remove rb_control_frame_t::klass

06/18/2015 11:30 AM - ko1 (Koichi Sasada)

Status: Closed
Priority: Normal
Assignee: ko1 (Koichi Sasada)
Target version: ruby -v: 2.3dev
Backport: 2.0.0: UNKNOWN, 2.1: UNKNOWN, 2.2: UNKNOWN

Description

Abstract

rb_control_frame_t has a field klass, which is used to search super class when super is called (and also several usages). super is only for methods. However, all of rb_control_frame_t requires to keep klass on other frames such as block and so on.

This patch solve this issue by introducing rb_callable_method_entry_t.

https://github.com/ko1/ruby/tree/remove_cf_klass

rb_callable_method_entry_t is similar to rb_method_entry_t (actually, same data layout), but it has defined_class.

Background

For methods defined to classes, then owner of these methods are also defined_class.

class C1 < C0
  def foo # foo's owner is C1, and foo's defined class is C0.
    super
  end
end

We can start to search super class from C1's super class (C0).

However, when we define methods in a modules, then defined class is not fixed.

module M
  def foo # foo's owner is M, however, defined class is not fixed.
    super
  end
end

We can not search super class from module M.
M is used when some classes include (extend, prepend). These classes determine super classes.

class C1 < C0
  include M
end

In this case, we can know super class of M#foo (included by C1) is C0.

To represent a correct class hierarchy, MRI uses special class T_ICLASS.
T_ICLASS is internal class points including (extending and prepending) modules like that:

```
C1 -> T_ICLASS -> C0
    |        
    +-- M
```

# Let's use notation I(M) to represent this data structure.
# C1 -> I(M) -> C0
We can't determine defined class of M#foo, but we can determine a defined class I(M)#foo (in this case, it is C0).

Current MRI pushes defined class of methods onto control frame stack (rb_control_frame_t::klass).
However, it becomes overhead, especially for non-method frames such as blocks and so on.

To overcome this issue, I introduced rb_callable_method_entry_t, which is similar to rb_method_entry_t, but has defined_class.

(rb_callable_method_entry_t is T_IMEMO/imemo_ment, same as rb_method_entry_t)

For C1#foo, the defined class is just C1. So rb_method_entry_t of C1#foo is also rb_callable_method_entry_t.

For M#foo, the defined class is not fixed. So rb_method_entry_t of M#foo is not a rb_callable_method_entry_t.

rb_callable_method_entry_t is created when M#foo is called by I(M).
We can find I(M) when we search M#foo in a class hierarchy C1 -> I(M) -> C0.
Let's call created rb_callable_method_entry_t for M#foo with I(M) as I(M)#foo.

It is inefficient that we make I(M)#foo everytime when M#foo is called.
So I(M)#foo is cached in a table pointed by I(M).
This table will be cleared when M is redefined.

**pros. and cons.**

**Advantage:**
- Faster pushing control frame especially for block invocation.
- Simplify codes around searching super classes.

**Disadvantage:**
- Increase memory consumption because of two reasons
  - Duplicate method entries for methods defined by modules.
  - Cache table kept by I(M)
- Increase complexity maintaining method entries. rb_method_entry_t was a simple enough data structure. We need to consider which data structures are required.

**Measurement**

**For performance.**

I do benchmark repeating 10 times (pickup the fastest results).

```plaintext
Speedup ratio: compare with the result of `trunk' (greater is better)

<table>
<thead>
<tr>
<th>name</th>
<th>modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>app_answer</td>
<td>1.032</td>
</tr>
<tr>
<td>app_aobench</td>
<td>0.989</td>
</tr>
<tr>
<td>app_erb</td>
<td>1.006</td>
</tr>
<tr>
<td>app_factorial</td>
<td>1.000</td>
</tr>
<tr>
<td>app_fib</td>
<td>1.026</td>
</tr>
<tr>
<td>app_lc_fizzbuzz</td>
<td>1.144</td>
</tr>
<tr>
<td>app_mandelbrot</td>
<td>1.032</td>
</tr>
<tr>
<td>app_pentomino</td>
<td>0.996</td>
</tr>
<tr>
<td>app_raise</td>
<td>0.996</td>
</tr>
<tr>
<td>app_strconcat</td>
<td>0.981</td>
</tr>
<tr>
<td>app_tak</td>
<td>0.999</td>
</tr>
<tr>
<td>app_tarai</td>
<td>1.004</td>
</tr>
<tr>
<td>app_uri</td>
<td>1.001</td>
</tr>
<tr>
<td>array_shift</td>
<td>0.913</td>
</tr>
<tr>
<td>hash_aref_flo</td>
<td>1.023</td>
</tr>
<tr>
<td>hash_aref_miss</td>
<td>1.097</td>
</tr>
<tr>
<td>hash_aref_str</td>
<td>1.074</td>
</tr>
<tr>
<td>hash_aref_sym</td>
<td>1.051</td>
</tr>
<tr>
<td>hash_aref_sym_long</td>
<td>1.047</td>
</tr>
<tr>
<td>hash_flatten</td>
<td>1.002</td>
</tr>
</tbody>
</table>
```

