# Feature Proposal: Method#super_method

**04/28/2014 10:07 PM - schneems (Richard Schneeman)**

<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>okkez (okkez _)</td>
</tr>
<tr>
<td>Target version:</td>
<td>2.2.0</td>
</tr>
</tbody>
</table>

## Description

When super is called in a method the Ruby VM knows how to find the next ancestor that has that method and call it. It is difficult to do this manually, so I propose we expose this information in Method#super_location.

Ruby Method class ([http://www.ruby-doc.org/core-2.1.1/Method.html](http://www.ruby-doc.org/core-2.1.1/Method.html)) is returned by calling `Object.method` and passing in a method name ([http://www.ruby-doc.org/core-2.1.1/Object.html#method-i-method](http://www.ruby-doc.org/core-2.1.1/Object.html#method-i-method)). This is useful for debugging:

```ruby
# /tmp/code.rb
class Foo
def bar
end
end
puts Foo.new.method(:bar).source_location
# => ['/tmp/code.rb', 3]
```

The `Object#method` allows a Ruby developer to easily track the source location of the method and makes debugging very easy. However, if the code is being invoked by a call to `super` it is difficult to track down:

```ruby
# /tmp/code.rb
class BigFoo
def bar
end
end
class Foo < BigFoo
def bar
  super
end
end
```

In this code sample it is easy to find the method definition inside of `Foo` but it is very difficult in large projects to find what code exactly `super` is calling. This simple example is easy, but it can be hard when there are many ancestors. Currently if I wanted to find this we can inspect ancestors:

```ruby
Foo.ancestors[1..-1].map do |ancestor|
  next unless ancestor.method_defined?(:bar)
  ancestor.instance_method(:bar)
end.compact.first.source_location
```

To make this process simpler I am proposing a method on the `Method` class that would return the result of `super`:

```ruby
Foo.new.method(:bar).super_method
```

I believe adding `Method#super_method`, or exposing this same information somewhere else, could greatly help developers to debug large systems easily.

## Related issues:

- Related to Ruby master - Feature #7836: Need a way to get Method and UnboundMethod source location... - Closed
- Has duplicate Ruby master - Feature #10216: Add methods to Method and UnboundMethod... - Closed

---

03/15/2022
I also have this problem especially when debugging code where modules are mixed in at runtime, so I have to do something like this:

class << obj; self; end.ancestors.find_all { |klass|
  klass.method_defined? :foo
}.map { |klass| klass.method(:foo).source_location }.last

(or something similar). A method to get the "super method" would be very convenient.

A patch.
No tests yet.
method_owner(VALUE obj)
return defined_class;
}

void
rb_method_name_error(VALUE klass, VALUE str)
+struct method_name_error {
+ VALUE class_name;
+ const char *type;
+};
+
+static struct method_name_error
+prepare_method_name_error(VALUE klass)
+
+const char s0 = " class";
+ VALUE c = klass;
+ struct method_name_error e;
+
+ if (FL_TEST(c, FL_SINGLETON)) {
+ VALUE obj = rb_ivar_get(klass, attached);
+}
else if (RB_TYPE_P(c, T_MODULE)) {
+  s0 = " module";
+}
+
+ rb_name_error_str(str, "undefined method `%"PRIsVALUE"' for%s `%"PRIsVALUE"'",
+  QUOTE(str), s0, rb_class_name(c));
+ e.class_name = rb_class_name(c);
+ e.type = s0;
+ return e;
+
+#define method_name_error(klass, str, t) do {  
+   struct method_name_error e = prepare_method_name_error(klass);
+   rb_name_error_str(str, t" method `%"PRIsVALUE"' for%s `%"PRIsVALUE"'",
+                     QUOTE(str), e.type, e.class_name);
+ } while (0)
+
+void
+rb_method_name_error(VALUE klass, VALUE str)
+
+  method_name_error(klass, str, "undefined");
+
+/*
+ method_proc(VALUE method)
+ return procval;
+
+static VALUE
+method_super_method(VALUE method)
+
+ TypedData_Get_Struct(method, struct METHOD, &method_data_type, data);
+ defined_class = data->defined_class;
+ if (BUILTIN_TYPE(defined_class) == T_MODULE) defined_class = data->rclass;
+ super_class = RCLASS_SUPER(defined_class);
+ if (!super_class) {
+   method_name_error(defined_class, rb_id2str(data->id), "no superclass");
+ }
+ return mnew(super_class, data->recv, data->id,
+               rb_obj_class(method), FALSE);
+*/
+/*
+ call-seq:
+   local_jump_error.exit_value -> obj
+*/
03/15/2022
I would also find this feature very useful when debugging large code bases with complicated class/module hierarchies.

