Ruby - Feature #10869

Add support for option to pre-compile Ruby files

02/19/2015 03:54 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Status:	Open	
Priority:	Normal	
Assignee:		
Target version:		

Description

I know this topic is tricky but please bear with me.

Goal: improve performance on files loading to speed up the boot process of some application requiring lots of files/gems.

Background:

Currently most frameworks/gems rely on the autoload feature to allow applications to load faster by lazy loading files as missing constants are referenced.

Autoload behavior may lead to hard-to-understand bugs and I believe this is the main reason why Matz discourages its usage:

https://bugs.ruby-lang.org/issues/5653

I described a bug involving autoload in a real scenario in this comment of this same issue:

https://bugs.ruby-lang.org/issues/5653#note-26

While I agree that autoload should be discouraged I think we should provide an alternative for speeding up application loading.

Overall benchmarks:

I decided to create a simple benchmark in order to measure how much time MRI would take to load 10_000 files containing a hundred methods each:

```
10000.times{|j| File.open("test#{j}.rb", 'w'){|f|f.puts "class A#{j}"; 100.times{|i| f.puts "  def m#{i}; end"}; f.puts "end"}}
```

```
time ruby -r benchmark -I. -e 'puts Benchmark.realtime{10000.times{|i|require "test#{i}"}}
```

```
8.766814350005006
```

```
real    0m10.068s
user    0m9.416s
sys     0m0.532s
```

```
time cat test*.rb > /dev/null
```

```
real    0m0.107s
user    0m0.068s
sys     0m0.040s
```

As you can see, most of the time is spent on MRI itself rather than on disk. Using require_relative doesn't make any real difference either.

Suggested solution: Pre-compiled files

I know nothing about MRI internals but I suspect that maybe if MRI could support some sort of database containing a precompiled version of the files (the bytecodes maybe). The database would store the size and a hash for each processed file. If the size and hash remain the same it would assume the bytecodes in the database are up-to-date, which should happen in most cases. In this case those files could be possibly loaded much faster.

In order to avoid additional overhead or some bugs in some cases, maybe an option to enable the pre-compile behavior would be better to allow us to test this approach.

I understand that it may be complicated to precompile all kind of Ruby files as they could execute code as well rather than simply declaring classes. In such cases I still think it would worth to detect such cases and skip pre-compiling for such files and only pre-compile those files containing simple class declarations only, which is the case for a lot of files already. Maybe this could potentially make gem owners move their statements to a separate file in order to allow the classes to be precompiled in the future...

Do you have any other suggestions to speed up application loading that do not involve autoload and conditional requires? Do you think precompilation is possible/worthy on MRI?

History

#1 - 02/19/2015 09:48 PM - normalperson (Eric Wong)

I profiled startup time a bit last year and much of it was in the
parser and malloc; so having this should be a win.

I've pondered having something like a self-managing cache in $HOME. It
would be similar to what ccache uses for caching gcc output, but for
Ruby iseqs. I've used ccache for over a decade it's never failed me.

Since 2.2 fixed iseq dumping/loading, we're closer to being able to
support this. We'll need to be careful and correctly invalidate the
cache on any internal ABI changes, of course, and I don't want us to be
stuck on an old/stable ABI for a cache.

I don't think this will make distributing most Ruby apps any easier
(because C extensions are usually the biggest problem)

#2 - 02/20/2015 01:11 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Good to know that you think this could speed up requires :) At least this seems like something feasible in case someone would be interested in giving it a try.

For the ABI case, maybe the easiest would be to separate the database location by ABI version, for example:

~/.mri/abi-xx/cache-database
~/.mri/abi-xy/cache-database

Or it could use the exact Ruby version/patch as the subdirectory name. I believe a first step towards precompiling would be great to allow us to evaluate how faster require could get with such approach. If the results are great then it would worth exploring more efficient solutions for caching format, invalidation rules and how to support most cases. But if we could get an initial version that only took care of the common cases it would already allow us to evaluate how faster we would be able to require files or gems with such approach...

Initially the database could be even some sort of Marshal dump if it makes it easier to implement...

#3 - 02/23/2015 10:28 AM - ko1 (Koichi Sasada)

We did trials.

- AOT compiler from YARV code to C
  - https://github.com/shyouhei/ruby/tree/shyouhei/yarvaot/ext/yarvaot?utm_source=twitterfeed&utm_medium=twitter
  - https://github.com/soba1104/CastOff

    ...
- [ruby-core:46896] to define protocl to make your own loading framework

It is my favorite topic.
Recent days, go-language is popular. I heard one of the reasons is packaging codes into one binary file. For Ruby, if we have AOT compiled code (special format or C extensions) and packaging them, we can provide similar feature.

#4 - 02/23/2015 11:08 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

I understand there are different reasons why one would want to pre-compile Ruby code but not all kind of pre-compilation would meet all goals.

Personally, I'm not currently worried about packing an application in a single file, or to hide code in some binary form and similar concerns. With this particular request I'm mostly interested in providing an alternative to autoload with regards to performance improvent over the application boot process.

I understand the other goals are also interesting ones but maybe it would be much simpler to implement some kind of solution that would simply take care of caching the Parser result to avoid reparsing the files everytime we require them in case such cache could improve a lot the load time of all files required by an application.

Eric believes the Parser take a significant amount of time when requiring files and such a simple and imperfect implementation of parser results caching could at least give us some numbers to evaluate how faster loading an application with thousands of files containing hundreds of methods each could be with such approach.

Even if such patch couldn't be used production-wide due to lots of edge cases which are not handled it would at least give us an idea on how much faster loading Ruby code could be...

I think it might be too early to think about defining a framework loading protocol or something like that before we play a bit with this idea for a little while...

If we try to handle all goals that pre-compilation support could help it may become too complex for an initial implementation.

I'm really interested in having an idea of how much faster code loading could be without autoload and a pretty basic implementation could get us such idea without requiring too much development time, thinking about how to handle all possible cases that pre-compilation could solve or which would be the best database format to store the cached parsing result and so on.

For instance, suppose it would be easier to just create some path/to/.source.rb.precompiled file for each path/to/source.rb than to think about a more complex database. Of course this wouldn't be acceptable as a final implementation for many reasons, including read-only locations, different MRI versions in the same computer and so on. But if this is the fastest to implement, maybe it would be enough to give us some numbers on how faster we could get by using a cache for the parser results... We could then continue the discussion based on some real numbers and the expected performance improvements with such approach.