Ruby master - Feature #5653

"I strongly discourage the use of autoload in any standard libraries" (Re: autoload will be dead)

11/21/2011 05:24 PM - matz (Yukihiro Matsumoto)

Status:	Closed
Priority:	Normal
Assignee:	matz (Yukihiro Matsumoto)
Target version:	Next Major

Description

Hi,

Today, I talked with NaHi about enhancing const_missing to enable autoload-like feature with nested modules. But autoload itself has fundamental flaw under multi-thread environment. I should have remove autoload when I added threads to the language (threads came a few months after autoload).

So I hereby declare the future deprecation of autoload. Ruby will keep autoload for a while, since 2.0 should keep compatibility to 1.9. But you don't expect it will survive further future, e.g. 3.0.

I strongly discourage the use of autoload in any standard libraries.

matz.

Related issues:

Related to Ruby master - Bug #11277: "code converter not found" error with mu...	**Closed**	
Related to Ruby master - Feature #7835: autoload will be dead	**Rejected**	**02/12/2013**
Related to Ruby master - Feature #15592: mode where "autoload" behaves like a...	**Open**	

History

#1 - 11/21/2011 05:28 PM - nahi (Hiroshi Nakamura)

- Subject changed from autoload will be dead to "I strongly discourage the use of autoload in any standard libraries" (Re: autoload will be dead)

- Category set to lib

- Target version set to 2.0.0

This ticket is for discussion about removing autoload from stdlib (or not)

% grep autoload {ext/,}lib/ | wc -l
442

#2 - 11/22/2011 04:53 AM - Anonymous

On Mon, Nov 21, 2011 at 05:28:25PM +0900, Hiroshi Nakamura wrote:

> Issue #5653 has been updated by Hiroshi Nakamura.
>
> Subject changed from autoload will be dead to "I strongly discourage the use of autoload in any standard libraries" (Re: autoload will be dead)
> Category set to lib
> Target version set to 2.0.0
>
> This ticket is for discussion about removing autoload from stdlib (or not)
>
> % grep autoload {ext/,}lib/ | wc -l
> 442

Looks like tk has most of them:

[aaron@higgins ruby (trunk)]$ git grep autoload ext | grep 'ext/tk' | wc -l
417
[aaron@higgins ruby (trunk)]$

I removed them from psych, and I'll do the same with dl.

--
Aaron Patterson
http://tenderlovemaking.com/

#3 - 11/22/2011 09:57 AM - jrochkind (jonathan rochkind)

My understanding was that plain old 'require' had much the same flaw in a multi-threaded environment as autoload. No?

#4 - 11/22/2011 10:29 AM - matz (Yukihiro Matsumoto)

Hi,

In message "Re: [ruby-core:41183] [ruby-trunk - Feature #5653] "I strongly discourage the use of autoload in any standard libraries" (Re: autoload will be dead)"
on Tue, 22 Nov 2011 09:57:54 +0900, jonathan rochkind jonathan@dnil.net writes:

|My understanding was that plain old 'require' had much the same flaw in a multi-threaded environment as autoload. No?

Since calling #require is explicit, there's plenty of chance to manage
them. Introducing single lock for #require loading is another issue.

```
matz.
```

#5 - 11/22/2011 02:23 PM - drbrain (Eric Hodel)

On Nov 21, 2011, at 11:51 AM, Aaron Patterson wrote:

> On Mon, Nov 21, 2011 at 05:28:25PM +0900, Hiroshi Nakamura wrote:
>
> > Issue #5653 has been updated by Hiroshi Nakamura.
> >
> > Subject changed from autoload will be dead to "I strongly discourage the use of autoload in any standard libraries" (Re: autoload will be dead)
> > Category set to lib
> > Target version set to 2.0.0
> >
> > This ticket is for discussion about removing autoload from stdlib (or not)
> >
> > % grep autoload {ext/*,}lib/* | wc -l
> > 442
>
> Looks like tk has most of them:
>
> [aaron@higgins ruby (trunk)]$ git grep autoload ext | grep 'ext/tk' | wc -l
> 417
> [aaron@higgins ruby (trunk)]$
>
> I removed them from psych, and I'll do the same with dl.

