descendants of a module don’t gain its future ancestors, but descendants of a class, do

```
p Class2.ancestors - Object.ancestors # [Class2, Class1]
Class1.include Mod1
```
```
p Class2.ancestors - Object.ancestors # [Class2, Class1, Mod1]
Mod1.include Mod2
```
```
p Mod1.ancestors - Object.ancestors # [Mod1, Mod2]
```
```
p Class2.ancestors - Object.ancestors # [Class2, Class1, Mod1]
```

note that descendants of a class do gain its future ancestors

so 2 issues:

1. It would seem natural that in dynamic language, dynamically added ancestors should propagate to descendants
2. Why is there a difference in ancestor propagation between modules and classes

Associated revisions
Revision 3556a834 - 02/27/2020 07:03 PM - jeremyevans (Jeremy Evans)
Make Module#include affect the iclasses of the module

When calling Module#include, if the receiver is a module, walk the subclasses list and include the argument module in each iclass.

This does not affect Module#prepend, as fixing that is significantly more involved.

Fixes [Bug #9573]

History
#1 - 02/27/2014 06:23 AM - nobu (Nobuyoshi Nakada)
- Description updated

First Last wrote:
so 2 issues:

1. It would seem natural that in dynamic language, dynamically added ancestors should propagate to descendants

It's a longstanding issue, a descendant knows its ancestors, but an ancestor doesn't know its descendants.

1. Why is there a difference in ancestor propagation between modules and classes

It is not between modules and classes, but caused by the order of inheritance and including.

#2 - 02/27/2014 06:43 AM - rits (First Last)
Nobuyoshi Nakada wrote:

First Last wrote:

so 2 issues:

1. It would seem natural that in dynamic language, dynamically added ancestors should propagate to descendants

It's a longstanding issue, a descendant knows its ancestors, but an ancestor doesn't know its descendants.

Is it the case that ancestors are cached in each descendant? So that it does not actually walk the ancestor tree each time. If so, is there any way to invalidate this cache for a given class or all, and have it reevaluate the ancestors?

1. Why is there a difference in ancestor propagation between modules and classes

It is not between modules and classes, but caused by the order of inheritance and including.

Please clarify. Mod1 is included in Class1 after Class2 extends Class1 and yet Class2 somehow learns of its new grandparent, Mod1. How does that happen if ancestors (Class1) do not know their descendants (Class2). So there is a difference, an ancestor added to a class, propagates to the descendant of this class, but an ancestor added to a module does not propagate to the descendant of this module.

#3 - 02/27/2014 08:30 AM - nobu (Nobuyoshi Nakada)
First Last wrote:

1. It would seem natural that in dynamic language, dynamically added ancestors should propagate to descendants

It's a longstanding issue, a descendant knows its ancestors, but an ancestor doesn't know its descendants.

Is it the case that ancestors are cached in each descendant? So that it does not actually walk the ancestor tree each time. If so, is there any way to invalidate this cache for a given class or all, and have it reevaluate the ancestors?

An included module is shared using an internal class (called as IClass), and IClasses are copied for each trees. Now subclasses/submodules are maintained in each classes/modules for method cache validation, so it may be possible.

1. Why is there a difference in ancestor propagation between modules and classes

It is not between modules and classes, but caused by the order of inheritance and including.

Please clarify. Mod1 is included in Class1 after Class2 extends Class1 and yet Class2 somehow learns of its new grandparent, Mod1. How does that happen if ancestors (Class1) do not know their descendants (Class2). So there is a difference, an ancestor added to a class, propagates to the descendant of this class, but an ancestor added to a module does not propagate to the descendant of this module.

Class1 only knows Mod1, and its ancestor tree is copied into Class2. And ditto for including a module.

#4 - 02/27/2014 10:12 AM - rits (First Last)
Nobuyoshi Nakada wrote:

First Last wrote:
Please clarify. Mod1 is included in Class1 after Class2 extends Class1 and yet Class2 somehow learns of its new grandparent, Mod1. How does that happen if ancestors (Class1) do not know their descendants (Class2). So there is a difference, an ancestor added to a class, propagates to the descendant of this class, but an ancestor added to a module does not propagate to the descendant of this module.

Class1 only knows Mod1, and its ancestor tree is copied into Class2.
And ditto for including a module.

I am inferring that the ancestor tree is copied into Class2 when it extends Class1, yes?
But at this time Mod1 has not yet been included, so how does Class2 learn of Mod1?

and why is the situation different with modules?

Do you see what I am pointing out, an ancestor added to a class, propagates to the past descendant of this class (Class2 extends Class1 before Mod1 is included in Class1), but an ancestor added to a module does not propagate to the past descendant of this module.

