Ruby 2.1 のすべて

Koichi Sasada
Heroku, Inc.
ko1@heroku.com
About this presentation

- In this presentation, I will show you about Ruby 2.1 which I know.
Ruby 2.1 release plan announcement

“I, Naruse, take over the release manager of Ruby 2.1.0 from mame. Ruby 2.1.0 is planed to release in 2013-12-25. I’m planning to call for feature proposals soon like 2.0.0 [ruby-core:45474], so if you have a suggestion you should begin preparing the proposal.”

- [ruby-core:54726] Announce take over the release manager of Ruby 2.1.0

by NARUSE, Yui
2013/12/25!

http://www.flickr.com/photos/htakashi/5285103341/ by Takashi Hososhima
Ruby 2.1 schedule (more)

- **We are here!**
- **2013/06**
  - Call for Feature Proposal (CFP)
- **2013/07, 8, 10**
  - Dev-meeting w/Matz
- **2013/09**
  - Feature freeze
- **2013/10**
  - Preview1
  - Preview2
  - RC
  - RubyConf2013
    - 11/8-10
  - Preview1
- **2013/12/25**
  - Ruby 2.1.0

Ruby 2.1 Changes

• Syntax changes
  • Required keyword argument
  • “r”, “i”, “f” suffix
  • “def” sentence returns a symbol of method name

• Runtime changes
  • String#scrub
  • Object tracing
  • Refinement is no longer experimental features

• Performance improvements
  • RGenGC: Generational GC
  • klasscache - the class hierarchy invalidation patch

• Other changes
Syntax

Required keyword argument

• “Keyword argument” from Ruby 2.0 is an alternative of optional argument
  • def foo(foo=1, bar: 2); end
  • foo(); foo(100); foo(100, bar: 200) # OK

• “Required keyword argument” is a not optional keyword argument
  • def foo(foo=1, bar: ); end
  • foo(); foo(100); # NG: missing keyword: bar
  • foo(100, bar: 200) # OK
Syntax
“r” suffix for Rational numbers

• To represent \( \frac{1}{2} \), in Ruby “Rational(1, 2)”
  \( \rightarrow \) Too long!!

• Introduce “r” suffix
  \( \frac{1}{2} \) \( \rightarrow \) 1/2r

• “[digits]r” represents “Rational([digits], 1)”

• 1/2r \( \rightarrow \) 1/Rational(2, 1)

• 1/Rational(2, 1) \( \rightarrow \) Rational(1/2)
Syntax

“i” suffix for Complex numbers

• We already have “Integer#i” method to make imaginary number like “1+2.i”
• We already introduced “r” suffix for Rational
  → No reason to prohibit “i” suffix!!
• [digits]i represents “Complex(0, [digits])”
• 1+2i #=> 1+Complex(0, 2)
• 1+Complex(0, 2) #=> Complex(1, 2)

• You can mix “r” and “i” suffix
Syntax
“f” suffix for String

- String literal “foo” creates new objects each time
  - 10.times{p “foo”.object_id} #=> show different objects
  - To support mutable strings
- “foo”f (‘f’ suffix) creates same/frozen string
  - 10.times{p “foo”f.object_id} #=> show only one object id
- Aggregate same frozen strings
  - p(“foo”f.object_id, “foo”f.object_id) #=> same object id

- Mainly for performance
  - Target is framework such as ERb, etc
Syntax
“def” returns a name symbol

• p(def foo; end) #=> nil @Ruby 2.0 and before
• p(def foo; end) #=> :foo @Ruby 2.1

• Usecase
  • private static void def main(args) ...; end
Runtime changes

String#scrub

• Problem: How to verify/fix invalid byte sequence?
• From Ruby 2.1, we introduce two methods “String#scrub” and “String#scrub!” to verify and fix invalid byte sequence.
Runtime changes
Object tracing

• **ObjectSpace. trace_object_allocations**
  • Trace object allocation and record allocation-site
    • Record filename, line number, creator method’s id and class
  • Usage:
    ```ruby
    ObjectSpace.trace_object_allocations{
      # record only in the block
      o = Object.new
      file = ObjectSpace.allocation_sourcefile(o) # => __FILE__
      line = ObjectSpace.allocation_sourcecode(o) # => __LINE__ - 2
    }
    ```
Performance improvement
RGenGC: Generational GC

