Lightweight Method Dispatch on MRI

Koichi Sasada
<ko1@heroku.com>

EURUKO2015
Koichi Sasada

A programmer from Japan
Koichi 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
• Generational/incremental GC for 2.x
“Why I wouldn’t use rails for a new company” by Jared Friedman

“The ruby interpreter is just a volunteer effort. Between 2007-2012, there were a number of efforts to fix the interpreter and make it fast (Rubinius, Jruby, YARV, etc) But lacking backers with staying power, the volunteers got bored and some of the efforts withered. JRuby is still active and recent versions are showing more promise with performance, but it’s been a long road.”

Koichi is an Employee
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”
<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matz</td>
<td>Designer/director of Ruby</td>
</tr>
<tr>
<td>Nobu</td>
<td>Quite active committer</td>
</tr>
<tr>
<td>Ko1</td>
<td>Internal Hacker</td>
</tr>
</tbody>
</table>
Matz
Title collector

• He has so many (job) title
  • Japanese teacher
  • Chairman - Ruby Association
  • Fellow - NaCl
  • Chief architect, Ruby - Heroku
  • Research institute fellow – Rakuten
  • Chairman – NPO mruby Forum
  • Senior researcher – Kadokawa Asciii 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
Nobu
Patch monster

Commit count of MRI
Nobu
The Ruby Hero
Ko1
EDD developer

Commit number of ko1 (last 3 years)

- RubyConf 2013
- RubyKaigi 2013
- Euruko 2013
- RubyConf 2012
- Ruby 2.0
- RubyConf 2013