Can someone please explain this phenomenon.

First Last wrote:

Nobuyoshi Nakada wrote:

Class1 only knows Mod1, and its ancestor tree is copied into Class2.
And ditto for including a module.

I am inferring that the ancestor tree is copied into Class2 when it extends Class1, yes?
But at this time Mod1 has not yet been included, so how does Class2 learn of Mod1?

and why is the situation different with modules?

Do you see what I am pointing out, an ancestor added to a class, propagates to the past descendant of this class (Class2 extends Class1 before Mod1 is included in Class1), but an ancestor added to a module does not propagate to the past descendant of this module.

Can someone please explain this phenomenon.

First Last wrote:

Nobuyoshi Nakada wrote:

Class1 only knows Mod1, and its ancestor tree is copied into Class2.
And ditto for including a module.

I am inferring that the ancestor tree is copied into Class2 when it extends Class1, yes?
But at this time Mod1 has not yet been included, so how does Class2 learn of Mod1?

and why is the situation different with modules?

Do you see what I am pointing out, an ancestor added to a class, propagates to the past descendant of this class (Class2 extends Class1 before Mod1 is included in Class1), but an ancestor added to a module does not propagate to the past descendant of this module.

Can someone please explain this phenomenon.
I am inferring that the ancestor tree is copied into Class2 when it extends Class1, yes? But at this time Mod1 has not yet been included, so how does Class2 learn of Mod1?

and why is the situation different with modules?

Do you see what I am pointing out, an ancestor added to a class, propagates to the past descendant of this class (Class2 extends Class1 before Mod1 is included in Class1), but an ancestor added to a module does not propagate to the past descendant of this module.

Can someone please explain this phenomenon.

What is the objection to explaining how this works?

#8 - 03/13/2014 08:34 PM - jeremyevans0 (Jeremy Evans)

First Last wrote:

What is the objection to explaining how this works?

nobu explained how it works. However, as he is not a native English speaker, let me attempt to clarify.

In ruby, there exist pseudo-copies of modules called iclasses. These copies share the same variable(class variable/instance variable/constant) tables and the same method tables, but have a different super pointer in their C struct. iclasses are made when you attempt to include a module in another module or class.

When you do:

```ruby
module M0; end
module M1
  include M0
end
```

This creates a module M1 that includes an iclass of M0 (notated below as i0M0) in its inheritance list. For a module, the inheritance list is the the super pointer in struct RClass).

When you do:

```ruby
class A
  include M1
end
```

What happens is iclasses of M1 and i0M0 are made (notated below as i0M1, i1M0). So method lookup for an instance of A will be:

A -> i0M1 -> i1M0 -> Object

Here's how the super pointers for the struct RClass should look:

M0: null
i0M0: null
M1: i0M0
i1M0: Object
i0M1: i1M0
A: i0M1

When you do:

```ruby
module M2; end
M1.include M2
```

This creates an iclass of M2 (i0M2) and updates the super pointer in M1, as shown:

M1: i0M2
i0M2: i0M0
i0M0: null

However, it has no effect on any of the iclasses of M1 already created.

Ruby doesn't have multiple inheritance. Ruby method lookup uses a linked listed, not a tree. This is the reason for iclasses, and why including module B in module A after A has been included in class C does not include B in C.
Note that I am not an expert on ruby internals, so if there are errors in the above description, hopefully a more knowledgeable person can correct me.

Jeremy Evans wrote:

Ruby doesn't have multiple inheritance. Ruby method lookup uses a linked listed, not a tree. This is the reason for iclasses, and why including module B in module A after A has been included in class C does not include B in C.

Conceptually Ruby does have multiple inheritance, an object is_a?(all included modules).

Is MRI's iclass snapshotting an implementation detail? Can it theoretically be done differently (e.g. tree that you mentioned)

Jeremy Evans wrote:

Previously it wasn't possible because there wasn't a way to go from the module to all iclasses generated from it. I think module_subclasses in struct rb_subclass_entry may now contain the necessary pointers, but I'm not sure. Actually, looks like the subclasses entry may contain it (module_subclasses is used by iclasses, not modules). This appears to be a linked list of iclasses for the module.

Is there a clean way to fix this issue?

Jeremy Evans wrote:

Attached is a patch that implements support for this for Module#include, but not Module#prepend. It passes make check. I'm not sure if we want to support this for Module#include but not Module#prepend, as it would make them inconsistent.