11/15/2021
<table>
<thead>
<tr>
<th>Function</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash_ident_flo</td>
<td>1.020</td>
</tr>
<tr>
<td>hash_ident_num</td>
<td>1.038</td>
</tr>
<tr>
<td>hash_ident_obj</td>
<td>1.036</td>
</tr>
<tr>
<td>hash_ident_str</td>
<td>1.055</td>
</tr>
<tr>
<td>hash_ident_sym</td>
<td>1.016</td>
</tr>
<tr>
<td>hash_keys</td>
<td>0.993</td>
</tr>
<tr>
<td>hash_shift</td>
<td>1.046</td>
</tr>
<tr>
<td>hash_values</td>
<td>1.006</td>
</tr>
<tr>
<td>io_file_create</td>
<td>0.983</td>
</tr>
<tr>
<td>io_file_read</td>
<td>0.985</td>
</tr>
<tr>
<td>io_file_write</td>
<td>1.014</td>
</tr>
<tr>
<td>io_select</td>
<td>0.958</td>
</tr>
<tr>
<td>io_select2</td>
<td>0.972</td>
</tr>
<tr>
<td>io_select3</td>
<td>1.027</td>
</tr>
<tr>
<td>loop_for</td>
<td>1.067</td>
</tr>
<tr>
<td>loop_generator</td>
<td>0.980</td>
</tr>
<tr>
<td>loop_times</td>
<td>1.078</td>
</tr>
<tr>
<td>loop_whileloop</td>
<td>0.995</td>
</tr>
<tr>
<td>loop_whileloop2</td>
<td>1.005</td>
</tr>
<tr>
<td>marshal_dump_flo</td>
<td>1.014</td>
</tr>
<tr>
<td>marshal_dump_load_geniv</td>
<td>0.989</td>
</tr>
<tr>
<td>marshal_dump_load_time</td>
<td>0.988</td>
</tr>
<tr>
<td>securerandom</td>
<td>0.944</td>
</tr>
<tr>
<td>so_ackermann</td>
<td>1.018</td>
</tr>
<tr>
<td>so_array</td>
<td>1.049</td>
</tr>
<tr>
<td>so_binary_trees</td>
<td>0.993</td>
</tr>
<tr>
<td>so_concatenate</td>
<td>1.036</td>
</tr>
<tr>
<td>so_count_words</td>
<td>1.012</td>
</tr>
<tr>
<td>so_exception</td>
<td>0.989</td>
</tr>
<tr>
<td>so_fannkuch</td>
<td>1.017</td>
</tr>
<tr>
<td>so_fasta</td>
<td>1.003</td>
</tr>
<tr>
<td>so_k_nucleotide</td>
<td>1.005</td>
</tr>
<tr>
<td>so_lists</td>
<td>1.001</td>
</tr>
<tr>
<td>so_mandelbrot</td>
<td>0.998</td>
</tr>
<tr>
<td>so_matrix</td>
<td>0.987</td>
</tr>
<tr>
<td>so_meteor_contest</td>
<td>1.035</td>
</tr>
<tr>
<td>so_nbody</td>
<td>0.997</td>
</tr>
<tr>
<td>so_nested_loop</td>
<td>1.054</td>
</tr>
<tr>
<td>so_nsieve</td>
<td>1.010</td>
</tr>
<tr>
<td>so_nsieve_bits</td>
<td>1.022</td>
</tr>
<tr>
<td>so_object</td>
<td>0.992</td>
</tr>
<tr>
<td>so_partial_sums</td>
<td>1.018</td>
</tr>
<tr>
<td>so_pidigits</td>
<td>0.993</td>
</tr>
<tr>
<td>so_random</td>
<td>0.981</td>
</tr>
<tr>
<td>so_reverse_complement</td>
<td>0.986</td>
</tr>
<tr>
<td>so_sieve</td>
<td>1.007</td>
</tr>
<tr>
<td>so_spectralnorm</td>
<td>1.014</td>
</tr>
<tr>
<td>vm1_attr_iivar*</td>
<td>0.991</td>
</tr>
<tr>
<td>vm1_attr_iivar_set*</td>
<td>0.987</td>
</tr>
<tr>
<td>vm1_block*</td>
<td>1.009</td>
</tr>
<tr>
<td>vm1_const*</td>
<td>0.983</td>
</tr>
<tr>
<td>vm1_ensure*</td>
<td>0.960</td>
</tr>
<tr>
<td>vm1_float_simple*</td>
<td>0.954</td>
</tr>
<tr>
<td>vm1_gc_short_lived*</td>
<td>1.002</td>
</tr>
<tr>
<td>vm1_gc_short_with_complex_long*</td>
<td>1.004</td>
</tr>
<tr>
<td>vm1_gc_short_with_long*</td>
<td>0.996</td>
</tr>
<tr>
<td>vm1_gc_short_with_symbol*</td>
<td>0.998</td>
</tr>
<tr>
<td>vm1_gc_wb_ary*</td>
<td>1.004</td>
</tr>
<tr>
<td>vm1_gc_wb_ary_promoted*</td>
<td>1.141</td>
</tr>
<tr>
<td>vm1_gc_wb_obj*</td>
<td>0.998</td>
</tr>
<tr>
<td>vm1_gc_wb_obj_promoted*</td>
<td>0.963</td>
</tr>
<tr>
<td>vm1_iivar*</td>
<td>0.982</td>
</tr>
<tr>
<td>vm1_iivar_set*</td>
<td>1.010</td>
</tr>
<tr>
<td>vm1_length*</td>
<td>1.006</td>
</tr>
<tr>
<td>vm1_iivar_init*</td>
<td>0.938</td>
</tr>
<tr>
<td>vm1_iivar_set*</td>
<td>0.990</td>
</tr>
<tr>
<td>vm1_neq*</td>
<td>0.987</td>
</tr>
</tbody>
</table>
Not so big change. vm2_super/zsuper should improve performance so I need to check it again.

