see #insert that might be non trivial to delegate, as the position would need to be translated, but at this stage I don't personally foresee an impossibility.

@ko1 (Koichi Sasada) at first glance, do any of these two solutions seem acceptable to you? If so I can make a more formal specification of the one of your choosing, and even try to implement it, or part of it as a proof of concept.

> In discussion at devmeeting, there is possibility to introduce a feature to help $LOAD_PATH observing which bootsnap does.

It would be nice as it would remove some fairly dirty and inefficient code from Bootsnap, but it kind of is a "solved" problem. That's no longer what is challenging in Bootsnap.

> Also Matz said he prefers to introduce Python's loader/finder mechanism.

That makes sense. If a change is to be made, might as well go all the way.

> To introduce them, maybe deep discussion is needed.

I suppose so. I obviously can't take API decisions, but know that I'm serious about this proposal and I'm willing to put as much work as needed into this.

#19 - 05/26/2020 01:12 PM - Dan0042 (Daniel DeLorme)

String duck-typing

It seems to me that apart from adding a path to $LOAD_PATH (which is a non-issue), the "vast majority of the cases" involve something like $LOAD_PATH.include?(path) and string duck-typing doesn't help with that.

$LOAD_PATH as a "view"

That's quite a nice, clean approach. $LOAD_PATH could also be a special Enumerable object that behaves like an array when adding elements but expands the Finder objects in #each. In gems I also see a fair amount of code like $LOAD_PATH.reject! / uniq! which might be tricky to handle for Finder objects.

#20 - 05/26/2020 02:01 PM - ko1 (Koichi Sasada)

> $LOAD_PATH as a "view"

This is what I proposed at comment #16.

Maybe there are two strategies:

(1) extend $LOAD_PATH.
(2) introduce $LOAD_PATH2 and deprecate $LOAD_PATH.
(3) other great idea?

Several years ago I tried this topic when I made an ISeq#to_bianry, but I gave up because of complexity.
It is good time to have a nice approach.

FYI: ruby-core:46896 new `require' framework (Half-baked DRAFT)

#21 - 05/26/2020 02:30 PM - byroot (Jean Boussier)

> the "vast majority of the cases" involve something like $LOAD_PATH.include?(path) and string duck-typing doesn't help with that.

@Dan0042 not sure what you mean by that. Like right now you can $LOAD_PATH.unshift(Object.new) and it won't break $LOAD_PATH.include?(path).

What I worry more about is things like $LOAD_PATH.any? { |p| p.start_with?("/blah") } or similar. String duck-typing helps with that.

What it might not help with is path operations, e.g. $LOAD_PATH.map! { |s| File.expand_path(s) }.

> This is what I proposed at comment 16

Ho, I see now. Sorry I initially missed it.

> extend $LOAD_PATH

Not sure what you mean by that. What would be the extension?

> introduce $LOAD_PATH2 and deprecate $LOAD_PATH.

Seems easy to me. Any method call on $LOAD_PATH trigger a deprecation telling to use $NEW_LOAD_PATH instead. If we managed to provide a decent $LOAD_PATH backward compatibility, it doesn't need a long deprecation cycle, users can test for the new load path existence, and do what they have to conditionally.

> FYI: ruby-core:46896 new `require' framework

I'm afraid that's a dead link.

#22 - 05/27/2020 04:16 PM - Dan0042 (Daniel DeLorme)

> not sure what you mean by that. Like right now you can $LOAD_PATH.unshift(Object.new) and it won't break $LOAD_PATH.include?(path).

If you look in existing gems you'll find lots of stuff like $LOAD_PATH.unshift(dir) unless $LOAD_PATH.include?(dir) which assumes that the gems' lib dirs are present in the $LOAD_PATH. So if you check for $LOAD_PATH.include?("/path/to/gems/foobar-1.2.3/lib") this would always return false even if the gem is actually present *via the finder/callable object*.

> (1) extend $LOAD_PATH.
> (2) introduce $LOAD_PATH2 and deprecate $LOAD_PATH.

(1) seems the best option to me. It allows a maximum of compatibility while introducing new functionality. The finder object would not be a simple callable (because it needs to expose the paths within) but that sounds good to me; a plain callable is starting to look too simple to support various existing/desirable functionality.

#23 - 05/27/2020 05:00 PM - byroot (Jean Boussier)

> $LOAD_PATH.unshift(dir) unless $LOAD_PATH.include?(dir)

That specific case would degrade properly. Assuming the include? returns false because that path is now handled by Rubygems/Bundler, it simply mean you'll disable caching for that specific path. That shouldn't break anything.

#24 - 05/27/2020 11:23 PM - Eregon (Benoit Daloze)

byroot (Jean Boussier) wrote in #note-21:

> I'm afraid that's a dead link.

I fixed the link on Redmine.

#25 - 06/06/2020 11:39 AM - shevegen (Robert A. Heiler)

This was recently discussed:

https://github.com/ruby/dev-meeting-log/blob/master/DevelopersMeeting20200514Japan.md

ko1 will reply.

I have no particular opinion per se on this suggestion. But one thing made me curious:

```
require "foo" # load my_app/foo.rb
require "bar" # load /gems/bar/bar.rb
require "baz" # load /gems/baz/baz.rb
require "qux" # load /stdlib/qux.rb
```

This would be quite nice in that we do not have to provide a specific path to a .rb file. This could also help if you relocate files, from one path to another - if you specify the hardcoded path, if that is changed, you have to modify all links that point to that .rb file.