• Issue: M&S is slow
• Issue: Introducing generational GC causes serious compatibility problem
• Proposal: Introduce new GC algorithm RGenGC (Restricted Generational GC) without compatibility problem
Performance improvement

Current GC: Mark & Sweep

1. Mark reachable objects from root objects

2. Sweep unmarked objects (collection and de-allocation)

Root objects

Collect unreachable objects
RGenGC: Background
Generational GC (GenGC)

• Weak generational hypothesis: Most objects die young →
  Concentrating reclamation effort on the youngest objects

• Separate young generation and old generation
  • Create objects as young generation
  • Promote to old generation after surviving \( n \)-th GC
  • In CRuby, \( n == 1 \) (after 1 GC, objects become old)

• Usually, GC on young space (minor GC)
• GC on both spaces if no memory (major/full GC)
RGenGC: Background
Difficulty of inserting write barriers

- To introduce generational garbage collector, WBs are necessary to detect [old→new] type reference
- “Write-barrier miss” causes terrible failure
  1. WB miss
  2. Remember-set registration miss
  3. (minor GC) marking-miss
  4. Collect live object → Terrible GC BUG!!
Performance improvement
RGenGC: Generational GC

• RGenGC: Restricted Generational GC
  • New GC algorithm allows mixing “Write-barrier protected objects” and “WB unprotected objects”
  • No (mostly) compatibility issue with C-exts

• Inserting WBs gradually
  • We can concentrate WB insertion efforts for major objects and major methods
  • Now, **Array, String, Hash, Object, Numeric** objects are WB protected
    • Array, Hash, Object, String objects are very popular in Ruby
    • Array objects using `RARRAY_PTR()` change to WB unprotected objects (called as Shady objects), so existing codes still works.
Performance improvement
klasscache: the class hierarchy invalidation

- Invalidate all classes’ method cache
Performance improvement
klasscache: the class hierarchy invalidation

“This patch adds class hierarchy method caching to CRuby. This is the algorithm used by JRuby and Rubinius.”

[ruby-core:55053] [ruby-trunk - Feature #8426][Open]
Implement class hierarchy method caching

by Charlie Somerville
Performance improvement
klasscache: the class hierarchy invalidation

• Invalid only sub-classes under effective class
Planned changes

• Sophisticated Symbol management
  • Collectable symbols
Current symbol management (-Ruby 2.0.0)

```
rb_intern(char *cstr, enc) /* pseudo code */
str = make_fake_str(cstr, enc)
if (st_lookup(sym_id, str, &id)) { return id }
else {
    str = make_true_str(cstr, enc);
    ID id = last_id++;
    st_insert(sym_id, cstr, id)
    st_insert(id_str, id, cstr)
}
```
Proposed symbol management v4 (Ruby 2.1.0-)

ID storage

- last_id
- imm_tbl
- id_str
- mark_tbl

<table>
<thead>
<tr>
<th>Imm</th>
<th>Str</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;foo&quot; (0x788)</td>
<td></td>
</tr>
<tr>
<td>&quot;bar&quot; (0x789)</td>
<td></td>
</tr>
<tr>
<td>&quot;baz&quot; (0x790)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Str</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x123</td>
<td></td>
</tr>
<tr>
<td>0x124</td>
<td></td>
</tr>
<tr>
<td>0x790</td>
<td>null</td>
</tr>
<tr>
<td>0x791</td>
<td></td>
</tr>
</tbody>
</table>

Heap

- Interned Symbol "foo"
  VALUE: ID2SYM(0x123)
- Frozen String "bar"
  VALUE: 0x789
- Frozen String "baz"
  VALUE: 0x790
- Dynamic Symbol "baz"
  VALUE: 0x791
- Interned Symbol "baz"
  VALUE: 0x791

Imm_str

<table>
<thead>
<tr>
<th>ID</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

flags:
HAS_ID,
HAS_FSTR,
HAS_DSYM

mark_tbl
Summary

• Schedule: Release at 2013/12/25
  • Preview1 2013/10
  • Preview2 2013/10 with feature freeze
  • Preview3 ?
  • Release candidate 2013/12/11
  • Release 2013/12/25

• New features

• Performance improvement

Thank you

Koichi Sasada
Heroku, Inc.
ko1@heroku.com