Evolution of Keyword parameters

Koichi Sasada
<ko1@heroku.com>

Rubyconf Portugal’15
Background
“Keyword parameters” from Ruby 2.0

# From Ruby 2.0 feature
def foo(k1: 1, k2: 2)
  p [k1, k2] #=> [345, 2]
end

foo(k1: 345)
Background
“Keyword parameters” was slow!!

Evaluation on Ruby 2.1

Repeat 10M times
Summary

Ruby 2.2 optimized “keyword parameters”

Repeat 10M times

But still x2 times slower compared with normal dispatch
Why was slow?

How to solve it?
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
• Generational/incremental GC for 2.x
Koichi Sasada from Japan

Japanese lesson

<table>
<thead>
<tr>
<th>English</th>
<th>Thank you</th>
</tr>
</thead>
<tbody>
<tr>
<td>Portuguese</td>
<td>Obrigado</td>
</tr>
<tr>
<td>Japanese</td>
<td>Arigato</td>
</tr>
<tr>
<td></td>
<td>ありがとう</td>
</tr>
</tbody>
</table>
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”
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
Nobu
Patch monster

Commit count of MRI
Nobu
The Ruby Hero
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

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
Ruby 2.2
Symbol GC

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
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>
RGenGC from Ruby 2.1: Micro-benchmark

- Total mark: 704.843669 ms
- Total sweep: 1699.805974 ms
- No RGenGC: x2.5 faster

- Total mark: 867.740319 ms
- Total sweep: 87.230735 ms

(no) vs (RGenGC)
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
Heroku Matz team and Ruby core team
Next target is

Ruby 2.3

No time to talk about it.
Please ask me later 😊
Back to the main topic
Ruby has many

Let’s play hangman game
Ruby has many
Ruby has many
Ruby has many
FU**
RubyConf 2008 Keynote by Dave Thomas (Pragmatic programmer)

Quoted from Dave Thomas Keynote
F**K RUBY

Quoted from Dave Thomas Keynote
Quoted from Dave Thomas Keynote
Ruby has many

RubyConf Portugal 2015
Koichi Sasada
Ruby has many Or Methods
Optimizing Func(tion)s or Methods is important for Ruby

• Syntax optimization (readable/writable)
  • Call without parenthesis
  • Passing blocks with braces or do/end
  • Splat/block arguments (*args, &block)
  • Optional/rest/post/block parameters by def
    • def foo(m1, m2, o1=1, o2=2, *rest, p1, p2, &block)
• Keyword arguments/parameters
• Performance optimization
  • By virtual machine implementation. My task 😊
Tough work to pass many arguments

# Quoted from my Ruby code
```
scinsn = Instruction.new(
    name, opes, pops, rets, comm,
    orig_insn.body, orig_insn.tvars, orig_insn.sp_inc,
    orig_insn, orig_insn.defopes, :sc, nextsc, pushs)
```

12 arguments.
Can’t understand what parameter mean.
Tough work to pass many arguments

```cpp
# Difficult to understand what we specify
GC::Tracer.start_logging(filename, false, false, false)
```

Only 4 arguments, but also it’s difficult to read
Keyword parameter helps you

```ruby
# quoted from my Ruby code
GC::Tracer.start_loggin(
    filename,
    gc_stat: false,
    gc_latest_gc_info: false,
    rusage: false
)

Easy to understand!
```
The History of Keyword parameter
Hash notation at the last argument from beginning of Ruby

foo(1, 2, :key1 => v1, :key2 => v2)

# Same as
# foo(1, 2, {:key1 => v1, :key2 => v2})
Symbol hash notation from Ruby 1.9.3

foo(1, 2, key1: v1, key2: v2)

# Same as
# foo(1, 2, :key1 => v1, :key2 => v2)
# foo(1, 2, {:key1 => v1, :key2 => v2})
Keyword parameters processing before Ruby 2.0

def foo(a, b, kw={}) # kw is Hash
  key1 = kw.fetch(:key1, 1)
  key2 = kw.fetch(:key2, 2)
  # check unknown kws
...
end