What should happen on #call method with method objects returned from super_method?

It's an ordinary Method (or UnboundMethod) instance, same as SuperClass.instance_method(:foo).bind(obj).

Oh, OK. I misunderstood the spec of Method class.

It's an ordinary Method (or UnboundMethod) instance, same as SuperClass.instance_method(:foo).bind(obj).

For one level, with simple class inheritance, it probably isn't, but throw in a prepended module or two, and the desire to go up more than one level (i.e. obj.method(:foo).super_method.super_method) and it's not completely trivial.

If ever we support a module being included more than once in the ancestry chain, then it's actually impossible (unless Method provides a more
I'm +1 for this feature, although I would favor the shorter name super. I don't think it would ever conflict with the keyword, and if it did there is still the same solution as with class, i.e. self.super.

Nobuyoshi Nakada wrote:

It's an ordinary Method (or UnboundMethod) instance, same as SuperClass.instance_method(:foo).bind(obj).

Agreed for Method, but I'm not sure I understand how we could define UnboundMethod#super_method, since a Module can be part of different ancestry chains.

module M
  def bar
  end
end

Foo.include bar

Foo.ancestors #=> [Foo, M, BigFoo, ...]
Foo.new.method(:bar).super_method.super_method.owner #=> BigFoo
Foo.instance_method(:bar).super_method.super_method.owner #=> Can't possible give meaningful result

(I didn't try the patch)

Nobu: Sorry, I was confused, there's no problem with UnboundMethod#super_method because we retain which class this method was accessed from.

Ryan: That's actually a good example, there is no api to get the "original owner" of an unbound method, so in general it's not possible to implement super_method in Ruby. E.g. from Hash.instance_method(:map), I don't know of a way to get back Hash in pure Ruby, and thus no way to go through Hash's ancestry chain.

Maybe I'm still not getting something. If you can call it (or super to it) you can grab it:

class BigFoo
  def bar
  end
end

class Middle1 < BigFoo; end
class Middle2 < Middle1; end
class Middle3 < Middle2; end
class Middle4 < Middle3; end
class Middle5 < Middle4; end

class Foo < Middle5
  def bar
    super
  end
end

class Object
  def super_method name
    self.class.superclass.instance_method name
  end
end

p Foo.new.method(:bar).source_location.last #=> 13
p Foo.new.class.superclass.instance_method(:bar).source_location.last #=> 2
p Foo.new.super_method(:bar).source_location.last #=> 2

Ryan, using superclass gets you really close, but doesn't handle extending object instances:

class BigFoo

#12 - 05/02/2014 03:02 PM - schneems (Richard Schneeman)
```ruby
def bar
end
end

class Foo < BigFoo
def bar
  super
end
end
class Object
def super_method
  self.class.superclass.instance_method(name)
end
end

module MixinFoo
def bar
  puts "MixinFoo"
end
end

foo = Foo.new
foo.extend(MixinFoo)
foo.bar # => "MixinFoo"

puts foo.super_method(:bar) #<UnboundMethod: BigFoo#bar> # wrong return

This is why my original code snippet uses ancestors rather than just superclass. We should see MixinFoo rather than BigFoo.

#13 - 05/08/2014 07:51 PM - josh.cheek (Josh Cheek)
I've only ever needed to do this in codebases like Rails, where there's an insane amount of inheritance. However, Rails also relies heavily on method_missing, which super_method would completely miss.

Anyway, I'd like this method to exist also.

Here's how pry does it: https://github.com/pry/pry/blob/06341dfadd53dc77e36761a974e6c99040dfb86/02/pry/method.rb#L382

Here's my crack at implementing it:

class Method
def super_method
  receiver.singleton_class.ancestors.drop_while { |ancestor| ancestor != owner }.first.instance_method(name).bind(receiver)
rescue NameError
  raise NameError, "No super method \"#{name}\" for \"#{inspect}\""
end
end
class BigFoo
def bar
end
end

# toss in some superclass spam to show it doesn't matter
class Foo < Class.new(Class.new(Class.new(BigFoo)))
def bar
end
end
module MixinFoo
def bar
end
end

Foo.new # => #<Foo:0x007fc123148728>
```

