Precompiling Ruby scripts

Myth & Fact

Koichi Sasada
ko1@heroku.com
Questions

Have you ever thought 

Ruby is slow?
Quick answer

- Try latest MRI contains optimized VM
  - Ruby 1.9 and later implement VMs
  - Ruby 2.3 (Dec/2015) also includes many improvements
  - VMs are written by Koichi Sasada
Questions

Have you ever thought Ruby’s GC is slow?
Quick answer

• **Try Ruby 2.1 and later**
  • Generational and incremental techniques to increase throughput and to reduce GC pause time
  • GCs are implemented by **Koichi Sasada**
Questions

Have you ever thought Ruby/Rails boot time is slow?
Quick answer

• Check out this presentation :p

• This presentation is by Koichi Sasada
  • A programmer living in Tokyo, Japan
  • Ruby core committer since 2007
Koichi is a member of Heroku Matz team

• Heroku employs three full time Ruby core developers in Japan named “Matz team”

Matz
Nobu
Koichi (ko1)
Mission of Heroku Matz’s team

Design Ruby language and improve quality of MRI

Latest achievement: Ruby 2.3
Next challenge: Ruby 2.4 and Ruby 3

Feel free to ask about Ruby itself later
Have you ever thought "Ruby/Rails boot time is slow?"
Myth

“If we have an AOT compiler, the boot time issue will be solved”
OK, let’s try it.
Today’s talk is about:

• New feature of Ruby 2.3
  “Pre-compilation primitives”
• Yomikomu gem: what is and how to use it.
• Evaluation results includes redmine boot time
New feature of Ruby 2.3
“Pre-compilation primitives”
Compilers for interpreters

• JIT (just in time) compilers
  • Compile to more efficient code at runtime
  • Runtime statistics information are available

• AOT (ahead of time) compilers
  • Program to native machine code (like C, ...)
  • Program to other languages code
    • Translate to C, Java, etc...
  • Program to persistent byte code (like Java, ...)
    • `RubyVM::InstructionSequence in Ruby’s case`
RubyVM::InstructionSequence or ISeq
Ruby’s bytecode

• All of Ruby programs are compiled to ISeqs
• MRI makes ISeqs at boot time
Before pre-compilation

Your application code

Bundled Libraries

Gem Libraries

Ruby script

Parse

Compile

Evaluator

Embedded classes and methods

Ruby Bytecode (ISeq)

Interpret on RubyVM
After pre-compilation

Pre-compilation utility
Parse/compile Ruby scripts
Compiled binary
Load
Interpret on RubyVM

Ruby script
Parse
Compile
Ruby Bytecode (ISeq)
Evaluator
Embedded classes and methods

Your application code
Bundled Libraries
Gem Libraries

Your application code
Bundled Libraries
Gem Libraries

Pre-compilation utility
Parse/compile Ruby scripts
Compiled binary
Load
Interpret on RubyVM

Ruby script
Parse
Compile
Ruby Bytecode (ISeq)
Evaluator
Embedded classes and methods

Your application code
Bundled Libraries
Gem Libraries

Pre-compilation utility
Parse/compile Ruby scripts
Compiled binary
Load
Interpret on RubyVM

Ruby script
Parse
Compile
Ruby Bytecode (ISeq)
Evaluator
Embedded classes and methods

Your application code
Bundled Libraries
Gem Libraries

Pre-compilation utility
Parse/compile Ruby scripts
Compiled binary
Load
Interpret on RubyVM

Ruby script
Parse
Compile
Ruby Bytecode (ISeq)
Evaluator
Embedded classes and methods

Your application code
Bundled Libraries
Gem Libraries
Purpose of pre-compilation

• Fast boot
• Reduce memory consumption
• Migrate compiled code to other nodes
Purpose of pre-compilation

