Ruby - Bug #16852

Refining Enumerable fails with ruby 2.7

05/12/2020 08:52 PM - parker (Parker Finch)

Status:	Closed		
Priority:	Normal		
Assignee:			
Target version:			
ruby -v:	ruby 2.7.1p83 (2020-03-31 revision a0c7c23c9c) [x86_64-darwin19]	**Backport:**	2.5: DONTNEED, 2.6: DONTNEED, 2.7: REQUIRED

Description

When using rspec and ruby 2.7 I am unable to refine Enumerable. I have created an issue in rspec but I'm wondering if there's an underlying ruby issue. (This bug does not occur when using ruby 2.6 or 2.5.)

Code to reproduce

(Also attached as a .tar.gz):

```
# file: refinement_bug.rb
module RefinementBug
  refine Enumerable do
    def refined_method
      puts "Called #refined_method"
    end
  end
end

# file: spec/spec_helper.rb
require_relative "../refinement_bug"

# file: spec/test.rb

require_relative "../refinement_bug"

using RefinementBug

puts "Using Ruby #{RUBY_VERSION}"
[].refined_method
```

After installing rspec (gem install rspec), call via rspec spec/test.rb --format progress --require spec_helper.

Expected Behavior

When run on Ruby 2.6 and 2.5 I get the expected behavior. The Enumerable module is refined, which allows an instance of Array to call #refined_method.

```
# This is output when running on ruby 2.6.6, and is expected.
~/minimal_refinement_failure> rspec spec/test.rb
Using Ruby 2.6.6
Called #refined_method
No examples found.

Finished in 0.0004 seconds (files took 0.08706 seconds to load)
0 examples, 0 failures
```

Actual Behavior

When I run the same code on ruby 2.7 I get a NoMethodError, suggesting that the Enumerable module has not been refined.

```
~/minimal_refinement_failure> rspec spec/test.rb
Using Ruby 2.7.1
```

```
An error occurred while loading ./spec/test.rb.
Failure/Error: [].refined_method

NoMethodError:
  undefined method `refined_method' for []:Array
# ./spec/test.rb:8:in `<top (required)>'
No examples found.

Finished in 0.00003 seconds (files took 0.12287 seconds to load)
0 examples, 0 failures, 1 error occurred outside of examples
```

Investigation

There are a few changes that prevent this bug from manifesting:

- If I refine Kernel instead of Enumerable then I get the expected behavior using Ruby 2.7.
- If I refine Array instead of Enumerable then I get the expected behavior using Ruby 2.7.

This bug does *not* manifest if run directly via ruby (i.e. ruby -r ./spec/spec_helper.rb spec/test.rb succeeds). It only manifests when run via rspec.

I can also get the expected behavior by changing the rspec options:

- If I don't use the --require option with rspec (e.g. run it as rspec spec/test.rb --format progress) then I get the expected behavior.
- If I don't use the --format option with rspec (e.g. run it as rspec spec/test.rb --require spec_helper) then I get the expected behavior.

This suggests the underlying issue might be in rspec. But if it is, I'm curious why the change to ruby 2.7 causes this to manifest, and whether Ruby 2.7 accidentally broke some expected behavior.

Thank you for looking into this, I am very confused :)
Let me know how I can help with this!

Associated revisions

Revision 98286e9850936e27e8ae5e4f20858cc9c13d2dde - 06/03/2020 04:50 PM - jeremyevans (Jeremy Evans)

Ensure origins for all included, prepended, and refined modules

This fixes various issues when a module is included in or prepended
to a module or class, and then refined, or refined and then included
or prepended to a module or class.

Implement by renaming ensure_origin to rb_ensure_origin, making it
non-static, and calling it when refining a module.

Fix Module#initialize_copy to handle origins correctly. Previously,
Module#initialize_copy did not handle origins correctly. For example,
this code:

```
module B; end
class A
  def b; 2 end
  prepend B
end
a = A.dup.new
class A
  def b; 1 end
end
p a.b
```

Printed 1 instead of 2. This is because the super chain for
a.singleton_class was:

```
a.singleton_class
A.dup
B(iclass)
B(iclass origin)
A(origin) # not A.dup(origin)
```

The B iclasses would not be modified, so the includer entry would be
still be set to A and not A.dup.

This modifies things so that if the class/module has an origin,
all iclasses between the class/module and the origin are duplicated
and have the correct includer entry set, and the correct origin
is created.

