What’s happening in your Rails app?

Introduction to

Introspection features of Ruby

Koichi Sasada

ko1@heroku.com
Short summary of this presentation
What's happening in your Rails app?

• You can use many existing tools to inspect your Rails application.

• Ruby/MRI provides many low-level features to make tools.
Koichi Sasada is a Programmer

• MRI committer since 2007/01
  • Original YARV developer since 2004/01
    • YARV: Yet Another RubyVM
    • Introduced into Ruby (MRI) 1.9.0 and later
  • Introduce generational/incremental GC

• Not a Rails programmer 😞
  • C language programmer
  • My wife is a professional Rails dev 😊
Koichi is a member of Heroku Matz team

Mission

Design Ruby language and improve quality of MRI

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

Matz
Designer/director of Ruby

Nobu
Quite active committer

Ko1
Internal Hacker
Matz
Title collector

- He has so many (job) title
  - Chairman - Ruby Association
  - Fellow - NaCl
  - Chief architect, Ruby - Heroku
  - Research institute fellow – Rakuten
  - Chairman – NPO mruby Forum
  - Senior researcher – Kadokawa Ascii Research Lab
  - Visiting professor – Shimane University
  - Honorable citizen (living) – Matsue city
  - Honorable member – Nihon Ruby no Kai
  - ...

- This margin is too narrow to contain
Nobu
Great Patch monster

Ruby’s bug
|> Fix Ruby
|> Break Ruby
|> And Fix Ruby
Commit count of MRI

Nobu 29%
akr 12%
svn 9%
naruse 8%
usa 4%
ko1 4%
derbrain 3%
drbrain 3%
ko1 3%
ussuru 4%
ussuru 4%
ussuru 8%
ussuru 9%
ussuru 9%
ussuru 9%
ussuru 9%
Ko1
EDD developer

 Commit number of ko1 (last 3 years)

EDD: Event Driven Development
Heroku Matz team and Ruby core team
Recent achievement

Ruby 2.2

http://www.flickr.com/photos/loginesta/5266114104

Current stable
Ruby 2.2 Syntax

• Symbol key of Hash literal can be quoted

```ruby
{“foo-bar”: baz}
#=> {"foo-bar" => baz}
#=> not {“foo-bar” => baz} like JSON
```

TRAP!!
Easy to misunderstand
(I wrote a wrong code, already...)
Ruby 2.2
Classes and Methods

• Some methods are introduced
  • Kernel#itself
  • String#unicode_normalize
  • Method#curry
  • Binding#receiver
  • Enumerable#slice_after, slice_before
  • File.birthtime
  • Etc.nprocessors
  • ...

Ruby 2.2

Improvements

• Improve GC
  • Symbol GC
  • Incremental GC
  • Improved promotion algorithm
    • Young objects promote after 4 GCs

• Fast keyword parameters

• Use frozen string literals if possible
before = Symbol.all_symbols.size
1_000_000.times{|i| i.to_s.to_sym} # Make 1M symbols
after = Symbol.all_symbols.size; p [before, after]

# Ruby 2.1

 #=> [2_378, 1_002_378] # not GCed 😞

# Ruby 2.2

 #=> [2_456, 2_456] # GCed! 😊
Ruby 2.2
Symbol GC (cont.)

• Upgrade Ruby 2.2.2
  • Memory (object) leak problem
    • Symbols has corresponding String objects
    • Symbols are collected, but Strings are not collected! (leak)
  • Ruby 2.2.1 solved this problem!!
    • However, 2.2.1 also has problem (rarely you encounter BUG at the end of process [Bug #10933] ← not big issue, I want to believe)
  • Finally Ruby 2.2.2 had solved it!!

TRAP!!
Ruby 2.2.0 has memory leak error!

Ruby 2.2
Fast keyword parameters

“Keyword parameters” introduced in Ruby 2.0 is useful, but slow!!

Ruby 2.2
Fast keyword parameters

Ruby 2.2 optimizes method dispatch with keyword parameters

<table>
<thead>
<tr>
<th>Function</th>
<th>Execution Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>foo6(1, 2, 3, 4, 5, 6)</code></td>
<td>15</td>
</tr>
<tr>
<td><code>foo_kw6(k1: 1, k2: 2, k3: 3, k4: 4, k5: 5, k6: 6)</code></td>
<td>1</td>
</tr>
</tbody>
</table>

Repeat 10M times

- `foo6(1, 2, 3, 4, 5, 6)` in Ruby 2.1 takes 15 seconds.
- `foo_kw6(k1: 1, k2: 2, k3: 3, k4: 4, k5: 5, k6: 6)` in Ruby 2.2 takes 1 second.

x14 faster!!

