Ruby master - Bug #11278

remove rb_control_frame_t::klass

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

<table>
<thead>
<tr>
<th>Status:</th>
<th>Closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority:</td>
<td>Normal</td>
</tr>
<tr>
<td>Assignee:</td>
<td>ko1 (Koichi Sasada)</td>
</tr>
<tr>
<td>Target version:</td>
<td>ruby -v: 2.3dev</td>
</tr>
<tr>
<td>Backport:</td>
<td>2.0.0: UNKNOWN, 2.1: UNKNOWN, 2.2: UNKNOWN</td>
</tr>
</tbody>
</table>

**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).

<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>
hash_ident_flo 1.020
hash_ident_num 1.038
hash_ident_obj 1.036
hash_ident_str 1.055
hash_ident_sym 1.016
hash_keys 0.993
hash_shift 1.046
hash_values 1.006
io_file_create 0.983
io_file_read 0.985
io_file_write 1.014
io_select 0.958
io_select2 0.972
io_select3 1.027
loop_for 1.067
loop_generator 0.980
loop_times 1.078
loop_whileloop 0.995
loop_whileloop2 1.005
marshal_dump_flo 1.014
marshal_dump_load_geniv 0.989
marshal_dump_load_time 0.988
securerandom 0.944
so_ackermann 1.018
so_array 1.049
so_binary_trees 0.993
so_concatenate 1.036
so_count_words 1.012
so_exception 0.989
so_fannkuch 1.017
so_fasta 1.003
so_k_nucleotide 1.005
so_lists 1.001
so_mandelbrot 0.998
so_matrix 0.987
so_meteor_contest 1.035
so_nbody 0.997
so_nested_loop 1.054
so_nsieve 1.010
so_nsieve_bits 1.022
so_object 0.992
so_partial_sums 1.018
so_pidigits 0.993
so_random 0.981
so_reverse_complement 0.986
so_sieve 1.007
so_spectralnorm 1.014
vm1_attr_ivar* 0.991
vm1_attr_ivar_set* 0.987
vm1_block* 1.009
vm1_const* 0.983
vm1_ensure* 0.960
vm1_float_simple* 0.954
vm1_gc_short_lived* 1.002
vm1_gc_short_with_complex_long* 1.004
vm1_gc_short_with_long* 0.996
vm1_gc_short_with_symbol* 0.998
vm1_gc_wb_ary* 1.004
vm1_gc_wb_ary_promoted* 1.141
vm1_gc_wb_obj* 0.998
vm1_gc_wb_obj_promoted* 0.963
vm1_iivar* 0.982
vm1_iivar_set* 1.010
vm1_length* 1.006
vm1_livar_init* 0.938
vm1_livar_set* 0.990
vm1_neq* 0.987
<table>
<thead>
<tr>
<th>Method</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>vm1_not*</td>
<td>1.013</td>
</tr>
<tr>
<td>vm1_rescue*</td>
<td>1.053</td>
</tr>
<tr>
<td>vm1_simplereturn*</td>