The newest version of rdoc has them, but I can remove them.

#6 - 11/23/2011 08:53 AM - normalperson (Eric Wong)

Yukihiro Matsumoto matz@ruby-lang.org wrote:

> Today, I talked with NaHi about enhancing const_missing to enable
> autoload-like feature with nested modules. But autoload itself has
> fundamental flaw under multi-thread environment. I should have remove
> autoload when I added threads to the language (threads came a few
> months after autoload).

Hi, many of my Ruby scripts/apps are single-threaded and I would
like to keep memory usage down.

How about keeping autoload unchanged for single-threaded use and have
modules registered via autoload instantly require everything registered
when Thread.new is called?

#7 - 11/23/2011 08:59 AM - wycats (Yehuda Katz)

It is common to use autoload to register a number of incompatible options. For instance, Rack registers all possible server adapters, and loading the adapters has side-effects.

I looked into this when I worked on Rails threadsafetiness and it is simply incorrect to preload anything registered as an autoload.

Yehuda Katz
(ph) 718.877.1325

On Tue, Nov 22, 2011 at 3:51 PM, Eric Wong normalperson@yhbt.net wrote:

> Yukihiro Matsumoto matz@ruby-lang.org wrote:
>
> > Today, I talked with NaHi about enhancing const_missing to enable autoload-like feature with nested modules. But autoload itself has fundamental flaw under multi-thread environment. I should have remove autoload when I added threads to the language (threads came a few months after autoload).
>
> Hi, many of my Ruby scripts/apps are single-threaded and I would like to keep memory usage down.
>
> How about keeping autoload unchanged for single-threaded use and have modules registered via autoload instantly require everything registered when Thread.new is called?

#8 - 11/23/2011 12:53 PM - luislavena (Luis Lavena)

On Nov 19, 2011 4:11 AM, "Yukihiro Matsumoto" matz@ruby-lang.org wrote:

> Hi,
>
> Today, I talked with NaHi about enhancing const_missing to enable autoload-like feature with nested modules. But autoload itself has fundamental flaw under multi-thread environment. I should have remove autoload when I added threads to the language (threads came a few months after autoload).
>
> So I hereby declare the future deprecation of autoload. Ruby will keep autoload for a while, since 2.0 should keep compatibility to 1.9. But you don't expect it will survive further future, e.g. 3.0.
>
> I strongly discourage the use of autoload in any standard libraries.

Thank you for the details matz, hope this means require and $LOADED_FEATURES along $LOAD_PATH will get a performance refactoring.

Saying because lazy loading no longer be an option, startup times will become a problem in some scenarios.

--
Luis Lavena
AREA 17

#9 - 11/23/2011 04:53 PM - now (Nikolai Weibull)

On Wed, Nov 23, 2011 at 04:34, Luis Lavena luislavena@gmail.com wrote:

> On Nov 19, 2011 4:11 AM, "Yukihiro Matsumoto" matz@ruby-lang.org wrote:
>
> > Today, I talked with NaHi about enhancing const_missing to enable autoload-like feature with nested modules. But autoload itself has fundamental flaw under multi-thread environment. I should have remove autoload when I added threads to the language (threads came a few months after autoload).
> >
> > So I hereby declare the future deprecation of autoload. Ruby will keep autoload for a while, since 2.0 should keep compatibility to 1.9. But you don't expect it will survive further future, e.g. 3.0.

> I strongly discourage the use of autoload in any standard libraries.

> Thank you for the details matz, hope this means require and $LOADED_FEATURES along $LOAD_PATH will get a performance refactoring.
>
> Saying because lazy loading no longer be an option, startup times will become a problem in some scenarios.

I second that. I have mostly switched to using load and require_relative, as they are the fastest alternative on Windows. They still do an insane amount of (seemingly) unnecessary work, but at least they're better than require. I currently still use autoload for loading bigger features that aren't immediately needed at start-up. I think that this use-case for autoload is still valid and perhaps should be in the future, in the same way Yehuda mentioned regarding Rack's server adapters.

#10 - 11/25/2011 03:23 PM - trans (Thomas Sawyer)

"autoloading" can still be done by putting the require within a method that is called only as needed.