Goal of this time

- Fast boot
- Reduce memory consumption
- Migrate compiled code to other nodes

Out of scope

No portable binary support
No verification at loading time

[Because we can’t not trust binaries by others]
Goal: Fast boot

Pre-compilation utility
Parse/compile Ruby scripts
Compiled binary
Load

Interpret on RubyVM

Your application code
Bundled Libraries
Gem Libraries

Parse/compile Ruby scripts
Compiled binary
Load

Skip

Evaluate

Embedded classes and methods
Evaluator

Bundled Libraries
Gem Libraries

Your application code

Interpret on RubyVM
Goal: Memory consumption

Current issue

ISeq consumes 15% (20MB) on simple Rails app

<table>
<thead>
<tr>
<th>Function</th>
<th>Memory Consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="mailto:iseq_setup@compile.c">iseq_setup@compile.c</a></td>
<td>15,595,764</td>
</tr>
<tr>
<td><a href="mailto:rb_iseq_new_with_opt@iseq.c">rb_iseq_new_with_opt@iseq.c</a></td>
<td>5,231,136</td>
</tr>
<tr>
<td><a href="mailto:heap_assign_page@gc.c">heap_assign_page@gc.c</a></td>
<td>40,518,400</td>
</tr>
<tr>
<td><a href="mailto:st_init_table_with_size@st.c">st_init_table_with_size@st.c</a></td>
<td>18,994,480</td>
</tr>
<tr>
<td><a href="mailto:rb_str_buf_new@string.c">rb_str_buf_new@string.c</a></td>
<td>4,817,252</td>
</tr>
<tr>
<td><a href="mailto:st_update@st.c">st_update@st.c</a></td>
<td>6,578,736</td>
</tr>
<tr>
<td><a href="mailto:onig_region_resize@regexec.c">onig_region_resize@regexec.c</a></td>
<td>4,891,968</td>
</tr>
<tr>
<td>others</td>
<td>37,676,810</td>
</tr>
</tbody>
</table>
Purpose: Memory consumption
Current issue on multi-processes

Actual

- Process
- Independent BCs

Expected

- Process
- (Partially) Shared BCs
- Shared Bytecode Data
Design and implementation of primitives on Ruby 2.3
We need two components

1. Serializer and deserializer for ISeq

2. Utility to control AOT compilation
   • When to compile scripts and load them?
   • Where/How to store compiled binaries?
Serializer and deserializer of ISeq

1. **Read, parse and compile**: Ruby script
2. **Pre-compilation**: Ruby process
3. **Serialize and store**: ISeq
4. **Compiled binary**: Compiled binary
5. **Load and deserialize**: ISeq
ISeq is a tree

- Basically, each scope has own ISeq
  - A top-level has class expressions
  - Class expression has method definitions
  - Method definition has blocks
  - Block has blocks, ...
  - Other bytecode blocks
    - ensure, rescue, ...
  - And other exceptional cases
Specify compiled binary data format

- Iseq (BC), ID, Objects are pointed by index of each lists in each data
- Referred objects are serialized
- **Dump machine dependent data (can’t migrate compiled code)**
- No verifier (because this file is not for migrations)
Optimization technique
Lazy loading
Lazy loading

• Do not load all of ISeq at once
  • Load ISeq if needed
  • Similar to “autoload” method
Technique
Lazy loading

(1) Load and make an empty toplevel ISeq

Ruby script

class C1
  def m1; end
  def m2; end
end

C1.new.m2

class C2; end

Compiled binary
toplevel
C1, C2,
m1, m2

Toplevel (empty)
### Technique

#### Lazy loading

(2) Load toplevel ISeq and make empty C1, C2 ISeqs and evaluate toplevel ISeq

**Ruby script**

```ruby
class C1
def m1; end
def m2; end
end
class C2; end
C1.new.m2
```

**Compiled binary**

- class C1 (empty)
- class C2 (empty)

**Toplevel**
Technique
Lazy loading

(3) Load C1 and evaluate C1
Define m1 and m2 with empty ISeqs

Ruby script

class C1
  def m1; end
  def m2; end
end
C1.new.m2

class C2; end
Technique
Lazy loading

(4) Load m2 and invoke m2

Ruby script
class C1
def m1; end
def m2; end
end

class C2; end

class C1

toplevel
C1,C2,
m1,m2

class C1

Toplevel

class C2 (empty)
def m1 (empty)
def m2
Technique
Lazy loading

(4) Load C2 and evaluate C2

Ruby script
class C1
def m1; end
def m2; end
end
C1.new.m2
class C2; end

Compiled binary
toplevel
C1,C2,
m1,m2

class C1
def m1 (empty)
class C2
def m2

Toplevel
Interface
API and Tools
How to store compiled binary?