<td>1.030</td>
</tr>
<tr>
<td>vm1_swap*</td>
<td>1.017</td>
</tr>
<tr>
<td>vm1_yield*</td>
<td>1.032</td>
</tr>
<tr>
<td>vm2_array*</td>
<td>0.987</td>
</tr>
<tr>
<td>vm2_bigarray*</td>
<td>1.014</td>
</tr>
<tr>
<td>vm2_bighash*</td>
<td>0.987</td>
</tr>
<tr>
<td>vm2_case*</td>
<td>1.001</td>
</tr>
<tr>
<td>vm2_defined_method*</td>
<td>1.003</td>
</tr>
<tr>
<td>vm2_dstr*</td>
<td>0.997</td>
</tr>
<tr>
<td>vm2_eval*</td>
<td>0.982</td>
</tr>
<tr>
<td>vm2_method*</td>
<td>1.011</td>
</tr>
<tr>
<td>vm2_method_missing*</td>
<td>0.973</td>
</tr>
<tr>
<td>vm2_method_with_block*</td>
<td>1.027</td>
</tr>
<tr>
<td>vm2_mutex*</td>
<td>1.065</td>
</tr>
<tr>
<td>vm2_newlambda*</td>
<td>1.014</td>
</tr>
<tr>
<td>vm2_poly_method*</td>
<td>0.962</td>
</tr>
<tr>
<td>vm2_poly_method_ov*</td>
<td>0.972</td>
</tr>
<tr>
<td>vm2_proc*</td>
<td>1.058</td>
</tr>
<tr>
<td>vm2_raise1*</td>
<td>0.977</td>
</tr>
<tr>
<td>vm2_raise2*</td>
<td>0.990</td>
</tr>
<tr>
<td>vm2_regexp*</td>
<td>1.006</td>
</tr>
<tr>
<td>vm2_send*</td>
<td>1.005</td>
</tr>
<tr>
<td>vm2_struct_big_aref_hi*</td>
<td>1.005</td>
</tr>
<tr>
<td>vm2_struct_big_aref_lo*</td>
<td>1.010</td>
</tr>
<tr>
<td>vm2_struct_big_aset*</td>
<td>1.005</td>
</tr>
<tr>
<td>vm2_struct_small_aref*</td>
<td>1.030</td>
</tr>
<tr>
<td>vm2_struct_small_aset*</td>
<td>1.019</td>
</tr>
<tr>
<td>vm2_super*</td>
<td>0.900</td>
</tr>
<tr>
<td>vm2_unif1*</td>
<td>1.031</td>
</tr>
<tr>
<td>vm2_zsuper*</td>
<td>0.913</td>
</tr>
<tr>
<td>vm3_backtrace</td>
<td>1.004</td>
</tr>
<tr>
<td>vm3_clearmethodcache</td>
<td>0.937</td>
</tr>
<tr>
<td>vm3_gc</td>
<td>0.996</td>
</tr>
<tr>
<td>vm_thread_alive_check1</td>
<td>0.963</td>
</tr>
<tr>
<td>vm_thread_close</td>
<td>1.028</td>
</tr>
<tr>
<td>vm_thread_create_join</td>
<td>1.007</td>
</tr>
<tr>
<td>vm_thread_mutex1</td>
<td>1.047</td>
</tr>
<tr>
<td>vm_thread_mutex2</td>
<td>1.842</td>
</tr>
<tr>
<td>vm_thread_mutex3</td>
<td>1.028</td>
</tr>
<tr>
<td>vm_thread_pass</td>
<td>0.665</td>
</tr>
<tr>
<td>vm_thread_pass_flood</td>
<td>0.960</td>
</tr>
<tr>
<td>vm_thread_pipe</td>
<td>0.998</td>
</tr>
<tr>
<td>vm_thread_queue</td>
<td>0.995</td>
</tr>
</tbody>
</table>