Memory consumption

Running this script to check process memory on Linux Ubuntu.

```ruby
N = 100_000
$mod = true
$cls = true

module M
  N.times{|i|
    define_method("foo#{i}"){}
  } if $mod
end

class C
  include M
  N.times{|i|
```

11/15/2021
```ruby
define_method("bar#{i}"){}
  ) if $cls
end

class D
  include M
  N.times{|i|
    define_method("bar#{i}"){}
  ) if $cls
end

class E
  include M
  N.times{|i|
    define_method("bar#{i}"){}
  ) if $cls
end

[C, D, E].each{|c|
  obj = c.new
  N.times{|i|
    obj.send "foo#{i}" if $mod
    obj.send "bar#{i}" if $cls
  }
}

puts File.readlines('/proc/self/status').grep(/VmHWM/)"
I will try class level cache proposed by funnyfalcon before, over there.

### Related issues:
- Related to Ruby master - Bug #11279: remove rb_control_frame_t::klass
  - Closed
- Related to Ruby master - Bug #12164: Binding UnboundMethod to BasicObject
  - Closed

### Associated revisions

**Revision Se8a1474 - 07/03/2015 11:24 AM - ko1 (Koichi Sasada)**

- method.h: introduce rb_callable_method_entry_t to remove rb_control_frame_t::klass. [Bug #11278], [Bug #11279] rb_method_entry_t data belong to modules/classes. rb_method_entry_t::owner points defined module or class. module M def foo; end end In this case, owner is M. rb_callable_method_entry_t data belong to only classes. For modules, MRI creates corresponding T_ICLASS internally. rb_callable_method_entry_t can also belong to T_ICLASS. rb_callable_method_entry_t::defined_class points T_CLASS or T_ICLASS. rb_method_entry_t data for classes (not for modules) are also rb_callable_method_entry_t data because it is completely same data. In this case, rb_method_entry_t::owner == rb_method_entry_t::defined_class. For example, there are classes C and D, and includes M, class C; include M; end class D; include M, end then, two T_ICLASS objects for C's super class and D's super class will be created. When C.new.foo is called, then MIfoo is searched and rb_callable_method_entry_t data is used by VM to invoke MIfoo. rb_method_entry_t data is only one for MIfoo. However, rb_callable_method_entry_t data are two (and can be more). It is proportional to the number of including (and prepending) classes (the number of T_ICLASS which point to the module). Now, created rb_callable_method_entry_t are collected when the original module M was modified. We can think it is a cache. We need to select what kind of method entry data is needed. To operate definition, then you need to use rb_method_entry_t. You can access them by the following functions.
  - rb_method_entry(VALUE klass, ID id);
  - rb_method_entry(VALUE klass, ID id);
  - rb_method_entry(VALUE klass, ID id);
  - rb_method_entry(VALUE klass, ID id);
  - rb_method_entry(VALUE klass, ID id);
  - rb_method_entry(VALUE klass, ID id);
  - rb_resolve_refined_method(VALUE refinements, const rb_method_entry_t *me); To invoke methods, then you need to use rb_callable_method_entry_t which you can get by the following APIs corresponding to the above listed functions.
  - rb_callable_method_entry(VALUE klass, ID id);
  - rb_callable_method_entry(VALUE klass, ID id);
  - rb_callable_method_entry(VALUE klass, ID id);
  - rb_callable_method_entry(VALUE klass, ID id);
  - rb_callable_method_entry(VALUE klass, ID id);
  - rb_callable_method_entry(VALUE klass, ID id);
  - rb_resolve_refined_method(VALUE refinements, const rb_callable_method_entry_t *me); VM pushes rb_callable_method_entry_t, so that rb_vm_frame_method_entry() returns rb_callable_method_entry_t. You can check a super class of current method by rb_callable_method_entry_t::defined_class.
  - method.h: renamed from rb_method_entry_t::klass to rb_method_entry_t::owner.
  - internal.h: add rb_classext_struct::callable_m_tbl to cache rb_callable_method_entry_t data. We need to consider abotu this field again because it is only active for T_ICLASS.
  - class.c (method_entry_i): ditto.
  - class.c (rb_define_attr): rb_method_entry() does not takes defiend_class_ptr.
  - gc.c (mark_method_entry): mark RCLASS_CALLABLE_M_TBL() for T_ICLASS.
  - cont.c (fiber_init): rb_control_frame_t::klass is removed.
  - proc.c: fix `struct METHOD` data structure because rb_callable_method_t has all information.
  - vm_core.h: remove several fields.
  - rb_control_frame_t::klass.
  - rb_block_t::klass. And catch up changes.
  - eval.c: catch up changes.
  - gc.c: ditto.
  - insns.def: ditto.
  - vm.c: ditto.
  - vm_args.c: ditto.
  - vm_backtrace.c: ditto.
  - vm_dump.c: ditto.
  - vm_eval.c: ditto.
  - vm_inshhelper.c: ditto.
  - vm_method.c: ditto.

