Ruby trunk - Feature #15663

Documenting autoload semantics

03/13/2019 06:25 PM - Eregon (Benoit Daloze)

Status:	Open
Priority:	Normal
Assignee:	
Target version:	

Description

The semantics of autoload are extremely complicated.

As far as I can see, they are unfortunately not documented.

ruby/spec tries to test many aspects of it, test/ruby/test_autoload.rb has a few tests, and e.g. zeitwerk tests some other parts.
One could of course read the MRI source code, but I find it very hard to follow around autoload.

For the context, I'm trying to implement autoload as correct as possible in TruffleRuby and finding it very difficult given the inconsistencies (see below) and lack of documentation.

There is nowhere a document on how it should behave, and given the complexity of it I am not even sure MRI behaves as expected.
Could we create this document?
For instance, there is such a document for refinements.

Here is an example how confusing autoload can be, and I would love to hear the rationale or have some written semantics on why it is that way.

main.rb:

```
require "pp"

$: << __dir__

Object.autoload(:Foo, "foo")

CHECK = -> state {
  checks = -> {
    {
      defined: defined?(Foo),
      const_defined: Object.const_defined?(:Foo),
      autoload?: Object.autoload?(:Foo),
      in_constants: Object.constants.include?(:Foo),
    }
  }

  pp when: state, **checks.call, other_thread: Thread.new { checks.call }.value
}

CHECK.call(:before_require)

if ARGV.first == "require"
  require "foo"
else
  Foo # trigger the autoload
end

CHECK.call(:after)

p Foo
```

foo.rb:

```
CHECK.call(:during_before_defining)
```

```
module Foo
end

CHECK.call(:during_after_defining)
```

Here are the results for MRI 2.6.1:

```
$ ruby main.rb
{:when=>:before_require,
 :defined=>"constant",
 :const_defined=>true,
 :autoload?=>"foo",
 :in_constants=>true,
 :other_thread=>
  {:defined=>"constant",
   :const_defined=>true,
   :autoload?=>"foo",
   :in_constants=>true}}
{:when=>:during_before_defining,
 :defined=>nil,
 :const_defined=>false,
 :autoload?=>nil,
 :in_constants=>true,
 :other_thread=>
  {:defined=>"constant",
   :const_defined=>true,
   :autoload?=>"foo",
   :in_constants=>true}}
{:when=>:during_after_defining,
 :defined=>"constant",
 :const_defined=>true,
 :autoload?=>nil,
 :in_constants=>true,
 :other_thread=>
  {:defined=>"constant",
   :const_defined=>true,
   :autoload?=>"foo",
   :in_constants=>true}}
{:when=>:after,
 :defined=>"constant",
 :const_defined=>true,
 :autoload?=>nil,
 :in_constants=>true,
 :other_thread=>
  {:defined=>"constant",
   :const_defined=>true,
   :autoload?=>nil,
   :in_constants=>true}}
Foo
```

Looking at during_before_defining, the constant looks not defined during the autoload for the Thread loading it, but looks defined and as an autoload for other threads.

Now we can discover other subtle semantics, by using require on the autoload file instead of accessing the constant:

```
$ ruby main.rb require
{:when=>:before_require,
 :defined=>"constant",
 :const_defined=>true,
 :autoload?=>"foo",
 :in_constants=>true,
 :other_thread=>
  {:defined=>"constant",
   :const_defined=>true,
   :autoload?=>"foo",
   :in_constants=>true}}
{:when=>:during_before_defining,
```

```
 :defined=>nil,
 :const_defined=>false,
 :autoload?=>nil,
 :in_constants=>true,
 :other_thread=>
  {:defined=>nil, :const_defined=>false, :autoload?=>nil, :in_constants=>true}}
{:when=>:during_after_defining,
 :defined=>"constant",
 :const_defined=>true,
 :autoload?=>nil,
 :in_constants=>true,
 :other_thread=>
  {:defined=>"constant",
   :const_defined=>true,
   :autoload?=>nil,
   :in_constants=>true}}
{:when=>:after,
 :defined=>"constant",
 :const_defined=>true,
 :autoload?=>nil,
 :in_constants=>true,
 :other_thread=>
  {:defined=>"constant",
   :const_defined=>true,
   :autoload?=>nil,
   :in_constants=>true}}
Foo
```

Looking at during_before_defining, now the other threads seem to see the constant not defined, although it is still in Object.constants.
But of course, the constant cannot be removed, as otherwise that would not be thread-safe and other threads would raise NameError when accessing the constant.
In fact, we can see other threads actually wait for the constant, by changing to Thread.new { Foo; checks.call }, and then we get a deadlock:

```
Traceback (most recent call last):
    2: from main.rb:20:in `<main>'
    1: from main.rb:17:in `block in <main>'
main.rb:17:in `value': No live threads left. Deadlock? (fatal)
3 threads, 3 sleeps current:0x00007f0124004cb0 main thread:0x000055929cc2c470
* #<Thread:0x000055929cc5b348 sleep_forever>
   rb_thread_t:0x000055929cc2c470 native:0x00007f013381d700 int:0
   main.rb:17:in `value'
   main.rb:17:in `block in <main>'
   main.rb:20:in `<main>'
* #<Thread:0x000055929ce2b380@main.rb:17 sleep_forever>
   rb_thread_t:0x000055929ce026d0 native:0x00007f0129007700 int:0
    depended by: tb_thread_id:0x000055929cc2c470
   main.rb:17:in `value'
   main.rb:17:in `block in <main>'
   foo.rb:1:in `<top (required)>'
   /home/eregon/.rubies/ruby-2.6.1/lib/ruby/2.6.0/rubygems/core_ext/kernel_require.rb:54:in `requi
re'
   /home/eregon/.rubies/ruby-2.6.1/lib/ruby/2.6.0/rubygems/core_ext/kernel_require.rb:54:in `requi
re'
   main.rb:17:in `block (2 levels) in <main>'
* #<Thread:0x000055929ce29c38@main.rb:17 sleep_forever>
   rb_thread_t:0x00007f0124004cb0 native:0x00007f0128e05700 int:0
    depended by: tb_thread_id:0x000055929ce026d0
   main.rb:17:in `block (2 levels) in <main>'
```

This is quite weird. Is the second behavior a bug?
Why should other threads suddenly see the constant as "not defined" while it is loading via require in the main thread?
It's also inconsistent with the first case.