• Compile timing
  • Use compiler explicitly
    • C/Java/... compilers
  • Loading time
    • Rubinius (*.rbc), Python (*.pyc), ...

• Location of compiled binary
  • A file in the same directory of *.rb files
  • A file in a special directory
  • DB

So many options!
Current (our) solution
Provides primitive APIs
• Serialize and de-serialize APIs
• Loading API

You can try to make your own pre-compilation controller
Current implementation

Primitive APIs

• Serialize and de-serialize APIs
  • RubyVM::InstructionSequence#to_binary
  • RubyVM::InstructionSequence.load_from_binary(binary)

• Loading API
  • RubyVM::InstructionSequence.load_iseq
    • Call this method at every loading time (if defined)
    • This method should return nil or loaded ISeq
Store serialized program and load

1. Read, parse and compile Ruby script
2. Serialize and store
3. Pre-compilation
4. Load and deserialize
5. Compiled binary
6. ISeq
Using ISeq.load_iseq

require/load process

File name (“x.rb”)
Load_internal(fname)
Read, parse and compile script named fname (x.rb)
ISeq of x.rb

New require/load process

File name (“x.rb”)
Load_internal(fname)
Call ISeq.load_iseq(fname)
Read, parse and compile script named fname (x.rb)
ISeq of x.rb
ISeq.load_iseq(fname)
Can’t load
Load and return iseq
Current implementation
APIs (again)

• Serialize and de-serialize APIs
  • RubyVM::InstructionSequence#to_binary
  • RubyVM::InstructionSequence.load_from_binary(binary)

• Loading API
  • RubyVM::InstructionSequence.load_iseq
    • Call this method at every loading time (if defined)
    • This method should return nil or loaded ISeq
Yomikomu.gem

Sample implementation of pre-compilation controller
When should we compile?

• Compile timing
  • Invoke a compiler explicitly
    • C/Java/... compilers
    • Invoke during gem installation is a good idea
  • Loading time (if not available, compile automatically)
    • Python (.pyc), Rubinius (.rbc)
Where to store?

• Make compiled binary files for each script?
• Store compiled binaries in one DB?

Store compiled binary in the same directory
/a/b/x.rb, x.rb.yarb
  y.rb, y.rb.yarb
  c/z.rb, z.rb.yarb

Store compiled binary in the specified directory
/a/b/x.rb, y.rb
  c/z.rb
/repos/a_b_x.rb.yarb
  a_b_y.rb.yarb
  a_c_z.rb.yarb

Store into DB
/a/b/x.rb
  Binary of x.rb
/a/b/y.rb
  Binary of y.rb
/a/c/z.rb
  Binary of z.rb

(Python and Rubinius do)
Where to store?

BTW, Matz doesn’t like storing binaries in same dir because he want to keep src dir clean.

Store compiled binary in the same directory
/a/b/x.rb, x.rb.yarb
  y.rb, y.rb.yarb
  c/z.rb, z.rb.yarb

Store compiled binary in the specified directory
/a/b/x.rb, y.rb
  c/z.rb
/repos/a_b_x.rb.yarb
  a_b_y.rb.yarb
  a_c_z.rb.yarb

(Python and Rubinius do)

Store into DB
/a/b/x.rb
  Binary of x.rb
/a/b/y.rb
  Binary of y.rb
/a/c/z.rb
  Binary of z.rb
Sample implementation
Yomikomu.gem

• “Yomikomu” = “読み込む” = “loading/reading”
• Implement many options
Usage of Yomikomu

3 steps

(1) **Set configuration** with environment variables
   • Storage options and so on. See documents for details

