**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 solves 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`.

```ruby
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.

```ruby
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.

```ruby
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).

Speedup ratio: compare with the result of `trunk' (greater is better)
name modified
---------------
app_answer 1.032
app_aobench 0.989
app.erb 1.006
app_factorial 1.000
app_fib 1.026
app_lc_fizzbuzz 1.144
app_mandelbrot 1.032
app_pentomino 0.996
app_raise 0.996
app_strconcat 0.981
app_tak 0.999
app_tarai 1.004
app_uri 1.001
array_shift 0.913
hash_aref_flo 1.023
hash_aref_miss 1.097
hash_aref_str 1.074
hash_aref_sym 1.051
hash_aref_sym_long 1.047
hash_flatten 1.002
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.

\[ N = 100\,000 \]
$mod = true
$cls = true

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

class C
    include M
    N.times{|i|
        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/)

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
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::klass | Closed |
| Related to Ruby master - Bug #12164: Binding UnboundMethod to BasicObject | Closed |

Associated revisions

Revision 5e8a1474 - 07/03/2015 11:24 AM - ko1 (Koichi Sasada)

- method.h: introduce rb_callable_method_entry_t to remove rb_control_frame_t::klass.
- 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 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);
    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 take 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.
- 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.

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.
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_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);
    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
  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.

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 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);
    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_classextend_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 RCCLASS_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]
ruby_method_entry_t data belong to modules/classes.
ruby_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 M#foo is searched and
ruby_callable_method_t data is used by VM to invoke M#foo.
ruby_method_entry_t data is only one for M#foo.
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 ruby_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.
ruby_method_entry(VALUE klass, ID id);
ruby_method_entry_with_refinements(VALUE klass, ID id);
ruby_method_entry_without_refinements(VALUE klass, ID id);
ruby_resolve_refined_method(VALUE refinements, const ruby_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 `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_traceback.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

```ruby
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.

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_entry_t` data is used by VM to invoke `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.

08/28/2022
To invoke methods, 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_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 `defined_class_ptr`.

You can check a super class of current method by `rb_callable_method_entry_t::defined_class`.

- `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_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_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);
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
  defined_class_ptr.
- gc.c (mark_method_entry): mark RCLASS_CALLABLE_M_TBL() for T_ICLASS.
- cont.c (fiber_deinit): 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]
r
  rb_method_entry_t data belong to modules/classes.
r
  rb_method_entry_t::owner points defined module or class.
r
  module M
  def foo; end
end
In this case, owner is M.
r
  rb_callable_method_entry_t data belong to only classes.
For modules, MRI creates corresponding T_ICLASS internally.
r
  rb_callable_method_entry_t can also belong to T_ICLASS.
  rb_callable_method_entry_t::defined_class points T_CLASS or
  T_ICLASS.
r
  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_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);
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.

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

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