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

Ruby 2.2

Current stable
Ruby 2.2
Syntax

• Symbol key of Hash literal can be quoted

{“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 introduces
  • Kernel#itself
  • String#unicode_normalize
  • Method#curry
  • Binding#receiver
  • Enumerable#slice_after, slice_before
  • File.birhtime
  • 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
Ruby 2.2
Symbol GC

```ruby
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 Issues history

• **Ruby 2.2.0** has 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)

• **Ruby 2.2.2** had solved [Bug #10933]!!
  • However, patch was forgot to introduce!!

• Finally, **Ruby 2.2.3** solved it!!
  • Please use newest version!!
Ruby 2.2
Fast keyword parameters

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

Evaluation on Ruby 2.1

Repeat 10M times

foo6(1, 2, 3, 4, 5, 6)
foo_kw6(k1: 1, k2: 2, k3: 3, k4: 4, k5: 5, k6: 6)

x30 slower
Ruby 2.2
Fast keyword parameters

Ruby 2.2 optimizes method dispatch with keyword parameters

Execution time (sec)

Repeat 10M times

foo6(1, 2, 3, 4, 5, 6)
foo_kw6(k1: 1, k2: 2, k3: 3, k4: 4, k5: 5, k6: 6)

x14 faster!!

But still x2 times slower
compare with normal dispatch
# Ruby 2.2

Incremental GC

<table>
<thead>
<tr>
<th></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>

**Goal**
RGenGC from Ruby 2.1: Micro-benchmark

The diagram shows the time taken for mark and sweep operations with and without RGenGC. Without RGenGC, the total time is 1699.805974 ms, and with RGenGC, it is 867.740319 ms, which is approximately 2.5 times faster.
RGenGC from Ruby 2.1: Pause time

Most of cases, FASTER 😊

(w/o rgengc)
RGenGC from Ruby 2.1: Pause time

Several peaks 😞

(w/o rgengc)
Ruby 2.2 Incremental GC

Short pause time 😊
Heroku Matz team and Ruby core team
Next target is

Ruby 2.3
New magic comment:
Frozen string literal

# -*- frozen-string-literal: true -*-
p 'foo'.frozen? #=> true

# There are many discussion.
# Please join us.
Break
Ruby has so many

Let’s play hangman game
Ruby has so many

*   *   *   *
Ruby has so many

F*****
Ruby has so many FU**
Ruby has so many

FU***
Ruby has so many functions or methods.
How many function/method call?

Importance of optimization for “function/method dispatch”
Easy way to measure method dispatch count

# at the beginning of your application
$c = 0; TracePoint.trace(:call, :a_call){$c+=1}
END{puts “call: #$c”}

# and your app...
Measuring method dispatch counts

- RDoc application
  - Make RDoc documents from Ruby’s source
  - $120M \approx 100M$ dispatches in 60 sec

- Tak(20, 10, 0) benchmark using recursive calls
  - Famous benchmark
    ```ruby
    def tak x, y, z
      y < x ? tak(tak(x-1, y, z), tak(y-1, z, x), tak(z-1, x, y)) : z
    end
    ```
  - $100M$ dispatches in 4.5 sec
Ruby has so many Functions or Methods
Execution time of method

1 Method call execution

Method dispatch overhead

Method body (your code)

Today’s topic
100M method dispatches
Estimation

- 1sec/method dispatch  #=> 100M sec  => about 3 years
- 1msec/method dispatch  #=> 100M ms  => about 1 days
- 1usec/method dispatch  #=> 100M us  => 100 sec
  - 1us is 3000 clocks on 3GHz CPU
- 10nsec/method dispatch  #=> 1,000M ns  => 1sec
  - 10ns is 30 clocks on 3GHz CPU
- 1nsec/method dispatch  #=> 100M nsec  => 0.1 sec
  - 1ns is 3 clocks on 3GHz CPU
Matter or not matter

• 1 sec method dispatch overhead in 60 sec application (rdoc)
  #=> doesn’t matter

• 1 sec method dispatch overhead in 4 sec application (tak)
  #=> big concern

• Maybe most of applications are located between these two applications
  • RDoc app has complex methods, so that dispatch cost is not a matter
  • Tak app has a simple method, so that dispatch cost slows app directly
BTW
1sec / method dispatch

# CAUTION:
# Do not insert this line in your friends’ application

TracePoint.trace(:call){sleep 1}
Requirements
Revisit Ruby’s method dispatch
# Simple method call

def simple_foo(x)
    ...
end

foo(123)
# Complex method call

protected # visibility
def complex_foo(m1, m2, o1=1, o2=2, *r, p1, p2, k1: 1, k2: 2, kr:, **kw, &b)
  ... # body
end
...
complex_foo(v1, v2, *a1, v3, v4, *a2, k1: 1, k2: 2, kr: 3, **kw, &block)

# RubyQuiz: can you explain everything?
Complex method dispatch
Caller side

• Normal arguments: m(v1, v2)
• Splatting arguments: m(*a1, *a2)
• Block argument: m(&block)
• Keyword arguments: m(k1: v1, k2: v2)
• Combination: m(v1, v2, *a1, v3, *a2, k1: 1, &b)
  • Ex) v1=v2=v3=a1=a2=b=v1=nil
    p(v1, v2, *a1, v3, *a2, k1: v1, &b) #=> ...?
Complex method dispatch
Callee side (defined methods)

• Parameters
  • Mandatory parameters
  • Optional parameters
  • Rest parameter
  • Post parameters
• Keyword parameters
  • Optional keyword parameters
  • Required keyword parameters
  • Rest keyword parameters
• Block parameter or block passing directly
Complex method dispatch
Callee side (defined methods)

• Visibility
  • Public
  • Private
  • Protected

# RubyQuiz: can you explain everything?
Complex method dispatch
Dynamic features

• Classes can extend methods
  • Open class
  • “include”/“prepend”
  • “extend” by instance objects
• Method missing
• Refinements (using)
• ...

…
Complex method dispatch
Calling interface

• Call by Ruby’s scripts
• Call by “send”
• Call by interpreters
  • E.g: Implicit conversions using to_int, to_a, …
• Call by C extensions
Design
What should we do?
Basic logic of method dispatch

1. Get class of receiver (`klass’)
2. Search a method `body’ from `klass’
3. Check availability, visibility and an arity of passed arguments
4. Construct a method frame with `body’
5. And continue VM execution
Basic logic of method dispatch

1. Get class of receiver (‘klass’)
2. **Search a method `body’ from `klass’**
3. Check availability, visibility and an arity of passed arguments
4. Construct a method frame with `body’
5. And continue VM execution
Search a method body

• Search method from `klass`
  1. Search method table of `klass`
     1. If method `body` is found, return `body`
     2. `klass` = super class of `klass` and repeat it
  2. If no method is given, exceptional flow
     • In Ruby language, `method_missing` will be called
Optimization
Method caching

• **Eliminate method search overhead**
  • Reuse search result
  • Invalidate cache entry with VM stat

• Two level method caching
  • Inline method caching (from Ruby 1.9.0)
  • Global method caching (from the beginning of Ruby)
Optimization
Method table (from Ruby 2.3)

• Make special Hash table for method table
  • To make search faster
  • To make more compact (lower memory usage, about 1/2)
  • https://bugs.ruby-lang.org/issues/11420

• Introduce method ID → method body related table
  • Ruby 2.2 and before use common table data structure shared with Hash objects. It is general and many features (ex: ordering), but over spec only for this purpose.
Basic logic of method dispatch

1. Get class of receiver (`klass’)
2. Search a method `body’ from `klass’
3. **Check availability, visibility and an arity of passed arguments**
4. Construct a method frame with `body’
5. And continue VM execution
Check the availability, visibility and an arity

- Method body checking
  - Not found → call method_missing

- Visibility checking
  - Not found → call method_missing

- Arity checking
  - Not matched → raise ArgumentError
Optimization (from Ruby 2.0)
Caching checking results into inline method cache

1\textsuperscript{st} time
1. Search method
2. Checks
3. Construct frame
4. Continue method

2\textsuperscript{nd} time
1. Search method
2. Checks
3. Construct frame
4. Continue method

Cache the checking result

[Skip!]
Basic logic of method dispatch

1. Get class of receiver (`klass’)
2. Search a method `body’ from `klass’
3. Check availability, visibility and an arity of passed arguments
4. **Construct a method frame with `body’**
5. And continue VM execution
Construct a method frame

• Each method needs a **method frame** to maintain:
  • Local variables (includes method parameters)
  • Passed block information
  • Current method information (used by backtrace and so on)
  • ...

Ruby 1.9 VM stacks structure