The downside of this is that requires get pushed down into deeper levels of code, making requirements less obvious to developers. Documentation not withstanding, it's also not hard to work around. Just link us something like:

```
$AUTOREQ = Hash.new{|h,k|h[k]=[]}

def req(key, path)
  $AUTOREQ[key.to_sym] << path
end

def use(key)
  $AUTOREQ[key.to_sym].each{ |path| require path }
end
```

Then

```
req :rdoc, 'rdoc'
req :markdown, 'redcarpet'

class Tmpl
  initialize(type)
    use type
```

I'm sure this can be greatly improved upon, maybe even generalized (and thread safe?) to make a useful library gem.

#11 - 12/01/2011 10:16 AM - stouset (Stephen Touset)

=begin
One thing to keep in mind is that (({const_missing})) *cannot* be used to replicate (({autoload})) currently due to Ruby cascading constant lookup to the Object namespace. Example:

```
>> class Foo; end
>> class Bar; end
>> class Baz
|    autoload :Foo, 'baz/foo'
|
|    def self.const_missing(name)
|      require "baz/#{name.downcase}"
|    end
|  end
>> Baz::Foo
LoadError: cannot load such file -- baz/foo
>> Baz::Bar
=> Bar
```

You can see here that Baz::Foo wasn't detected (even though there's a Foo in the Object namespace) and the autoload triggers. Baz::Bar, however, resolves to Object::Bar and the (({const_missing}))-based autoload does not fire as expected.
=end

#12 - 12/01/2011 10:34 AM - stouset (Stephen Touset)

=begin
@ThomasSawyer That kind of approach falls apart when you have multiple entry points into your library that require various features.

I've long considered it (perhaps incorrectly) a best practice to organize my hierarchy as this gist:

```
https://gist.github.com/1412552
```

It has several advantages. Users can require everything in my library with only using the top-level file ((({require 'foo'}))), but without incurring an immediate performance penalty for loading the entire library. It becomes loaded progressively as modules are needed, and only needed modules are ever loaded. It also allows a user to require only a nested component of my library ((({require 'foo/baz/qux'}))) and have everything above it automatically pulled in; again, without pre-loading unnecessary parts of the library.

Of course, the disadvantage is that (({autoload})) is being removed. So what alternative exists that lets library authors be considerate to their users, while still ensuring thread-safety?
=end

#13 - 12/01/2011 10:36 AM - wycats (Yehuda Katz)

Stephen: There is an open request, together with a proposal and working patch to address the issue. Check it out:
http://redmine.ruby-lang.org/issues/2740

Make sure to read down towards the bottom as the proposal changed.

#14 - 12/01/2011 11:14 PM - stouset (Stephen Touset)

=begin
After discussion last night with Yehuda, we both agreed that this issue isn't resolved by #2740. Since (({const_missing})) is never called when Ruby resolves a constant like (({Foo::Bar})) to (({Object::Bar})), it cannot be used as a replacement to (({autoload})), which ((*does*)) trigger before the constant lookup is delegated to (({Object})).

This is a more common occurrence than you might think. Requiring any gem or outside library that defines a top-level constant named the same as a nested constant you've autoloaded (via (({const_missing}))) in your project will prevent that nested constant from ever being visible.
=end

#15 - 03/28/2012 12:33 AM - mame (Yusuke Endoh)

- Status changed from Open to Assigned

- Assignee set to nahi (Hiroshi Nakamura)

Hello, NaHi-san

Hiroshi Nakamura wrote:

> This ticket is for discussion about removing autoload from stdlib (or not)

It would be good to open tickets for each library that uses autoload.

And, what do you think about Stephen and Yehuda's opinion?
It looks reasonable to me.

Stephen Touset wrote:

> After discussion last night with Yehuda, we both agreed that this issue isn't resolved by #2740. Since const_missing is never called when Ruby resolves a constant like Foo::Bar to Object::Bar, it cannot be used as a replacement to autoload, which does trigger before the constant lookup is delegated to Object.
>
> This is a more common occurrence than you might think. Requiring any gem or outside library that defines a top-level constant named the same as a nested constant you've autoloaded (via const_missing) in your project will prevent that nested constant from ever being visible.