But still x2 times slower compared with normal dispatch.

Ruby 2.2
Incremental GC

<table>
<thead>
<tr>
<th>Goal</th>
<th>Before Ruby 2.1</th>
<th>Ruby 2.1 RGenGC</th>
<th>Incremental GC</th>
<th>Ruby 2.2 Gen+IncGC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Pause time</td>
<td>Long</td>
<td>Long</td>
<td>Short</td>
<td>Short</td>
</tr>
</tbody>
</table>
RGenGC from Ruby 2.1: Micro-benchmark

![Bar chart comparing time (ms) for 'no' and 'RGenGC' with 'total mark' and 'total sweep' categories. The chart shows 'no' has a total mark of 1699.805974 ms and a total sweep of 704.843669 ms, while 'RGenGC' has a total mark of 867.740319 ms and a total sweep of 87.230735 ms.]

x2.5 faster
RGenGC from Ruby 2.1: Pause time

Most of cases, FASTER 😊
RGenGC from Ruby 2.1: Pause time

Several peaks 😞

GC pause time (sec)

1. pause time (raw)
2. pause time (rgengc) (w/o rgengc)

Ruby 2.2 Incremental GC

Short pause time 😊

---

Ruby 2.3

Next version will be released the end of this year

Catch me and give me your feedback later

http://www.flickr.com/photos/adafruit/8483990604
What’s happening in your Rails app?

Introduction to

Introspection features of Ruby

Today’s topic

#駄洒落クラブ
#駄洒落クラブ (DAJARE Club)

- DAJARE is something like PUN in Japanese
- #駄洒落クラブ is associated by our HERO Aaron Patterson @tenderlove
- Many (Japanese) Rubyists have joined
- Recommendation: Join our club if you are learning Japanese
#駄洒落クラブ (DAJARE Club)

@_ko1 @_ko1 • 2014年3月25日

my first gem: rubygems.org/gems/gc_tracer

Aaron Patterson
@tenderlove

@_ko1 that's quite a gem! #駄洒落クラブ

11:16 - 2014年3月25日
Introspection features of Ruby for MRI
Launch your application on the Layered computer structure

Your Rails app

Framework / Libraries / Tools

Ruby interpreter

Operating System

Hardware

Launch your application on the Layered computer structure

Your Rails app
Framework / Libraries / Tools
Ruby interpreter
Operating System
Hardware

Black box

Launch your application on the Layered computer structure.

- Your Rails app (3 days later)
- Framework / Libraries / Tools
- Ruby interpreter
- Operating System
- Hardware

Trouble will come from a black box

- Slow requests...
- Memory consuming...

- Unexpected values...
- Difficult debugging...

https://www.flickr.com/photos/wingedwolf/5471047557/
Layered computer structure
How to inspect your Rails application?

- Your Rails app
- Framework / Libraries / Tools
- Ruby interpreter
- Operating System
- Hardware
1. Use existing tools
2. Make your own tools
1. Use existing tools

- Your Rails app
- Framework / Libraries / Tools
- Ruby interpreter

Tools & Services

Operating System

Hardware
Great presentations at RailsConf2015

“Prying Open The Black Box” by Godfrey Chan

“Speed Science” by Richard Schneeman

Keynote: Aaron Patterson

...
Performance issue

Slow requests...
Memory consuming ...
The easiest way to solve performance issue is...

Use PX: Performance dyno on Heroku
(or another high-performance machine)

<table>
<thead>
<tr>
<th>Dyno Size</th>
<th>Memory (RAM)</th>
<th>CPU Share</th>
<th>Multitenant</th>
<th>Compute</th>
<th>Price/dyno-hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>1X</td>
<td>512MB</td>
<td>1x</td>
<td>yes</td>
<td>1x-4x</td>
<td>$0.05</td>
</tr>
<tr>
<td>2X</td>
<td>1024MB</td>
<td>2x</td>
<td>yes</td>
<td>4x-8x</td>
<td>$0.10</td>
</tr>
<tr>
<td>PX</td>
<td>6GB</td>
<td>100%</td>
<td>no</td>
<td>40x</td>
<td>$0.80</td>
</tr>
</tbody>
</table>

https://devcenter.heroku.com/articles/dyno-size
https://blog.heroku.com/archives/2014/2/3/heroku-xl
Performance issue
We need to know what happen in a black box