**Revision 51126 - 07/03/2015 11:24 AM - ko1 (Koichi SASADA)**

- method.h: introduce rb_callable_method_entry_t to remove rb_control_frame_t::klass. [Bug #11278], [Bug #11279] rb_method_entry_t data belong to modules/classes. rb_method_entry_t::owner points defined module or class. module M def foo; end end In this case, owner is M. rb_callable_method_entry_t data belong to only classes. For modules, MRI creates corresponding T_ICLASS internally. rb_callable_method_entry_t can also belong to T_ICLASS. rb_callable_method_entry_t::defined_class points T_CLASS or T_ICLASS. rb_method_entry_t data for classes (not for modules) are also rb_callable_method_entry_t data because it is completely same data. In this case, rb_method_entry_t::owner == rb_method_entry_t::defined_class. For example, there are classes C and D, and includes M, class C; include M; end class D; include M, end then, two T_ICLASS objects for C's super class and D's super class will be created. When C.new.foo is called, then MIfoo is searched and rb_callable_method_entry_t data is used by VM to invoke MIfoo. rb_method_entry_t data is only one for MIfoo. However, rb_callable_method_entry_t data are two (and can be more). It is proportional to the number of including (and prepending) classes (the number of T_ICLASS which point to the module). Now, created rb_callable_method_entry_t are collected when the original module M was modified. We can think it is a cache. We need to select what kind of method entry data is needed. To operate definition, then you need to use rb_method_entry_t. You can access them by the following functions.
  - rb_method_entry(VALUE klass, ID id);
  - rb_method_entry(VALUE klass, ID id);
  - rb_method_entry(VALUE klass, ID id);
  - rb_method_entry(VALUE klass, ID id);
  - rb_method_entry(VALUE klass, ID id);
  - rb_method_entry(VALUE klass, ID id);
rb_method_entry_without_refinements(VALUE klass, ID id);
rb_resolve_refined_method(VALUE refinements, const rb_method_entry_t *me); To invoke methods, then you need to use
rb_callable_method_entry_t which you can get by the following APIs corresponding to the above listed functions.
rb_callable_method_entry(VALUE klass, ID id);
rb_callable_method_entry_with_refinements(VALUE klass, ID id);
rb_callable_method_entry_without_refinements(VALUE klass, ID id);
rb_resolve_refined_method VALUE refinements, const rb_callable_method_entry_t *me); VM pushes rb_callable_method_entry_t,
so that rb_vm_frame_method_entry() returns rb_callable_method_entry_t. You can check a super class of current method by
rb_callable_method_entry_t::defined_class.

- Method.h: renamed from rb_method_entry_t::klass to rb_method_entry_t::owner.
- Internal.h: add rb_classext::callable_m_tbl to cache rb_callable_method_entry_t data. We need to consider about this field again because
it is only active for T_ICLASS.
- Class.c (method_entry_i): ditto.
- Class.c (rb_define_attr): rb_method_entry() does not takes defiend_class_ptr.
- GC.c (mark_method_entry): mark RCLASS_CALLABLE_M_TBL() for T_ICLASS.
- Cont.c (fiber_init): rb_control_frame_t::klass is removed.
- Proc.c: fix 'struct METHOD' data structure because rb_callable_method_t has all information.
- VM core: h: remove several fields.
- rb_control_frame_t::klass.
- rb_block_t::klass. And catch up changes.
- Eval.c: catch up changes.
- GC.c: ditto.
- Insns.def: ditto.
- VM.c: ditto.
- VM args.c: ditto.
- VM backtrace.c: ditto.
- VM dump.c: ditto.
- VM eval.c: ditto.
- VM insnhelper.c: ditto.
- VM method.c: ditto.

Revision 51126 - 07/03/2015 11:24 AM - ko1 (Koichi Sasada)

- Method.h: introduce rb_callable_method_entry_t to remove rb_control_frame_t::klass. [Bug #11278], [Bug #11279] rb_method_entry_t data
belong to modules/classes. rb_method_entry_t::owner points defined module or class. module M def foo; end End In this case, owner is M.
rb_callable_method_entry_t data belong to only classes. For modules, MRI creates corresponding T_ICLASS internally.
rh_callable_method_entry_t can also belong to T_ICLASS. rh_callable_method_entry_t::defined_class points T_CLASS or T_ICLASS.
rb_method_entry_t data for classes (not for modules) are also rb_callable_method_entry_t data because it is completely same data. In this case,
rb_method_entry_t::owner == rb_method_entry_t::defined_class. For example, there are classes C and D, and includes M, class C; include M;
end class D; include M; end then, two T_CLASS objects for C's super class and D's super class will be created. When C.new.foo is called, then
Mfoo is searched and rb_callable_method_t data is used by VM to invoke Mfoo. rb_method_entry_t data is only one for Mfoo. However,