--
Yusuke Endoh mame@tsg.ne.jp

#16 - 06/15/2012 02:37 AM - trans (Thomas Sawyer)

If autoload is still going to be around for awhile, can we at least get it modified to call Kernel#require? As I've said before, I use a customized load system and b/c of this inability to effect how autoload requires, I can't use my load system with any library that uses autoload.

#17 - 07/01/2012 12:24 AM - nahi (Hiroshi Nakamura)

- File 5653.pdf added

Endoh-san, here's "slide-show" of this proposal.

#18 - 07/01/2012 01:35 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

I totally support this slide. The differences in start-up time of a typical Rails application will reveal very noticeable with and without auto-loading. It would be great to include this demonstration during the slide shows. Would that be possible?

#19 - 07/01/2012 07:23 AM - nahi (Hiroshi Nakamura)

Slightly updated version. You can comment it, too:
https://docs.google.com/presentation/d/1pP8XZBzoA5HehA5xyWBHda6qgpvFvmdMsVWaWauyr2o/edit

I'm not sure if we need to add the demonstration at this moment because the most important part is 'the problem does not exist' :)

#20 - 07/01/2012 10:10 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

sorry, but I didn't get it. It is pretty easy to demonstrate the difference in performance between using autoload and require.

Just change "config.cache_classes = false" to "true" in config/environments/development.rb.

Then, time rails r 'p 1'. For a fresh Rails application here:

with cache_classes set to false: 1.5s
with cache_classes set to true: 1.9s

In my actual app it goes from 4.5s to 6.3s.

#21 - 07/02/2012 03:13 AM - mame (Yusuke Endoh)

NaHi-san,
Please make a presentation yourself at the meeting!

--
Yusuke Endoh mame@tsg.ne.jp

#22 - 07/02/2012 03:32 AM - trans (Thomas Sawyer)

I just want to note that start-up times should be able to be improve somewhat by optimizing #require_relative.

Also, by putting #require_relative in #initialize methods of classes that need library instead of at toplevel and speed improvement is likewise gained.

Nonetheless, I agree that autoload is nice feature and is unfortunate loss.

#23 - 07/02/2012 03:56 AM - funny_falcon (Yura Sokolov)

rosenfeld (Rodrigo Rosenfeld Rosas) wrote:

> sorry, but I didn't get it. It is pretty easy to demonstrate the difference in performance between using autoload and require.
>
> Just change "config.cache_classes = false" to "true" in config/environments/development.rb.
>
> Then, time rails r 'p 1'. For a fresh Rails application here:
>
> with cache_classes set to false: 1.5s
> with cache_classes set to true: 1.9s
>
> In my actual app it goes from 4.5s to 6.3s.

Considering start-up time: many people uses patch #5767 (https://github.com/ruby/ruby/pull/68) to speedup startup time.
(Currently, it is part of falcon.patch in rvm. It could be easily installed with rvm reinstall 1.9.3-falcon --patch falcon)

With this patch difference becomes much lesser. time rails r 'p 1' for fresh rails app:

with cache_classes set to false: 1.43s
with cache_classes set to true: 1.58s

#24 - 07/03/2012 10:53 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Yura, thanks for letting me know. I'll give it a try when I find some time.

#25 - 11/24/2012 08:36 AM - mame (Yusuke Endoh)

- Assignee changed from nahi (Hiroshi Nakamura) to matz (Yukihiro Matsumoto)

matz, how do you feel about this ticket?
autoload should be marked as "deprecated" in 2.0.0?

--
Yusuke Endoh mame@tsg.ne.jp

#26 - 02/12/2013 10:38 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

+1 for removing autoload. It not only has troubles with thread safety, which I believe is fixable (although I confess I haven't thought much about it).

There is a more serious issue in my opinion that is unfixable with current specs.

For a while I've been experiencing a subtle bug in my Rails application that only happened on development mode (when autoload is enabled) and only for the first request. Recently I decided to understand what was happening and Xavier Noria showed me what was happening.

The problem is that I had some custom classes that relied on ActionView::Helpers::NumberHelper. Since all code in number_helper.rb was self-contained and it had no extra dependencies on ActionView, I just used it directly and my unit tests passed and my application always worked except for the first request when my custom class was used by the controller.

