Refinement doesn't work if using locate after method.
(I eliminate discussion because my laptop doesn't have enough power...)

class C
def foo
p :C_foo
end
end

module M1
refine C do
def foo
p :M1_foo
super
end
end
end

module M2
refine C do
def foo
p :M2_foo
super
end
end
end

class D
using M1
def x
C.new.foo
end
using M2
end

p :x
D.new.x

#=>
:x
:M1_foo
:C_foo

I don't like the idea that using should affect methods already defined. At this point I view using similar to private/protected/etc, which also do not affect methods defined before you call them.
I don't like the idea that using should affect methods already defined. At this point I view using similar to private/protected/etc, which also do not affect methods defined before you call them.

I feel that using' is similar to include' and prepend' rather than public' and 'private'.

--
// SASADA Koichi at atdot dot net

The following code (C_User2#x) affect refinement 'after' defining method. Is that intentional?

```ruby
###
class C
def foo
  p :C_foo
end
end

module RefineC
  refine C do
    def foo
      p :RefineC_foo
      super
    end
  end
end

class C_User
  using RefineC
  def x
    C.new.foo
  end
end

class C_User2
  def x
    self.class.send(:using, RefineC)
    C.new.foo
  end
end

puts "C.new.foo"
C.new.foo

puts "C_User.new.x"
C_User.new.x

puts "C_User2.new.x"
C_User2.new.x

#=>
C.new.foo
  :C_foo
C_User.new.x
```

8/07/2021
class C
  def foo
    p :C_foo
  end
end

module RefineC
  refine C do
    def foo
      p :RefineC_foo
      super
    end
  end
end

class C_User
  using RefineC
  def x
    C.new.foo
  end
end

class C_User2
  def x
    self.class.send(:using, RefineC)
    C.new.foo
  end
end

puts "C.new.foo"
C.new.foo
puts "C_User.new.x"
C_user.new.x
puts "C_User2.new.x"
C_User2.new.x

#=>
C.new.foo
:C_foo
C_User.new.x
:RefineC_foo
:C_foo
C_User2.new.x
:RefineC_foo
:C_foo

# is that corner case? :)
class C_User2
  def x
    2.times{
      C.new.foo
      self.class.send(:using, RefineC)
    }
  end
end
is that corner case? :)

class C_User2
  def x
    2.times{
      C.new.foo
      self.class.send(:using, RefineC)
    }
  end
end

I commented on the other bug about how refinements need to be temporal to limit their impact (implementation-wise), but there are obvious flaws in making them temporal too. Your example above shows how ordering can change what method will be called. My example was using calls that happen up-hierarchy in different files.

Even if we ignore implementation/performance concerns (and there are lots of them), there are many behavioral problems like this with refinements.

I've started to wonder if it's better to limit refinements based on modules instead of lexical scopes.

In the current implementation a cref has a table for activated refinements, but in module-based refinements a module linked from the cref have the table. Furthermore, in the current implementation the table is shared and copied-on-write by nested crefs, but it may be better to separate them and search refinements in outer modules when a method is not found in the table of the current module.

For example,

module Foo
  using X
  # Foo's table has refinements in X.
module Bar
  using Y
  # Bar's table has refinements in Y, not X.

C.new.foo  # First, Bar's table is searched, then Foo's table is searched.
end
end

In module-based refinements, toplevel using should affects not only the current file, but global.

OK, I understand the behavior. What are pros and cons of module based refinement?

Matz.

#8 - 11/05/2012 01:42 PM - headius (Charles Nutter)
Making refinements less lexical seems like the wrong direction to me. It means all calls everywhere have to check for refinements all the time, similar to the module_eval problem.

In the following code:

a.rb:
```ruby
module Foo
  def self.go
    C.new.foo
  end
end
```

b.rb:
```ruby
module Foo
  using X
end
```

I assume that regardless of the order of loading these two files, calling the "go" method would require searching the refinements table of Foo, correct? That seems pretty clearly to indicate that module-based refinements require all call sites everywhere to check for refinements every time.

#9 - 11/05/2012 01:49 PM - headius (Charles Nutter)
I should also mention that if we can't tell ahead of time that a call site has to search refinements, it means all calls everywhere will have to have cref available. In JRuby, this could easily be a crippling blow to performance, and in MRI it would make it impossible to eliminate Ruby call frames (or eliminate cref management) ever in the future (which I'm sure ko1 would like to be able to do).

Currently JRuby only needs the cref to be available if there's a closure, binding-related call, or constant lookup. We'd now have to make it available 100% of the time.

It also means that you will never again be able to look at a piece of code and know if refinements will affect it. Refinements would become one of the most confusing features in Ruby.

After talking with Yehuda a bit, I am more and more of the opinion that "using" should only affect call sites lexically in the same scope as (or a child scope of) the "using" call, and only call sites that appear after the "using" call. This would mean having to use "using" in every file where you want refinements active, but it would make it very where refinements are active both to the programmer and to the VM.

#10 - 11/05/2012 02:49 PM - shugo (Shugo Maeda)
matz (Yukihiro Matsumoto) wrote:

> OK, I understand the behavior. What are pros and cons of module based refinement?

pros:
- not affected by the order of loading/using
- consistent with module_eval and behavior when a module is reopened (Bug #7271)

cons:
- might be hard to implement efficiently
  - especially super when refinements are cascaded might be hard to implement

#11 - 12/13/2012 02:16 PM - shugo (Shugo Maeda)
- Status changed from Open to Closed

Module#using is removed, so I close this ticket.