(2) **Compile Ruby scripts** with “kakidasu” command
   • “kakidasu” = “書き出す” = “write/output”
   • `$ kakidasu [script or directory]`

(3) **put “require ‘yomikomu’”** on your application
   • Compiled binaries are loaded automatically
Configuration
Yomikomu supports several storages

• **YOMIKOMU_STORAGE** specifies how and where to store and load compiled binaries
  
  • fs (default)
  • fs2
  • fsgz
  • Fs2gz
  • dbm
  • flatfile
Configuration

Yomikomu supports 4 basic storages

- **fs**: put compiled binary files on same directory
- **fs2**: put compiled binary files on one directory
- **dbm**: put compiled binaries on one DB (dbm)

---

**fs**

Store compiled binary in the same directory

- `/a/b/x.rb, x.rb.yarb`
- `/y.rb, y.rb.yarb`
- `/c/z.rb, z.rb.yarb`

(Python and Rubinius do)

**fs2**

Store compiled binary in the specified directory

- `/a/b/x.rb, y.rb`
- `/c/z.rb`
- `/repos/a_b_x.rb.yarb`
  - `/a_b_y.rb.yarb`
  - `/a_c_z.rb.yarb`

**dbm**

Store into DB

- `/a/b/x.rb`
  - Binary of x.rb
- `/a/b/y.rb`
  - Binary of y.rb
- `/a/c/z.rb`
  - Binary of z.rb
Configuration
Yomikomu supports 4 basic storages

• flatfile: put compiled binaries into one file sequentially (and make index)
• 😊 we can locate binaries in loading order
• 😞 it does not support rewriting
Configuration
Yomikomu supports compactions

• Store Gzip compressed compiled binary
  • fsgz, fs2gz, flatfilegz
Configuration
Yomikomu supports auto compilation

• YOMIKOMU_AUTO_COMPILE
  • If required script is not compiled, compile it and store to somewhere automatically
  • Similar to Python and Rubinius
  • You don’t need to use “kakidasu” command
Demonstration

(if I have time...
Evaluation
Evaluation

- Measure loading time of same script 1,000 times
  - Use `remove_const` to cleanup each loading
  - Choose from `lib/* .rb`

<table>
<thead>
<tr>
<th>Target script</th>
<th>Lines</th>
<th>Size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>resolv.rb</td>
<td>2,855</td>
<td>73</td>
</tr>
<tr>
<td>csv.rb</td>
<td>2,346</td>
<td>83</td>
</tr>
<tr>
<td>fileutils.rb</td>
<td>1,761</td>
<td>48</td>
</tr>
<tr>
<td>forwardable.rb</td>
<td>290</td>
<td>8</td>
</tr>
</tbody>
</table>
**Evaluation**

**Loading time (x1,000)**

<table>
<thead>
<tr>
<th></th>
<th>Normal (sec)</th>
<th>Load (sec)</th>
<th>Lazy load (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>resolve.rb</td>
<td>13.19</td>
<td>3.92 (x3.36)</td>
<td>2.42 (x5.45)</td>
</tr>
<tr>
<td>csv.rb</td>
<td>7.88</td>
<td>4.19 (x1.88)</td>
<td>2.85 (x2.76)</td>
</tr>
<tr>
<td>fileutils.rb</td>
<td>8.55</td>
<td>4.64 (x1.84)</td>
<td>3.61 (x2.37)</td>
</tr>
<tr>
<td>forwardable.rb</td>
<td>0.48</td>
<td>0.18 (x2.67)</td>
<td>0.12 (x4.00)</td>
</tr>
</tbody>
</table>

