Ruby master - Bug #7844

include/prepend satisfiable module dependencies are not satisfied

02/13/2013 10:55 PM - mame (Yusuke Endoh)

Status:	Assigned		
Priority:	Normal		
Assignee:	matz (Yukihiro Matsumoto)		
Target version:			
ruby -v:	ruby 2.0.0dev (2013-02-13 trunk 39225) [x86_64-linux]	**Backport:**	

Description

Hello,

```
module P
  def m; puts "P"; super; end
end
module Q
  def m; puts "Q"; super; end
  include P
end
module R
  def m; puts "R"; super; end
  prepend Q
end
module S
  def m; puts "S"; super; end
  include R
end
class A
  def m; puts "A"; super; end
  prepend P
  include S
end
A.new.m #=> P, R, A, S, Q
```

This code has five module dependencies, but only two are satisfied.

- Q includes P, which is not satisfied: P#m precedes Q#m.
- R prepends Q, which is not satisfied: R#m precedes Q#m.
- S includes R, which is not satisfied: R#m precedes S#m.
- A prepends P, which is satisfied: P#m precedes A#m.
- A includes S, which is satisfied: A#m precedes S#m.

Note that all the dependencies can be satisfied at all:

```
A.new.m #=> Q, P, A, S, R
```

--
Yusuke Endoh mame@tsg.ne.jp

Related issues:

Related to Ruby master - Bug #14704: Module#ancestors looks wrong when a modu...	**Open**

History

#1 - 02/13/2013 11:23 PM - matz (Yukihiro Matsumoto)

I believe the behavior is undefined (or implementation defined) when module dependency has contradiction.
And preferably error should be raised when contradiction detected.

Matz.

#2 - 02/13/2013 11:38 PM - mame (Yusuke Endoh)

matz (Yukihiro Matsumoto) wrote:

> I believe the behavior is undefined (or implementation defined) when module dependency has contradiction.
> And preferably error should be raised when contradiction detected.

Thank you, I agree with the policy.

However, is there any 'contradiction' in this case? No pair of these dependencies conflicts. There are no cycles.
Actually, they are satisfiable. A solution can be found by using the topological sorting.

--
Yusuke Endoh mame@tsg.ne.jp

#3 - 02/15/2013 01:25 AM - marcandre (Marc-Andre Lafortune)

As I stated before (#1586), I feel that the solution is easy:

```
A.ancestors = [*A.prepended_modules.flat_map(&:ancestors), A, *A.included_modules.flat_map(&:ancestors), *A.superclass.ancestors]
```

In the given example, this would be:

```
  P, A, S, Q, P, R, Object, Kernel, BasicObject
```

It makes absolutely no sense to me that R could come before A and believe it is clearly a major problem. R is never prepended, nor included in a prepended module!

Matz: how would you explain that R can be called before A in that example?

#4 - 05/31/2014 03:30 PM - nobu (Nobuyoshi Nakada)

- Description updated

#5 - 12/25/2017 06:15 PM - naruse (Yui NARUSE)

- Target version deleted (2.6)

#6 - 08/08/2019 05:09 PM - jeremyevans0 (Jeremy Evans)

- File include-after-origin-7844.patch added

This is still a bug in the master branch. I think the main problem in this example is that when calling Module#include, you can get a module inserted before the receiver in the lookup chain instead of behind the receiver.

mame correctly pointed out that using a topological sort you can satisfy all dependencies in the example. However, you can only do so by inserting a module before the receiver in the lookup chain, which I don't think should be allowed.

In the example, at the time A.include S is called:

```
A.ancestors # => [P, A, Object, Kernel, BasicObject]
S.ancestors # => [S, Q, P, R]
```

In my opinion, Module#include should have these properties:

(1) It should not add a module to the receiver's lookup chain if it already exists anywhere in the receiver's lookup chain
(2) It should not move the position of any module in the receiver's lookup chain, only add modules to the receiver's lookup chain
(3) It should not insert any modules before the receiver in the receiver's lookup chain
(4) It should insert all modules as a group, not some modules in the receiver's lookup chain at a different place than other modules

Applied to this example, assuming the above properties, after A.include S, we should have the following

```
A.ancestors # => [P, A, S, Q, R, Object, Kernel, BasicObject]
```

In this example, P in the included module lookup chain is ignored as it is already in the lookup chain for A. The remaining modules in the included module lookup chain are inserted as a group after A.

Attached is a patch that implements the above behavior, implementing property (3). The only change is that before inserting a module into the lookup chain, we check to make sure we are inserting it after the origin class. make check passes with this patch. I'm not sure this implements property (4). Still, I believe this is an improvement and should be committed.

#7 - 10/17/2019 10:55 PM - jeremyevans0 (Jeremy Evans)

- Related to Bug #14704: Module#ancestors looks wrong when a module is both included and prepended in the same class. added

#8 - 07/28/2020 08:19 PM - jeremyevans0 (Jeremy Evans)

matz (Yukihiro Matsumoto) considered this during the September 2019 and December 2019 developers meetings, but has not yet made a decision on it.

Files

include-after-origin-7844.patch	2.61 KB	08/08/2019	jeremyevans0 (Jeremy Evans)