Default values
Keyword parameters from Ruby 2.0 (1)

def foo(a, b, key1: 1, key2: 2)
  ...
end

We don’t need to write Hash access any more!
Keyword parameters from Ruby 2.0 (2)

• Raise an exception when unknown keywords are passed

• Rest keyword parameter (**kw) can receive non-specified keyword parameters

```ruby
def foo(k1: v1, **kw)
  p kw #=> {k2: 2, k3: 3}
end

foo(k1: 1, k2: 2, k3: 3)
```

• Blocks also can accept keyword parameters

```ruby
foo{|k1: 1, k2: 2| ...}
```
Required keyword parameter from Ruby 2.1

def foo(a, b, key1: 1, key2:)
    ...
end

# Similar to
def foo(a, b, key1: 1, key2: raise("err"))
    ...
end

No default value
Need to specify by caller
The Implementation of Keyword parameter
Implementation of keyword parameter
Ruby 2.0 and Ruby 2.1

• Caller: make a Hash object and pass it normally
  • Same as Ruby 2.0 and before

• Callee: decompose a Hash object and assign to local variables correctly
  • Mostly same code of decomposing code in Ruby
  • Need some more error checking
Implementation of keyword parameter Ruby 2.0 and Ruby 2.1

def foo(k1: v1, k2: v2)
  ...
end

Compiler translate to

def foo(h={})
  k1 = h.fetch(:k1, v1)
  k2 = h.fetch(:k2, v2)
  #(and error check code)
  ...
end
Bad NEWS
Slow keyword parameters

Evaluation on Ruby 2.1

x30 slower

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

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

1. Hash creation
2. Hash access

def foo(h = {})
    k1 = h.fetch(:k1, v1)
    k2 = h.fetch(:k2, v2)
    ...  
end
foo(k1: 1, k2: 2)
Optimization technique of keyword parameters from Ruby 2.2

• Key technique
  → Pass “a keyword list” instead of a Hash object
Preparation: Make “keyword list” and “default value list” at compile time

• We can see all source code at compile time
• Collect keywords in a list for each method call
  • ex: “foo(k1: x, k2: y)” #=> kwlist is [:k1, :k2]
• Collect “Receive keyword list (Rkwlist)” and “Default values list (dvlist)” in each method definition
  • ex: “def foo(k1: 1, k2: 2)” #=> Rkwlist is [:k1, :k2],
    dvlist is [1, 2]
  • ex: “def foo(k1: 1, k2: f2())” #=> dvlist is [1, Qundef]

NOTE: Qundef is internal special value which should not expose Ruby world
Call with keyword parameter [Sender]
Pass “kwlist” instead of making a Hash

• Pass values as a “keyword list”

```python
foo(k1: 1, k2: 2)
```

```
foo(1, 2, kwlist)
```

NOTE: This is pseudo code. kwlist is not passed as an argument, but passed as calling information.
Call with keyword parameter [Receiver]
Manipulate passed kwlist

• Assign local variables with passed keyword list

```ruby
def foo(*kvs, kwlist)
    Rkwlist.each.with_index{|k, i|
        ki = kwlist.index(k)
        assign(k, ki ? kvs[ki] : dvlist[i])
    }

def foo(k1: 1, k2: 2, k3: 3)
    kvs = [1, 2]; kwlist=[:k1, :k2]
    Rkwlist = [:k1, :k2, :k3]
    dvlist = [1, 2, 3]
```

Pseudo code
Call with keyword parameter [Receiver]
Treat with default values with expressions

```ruby
def foo(k1: 1, k2: f2(), k3: f3())
  Rkwlist = [:k1, :k2, :k3]
  dvlist = [1, Qundef, Qundef]

  # k1 is already initialized
  k2 = f2() unless unset_bits[1]
  k3 = f3() unless unset_bits[2]

  ... # start of method body
end
```

