Ruby master - Bug #17048

Calling initialize_copy on live modules leads to crashes

07/24/2020 04:09 PM - alanwu (Alan Wu)

Status:	Open		
Priority:	Normal		
Assignee:			
Target version:			
ruby -v:	ruby 2.8.0dev (2020-07-23T14:44:25Z master 098e8c2873) [x86_64-linux]	**Backport:**	2.5: UNKNOWN, 2.6: UNKNOWN, 2.7: UNKNOWN

Description

Here's a repro script

```
loop do
  m = Module.new do
    prepend Module.new
    def hello
    end
  end

  klass = Class.new { include m }
  m.send(:initialize_copy, Module.new)
  GC.start

  klass.new.hello rescue nil
end
```

Here's a script that shows that it has broken semantics even
when it happens to not crash.

```
module A
end

class B
  include A
end

module C
  Const = :C
end

module D
  Const = :D
end

A.send(:initialize_copy, C)
p B::Const # :C, makes sense
A.send(:initialize_copy, D)
p B::Const # :D, makes sense
A.send(:initialize_copy, Module.new)
p (begin B::Const rescue NameError; 'NameError' end) # NameError, makes sense
A.send(:initialize_copy, C)
p B::Const # still NameErorr. Weird
```

This example shows that the problem exists as far back as 2.0.0.

I think the easiest way to fix this is to forbid calling :initialize_copy
on modules that have children. Another option is to try to decide on
the semantics of this. Though I don't think it's worth the effort as this
has been broken for a long time and people don't seem to to be using it.
Thoughts?

History

#1 - 07/24/2020 04:19 PM - alanwu (Alan Wu)

- Description updated

#2 - 07/24/2020 04:24 PM - alanwu (Alan Wu)

- Description updated

#3 - 07/24/2020 04:40 PM - jeremyevans0 (Jeremy Evans)

Removing Module#initialize_copy would probably require changes to Module#{dup,clone}. Maybe we can set a flag in Module#initialize_copy such that calling the method raises if called again on the same module (not sure if that is what you meant)?

#4 - 07/24/2020 05:23 PM - alanwu (Alan Wu)

I'm not proposing that we remove initialize_copy, but to make it raise when the receiver has children. So this is what it would look like:

```
module A
end

A.send(:initialize_copy, Module.new) # fine, no one inherits from A

class B
  include A
  A.send(:initialize_copy, Module.new) # raise, since A now has one child
end
```

Admittedly this proposed behavior is designed to sidestep the problem that the current implementation has.
Though maybe it's fine since this is a seldom used private method and other Ruby implementations don't have to follow this behavior?

Now that you've mentioned it, removing the method and moving the logic into Module#{dup,clone} feels like the cleanest solution. It's more likely to be a breaking change though.

#5 - 07/24/2020 05:52 PM - jeremyevans0 (Jeremy Evans)

I would recommend to have initialize_copy always raise instead of only raising if the module has children. It's more consistent, and there is no reason anyone should be calling initialize_copy more than once. This shouldn't be considered a breaking change, as initialize_copy is a private method. I think it is better than moving the logic to dup/clone, just in case a Module subclass is overridding initialize_copy and calling super.

#6 - 07/24/2020 10:29 PM - alanwu (Alan Wu)

In principle I agree that allowing initialization only once is a good way to go, but the way Module.allocate is currently setup makes implementing this a bit complicated. At the moment there is not really a pre-init state for modules and the result from Module.allocate is the same as Module.new. If we want to do this we would have to implement a pre-init state for modules, and make sure that all the operations on modules (adding methods, constants, etc) are aware of this state so they can do the initialization in case they receive a pre-init module.

It's doable, but I don't know if the extra consistency is worth the added complexity. It would slow things down a small bit for operations like method addition and has the potential to introduce crashes if we miss places where we need the initialization check.