My totally unfinished idea is to propose being able to omit all require statements. :)

Perhaps only make it limited to gem releases where the ruby user no longer has to supply a full path, and then update it if it changes. Something like an API for requires that allows us to be unspecific.

I understand that this is not what byroot suggested, so I should make a separate issue suggestion. But I wanted to mention it because it sort of taps into a related idea - make it easier to handle require of files.

This is something I have to do almost daily when I write ruby code - I have to find where code is, then load that code and make sure all require lines are correct. This is a bit cumbersome.

#26 - 06/11/2020 02:00 PM - byroot (Jean Boussier)

In order to try to move this forward, here's some code snippet of what I think the interface could look like:

```
class LoaderInterface
  def find_feature(relative_path)
    raise NotImplementedError
  end

  def load_feature(relative_path, absolute_path)
    # This should likely be the default behavior.
    # But defining load also allows to generate some source code and call `eval`.
    ::Kernel.load(absolute_path)
  end
end

class CacheLoader < LoaderInterface
  def initialize(cache)
    @cache = cache
  end

  def find_feature(relative_path)
    @cache[relative_path] # absolute_path or nil
  end
end

class ZipLoader < LoaderInterface
  def initialize(zip_path)
    @zip_path = zip_path
    @zip = Zip.open(zip_path)
  end

  def find_feature(relative_path)
    if @zip.file?(relative_path)
      "#{zip_path}:#{relative_path}"
    end
  end

  def load_feature(relative_path, absolute_path)
    ::Kernel.eval(@zip.read(relative_path), relative_path, absolute_path)
  end
end

LOAD_PATH = [
  '/path/to/lib',
  CacheLoader.new('delegate.rb' => '/opt/rubies/2.7.1/lib/ruby/2.7.0/delegate.rb'),
  ZipLoader.new('/path/to/file.zip'),
]

LOADED_FEATURES = []

def require_file(relative_path)
  $LOAD_PATH.each do |path|
    case path
    when String
      # old behavior
      feature_path = File.join(path, feature_path)
      if File.exist?(feature_path)
        ::Kernel.load(feature_path)
        LOADED_FEATURES << feature_path
        return true
      end
    else
      if feature_path = path.find_feature(relative_path)
        path.load_feature(relative_path, feature_path)
        # The loader doesn't have to care about LOADED_FEATURES management
        # if the feature was already loaded it simply won't be called.
```

```
        LOADED_FEATURES << feature_path
        return true
      end
    end
  end
  false
end

def require(feature)
  # Insert LOADED_FEATURES check here.
  if feature.end_with?('.rb')
    require_file(feature)
  else
    require_file("#{feature}.rb") || require_file("#{feature}.so")
  end
end
```

In short:

- In the context of require "foo/bar"
- First we call find_feature("foo/bar.rb"), loaders should return nil on miss, or a String that will be inserted in $LOADED_FEATURES on hit. e.g. /absolute/path/to/foo/bar.rb
- If a loader hits, we then call load_feature("foo/bar.rb", "/absolute/path/to/foo/bar.rb").
- MAYBE: if the loader doesn't respond to load_feature, we can fallback to load the absolute path that was returned.
- If none of the loader hit, we do a second pass with find_feature("foo/bar.so") (or the platform equivalent).

Important parts:

- I think the $LOADED_FEATURES management should remain a responsibility of Kernel#require. Loaders shouldn't have to worry about it.
- The double pass is a bit unfortunate, but required to maintain backward compatibility with the current behavior on LOAD_PATH precedence.

Any thoughts?

#27 - 06/26/2020 02:15 PM - Dan0042 (Daniel DeLorme)

That's interesting...
My understanding was that the "find" and "load" operations were supposed to be orthogonal, so that we could have e.g.

1. find in:
   - directory
   - gems (cache)
   - zip file
2. load:
   - .rb
   - .so
   - .rbc (compiled ruby)

My rough idea of how that could work is something like

```
$LOAD_PATH.loaders[".rb"] = ->(filename, io_open) do
  if io_open.nil?
    ::Kernel.load(filename)
  else
    io_open.call(filename) do |io| #e.g. from zip file
      ::Kernel.eval(io.read, nil, filename)
    end
  end
end

def $LOAD_PATH.find_by_base_name(relative_name)
  extglob = "{" + $LOAD_PATH.loaders.keys.join(",") + "}" # for simplicity pretend globbing is ok
  so = nil
  $LOAD_PATH.each do |finder|
    case finder
    when String # old behavior
      founds = Dir.glob(relative_name + extglob, base: finder)
    else
      founds = finder.find_file(relative_name, extglob)
    end
    founds&.each do |filename, io_open|
      return [filename, io_open] unless filename.end_with?(".so")
      so ||= [filename, io_open]
    end
  end
  return so
```

```
end

def require(relative_name)
  found = $LOAD_PATH.find_by_exact_name(relative_name) if $LOAD_PATH.loaders[File.extname(relative_name)]
  found ||= $LOAD_PATH.find_by_base_name(relative_name)
  found or raise LoadError
  filename, io_open = *found
  loader = $LOAD_PATH.loaders[File.extname(filename)]
  LOADED_FEATURES << filename
  loader.call(filename, io_open)
  return true
end
```