rh_callable_method_entry_t data are two (and can be more). It is proportional to the number of including (and prepending) classes (the number
of T_ICLASS which point to the module). Now, created rb_callable_method_entry_t are collected when the original module M was modified. We
can think it is a cache. We need to select what kind of method entry data is needed. To operate definition, then you need to use
rb_method_entry_t. You can access them by the following functions.
- rb_method_entry(VALUE klass, ID id);
- rb_method_entry_with_refinements(VALUE klass, ID id);
- rb_method_entry_without_refinements(VALUE klass, ID id);
- rb_resolve_refined_method(VALUE refinements, const rb_method_entry_t *me); To invoke methods, then you need to use
rb_callable_method_entry_t which you can get by the following APIs corresponding to the above listed functions.
- rb_callable_method_entry(VALUE klass, ID id);
- rb_callable_method_entry_with_refinements(VALUE klass, ID id);
- rb_callable_method_entry_without_refinements(VALUE klass, ID id);
- rb_resolve_refined_method(VALUE refinements, const rb_callable_method_entry_t *me); VM pushes rb_callable_method_entry_t,
so that rb_vm_frame_method_entry() returns rb_callable_method_entry_t. You can check a super class of current method by
rb_callable_method_entry_t::defined_class.

- Method.h: renamed from rb_method_entry_t::klass to rb_method_entry_t::owner.
- Internal.h: add rb_classext::callable_m_tbl to cache rb_callable_method_entry_t data. We need to consider about this field again because
it is only active for T_ICLASS.
- Class.c (method_entry_i): ditto.
- Class.c (rb_define_attr): rb_method_entry() does not takes defiend_class_ptr.
- GC.c (mark_method_entry): mark RCLASS_CALLABLE_M_TBL() for T_ICLASS.
- Cont.c (fiber_init): rb_control_frame_t::klass is removed.
- Proc.c: fix 'struct METHOD' data structure because rb_callable_method_t has all information.
- VM core: h: remove several fields.
- rb_control_frame_t::klass.
- rb_block_t::klass. And catch up changes.
- Eval.c: catch up changes.
- GC.c: ditto.
- Insns.def: ditto.
- VM.c: ditto.
- VM args.c: ditto.
- VM backtrace.c: ditto.
- VM dump.c: ditto.
rb_method_entry_t. You can access them by the following functions.

- rb_method_entry(VALUE klass, ID id);
- rb_method_entry_with_refinements(VALUE klass, ID id);
- rb_method_entry_without_refinements(VALUE klass, ID id);
- rb_resolve_refined_method(VALUE refinements, const rb_method_entry_t *me); To invoke methods, then you need to use
  rb_callable_method_entry_t::defined_class.

rb_callable_method_entry_t which you can get by the following APIs corresponding to the above listed functions.

- rb_callable_method_entry(VALUE klass, ID id);
- rb_callable_method_entry_with_refinements(VALUE klass, ID id);
- rb_callable_method_entry_without_refinements(VALUE klass, ID id);
- rb_resolve_refined_method_callable(VALUE refinements, const rb_callable_method_entry_t *me); VM pushes rb_callable_method_entry_t,
  so that rb_vm_frame_method_entry() returns rb_callable_method_entry_t. You can check a super class of current method by
  rb_callable_method_entry_t::defined_class.

method.h: renamed from rb_method_entry_t::klass to rb_method_entry_t::owner.

internal.h: add rb_classext_t::callable_m_tbl to cache rb_callable_method_entry_t data. We need to consider about this field again because
it is only active for T_ICLASS.

class.c (method_entry_i): ditto.

class.c (rb_define_attr): rb_method_entry() does not take définiend_class_ptr.

gc.c (mark_method_entry): mark RCLASS_CALLABLE_M_TBL() for T_ICLASS.

cont.c (fiber_init): rb_control_frame_t::klass is removed.

proc.c: fix `struct METHO'D data structure because rb_callable_method_t has all information.

tclass.c: remove several fields.

  - rb_control_frame_t::klass.
  - rb_block_t::klass. And catch up changes.

eval.c: catch up changes.

gc.c: ditto.

insns.def: ditto.

vm.c: ditto.

vm_args.c: ditto.

vm_backtrace.c: ditto.

vm_dumpc.c: ditto.

vm_eval.c: ditto.

vm_inshelper.c: ditto.

vm_method.c: ditto.

---

Revision 51126 - 07/03/2015 11:24 AM - ko1 (Koichi Sasada)

- method.h: introduce rb_callable_method_entry_t to remove rb_control_frame_t::klass. [Bug #11278], [Bug #11279] rb_method_entry_t data
  belong to modules/classes. rb_method_entry_t::owner points defined module or class. module M  def foo; end end In this case, owner is M.
ruby_callable_method_entry_t data belong to only classes. For modules, MRI creates corresponding T_ICLASS internally.

rb_callable_method_entry_t can also belong to T_ICLASS. rb_callable_method_entry_t::defined_class points T_CLASS or T_ICLASS.