file.copipa-temp-image.png

Not so big change. vm2_super/zsuper should improve performance so I need to check it again.

**Memory consumption**

Runing this script to check process memory on Linux Ubuntu.

```
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|
```
def define_method("bar#{i}"){}
  if $cls
    end
  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/)

This program makes 100_000 methods for a module and classes. Maybe it is too big example.

Making methods on classes and a module.

ruby 2.2
VmHWM:  247624  kB
trunk
VmHWM:  234004  kB
modified
VmHWM:  252236  kB

Making methods only on a module.

ruby 2.2
VmHWM:  77848  kB
trunk
VmHWM:  86452  kB
modified
VmHWM:  108756  kB

Making methods only on classes.

ruby 2.2
VmHWM:  175780  kB
trunk
VmHWM:  182944  kB
modified
VmHWM:  179216  kB

As you can see, first result shows 2% increase for memory usage compare to Ruby 2.2. Second result shows 40% increase, but it is worst case. Third result is best case (no methods in modules).

We need to check real usage.

**Future work**
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::class
  - 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::class. [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 Mfoo is searched and rb_callable_method_entry_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(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: renamed from rb_method_entry_t::class 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::class 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::class.
- rb_block_t::class. 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.

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@51126 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

07/24/2021
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::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.
ruby_callable_method_entry_t data belong to only classes. For modules, MRI creates corresponding T_ICLASS internally.
ruby_callable_method_entry_t can also belong to T_ICLASS. ruby_callable_method_entry_t::defined_class points T_CLASS or T_ICLASS.
ruby_method_entry_t data for classes (not for modules) are also ruby_callable_method_entry_t data because it is completely same data. In this case,
ruby_method_entry_t::owner == ruby_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
Mfoo is searched and rb_callable_method_t data is used by VM to invoke Mfoo. ruby_method_entry_t data is only one for Mfoo. However,
ruby_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
ruby_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
ruby_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
ruby_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.
- vm.c: ditto.
- vm_args.c: ditto.
- vm_backtrace.c: ditto.
- vm_dump.c: ditto.

07/24/2021
rb_callable_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(VALUE klass, ID id);`

rb_callable_method_entry_t::defined_class. For example, there are classes C and D, and include C, 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_entry_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);`

You can check a super class of current method by `rb_callable_method_entry(VALUE klass, ID id);`

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

- `rb_callable_method_entry_without_refinements(VALUE klass, ID id);`
- `rb_callable_method_entry_with_refinements(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(VALUE klass, ID id);`

Method.h: introduce rb_callable_method_entry_t to remove rb_control_frame_t::class. [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::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 M.foo is searched and rb_callable_method_entry_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);`

Method.h: renamed from rb_method_entry_t::class to rb_method_entry_t::owner.

Method.h: add rb_classextend::callable_method_table to cache rb_callable_method_entry_t data. We need to consider about this field again because it is only active for T_ICLASS.

Method.h: introduce rb_callable_method_entry_t to remove rb_control_frame_t::class. [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::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 M.foo is searched and rb_callable_method_entry_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);`

Method.h: introduce rb_callable_method_entry_t to remove rb_control_frame_t::class. [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::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 M.foo is searched and rb_callable_method_entry_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);`
- `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_ICALASS.

- `class.c (method_entry_i):` ditto.

- `class.c (rb_define_attr):` `rb_method_entry()` does not takes defined_class_ptr.

- `gc.c (mark_method_entry):` mark RCLASS_CALLABLE_M_TBL() for T_ICALASS.

- `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_ICALASS internally. `rb_callable_method_entry_t` can also belong to T_ICALASS. `rb_callable_method_entry_t::defined_class points T_CLASS or T_ICALASS. 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_ICALASS 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_entry_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_ICALASS 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()`. 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_ICALASS.

- `class.c (method_entry_i):` ditto.

- `class.c (rb_define_attr):` `rb_method_entry()` does not takes defined_class_ptr.

- `gc.c (mark_method_entry):` mark RCLASS_CALLABLE_M_TBL() for T_ICALASS.

- `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.

History
#1 - 06/18/2015 11:36 AM - ko1 (Koichi Sasada)
- Related to Bug #11279: remove rb_control_frame_t::klass added

#2 - 07/03/2015 11:25 AM - ko1 (Koichi Sasada)
- Status changed from Open to Closed

Applied in changeset r51126.

- 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 defines 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_CLASS or T_ICLASS internally.
  - rb_callable_method_entry_t can also belong to T_CLASS or T_ICLASS. The 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_callable_method_entry_t::defined_class. For example, there are classes C and D, and include M, class C; include M; 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 M#foo is searched and rb_callable_method_entry 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_CLASS 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_class_entry_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 takes defined_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.

#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 gem. This gem reports class/module/T_ICLASS usage.

For example, my 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

<table>
<thead>
<tr>
<th>file</th>
<th>Size</th>
<th>Date</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>file.copipa-temp-image.png</td>
<td>72.7 KB</td>
<td>06/18/2015</td>
<td>ko1 (Koichi Sasada)</td>
</tr>
</tbody>
</table>