```ruby
def foo
  bar {
    ...
  }
end
end
foo
```
Local variables with complex passed arguments and method parameters

# complex method parameters
def complex_foo(m1, m2, o1=1, o2=2, *r, p1, p2, k1: 1, k2: 2, kr:, **kw, &b)
    ... # body
end

... # complex method dispatch
complex_foo(v1, v2, *a1, v3, v4, *a2, k1: 1, k2: 2, kr: 3, **kw, &block)
Basic logic of method dispatch

1. Get class of receiver (`klass’)
2. Search a method `body’ from `klass’
3. Check availability, visibility and an arity of passed arguments
4. **Construct a method frame with `body’**
5. And continue VM execution
Detailed logic of method dispatch

1. Get class of receiver (`klass`)
2. Search a method `body` from `klass`
3. Check availability, visibility and an arity of passed arguments
   1. Check arity (expected args # and given args #) and process
      1. Post arguments
      2. Optional arguments
      3. Rest argument
      4. Keyword arguments
      5. Block argument
4. **Construct a method frame with `body`**
   1. Push new control frame
      1. Store `PC` and `SP` to continue after method returning
      2. Store `block information`
      3. Store `defined class`
      4. Store bytecode info (iseq)
      5. Store recv as self
5. And continue VM execution
Optimization (from Ruby 1.9.0)
Specialized instruction

• Make special VM instruction for several methods
  • +, -, *, /, ...

```ruby
def opt_plus(recv, obj)
  if recv.is_a(Fixnum) and obj.is_a(Fixnum) and Fixnum#+ is not redefined
    return Fixnum.plus(recv, obj)
  else
    return recv.send(:+, obj) # not prepared
  end
end```

Keyword parameters from Ruby 2.0

```ruby
# def with keywords
def foo(a, b, key1: 1, key2: 2)
  ...
end

# call with keywords
foo(1, 2, key1: 123, key2: 456)
```
Slow keyword parameters

Evaluation on Ruby 2.1

x30 slower

Repeat 10M times
Why slow, compare with normal parameters?