#7 - 07/25/2020 02:35 AM - nobu (Nobuyoshi Nakada)

I agree with alanwu (Alan Wu), that it won't be worth.

```
diff --git c/class.c i/class.c
index 6835d2d7289..f7a56601634 100644
--- c/class.c
+++ i/class.c
@@ -354,6 +354,13 @@ static void ensure_origin(VALUE klass);
 VALUE
 rb_mod_init_copy(VALUE clone, VALUE orig)
 {
+    if (RCLASS_EXT(clone)->subclasses ||
+        RCLASS_EXT(clone)->parent_subclasses ||
+        RCLASS_EXT(clone)->module_subclasses) {
+        rb_raise(rb_eTypeError, "cannot replace %s in use",
+                 (RB_TYPE_P(clone, T_MODULE) ? "module" : "class"));
+    }
+
     /* cloned flag is refer at constant inline cache
      * see vm_get_const_key_cref() in vm_insnhelper.c
      */
diff --git c/test/ruby/test_module.rb i/test/ruby/test_module.rb
index d2da384cbd1..8d986f13413 100644
```

```
--- c/test/ruby/test_module.rb
+++ i/test/ruby/test_module.rb
@@ -435,6 +435,12 @@
     assert_empty(m.constants, bug9813)
   end

+  def test_initialize_copy_in_use
+    m = Module.new
+    Class.new {include m}
+    assert_raise(TypeError) {m.send(:initialize_copy, Module.new)}
+  end
+
   def test_dup
     OtherSetup.call
```

#8 - 07/25/2020 10:32 AM - Eregon (Benoit Daloze)

Should we rather design a good way to allow copying but yet not have to deal with uninitialized state?

Right now the only well-defined protocols for copying are

- dup = allocate, copy @ivars, initialize_dup (which calls initialize_copy)
- clone = allocate, copy @ivars, initialize_clone (which calls initialize_copy), clone also copies extra state like frozen and the singleton class

This means some classes have to support an "unintialized state" when otherwise they would not need to.
And in some cases it even means instances have to be mutable when they would otherwise not need to (e.g., MatchData, #16294).

So maybe we should make Module.allocate and #initialize_copy always raise, and override dup and clone for Module?
It's still unfortunate that this would mean duplicating the logic for dup/clone there.
So I think a better copying protocol is warranted here as it's not just an issue for Module.

Re nobu (Nobuyoshi Nakada)'s patch I don't like this ad-hoc condition which leaks implementation details into semantics.
How about having an initialized flag that's set by #initialize and #initialize_copy and checked in both of these methods if we want a quick fix?

#9 - 07/25/2020 06:33 PM - alanwu (Alan Wu)

> How about having an initialized flag that's set by #initialize and #initialize_copy and checked in both of these methods if we want a quick fix?

That doesn't work because you can trigger the bug without ever calling initialize on the module:

```
m = Module.allocate
m.prepend(Module.allocate)
m.define_method(:hello) {}
klass = Class.new { include m }
m.send(:initialize_copy, Module.allocate)
GC.start
klass.new.hello rescue nil
# you may need to run this multiple times to get it to crash
```

If we want something like that we would have to implement an uninitialized state.

#10 - 07/26/2020 06:21 AM - nobu (Nobuyoshi Nakada)

Tried it.
https://github.com/nobu/ruby/tree/uninitialized-module

#11 - 07/26/2020 04:23 PM - Eregon (Benoit Daloze)

nobu (Nobuyoshi Nakada) wrote in #note-10:

> Tried it.
> https://github.com/nobu/ruby/tree/uninitialized-module

Right, removing Module.allocate is one way since dup/clone uses the alloc function directly (and does not call allocate).

I think calling Module.allocate in user code makes no sense, so that approach seems a good way to me.

#12 - 07/28/2020 11:45 PM - alanwu (Alan Wu)

Thank you for the code, nobu (Nobuyoshi Nakada) . I think with your branch we could even keep .allocate, though people wouldn't be able to do much with it.

As long as no one is able to call initialize_copy after children (iclasses) exist, it's fine.
I think I was wrong about the number of places we would have to plug to implement an uninitialized state that resolves the issue.
Only the places that make new iclasses need to check for the uninitilaized state, so jsut prepend, include and maybe refinements.

Side note about the branch (57c7f9b), it's possible to get access to an uninitialized module in Ruby land by subclassing from Module:

```
class Sub < Module
  def initialize_copy(other)
    p ancestors
  end
end

Sub.new.dup # [#<Sub:0x00007fa4ec015b10>, BasicObject]
```

It doesn't cause anything bad to happen AFAICT. I just found it interesting that the branch adds a normally impossible-to-construct module.
Maybe it's a positive because it makes Ruby more weird :D

#13 - 08/12/2020 07:30 AM - ko1 (Koichi Sasada)

sorry I didn't check all threads, but nobu's patch can close it?

#14 - 08/26/2020 08:27 PM - alanwu (Alan Wu)

Yes, nobu's patch fixes the crash. It is technically a breaking change though, so maybe it needs approval from Matz?
Side note, the bug still exists on master as of today.