<table>
<thead>
<tr>
<th>Slow requests</th>
<th>Memory consuming</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Which part is slow?</td>
<td>• Who is consuming memory?</td>
</tr>
<tr>
<td>• DB?</td>
<td>• Correct GC Parameter?</td>
</tr>
<tr>
<td>• External API access?</td>
<td></td>
</tr>
<tr>
<td>• Your Ruby app?</td>
<td></td>
</tr>
<tr>
<td>• Application?</td>
<td></td>
</tr>
<tr>
<td>• GC?</td>
<td></td>
</tr>
</tbody>
</table>
Analyze slow requests
Use New Relic

http://newrelic.com/application-monitoring/features
Analyze slow requests
Use New Relic

• “Dive into Ruby VM Stats with New Relic”
  http://blog.newrelic.com/2014/04/23/ruby-vm-stats/

• “Ruby VM measurements”
  https://docs.newrelic.com/docs/ruby/ruby-vm-stats

IMPORTANT
You can use New Relic very easily on Heroku as an Add-on
Analyze slow requests
Performance profilers

“Debugging Ruby Performance” by Aman Gupta
• https://speakerdeck.com/tmm1/debugging-ruby-performance

“Ruby 2.1 in Production” by Aman Gupta
• http://rubykaigi.org/2014/presentation/S-AmanGupta
Analyze memory consumption issues

Background

• Ruby has GC
  • Automatic object recycling and memory management
  • Ruby 2.2 has “Incremental and generational” GC
    • https://engineering.heroku.com/blogs/2015-02-04-incremental-gc

• Issues
  • Incorrect GC parameters (environment variables)
  • Object leaking
  • MRI bugs
Analyze memory consumption issues

Background

• Generational GC (from Ruby 2.1)
  • Collect only newer objects
  • Old objects can remain long time unexpectedly

→ Object leaking
Analyze memory consumption issues

Tools (pickup my original two)

Measure GC statistics

• gc_tracer.gem

Find object creation locations

• allocation_tracer.gem
Analyze memory consumption issues
GC Tracer in your app

• Require and call “GC::Tracer.start_logging(filename)”
  • You can specify logging filename by env val “GC_TRACER_LOGFILE”
• All GC related information are recorded into “filename” (or STDERR)
Analyze memory consumption issues
Allocation Tracer in your app

# Source code

```ruby
require 'allocation_tracer'
require 'pp'

ObjectSpace::AllocationTracer.setup(%i{path line class})

pp ObjectSpace::AllocationTracer.trace{
  50_000.times{|i|
    i.to_s
    i.to_s
    i.to_s
  }
}
```

# result

```ruby
{["t.rb", 7, String]=>[50000, 1, 40960, 0, 6, 1745520],
["t.rb", 8, String]=>[50000, 0, 40950, 0, 1, 1745560],
["t.rb", 9, String]=>[50000, 1, 40963, 0, 6, 1745480]}
```
GC/Allocation tracer
Use as Rack middleware

You only need to write the following lines

# Sample in config.ru

# GC tracer
require 'rack/gc_tracer'
use Rack::GCTracerMiddleware, view_page_path: '/gc_tracer', filename: '/tmp/rails-gc_tracer'

# Allocation Tracer
require 'rack/allocation_tracer'
use Rack::AllocationTracerMiddleware

Warning: You should not use Allocation Tracer in production because it has big overhead
Demo App

http://atdot.net/ab/
GC Tracer
Events

• Capture information on every GC related events
  • Start marking, start sweeping, end sweeping (default events)
  • Enter GC, exit GC, newobj, freeobj (optional events)
GC Tracer
Captured information

• GC.stat results (statistics for current GC)
  • https://docs.google.com/spreadsheets/d/11Ua4uBr6o0k-nORrZLEIIUkHJ9JRzRR0NyZfrhEEnc8/edit?usp=sharing

• GC. latest_gc_info

• Result of get_gusage (on supported OSs)

• Custom fields
  • You can add any numbers
  • “Access number” field on Rack mode
GC Tracer
How to see the log

• “Debugging memory leaks in Ruby” by Sam Saffron
  • http://samsaffron.com/archive/2015/03/31/debugging-memory-leaks-in-ruby

• Send me ko1@heroku.com your gc_tracer log
Allocation tracer

• Object allocation status for each line
  • # of created objects
  • # of old objects
  • Total ages (total ages / # of created objects = the average age)
  • Minimum / maximum ages
  • Consumed memory bytes

• # of created objects helps to find out newobj overheads
• # of old objects helps to find out unexpected long lifetime objects
• A bit slow (tracking all object creation and recycling)
Unknown behavior
Tools to reveal unknown behaviors

• Debugger
  • pry
  • Byebug
  • ...