rb_method_entry_t data for classes (not for modules) are also rb_callable_method_entry_t data because it is completely same data. In this case,
rb_method_entry_t::owner == rb_method_entry_t::defined_class. For example, there are classes C and D, and include M, class C; include M;
end class D; include M; end then, two T_ICLASS objects for C’s super class and D’s super class will be created. When C.new.foo is called, then
Mfoo is searched and rb_callable_method_t data is used by VM to invoke Mfoo. rb_method_entry_t data is only one for Mfoo.

However, rb_callable_method_entry_t data are two (and can be more). It is proportional to the number of including (and prepending) classes (the number
of T_ICLASS which point to the module). Now, created rb_callable_method_entry_t are collected when the original module M was modified. We

Can think it is a cache. We need to select what kind of method entry data is needed. To operate definition, then you need to use

rb_method_entry_t. You can access them by the following functions.

- rb_method_entry(VALUE klass, ID id);
- rb_method_entry_with_refinements(VALUE klass, ID id);
- rb_method_entry_without_refinements(VALUE klass, ID id);
- rb_resolve_refined_method(VALUE refinements, const rb_method_entry_t *me); To invoke methods, then you need to use
  rb_callable_method_entry_t which you can get by the following APIs corresponding to the above listed functions.

- rb_callable_method_entry(VALUE klass, ID id);
- rb_callable_method_entry_with_refinements(VALUE klass, ID id);
- rb_callable_method_entry_without_refinements(VALUE klass, ID id);
- rb_resolve_refined_method_callable(VALUE refinements, const rb_callable_method_entry_t *me); VM pushes rb_callable_method_entry_t,
  so that rb_vm_frame_method_entry() returns rb_callable_method_entry_t. You can check a super class of current method by
  rb_callable_method_entry_t::defined_class.

---

Revision 51126 - 07/03/2015 11:24 AM - ko1 (Koichi Sasada)

- method.h: introduce rb_callable_method_entry_t to remove rb_control_frame_t::klass. [Bug #11278], [Bug #11279] rb_method_entry_t data
  belong to modules/classes. rb_method_entry_t::owner points defined module or class. module M  def foo; end end In this case, owner is M.
ruby_callable_method_entry_t data belong to only classes. For modules, MRI creates corresponding T_ICLASS internally.

rb_callable_method_entry_t can also belong to T_ICLASS. rb_callable_method_entry_t::defined_class points T_CLASS or T_ICLASS.

rb_method_entry_t data for classes (not for modules) are also rb_callable_method_entry_t data because it is completely same data. In this case,
rb_method_entry_t::owner == rb_method_entry_t::defined_class. For example, there are classes C and D, and include M, class C; include M;
end class D; include M; end then, two T_ICLASS objects for C’s super class and D’s super class will be created. When C.new.foo is called, then
Mfoo is searched and rb_callable_method_t data is used by VM to invoke Mfoo. rb_method_entry_t data is only one for Mfoo.

However, rb_callable_method_entry_t data are two (and can be more). It is proportional to the number of including (and prepending) classes (the number
of T_ICLASS which point to the module). Now, created rb_callable_method_entry_t are collected when the original module M was modified. We

Can think it is a cache. We need to select what kind of method entry data is needed. To operate definition, then you need to use

rb_method_entry_t. You can access them by the following functions.

- rb_method_entry(VALUE klass, ID id);
- rb_method_entry_with_refinements(VALUE klass, ID id);
- rb_method_entry_without_refinements(VALUE klass, ID id);
- rb_resolve_refined_method(VALUE refinements, const rb_method_entry_t *me); To invoke methods, then you need to use
  rb_callable_method_entry_t which you can get by the following APIs corresponding to the above listed functions.

- rb_callable_method_entry(VALUE klass, ID id);
- rb_callable_method_entry_with_refinements(VALUE klass, ID id);
- rb_callable_method_entry_without_refinements(VALUE klass, ID id);
- rb_resolve_refined_method_callable(VALUE refinements, const rb_callable_method_entry_t *me); VM pushes rb_callable_method_entry_t,
  so that rb_vm_frame_method_entry() returns rb_callable_method_entry_t. You can check a super class of current method by
  rb_callable_method_entry_t::defined_class.

---
rb_resolve_refined_method_callable(VALUE refinements, const rb_callable_method_entry_t *me): VM pushes rb_callable_method_entry_t, so that rb_vm_frame_method_entry() returns rb_callable_method_entry_t. You can check a super class of current method by rb_callable_method_entry_t::defined_class.