This requires other changes to make sure all tests still pass:

- rb_undef_methods_from doesn't automatically handle classes with
  origins, so pass it the origin for Comparable when undefing
  methods in Complex. This fixed a failure in the Complex tests.

- When adding a method, the method cache was not cleared
  correctly if klass has an origin. Clear the method cache for
  the klass before switching to the origin of klass. This fixed
  failures in the autoload tests related to overridding require,
  without breaking the optimization tests. Also clear the method
  cache for both the module and origin when removing a method.

- Module#include? is fixed to skip origin iclasses.

- Refinements are fixed to use the origin class of the module that
  has an origin.

- RCLASS_REFINED_BY_ANY is removed as it was only used in a single
  place and is no longer needed.

- Marshal#dump is fixed to skip iclass origins.

- rb_method_entry_make is fixed to handled overridden optimized
  methods for modules that have origins.

Fixes [Bug #16852]

Revision 98286e9850936e27e8ae5e4f20858cc9c13d2dde - 06/03/2020 04:50 PM - jeremyevans (Jeremy Evans)

Ensure origins for all included, prepended, and refined modules

This fixes various issues when a module is included in or prepended
to a module or class, and then refined, or refined and then included
or prepended to a module or class.

Implement by renaming ensure_origin to rb_ensure_origin, making it
non-static, and calling it when refining a module.

Fix Module#initialize_copy to handle origins correctly. Previously,
Module#initialize_copy did not handle origins correctly. For example,
this code:

```
module B; end
class A
  def b; 2 end
  prepend B
end
a = A.dup.new
class A
  def b; 1 end
end
p a.b
```

Printed 1 instead of 2. This is because the super chain for
a.singleton_class was:

```
a.singleton_class
A.dup
B(iclass)
B(iclass origin)
A(origin) # not A.dup(origin)
```

The B iclasses would not be modified, so the includer entry would be
still be set to A and not A.dup.

This modifies things so that if the class/module has an origin,
all iclasses between the class/module and the origin are duplicated
and have the correct includer entry set, and the correct origin
is created.

This requires other changes to make sure all tests still pass:

- rb_undef_methods_from doesn't automatically handle classes with origins, so pass it the origin for Comparable when undefing methods in Complex. This fixed a failure in the Complex tests.

- When adding a method, the method cache was not cleared correctly if klass has an origin. Clear the method cache for the klass before switching to the origin of klass. This fixed failures in the autoload tests related to overridding require, without breaking the optimization tests. Also clear the method cache for both the module and origin when removing a method.

- Module#include? is fixed to skip origin iclasses.

- Refinements are fixed to use the origin class of the module that has an origin.

- RCLASS_REFINED_BY_ANY is removed as it was only used in a single place and is no longer needed.

- Marshal#dump is fixed to skip iclass origins.

- rb_method_entry_make is fixed to handled overridden optimized methods for modules that have origins.

Fixes [Bug #16852]

Revision 98286e98 - 06/03/2020 04:50 PM - jeremyevans (Jeremy Evans)

Ensure origins for all included, prepended, and refined modules

This fixes various issues when a module is included in or prepended
to a module or class, and then refined, or refined and then included
or prepended to a module or class.

Implement by renaming ensure_origin to rb_ensure_origin, making it
non-static, and calling it when refining a module.

Fix Module#initialize_copy to handle origins correctly. Previously,
Module#initialize_copy did not handle origins correctly. For example,
this code:

```
module B; end
class A
  def b; 2 end
  prepend B
end
a = A.dup.new
class A
  def b; 1 end
end
p a.b
```

Printed 1 instead of 2. This is because the super chain for
a.singleton_class was:

```
a.singleton_class
A.dup
B(iclass)
B(iclass origin)
A(origin) # not A.dup(origin)
```

The B iclasses would not be modified, so the includer entry would be
still be set to A and not A.dup.

This modifies things so that if the class/module has an origin,
all iclasses between the class/module and the origin are duplicated
and have the correct includer entry set, and the correct origin

is created.

This requires other changes to make sure all tests still pass:

- rb_undef_methods_from doesn't automatically handle classes with origins, so pass it the origin for Comparable when undefing methods in Complex. This fixed a failure in the Complex tests.

- When adding a method, the method cache was not cleared correctly if klass has an origin. Clear the method cache for the klass before switching to the origin of klass. This fixed failures in the autoload tests related to overridding require, without breaking the optimization tests. Also clear the method cache for both the module and origin when removing a method.

- Module#include? is fixed to skip origin iclasses.