Module#include support isn't too difficult to implement. Conceptually, the patch is similar to the pull request nobu was working on (https://github.com/ruby/ruby/pull/549). nobu's approach using rb_class_foreach_subclass is definitely simpler, but I'm not sure what the issues were with it that caused the pull request to be closed (I didn't see nobu's pull request until after working on my patch).

It is probably possible to support this for Module#prepend, but I believe it would at least require creating an origin module for all modules that are included/prepended to other modules/classes. I think that's a necessary condition, but not a sufficient one, as doing that by itself doesn't allow Module#prepend to work similarly.

Jeremy Evans wrote:

Attached is a work-in-progress patch that includes similar support for Module#prepend. It does work in terms of Module#prepend affecting classes/modules that have already included the receiver, but it causes failures in the tests that would need to be fixed. Example:

class Object
  def foo; [] end
end

module A
  def foo; [:A] + super end
end

module B
  def foo; [:B] + super end
end

module C
  def foo; [:C] + super end
end

module D
  def foo; [:D] + super end
end

module E
  include C
  prepend D
  def foo; [:E] + super end
end

module Enumerable
  def foo; [:Enumerable] + super end
end

Enumerable.include A
Enumerable.prepend B
Enumerable.include E
Output:

```
```

This does require creating origin iclasses for all modules that are included or prepended to other modules. It changes the origin pointer handling for iclasses such that the iclass origin pointer points not to the module origin but to the iclass origin. Due to the way module inclusion works, the iclass origin is not created at the point the iclass is created, so we need to keep a record of the created iclass, and when we come across the module origin and create the iclass origin, we set the origin for the iclass to the iclass origin.

Before doing more work in this area to attempt to fix the test failures, we should decide if we want this behavior and whether the necessary tradeoff of doubling the number of iclasses is worth it. Here are a few of the 43 test-all failures:

1) Failure: Complex_Test#test_respond [\home\jeremy\tmp\ruby\test\ruby\test_complex.rb:934]:
Complex#clamp.
Expected (1+1i) to not respond to clamp.

2) Failure: TestModule#test_override_optmethod_after-prepend [\home\jeremy\tmp\ruby\test\ruby\test_module.rb:1990]:
<1/2> expected but was <0>.
...

42) Error: TestAlias#testスーパーinaliased_module_method:
NoMethodError: super: no superclass method `foo' for #<TestAlias::SuperInAliasedModuleMethod::Derived:0x000006e8b02d3d78>
Did you mean? for /
\home\jeremy\tmp\ruby\test\ruby\test_alias.rb:96:in `foo'
\home\jeremy\tmp\ruby\test\ruby\test_alias.rb:115:in `testスーパーinaliased_module_method'

43) Error: TestRefinement#test_refine_module:
NoMethodError: super: no superclass method `bar' for #<TestRefinement::RefineModule::C:0x0000006a8afad0018>
Did you mean? baz
\home\jeremy\tmp\ruby\test\ruby\test_refinement.rb:453:in `bar'
\home\jeremy\tmp\ruby\test\ruby\test_refinement.rb:469:in `call_bar'
\home\jeremy\tmp\ruby\test\ruby\test_refinement.rb:479:in `test_refine_module'

#15 - 02/27/2020 08:23 AM - matz (Yukihiro Matsumoto)
The patch for include looks OK to me. To ensure there are no unseen compatibility issues, I'd like to experiment with it during the 2.8 development cycle. I am not yet sure how prepend should work here. I need to think about it more deeply. Keep it as it is for prepend for the moment.

Matz.

#16 - 02/27/2020 07:03 PM - jeremyevans (Jeremy Evans)
- Status changed from Open to Closed

Applied in changeset git|3556a834a2847e52162d1d3302d4c64390df1694.

Make Module#include affect the iclasses of the module

When calling Module#include, if the receiver is a module, walk the subclasses list and include the argument module in each iclass.

This does not affect Module#prepend, as fixing that is significantly more involved.

Fixes [Bug #9573]

#17 - 02/27/2020 07:08 PM - jeremyevans0 (Jeremy Evans)
- Status changed from Closed to Open

05/09/2020
jeremyevans0 (Jeremy Evans) thanks so much for fixing this at least in the #include case.

Files

<table>
<thead>
<tr>
<th>File Name</th>
<th>Size</th>
<th>Date</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>include-future-ancestors-9573.patch</td>
<td>3.49 KB</td>
<td>01/07/2020</td>
<td>jeremyevans0 (Jeremy Evans)</td>
</tr>
<tr>
<td>prepend-future-ancestors-9573.patch</td>
<td>6.42 KB</td>
<td>01/08/2020</td>
<td>jeremyevans0 (Jeremy Evans)</td>
</tr>
</tbody>
</table>