I would have thought require autoload_path would basically do the same as triggering the autoload of the constant (such as Foo). But the results above show they differ.

There are many more complex cases for autoload, such as this spec, or how is thread-safety is achieved when methods are defined incrementally in Ruby but the module is defined immediately.

Who is knowledgeable about autoload and could answer these questions?
Could we start a document specifying the semantics?

History

#1 - 03/13/2019 09:49 PM - shevegen (Robert A. Heiler)

May explain why matz wants to remove autoload in the long run. :)

#2 - 03/20/2019 02:48 AM - akr (Akira Tanaka)

Eregon (Benoit Daloze) wrote:

> I would have thought require autoload_path would basically do the same as triggering the autoload of the constant (such as Foo). But the results above show they differ.

Agreed.

I think no one seriously considered about requiring a library used for autoload.

> Who is knowledgeable about autoload and could answer these questions?
> Could we start a document specifying the semantics?

I think we should start to make autoload semantics simpler by introducing "global autoload lock"
as I described in https://bugs.ruby-lang.org/issues/15598 .
This is needed because ruby doesn't (cannot) know dependencies of autoloaded libraries before loading.
It makes autoload related procedure single threaded which is much simpler than multi threads.

#3 - 03/20/2019 03:33 AM - akr (Akira Tanaka)

akr (Akira Tanaka) wrote:

> > Could we start a document specifying the semantics?
>
> I think we should start to make autoload semantics simpler by introducing "global autoload lock"
> as I described in https://bugs.ruby-lang.org/issues/15598 .

Of course, there is no problem to describe the single thread semantics.

#4 - 03/20/2019 11:03 AM - Eregon (Benoit Daloze)

akr (Akira Tanaka) wrote:

> Eregon (Benoit Daloze) wrote:
>
> > I would have thought require autoload_path would basically do the same as triggering the autoload of the constant (such as Foo). But the results above show they differ.
>
> Agreed.

It seems tricky implementation-wise, constant resolution (e.g. #const_get) has the constant data structure, and #require has the expanded file path + related lock but none of them has both.

I was thinking maybe the require lock per path should be used for everything, but then since #autoload calls require dynamically, how to keep track which thread is loading the constant and so should observe the autoload constant as "not defined" while loading it, without deadlocks?
Other threads might try to load the constant too, or require the autoload path, and only one thread should be the loading thread for that constant.

I'd be tempted for constant resolution to basically do nothing more than call require, but then we need Kernel#require when it starts loading the file to also mark the autoload constant as being loaded by the current thread (such that the constant looks as "not defined").
How to pass that information (e.g., the constant data structure) from constant resolution down to Kernel#require?
The require feature could be changed by user-defined require. And a user-defined require might very well require other files for its own logic (e.g., rubygems files).

> I think no one seriously considered about requiring a library used for autoload.

lib/net/http.rb has autoload :OpenSSL, 'openssl' and therefore just require "net/http"; require "openssl" produces such a case.

> I think we should start to make autoload semantics simpler by introducing "global autoload lock"
> as I described in https://bugs.ruby-lang.org/issues/15598 .
> This is needed because ruby doesn't (cannot) know dependencies of autoloaded libraries before loading.
> It makes autoload related procedure single threaded which is much simpler than multi threads.

That would simplify things, but then I think it would also need to be a global require lock, and I think that can be problematic for compatibility: e.g., what if a required file starts a server and so the require never ends, and later on another thread wants to require some code?

#5 - 03/20/2019 01:30 PM - akr (Akira Tanaka)

Eregon (Benoit Daloze) wrote:

> I was thinking maybe the require lock per path should be used for everything, but then since #autoload calls require dynamically, how to keep track which thread is loading the constant and so should observe the autoload constant as "not defined" while loading it, without deadlocks? Other threads might try to load the constant too, or require the autoload path, and only one thread should be the loading thread for that constant.

I think "lock per path" can cause deadlock with "mutual require" similar to https://bugs.ruby-lang.org/issues/15598 .

> That would simplify things, but then I think it would also need to be a global require lock, and I think that can be problematic for compatibility: e.g., what if a required file starts a server and so the require never ends, and later on another thread wants to require some code?

Why "it would also need to be a global require lock"?

#6 - 04/20/2019 01:44 AM - fxn (Xavier Noria)

Let me share some thoughts that won't help much, but would like to contribute anyway :).

To me it is a surprise that constants for which there is an autoload are treated as existing by the constants API. My basic observation is that you don't know if the constant will actually be there until you execute the require. Since the require could fail, from non-existing files, to syntax errors, to files not actually defining the constants. The constant may never materialize.

For me the semantics would be easier if autoloads were treated separately. For example, if const_defined? or defined? returned false for autoloads, constants would not include them, etc. You have actually existing constants, and autoloads, you have autoload? for autoloads if you need to introspect them. You would need remove_autoload perhaps... you see the mental model: two separate collections.

Of course, nothing of this is backwards compatible, so of no practical value for this thread surely.