1. Hash creation
2. Hash access

```python
def foo(k1: v1, k2: v2):
    ...  
end
foo(k1: 1, k2: 2)
```

```python
def foo(h = {}):
    k1 = h.fetch(:k1, v1)
    k2 = h.fetch(:k2, v2)
    ...  
end
foo( {k1: 1, k2: 2} )
```
Optimization (from Ruby 2.2)
Fast Keyword parameters

• Key technique
  → Pass “a keyword list” instead of a Hash object

Check “Evolution of Keyword parameters” at Rubyconf portugal'15
Result: Fast keyword parameters (Ruby 2.2.0)

Ruby 2.2 optimizes method dispatch with keyword parameters

- foo6(1, 2, 3, 4, 5, 6)
- foo_kw6(k1: 1, k2: 2, k3: 3, k4: 4, k5: 5, k6: 6)

- Execution time (sec)
- Repeat 10M times

Ruby 2.2 optimizes method dispatch with keyword parameters

- x14 faster!!
  (best case)

- But still x2 times slower compared with normal dispatch
Another Idea:
90% of methods are like simple method calls

```ruby
# Simple method call
def simple_foo(x)
    ...
    ...
end
...
foo(123)
```
Optimization (from Ruby 2.3)
Caching checking results into inline method cache

1\textsuperscript{st} time
1. Search method
2. Checks
3. Construct frame
4. Continue method

1. Search method
2. Checks
3. Construct frame

2\textsuperscript{nd} time
4. Continue method

Cache the checking result

Simple code setup
• Call inline code for 0 param, 0 locals
• Call inline code for 0 param, 1 locals
• Call inline code for 1 param, 0 locals
• ....
2. Complex code setup
Optimization (from Ruby 2.3)
Caching checking results into inline method cache

• Make dispatch function
  • Base C function: dispatch(..., param, local){ /* setup frame */ }
  • Make several inline codes
    • dispatch_0_0(...){dispatch(..., 0, 0);}
    • dispatch_0_1(...){dispatch(..., 0, 1);}
    • dispatch_1_0(...){dispatch(..., 1, 0);}
    • dispatch_0_1(...){dispatch(..., 0, 1);}
    • ...

• And call inline dispatch function (if it is possible)
Evaluation
Dispatch same method (hit inline cache)

Execution time (sec)

<table>
<thead>
<tr>
<th></th>
<th>Execution time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>vm2_method*</td>
<td>1.367</td>
</tr>
<tr>
<td></td>
<td>1.35</td>
</tr>
<tr>
<td></td>
<td>1.299</td>
</tr>
<tr>
<td>trunk</td>
<td>1.19</td>
</tr>
<tr>
<td>modified</td>
<td>1.058</td>
</tr>
</tbody>
</table>
Evaluation
Dispatch different methods (miss inline cache)

vm2_poly_method* (48M dispatches)

<table>
<thead>
<tr>
<th></th>
<th>Execution time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>2.089</td>
</tr>
<tr>
<td>21</td>
<td>2.187</td>
</tr>
<tr>
<td>22</td>
<td>2.17</td>
</tr>
<tr>
<td>trunk</td>
<td>2.16</td>
</tr>
<tr>
<td>modified</td>
<td>2.278</td>
</tr>
</tbody>
</table>
Evaluation
Tak function

![Bar chart showing execution times for different inputs.]

- **app_tak**
  - Execution time (sec)
  - 200: 0.789
  - 21: 0.762
  - 22: 0.767
  - trunk: 0.732
  - modified: 0.694
Rough estimation

• Hit inline cache: about 1.1 sec on 48M call
  → 23ns / call
  → 78 clocks on 3.4GHz CPU

• Miss inline cache: about 2.3 sec on 48M call
  → 48ns / call
  → 163 clocks on 3.4GHz
Summary

• Method dispatch is key feature for Ruby
• Ruby’s method has rich features
• Many optimization techniques on MRI are invented by many people
Summary

Ruby/MRI is getting better and better.
Thank you for your attention

Koichi Sasada
<ko1@heroku.com>