The problem is that once Rails is loaded it will declare an auto-load dependency on ActionView. Then when I "require 'action_view/helpers/number_helper'" that file will be something like:

module ActionView
module Helpers
module NumberHelper

But since Helpers is set up to use autoload, once "module Helpers" is found, it will load "action_view/helpers.rb". And this file will then require "action_view/helpers/number_helper.rb" which causes a circular dependency problem.

Here is a complete Ruby-only code exemplifying the issue:

./test.rb:
```
autoload :A, 'a'
require 'a/b'
```

./lib/a.rb:
```
require 'a/b'
```

./lib/a/b.rb:
```
module A
module B
end
end
```

ruby -I lib test.rb

This kind of bug may be hard to track, so I agree with Matz that autoload should be dead.

#27 - 02/17/2013 03:17 PM - ko1 (Koichi Sasada)

- Target version changed from 2.0.0 to 2.1.0

time up. 2.0.0 was fixed.
Matz, could you consider it on 2.1.0?
Or please reject it.

#28 - 05/22/2013 12:46 PM - reset (Jamie Winsor)

Matz,

It appears that a patch was accepted into the soon to be released 1.9.4 branch (http://bugs.ruby-lang.org/issues/921) to fix the thread safety issues with autoload. Is this something that you still wish to see deprecated and removed from the language?

I agree with Yehuda's point regarding intelligent loading of modular, and possibly incompatible, components (http://bugs.ruby-lang.org/issues/5653#note-7).

Are there other reasons aside from the thread safety issues which would still make deprecating this feature a good choice?

#29 - 06/24/2013 05:49 PM - Anonymous

Where are we now ? Will the autoload be deprecated in the future ?

#30 - 08/31/2013 06:17 PM - Anonymous

I would also be interested in the answer to se8's question.

#31 - 01/30/2014 06:16 AM - hsbt (Hiroshi SHIBATA)

- Target version changed from 2.1.0 to 2.2.0

#32 - 04/07/2015 12:40 PM - plribeiro3000 (Paulo Henrique Lopes Ribeiro)

Do we have a decision for this?

#33 - 06/19/2015 07:40 AM - ngoto (Naohisa Goto)

- Related to Bug #11277: "code converter not found" error with multi-thread (high occurrence rate since r50887) added

#34 - 04/28/2017 01:22 PM - hsbt (Hiroshi SHIBATA)

- Target version changed from 2.2.0 to Next Major

#35 - 04/28/2017 01:23 PM - hsbt (Hiroshi SHIBATA)

- Description updated

#36 - 11/23/2018 08:23 AM - matz (Yukihiro Matsumoto)

- Related to Feature #7835: autoload will be dead added

#37 - 01/21/2019 05:57 PM - rafaelfranca (Rafael França)

matz (Yukihiro Matsumoto) when taking in consideration deprecation/removal of autoload I'd like you to consider that Rails built and plan to use a thread-safe code loader based on autoload. See https://medium.com/@fxn/zeitwerk-a-new-code-loader-for-ruby-ae7895977e73

#38 - 01/23/2019 10:19 AM - shyouhei (Shyouhei Urabe)

rafaelfranca (Rafael França) wrote:

> matz (Yukihiro Matsumoto) when taking in consideration deprecation/removal of autoload I'd like you to consider that Rails built and plan to use a thread-safe code loader based on autoload. See https://medium.com/@fxn/zeitwerk-a-new-code-loader-for-ruby-ae7895977e73

That's pretty unfair. This thread is 7yrs old and the library only started in a week or two.
Consideration shall be exercised by its authors, not by matz.

#39 - 01/23/2019 10:55 AM - Eregon (Benoit Daloze)

It would be good to have a definite choice on this thread, I'll add it to the developer meeting (#15546).
I think at the last RubyKaigi during the Q/A, it was becoming clear removing #autoload would be a huge breaking change and there are usages which cannot be replaced easily.
The ability to load code lazily is increasingly important with larger applications, so that if #autoload was removed, we'd need another easy way to achieve that to maintain reasonable boot times.

I think the observation that Rails uses autoload quite a bit is something worth consideration for this ticket, which I believe is what rafaelfranca (Rafael França) meant.