method.h: renamed from rb_method_entry_t::klass to rb_method_entry_t::owner.

internal.h: add rb_classext_struct::callable_m_tbl to cache rb_callable_method_entry_t data. We need to consider about this field again because it is only active for T_ICLASS.

class.c (method_entry_i):: ditto.

class.c (rb_define_attr): rb_method_entry() does not takes defiend_class_ptr.

gc.c (mark_method_entry): mark RCLASS_CALLABLE_M_TBL() for T_ICLASS.

cont.c (fiber_init): rb_control_frame_t::klass is removed.

proc.c: fix 'struct METHOD' data structure because rb_callable_method_t has all information.

vm_core.h: remove several fields.

rb_control_frame_t::klass.

rb_block_t::klass. And catch up changes.

eval.c: catch up changes.

gc.c: ditto.

insns.def: ditto.

class.c: ditto.

vm.c: ditto.

vm_args.c: ditto.

vm_backtrace.c: ditto.

vm_dump.c: ditto.

vm_eval.c: ditto.

vm_inshelper.c: ditto.

vm_method.c: ditto.

Revision 51126 - 07/03/2015 11:24 AM - ko1 (Koichi Sasada)

method.h: introduce rb_callable_method_entry_t to remove rb_control_frame_t::klass. [Bug #11278], [Bug #11279] rb_method_entry_t data belong to modules/classes. rb_method_entry_t::owner points defined module or class. module M def foo; end end In this case, owner is M. rb_callable_method_entry_t data belong to only classes. For modules, MRI creates corresponding T_ICLASS internally. rb_callable_method_entry_t can also belong to T_ICLASS. rb_callable_method_entry_t::defined_class points T_CLASS or T_ICLASS. rb_method_entry_t data for classes (not for modules) are also rb_callable_method_entry_t data because it is completely same data. In this case, rb_method_entry_t::owner == rb_method_entry_t::defined_class. For example, there are classes C and D, and includes M, class C; include M; end class D; include M; end then, two T_ICLASS objects for C's super class and D's super class will be created. When C.new.foo is called, then M#foo is searcheed and rb_callable_method_t data is used by VM to invoke M#foo. rb_method_entry_t data is only one for M#foo. However, rb_callable_method_entry_t data are two (and can be more). It is proportional to the number of including (and prepending) classes (the number of T_ICLASS which point to the module). Now, created rb_callable_method_entry_t are collected when the original module M was modified. We can think it is a cache. We need to select what kind of method entry data is needed. To operate definition, then you need to use rb_method_entry_t. You can access them by the following functions.

rb_method_entry(VALUE klass, ID id);
rb_method_entry_with_refinements(VALUE klass, ID id);
rb_method_entry_without_refinements(VALUE klass, ID id);
rb_resolve_refined_method(VALUE refinements, const rb_method_entry_t *me); To invoke methods, then you need to use rb_callable_method_entry_t which you can get by the following APIs corresponding to the above listed functions.

rb_callable_method_entry(VALUE klass, ID id);
rb_callable_method_entry_with_refinements(VALUE klass, ID id);
rb_callable_method_entry_without_refinements(VALUE klass, ID id);
rb_resolve_refined_method(VALUE refinements, const rb_callable_method_entry_t *me); VM pushes rb_callable_method_entry_t, so that rb_vm_frame_method_entry() returns rb_callable_method_entry_t. You can check a super class of current method by rb_callable_method_entry_t::defined_class.

method.h: renamed from rb_method_entry_t::klass to rb_method_entry_t::owner.

internal.h: add rb_classext_struct::callable_m_tbl to cache rb_callable_method_entry_t data. We need to consider about this field again because it is only active for T_ICLASS.

class.c (method_entry_i):: ditto.

class.c (rb_define_attr): rb_method_entry() does not takes defiend_class_ptr.

gc.c (mark_method_entry): mark RCLASS_CALLABLE_M_TBL() for T_ICLASS.

cont.c (fiber_init): rb_control_frame_t::klass is removed.

proc.c: fix 'struct METHOD' data structure because rb_callable_method_t has all information.

vm_core.h: remove several fields.

rb_control_frame_t::klass.

rb_block_t::klass. And catch up changes.

eval.c: catch up changes.

gc.c: ditto.

insns.def: ditto.

class.c: ditto.

vm.c: ditto.

vm_args.c: ditto.

vm_backtrace.c: ditto.

vm_dump.c: ditto.

vm_eval.c: ditto.

vm_inshelper.c: ditto.

vm_method.c: ditto.

History

11/15/2021