- Refinements are fixed to use the origin class of the module that has an origin.

- RCLASS_REFINED_BY_ANY is removed as it was only used in a single place and is no longer needed.

- Marshal#dump is fixed to skip iclass origins.

- rb_method_entry_make is fixed to handled overridden optimized methods for modules that have origins.

Fixes [Bug #16852]

History

#1 - 05/17/2020 09:34 AM - osyo (manga osyo)

hi.
Thanks for issues :)

I got the same error when include M after refine M.

```
module M; end
class X
  include M
end

module RefinementBug
  refine M do
    def refined_method
      puts "Called #refined_method"
    end
  end
end

class Y
  # Error if include M
  include M
end

using RefinementBug

puts "Using Ruby #{RUBY_VERSION}"

# error: undefined method `refined_method' for #<X:0x000055d4e22468c0> (NoMethodError)
X.new.refined_method
```

see: https://wandbox.org/permlink/bE3AvLBRnu07wtJY

By the way, RSpec caused an error due to require "stringio".
This is because stringio does include Enumerable.

```
module RefinementBug
  refine Enumerable do
    def refined_method
      puts "Called #refined_method"
    end
  end
```

```
end

# Error if `require "stringio"`
require "stringio"

using RefinementBug

puts "Using Ruby #{RUBY_VERSION}"

# error: undefined method `refined_method' for []:Array (NoMethodError)
[].refined_method
```

#2 - 05/19/2020 09:41 PM - jeremyevans0 (Jeremy Evans)

- Backport changed from 2.5: UNKNOWN, 2.6: UNKNOWN, 2.7: UNKNOWN to 2.5: DONTNEED, 2.6: DONTNEED, 2.7: REQUIRED

I can confirm this bug. It was introduced in a0579f3606561a74e323f6193b9504c06845236c, which fixed prepending a refined module after inclusion. I'll look into fixing this.

#3 - 05/24/2020 03:33 AM - jeremyevans0 (Jeremy Evans)

I've added https://github.com/ruby/ruby/pull/3140 to fix this. It works by ensuring an origin for Enumerable directly after it is created. It does the same for Comparable and Kernel, since I think the issue could affect them.

#4 - 06/03/2020 04:51 PM - jeremyevans (Jeremy Evans)

- Status changed from Open to Closed

Applied in changeset git|98286e9850936e27e8ae5e4f20858cc9c13d2dde.

Ensure origins for all included, prepended, and refined modules

This fixes various issues when a module is included in or prepended
to a module or class, and then refined, or refined and then included
or prepended to a module or class.

Implement by renaming ensure_origin to rb_ensure_origin, making it
non-static, and calling it when refining a module.

Fix Module#initialize_copy to handle origins correctly. Previously,
Module#initialize_copy did not handle origins correctly. For example,
this code:

```
module B; end
class A
  def b; 2 end
  prepend B
end
a = A.dup.new
class A
  def b; 1 end
end
p a.b
```

Printed 1 instead of 2. This is because the super chain for
a.singleton_class was:

```
a.singleton_class
A.dup
B(iclass)
B(iclass origin)
A(origin) # not A.dup(origin)
```

The B iclasses would not be modified, so the includer entry would be
still be set to A and not A.dup.

This modifies things so that if the class/module has an origin,
all iclasses between the class/module and the origin are duplicated
and have the correct includer entry set, and the correct origin
is created.

This requires other changes to make sure all tests still pass:

- rb_undef_methods_from doesn't automatically handle classes with origins, so pass it the origin for Comparable when undefing methods in Complex. This fixed a failure in the Complex tests.

- When adding a method, the method cache was not cleared correctly if klass has an origin. Clear the method cache for the klass before switching to the origin of klass. This fixed failures in the autoload tests related to overridding require, without breaking the optimization tests. Also clear the method cache for both the module and origin when removing a method.

- Module#include? is fixed to skip origin iclasses.

- Refinements are fixed to use the origin class of the module that has an origin.

- RCLASS_REFINED_BY_ANY is removed as it was only used in a single place and is no longer needed.

- Marshal#dump is fixed to skip iclass origins.

- rb_method_entry_make is fixed to handled overridden optimized methods for modules that have origins.

Fixes [Bug #16852]

#5 - 12/05/2021 11:17 PM - osyo (manga osyo)

Can backport this fix?

Files

minimal_refinement_failure.tar.gz	415 Bytes	05/12/2020	parker (Parker Finch)