#40 - 01/23/2019 01:39 PM - fxn (Xavier Noria)

Hey! I am the author of Zeitwerk, let me do a quick followup here.

First and foremost, I'd like to be very straightforward saying that if Kernel#autoload is killed tomorrow, that would be fine with me. I'd shutdown Zeitwerk and Rails 6 plans without regrets. The Ruby core team are the stedwards of the language. You know better than anybody else the rationale for your decisions, and if you believe you have to definitely say goodbye to Kernel#autoload I'd profoundly respect it.

Having say that. Why have I worked on Zeitwerk while being well aware of this 7-years old issue? For several reasons: The first one is that being so old means to me this issue is not black/white and perhaps is still open for consideration. Another one is that I wanted to solve some very specific problems for which I don't know another solution today. And a third one is that the existence of something like Zeitwerk, Rails adoption, and the kind of problems that it solves could bring more information or context when pondering the future of Kernel#autoload.

Zeitwerk is motivated specifically by two problems: One problem is Rails autoloading, which cannot reproduce Ruby semantics due to lack of information in const_missing, whereas Kernel#autoload logic is builtin in the interpreter and works perfectly. The other one is brittle requires in non-trivial projects. Both of them are explained in the post and Zeitwerk README.

1) Being able to reload code is handy in web applications development. That is replacing the objects stored in the autoloaded constants, not reopening classes by reevaluating the files.

2) In any non-trivial project, getting the require calls right is difficult, you always forget some and gives load order bugs.

3) If you structure your project in a conventional manner in which file paths match constant paths, the requires don't feel DRY. You are repeating something all the time that could be automated.

4) Being able to work as if all your classes and modules are just available everywhere (as in Rails) is a great user experience.

5) Being able to transparently load code on demand in development speeds things up in large code bases.

Now, we have to differentiate the problem from the solution. The points above are the problem, the use cases, and a solution based on Kernel#autoload is possible today. If there were other ways to solve the same problems, that would be also great.

And, finally, let me say it again: Whatever decision the Ruby core team agrees on, I'd respectfully accept without regrets.

#41 - 01/24/2019 10:19 AM - fxn (Xavier Noria)

Let me add a couple of things.

In case const_missing is discussed, these are the reasons why implementing autoloading based on const_missing doesn't fly (from memory):

1) The nesting is unknown.
2) Whether the missing constant was relative is unknown.
3) Since const_missing is the last resort in the constant resolution algorithms, you may miss an autoload if some class/module up in the nesting or the ancestor chain happens to have a constant with the same name.
4) It is not thread-safe.

On the other hand, Zeitwerk is based on these techniques (some are not portable as of this writing, but would hope they are in the future for compatibility with CRuby):

1) Kernel#autoload, related API, and the fact that it is thread-safe.
2) Constants API.
3) Kernel#autoload calls Kernel#require (Zeitwerk decorates it).
4) TracePoint for the :class event if there is at least one explicit namespace (hotel.rb & hotel/pricing.rb), to support the very common edge case in which Hotel includes Hotel::Pricing and you need to set an autoload on Hotel for :Pricing before that line is reached. (I am aware of https://bugs.ruby-lang.org/issues/14104 and this week I have made some changes in that line, as in https://github.com/fxn/zeitwerk/commit/a1bd83b10521f41f7f74e921602839a1813d11e4).

Additionally, Zeitwerk by design uses internally absolute paths only, there are no lookups in $LOAD_PATH or autoload paths, gems using Zeitwerk should load a tad faster. Even more, a code base managed by Zeitwerk does not even need to be in $LOAD_PATH.

Any other information that you'd like to know please just tell me!

#42 - 01/28/2019 04:22 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Hi, this is a discussion which I'm pretty much interested in and I'd like to add a few comments.

It seems to me that there are basically two reasons why people rely on autoload:

1. to achieve fast boot;
2. for convenience (not having to explicitly require the dependencies);

So, I'd just like to point out that there are options to achieve fast boot rather than resorting to autoload. Loading too many code upfront can be very slow not only in Ruby and there is another very well known solution for that problem that is applied to all other languages and which is available to Ruby as well: on-demand loading or lazy loading.