03/15/2022
14 - 05/30/2014 07:41 AM - nobu (Nobuyoshi Nakada)
- Related to Feature #7836: Need a way to get Method and UnboundMethod objects to methods overridden by prepended modules added

15 - 06/10/2014 08:32 AM - ko1 (Koichi Sasada)
- Assignee set to matz (Yukihiro Matsumoto)
- Target version set to 2.2.0

16 - 06/10/2014 08:51 AM - matz (Yukihiro Matsumoto)
- Status changed from Open to Feedback

Having something like Method/UnboundMethod#super_method for debugging purpose is OK, but

- Is super_method is the right name? It sounds little bit weird for me.
- I don't like Object#super_method.

Matz.

17 - 06/10/2014 06:07 PM - schneems (Richard Schneeman)
I am also not in love with the naming. I would have preferred Method#super but we don't want to overwrite super or confuse functionality here. Maybe something along the lines of get_super. Does anyone have a better naming suggestion?

18 - 06/11/2014 11:43 AM - marcandre (Marc-Andre Lafortune)
I also favor super (as I wrote before). It wouldn't overwrite the super keyword in most cases, and if it does (say from an instance method of Method) then there is still self.super.

19 - 07/26/2014 08:23 AM - matz (Yukihiro Matsumoto)
OK, accepted.

The name should be 'super_method', since 'super' make me feel invocation of a method in super class.
The 'super_method' should return nil when there's no method in superclasses.

Matz.

20 - 07/26/2014 08:24 AM - matz (Yukihiro Matsumoto)
- Status changed from Feedback to Open
- Assignee changed from matz (Yukihiro Matsumoto) to okkez (okkez _)

21 - 07/26/2014 04:22 PM - nobu (Nobuyoshi Nakada)
- Status changed from Open to Closed
- % Done changed from 0 to 100

Applied in changeset r46964.

---

proc.c: method_super_method

- proc.c (method_super_method): new method Method#super_method, which returns a method object of the method to be called by super in the receiver method object. [ruby-core:62202] [Feature #9781]

22 - 08/02/2014 11:04 AM - aledovsky (Aleksandrs Žedovskis)
Yukihiro Matsumoto wrote:

OK, accepted.

The name should be 'super_method', since 'super' make me feel invocation of a method in super class.
The 'super_method' should return nil when there's no method in superclasses.

Matz.
Wouldn't it be more consistent to name this method supermethod, i.e. without "_" in the middle? Thus it would be logical extension to super<something> family, which now consists of Class#superclass

#23 - 09/08/2014 01:51 PM - nobu (Nobuyoshi Nakada)
- Has duplicate Feature #10216: Add methods to Method and UnboundMethod classess to retrieve method instance for super added

#24 - 02/10/2015 01:38 PM - Eregon (Benoit Daloze)
Marc-Andre Lafontune wrote:

I must be tired.

Nobu: Sorry, I was confused, there's no problem with UnboundMethod#super_method because we retain which class this method was accessed from.

Ryan: That's actually a good example, there is no api to get the "original owner" of an unbound method, so in general it's not possible to implement super_method in Ruby. E.g. from Hash.instance_method(:map), I don't know of a way to get back Hash in pure Ruby, and thus no way to go through Hash's ancestry chain.

Actually there is a caveat with an UnboundMethod created with Module#instance_method, from a module (and not a class), as it has no idea what is the actual super method (as that can change in different ancestry chains). Instead it will just return a method if there is a module "M" included in the former module with a corresponding method, and that method might be somewhere in the ancestry of an object including M or not (if "M" was included after).