Pseudo code

```ruby
def foo(*kvs, kwlist)
  unset_bits = 0
  Rkwlist.each.with_index{|k, i|
    if ki = kwlist.index(k)
      v = kvs[ki]
    else if (v = dvlist[i]) == Qundef
      v = nil
      unset_bits[i] = 1
    end
    assign(k, v)
  } # cont to right

  # k1 is already initialized
  k2 = f2() unless unset_bits[1]
  k3 = f3() unless unset_bits[2]

  ... # start of method body
end
```

NOTE: Qundef is internal special value which should not expose Ruby world

Using BITMAP to remember unspecified keywords
Q. Why not assign Qundef directly? (instead of using bitmap)

```python
def foo(k1: 1, k2: f2(), k3: f3())
```

Pseudo code

```python
def foo(*kvs, kwlist):
    unset_bits = 0
    Rkwlist.each.with_index{|k, i|ki = kwlist.index(k)
v = ki ? kvs[i] : dvlist[i]
    assign(k, v)
}
k2 = f2() unless k2 == Qundef
k3 = f3() unless k3 == Qundef
... # start of method body
end
```

Rkwlist = [:k1, :k2, :k3]
dvlist = [1, Qundef, Qundef]
A. We can access initializing keyword variables with `eval()`

```python
def foo(k1: 1,
    k2: eval("k3"), # should be nil
    k3: f3())
```
Evaluation result

Compare 3 types methods

1. def foo6(a, b, c, d, e, f); end
   • Normal method dispatch with 6 parameters

2. def foo_kw6(k1: 1, k2: 2, k3: 3, k4: 4, k5: 5, k6: 6); end
   • Default values are immediate values

3. def foo_complex_kw6(k1: 1+1, k2: 2+1, k3: 3+1, k4: 4+1, k5: 5+1, k6: 6+1); end
   • Default values are expressions (not immediate values)
Result: Fast keyword parameters

Ruby 2.2 optimizes method dispatch with keyword parameters

Ruby 2.2 optimizes method dispatch with keyword parameters

x14 faster!! (best case)

Repeat 10M times

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

But still x2 times slower compared with normal dispatch
Result: Ruby 2.1 vs. Ruby 2.2

Removing Hash creation is dramatically effective.
Using immediate default values is effective

<table>
<thead>
<tr>
<th>Function</th>
<th>Execution time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo6(1, 2, 3, 4, 5, 6)</td>
<td>0.5</td>
</tr>
<tr>
<td>foo_kw6</td>
<td>1</td>
</tr>
<tr>
<td>foo_kw6(k1: 1, k2: 2, k3: 3, k4: 4, k5: 5, k6: 6)</td>
<td>1.5</td>
</tr>
<tr>
<td>foo_complex_kw6</td>
<td>2</td>
</tr>
<tr>
<td>foo_complex_kw6(k1: 1, k2: 2, k3: 3, k4: 4, k5: 5, k6: 6)</td>
<td>2</td>
</tr>
</tbody>
</table>

Using immediate values

Using non-immediate values
Challenge: Improve computational complexity

• Computational complexity of current impl. is $O(mn)$
  • Now, $m$ and $n$ is enough small (only a few keywords), but...

$n = \text{kwlist}.\text{length}$
$m = \text{Rkwlist}.\text{length}$

Total computationan complexity: $O(mn)$

```ruby
def foo(*kvs, kwlist)
  Rkwlist.each.with_index{|k, i| # m times
    ki = kwlist.index(k) # O(m)
  }
  # O(n)
```

Pseudo code

$n = \text{kwlist}.\text{length}$
$m = \text{Rkwlist}.\text{length}$

Total computationan complexity: $O(mn)$

```ruby
def foo(*kvs, kwlist)
  Rkwlist.each.with_index{|k, i| # m times
    ki = kwlist.index(k) # O(m)
  }
  # O(n)
```
Summary
Ruby 2.2 optimized “keyword parameters”

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

Repeat 10M times

Execution time (sec)

Ruby 2.1  Ruby 2.2

But still x2 times slower compare with normal dispatch
Thank you for your attention.
Arigato!

Koichi Sasada
<ko1@heroku.com>