This is not a theory only. I've been coding this way for over an year. I maintain a large application for a long time. It was written in Groovy when I joined the project about 7 years ago, and it wasn't a small application already by that time and it's only getting bigger. Some years ago I had gradually moved the source from Grails to Rails and a few years ago I moved it from Rails to Roda. While doing that, I knew already that loading all the application upfront could be pretty slow and I didn't like the autoload or constant-missing approach already by that time to load code on-demand automatically and I would prefer to do that explicitly instead. Another common requirement when developing server-based apps such as web servers is the auto-reloading feature.

So, the solution I found for my use case, which would also scale to any code base size, is to split the application in multiple apps, one per route base path. With Roda an option, the one I opted for, would be to use the multi_run plugin:

http://roda.jeremyevans.net/rdoc/classes/Roda/RodaPlugins/MultiRun.html

Instead of loading the sub-apps upfront, I'd call it something like this:

```
run 'some_app', ->(env){ require_relative 'apps/some_app'; Apps::SomeApp.call env }
```

I actually use a helper to register the apps, but this is basically what it happens behind the scenes.

When some of the relevant files (source, settings) change, I'd unload all reloadable constants and run the application, using a gem I've authored a few years ago:

https://github.com/rosenfeld/auto_reloader

With that configuration my application boots instantly (much faster than a fresh Rails app) and with a one-line code change I could decide to load it all up-front for the production environment (I've opted for keeping it loading on-demand in production too, so that I could deploy my app as if the app was

comprised by several micro-services in the case some of the requests would need more workers, so that such app wouldn't have to use too much RAM, for example).

So, I'd say that there are better solutions to achieve a fast boot time than complicated frameworks such as bootsnap and similar ones and without relying on autoload or constant missing to automatically load code on demand.

Now, I'd like to discuss the second reason why people want autoload: convenience.

While I do have to type all my dependencies explicitly in my code base I actually like this. I mean, I prefer to type my dependencies explicitly. I'd certainly prefer an approach such as the import/export one used by ES6, for example, but I think such change would be too big for Ruby.

I find it particularly annoying when I'm trying to follow what some third-party library is doing and I have to guess the dependencies based on their several lines of autoload calls distributed anywhere in the code base. I like to be able to quickly inspect the dependencies of each file separately. There was a time I'd like to use one method from Rails helpers to format a currency value and I couldn't just require the file containing the method definition because it wouldn't work unless I required the full action_view gem due to issues caused by the use of autoload, since not all dependencies were explicitly required by that file. I can't call this a best practice, so I'm not convinced that this approach actually leads to convenience. It was quite inconvenient to me when I wanted to use just a small subset of the actionview gem, for example.

Since Ruby strives to make programming a joy I believe there's a conflict here. In one hand there's a bunch of people that find it convenient not to have to type their dependencies in all sources files using them because it feels too "Javay" and consider specifying the dependencies explicitly annoying, boilerplate code. Those see autoload or automatic loading through constant-missing hooks like a huge win for making application development more convenient.

On the other hand there are people like me, who find annoying not to be able to require an specific file from a third-party library because it can't be required separately because it relies on autoload calls that are supposed to be registered in another file, which is not required by this particular file, by the way.

That's why I find this very topic so tricky. Because it's virtually impossible to make all users happy. If you opt for keeping the autoload feature, those who want to be able to require individual files from other projects, or those who want to view all dependencies for every file explicitly available, will become disappointed (like me). And if autoload is removed, a lot of code would simply stop working since this seems to be a very common practice among gem authors. And, of course, most Ruby users would be upset as well because the gems they relied on are no longer working, which is very understandable. Also, I believe most Rails users actually like all the conventions and not having to explicitly require their dependencies, so I bet most Ruby users would prefer autoload to remain available in the language.

That's why, even though I'd personally love to see autoload gone once and for all from the language, I actually sympathize with most Ruby devs who actually seem to like autoload and I think that actually removing it from the language wouldn't be the right move at this moment.

The way I see this issue is like Matz was begging Rails to find an alternative because he wanted very much to get rid of autoload because he realized that it's not a good feature to have in the language but he also realizes that he can't just remove autoload while Rails keeps using it. If the Rails team stopped using it, maybe other popular gem authors would follow the move and hopefully one day we could get rid of it. Instead, the Rails core team has decided not only to disregard Matz's request, but to actually do the opposite and rely even more on autoload so that removing autoload would actually break all Rails apps because it would become impossible to replace it with another solution without requiring changes to the application.