method.h: introduce rb_callable_method_entry_t to remove rb_control_frame_t::klass. [Bug #11279], [Bug #11279] rb_method_entry_t data belong to modules/classes. rb_method_entry_t::owner points defined module or class. module M def foo; end in this case, owner is M. rb_callable_method_entry_t data belong to only classes. For modules, MRI creates corresponding T_ICLASS internally. rb_callable_method_entry_t can also belong to T_ICLASS. rb_callable_method_entry_t::defined_class points T_CLASS or T_ICLASS. rb_method_entry_t data for classes (not for modules) are also rb_callable_method_entry_t data because it is completely same data. In this case, rb_method_entry_t::owner == rb_method_entry_t::defined_class. For example, there are classes C and D, and includes M, class C; include M; end class D; include M; end then, two T_ICLASS objects for C's super class and D's super class will be created. When C.new.foo is called, then M#foo is searched and rb_callable_method_t data is used by VM to invoke M#foo. rb_method_entry_t data is only one for M#foo. However, rb_callable_method_entry_t data are two (and can be more). It is proportional to the number of including (and prepending) classes (the number of T_ICLASS which point to the module). Now, created rb_callable_method_entry_t are collected when the original module M was modified. We can think it is a cache. We need to select what kind of method entry data is needed. To operate definition, then you need to use rb_method_entry_t. You can access them by the following functions.

- rb_method_entry(VALUE klass, ID id);
- rb_method_entry_with_refinements(VALUE klass, ID id);
- rb_method_entry_without_refinements(VALUE klass, ID id);
- rb_resolve_refined_method(VALUE refinements, const rb_method_entry_t *me); To invoke methods, then you need to use rb_callable_method_entry_t which you can get by the following APIs corresponding to the above listed functions.

- rb_callable_method_entry(VALUE klass, ID id);
- rb_callable_method_entry_with_refinements(VALUE klass, ID id);
- rb_callable_method_entry_without_refinements(VALUE klass, ID id);
- rb_resolve_refined_method_callable(VALUE refinements, const rb_callable_method_entry_t *me); VM pushes rb_callable_method_entry_t, so that rb_vm_frame_method_entry() returns rb_callable_method_entry_t. You can check a super class of current method by rb_callable_method_entry_t::defined_class.

method.h: renamed from rb_method_entry_t::klass to rb_method_entry_t::owner.

internal.h: add rb_classext_struct::callabe_m tbl to cache rb_callable_method_entry_t data. We need to consider about this field again because it is only active for T_ICLASS.

class.c (method_entry_i): ditto.

class.c (rb_define_attr): rb_method_entry() does not takes defiend_class_ptr.

gc.c (mark_method_entry): mark RCLASS_CALLABLE_M_TBL() for T_ICLASS.

cont.c (fiber_init): rb_control_frame_t::klass is removed.

proc.c: fix 'struct METHOD' data structure because rb_callable_method_t has all information.

vm_core.h: remove several fields.

- rb_control_frame_t::klass.
- rb_block_t::klass. And catch up changes.
- eval.c: catch up changes.
- gc.c: ditto.
- insns.def: ditto.
- vm.c: ditto.
- vm_args.c: ditto.
- vm_backtrace.c: ditto.
- vm_dump.c: ditto.
- vm_eval.c: ditto.
- vm_inshelper.c: ditto.
- vm_method.c: ditto.

#3 - 07/03/2015 11:37 AM - ko1 (Koichi Sasada)

I committed this change. If you find any regression, please report about it.

I measured some applications with [https://github.com/ko1/class_stat](https://github.com/ko1/class_stat) gem. This gem reports class/module/T_ICLASS usage.

For example, my rails app [https://github.com/ko1/tracer_demo_rails_app](https://github.com/ko1/tracer_demo_rails_app):

total_klasses 6204
total_included 398
total_iclasses 979
total_methods 23539
total_dup 10149

In this case,

- there are 6,000 classes and modules.
- 400 modules are included (or prepended).
- 1,000 T_ICLASSes are created.
• 24,000 methods are defined.
• 10,000 methods can be duplicated by this patch.

Last line needs explanation.
Without this patch, each method has one rb_method_entry_t (VALUE).
However, this patch makes that methods of modules needs additional rb_callable_method_entry_t for each T_ICLASS.

Roughly, 10,000 objects can be allocated additionally in this case.
(rb_callable_method_entry_t for methods in modules are allocated when called, so it does not mean increasing 10,000 objects immediately)

Recently, I reduced one objects per methods in trunk.
In this case, 24,000 objects. So I decided increasing 10,000 objects is acceptable.
This is why I commit-ed it.

We need to consider how to cache rb_callable_method_entry_t.
This is future work.

#4 - 03/24/2016 07:30 AM - usa (Usaku NAKAMURA)
- Related to Bug #12164: Binding UnboundMethod to BasicObject added

Files

| file.copipa-temp-image.png | 72.7 KB | 06/18/2015 | ko1 (Koichi Sasada) |