😊 5 times faster on resolv.rb seems good

😢 Nobody load resolv.rb 1,000 times
## Evaluation

### Compiled binary size

<table>
<thead>
<tr>
<th>Target script</th>
<th>Lines</th>
<th>Script size (KB)</th>
<th>Binary size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>resolv.rb</td>
<td>2,855</td>
<td>73</td>
<td>337 (x4.6)</td>
</tr>
<tr>
<td>csv.rb</td>
<td>2,346</td>
<td>83</td>
<td>170 (x2.0)</td>
</tr>
<tr>
<td>fileutils.rb</td>
<td>1,761</td>
<td>48</td>
<td>202 (x4.2)</td>
</tr>
<tr>
<td>forwardable.rb</td>
<td>290</td>
<td>8</td>
<td>14 (x1.7)</td>
</tr>
</tbody>
</table>
Evaluation
Rails launch time
• Loading time of Redmine 3.2.1 (rails app)
  • $ bundle exec rails r "p:success"
  • YOMIKOMU_STORAGE=fs

<table>
<thead>
<tr>
<th>Execution time</th>
<th>Normal (sec)</th>
<th>Use Yomikomu (sec)</th>
<th>Use Yomikomu w/ lazy loading (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>w/o bundle</td>
<td>2.65</td>
<td>2.22 (x1.19)</td>
<td>2.03 (x1.31)</td>
</tr>
<tr>
<td>w/ bundle</td>
<td>2.94</td>
<td>2.45 (x1.20)</td>
<td>2.24 (x1.31)</td>
</tr>
</tbody>
</table>
Evaluation
Compare only loading time

- Check the (load file + parse + compile) time and corresponding (load file + deserializing) time
  - `YOMIKOMU_STORAGE=fs`

<table>
<thead>
<tr>
<th>Loading time</th>
<th>Normal: load file + parse + compile (sec)</th>
<th>Use Yomikomu: deserialize (sec)</th>
<th>Use Yomikomu w/ lazy loading (sec) (*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>w/o bundle</td>
<td>0.87 (33% of exec)</td>
<td>0.43 (x2.02)</td>
<td>0.23 (x3.78)</td>
</tr>
</tbody>
</table>

(*) Does not contain actual lazy loading time
Evaluation
Loading (parse & compile) overhead

- Normal
- Yomikomu
- Yomikomu + Lazy

- Load time
- Execution time

- 😊 4 times faster!
- 😞 No drastic reduction
Evaluation
Rails launch time w/ flatfile

- Loading time of Redmine 3.2.1 (rails app)
  - `$ bundle exec rails r "p:success"`
  - `YOMIKOMU_STORAGE=flatfile`

<table>
<thead>
<tr>
<th>Execution time</th>
<th>Normal (sec)</th>
<th>Use Yomikomu (sec)</th>
<th>Use Yomikomu w/ lazy loading (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>w/o bundle</td>
<td>2.65</td>
<td>2.11 (x1.26)</td>
<td>2.05 (x1.29)</td>
</tr>
<tr>
<td>w/ bundle</td>
<td>2.94</td>
<td>2.46 (x1.20)</td>
<td>2.45 (x1.20)</td>
</tr>
</tbody>
</table>
Evaluation

Compare loading time w/ flatfile

- Check the (load file + parse + compile) time and corresponding (load file + deserializing) time
  - `YOMIKOMU_STORAGE=flatfile`

<table>
<thead>
<tr>
<th>Loading time</th>
<th>Normal: parse + compile (sec)</th>
<th>Use Yomikomu: deserialize (sec)</th>
<th>Use Yomikomu w/ lazy loading (sec) (*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>w/o bundle</td>
<td>0.87</td>
<td>0.43 (x2.02)</td>
<td>0.22 (x3.95)</td>
</tr>
</tbody>
</table>

(*) Does not contain actual lazy loading time
Future work

• Reduce memory consumption by memory sharing with mmap (and so on)
• Reduce binary size with some techniques
  • Smart serialization technique
  • Compaction technique
• And more…
Today’s talk was about:

• New feature of Ruby 2.3
  “Pre-compilation primitives”
• Yomikomu gem: what is and how to use it.
• Evaluation results includes redmine boot time
Myth

“If we have an AOT compiler, the boot time issue will be solved”
Fact

“The world is not so easy”
Message

“Please enjoy making your own Yomikomu utility”
Thank you for your attention

Koichi Sasada  
<ko1@heroku.com>