I actually feel sad about this decision because Rails is now forcing the development of Ruby in a direction that is the opposite of its creator will.

I'd love to see the Rails core team reconsider their decision although I don't think this will ever happen :(

Sadly it seems we'll have to live with autoload in the language, not because most Rubyists seem to agree that it's a great feature, but because the Rails core team decided it had to stay instead. This is the part that makes me feel more sad :(

#43 - 01/28/2019 06:38 PM - rafaelfranca (Rafael França)

I think you missed an important point here. Like fxn (Xavier Noria) said above:

> First and foremost, I'd like to be very straightforward saying that if Kernel#autoload is killed tomorrow, that would be fine with me. I'd shutdown Zeitwerk and Rails 6 plans without regrets.

So Rails team is not forcing anything. If Matz decide remove autoload we would gladly change Rails to use something else.

Right now is exactly the best time to make that decision since Zeitwerk is pretty new and we still didn't bet even more in autoload. Making this deciding this two or three years from now would make it even harder since more Rails applications would be using autoload.

That was my initial reason to comment in this thread. Give Matz one more data point on what problems autoload solves that other alternatives doesn't and ask feedback if we should or should not bet more in autoload.

There is no need to be sad ;)

#44 - 02/03/2019 11:01 PM - MSP-Greg (Greg L)

One issue that autoload helps with is stability when the code 'entry point' is not known. Many applications (eg RubyGems) are largely CLI based, but testing may start anywhere. Non-typical applications may also use something like RubyGems in a unexpected manner.

With an 'autoload' app, neither the test files nor the application files are burdened with extra require statements due the indeterminate nature of the code 'entry point'.

Issues also appear from parallel testing, conditional requires, running a subset of test files which result in a different set of $LOADED_FEATURES, etc...

#45 - 02/05/2019 02:27 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Hi Greg, could you please expand on your argument? Testing may start anywhere but tests should require the files they are testing, so I don't understand why autoload could make things more stable. If the tests are failing because of a missing require, including the require will fix the test anyway.

I couldn't get the point regarding CLI based gems. Why would they become more stable by using autoload instead of require?

> Non-typical applications may also use something like RubyGems in an unexpected manner

What are non-typical applications? What does it mean to use something like RubyGems? What would be the unexpected manner? Could you please provide examples?

> With an 'autoload' app, neither the test files nor the application files are burdened with extra require statements"...

Convenience is the only reason I can currently see to use autoload, although I find this to be a subjective reason because not everyone find them convenient.

> ...due the indeterminate nature of the code 'entry point'.

What would be indeterminate in finding the entry point of some CLI or any other Ruby type of code? As long as you type all dependencies in each file relying on them you should be fine no matter what the entry point is. The only reason you might worry about entry points is in case you're worried about fast start-up time and want to apply lazy load to your code.

> Issues also appear from parallel testing, conditional requires, running a subset of test files which result in a different set of $LOADED_FEATURES, etc...

How could those be an issue, could you please provide concrete examples? As long as you always require your dependencies parallel tests should pass as usual. Why are conditional requires a problem without autoload? What is the issue with running a subset of test files? I do that all the time and I don't rely on autoload. What are the issues with having a different set of $LOADED_FEATURES?

#46 - 02/07/2019 06:20 AM - matz (Yukihiro Matsumoto)

- Status changed from Assigned to Closed

OK, I withdraw the proposal. The autoload method will stay (for Ruby3.0).

Matz.

#47 - 02/07/2019 07:02 AM - akr (Akira Tanaka)

I created [Feature #15592] for a feature to switch
"autoload" behavior to "require" immediately to obtain both
safety of eager loading and easier development of lazy loading.

#48 - 02/07/2019 07:02 AM - akr (Akira Tanaka)

- Related to Feature #15592: mode where "autoload" behaves like an immediate "require" added

Files

noname	500 Bytes	11/22/2011	Anonymous
5653.pdf	38.3 KB	07/01/2012	nahi (Hiroshi Nakamura)