• Error messages
  • better_errors
  • did_you_mean
  • pretty_backtrace
  • …
“Did you mean?” gem
https://github.com/yuki24/did_you_mean

class User
  attr_accessor :first_name, :last_name
  def to_s; "#{first_name} #{last_name}"; end
end

user.to_s

# => NameError: undefined local variable or method `f1rst_name' for
#<User:0x0000000928fad8>

# Did you mean? #first_name

#
“Pretty backtrace” gem

https://github.com/ko1/pretty_backtrace

# Source code

```ruby
require 'pretty_backtrace/enable'

def fib n
  n < 1 ? raise : fib(n-1)
End

fib(3)
```

# Error message

test.rb:5:in `fib' (n = 0): unhandled exception

from test.rb:5:in `fib' (n = 1)

from test.rb:5:in `fib' (n = 2)

from test.rb:5:in `fib' (n = 3)

from test.rb:8:in `<main>,'

No suitable tools.
How to do it?
2. Make your own tools

You can make it!

Your Rails app
Framework / Libraries / Tools
Ruby interpreter
Ruby provides primitives

Operating System

Hardware
Low level Ruby APIs

- TracePoint
  - Hook events
- Exception
  - Add your favorite information to error messages
- Reflection features
- debug_inspect gem
- And more…
Easy tracking with TracePoint
Example: Where is the “#index” method called?

# insert this part of code into boot code
TracePoint.new(:call) do |tp|
  if tp.method_id == :index
    pp [tp, caller_locations]
  end
end
end.enable
Modify exceptions
Example: Modify raised exception backtraces with ♥

```
# src
TracePoint.new(:raise){ |tp|
  e = tp.raised_exception
  e.set_backtrace(  
    e.backtrace.map{|line|  
      line + "<3<3<3"}
  ).enable{
    raise
  }
}

# result
try: in `block in <main>'<3<3<3:  
  unhandled exception
    from try: in `enable'<3<3<3
    from try: in `<main>'<3<3<3
```
Reflection API
Example: Getting local variable name

<table>
<thead>
<tr>
<th># src</th>
<th># result</th>
</tr>
</thead>
</table>
| def foo of: STDOUT, if: STDIN
  binding.local_variables.each{|lv|
    p [lv, binding.local_variable_get(lv)] }
  # BTW: how to get the value of “if”? |
| [:of, #<IO:<STDOUT>>]
[:if, #<IO:<STDOUT>>] |
Debug Inspect gem
Example: Getting bindings for all frames

# source code

```ruby
require 'debug_inspector' # gem

def local_variables_for_frames
  RubyVM::DebugInspector.open{|dc|
    dc.backtrace_locations.size.times{|i|
      p dc.frame_binding(i).local_variables if dc.frame_binding(i)
    }
  }
end

def fib n; n < 1 ? local_variables_for_frames : fib(n-1); end

fib(3)
```

# result

```ruby
[]
[:n]
[:n]
[:n]
[:n]
[]
```
Combination of techniques

• Examples
  • Track “raise” events
  • Modify backtrace
  • Get local variables from binding
  • Get bindings for each method frames

Combining all examples makes “Pretty Backtrace” gem
Advanced hacking

Escape from Ruby world
Make C extensions with low level C APIs

• See “ruby/debug.h”
  • TracePoint with internal events
  • rb_profile APIs
  • rb_debug_inspect APIs
  • GC inspection APIs
Hack Ruby

• “Ruby Under a Microscope”
  • By Pat Shaughnessy
  • http://patshaughnessy.net/ruby-under-a-microscope

• “Ruby Hacking Guide”
  • By Minero Aoki, written in Japanese
  • English translation: http://ruby-hacking-guide.github.io/
Combine with low level tools

Your Rails app

Framework / Libraries / Tools

Ruby interpreter

Operating System

Hardware

gdb

perf (linux)

valgrind

...
Hacking low level systems

Hack here!!

Your Rails app
Framework / Libraries / Tools
Ruby interpreter

Operating System

Hardware

Rails programming is fun.

Low-level programming is also fun!

(sometimes)
Message of this talk

You can introspect your Rails application with existing tools

• So many efforts on this purpose
• Please find them

You can make your own inspection tools for your Rails application

• Recent Ruby/MRI will help you
• I can help you, too
Thank you for your attention

Koichi Sasada
<k01@heroku.com>