## Feature Proposal: Extend 'protected' to support module friendship

11/19/2016 01:15 PM - matthewd (Matthew Draper)

<table>
<thead>
<tr>
<th>Status:</th>
<th>Rejected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority:</td>
<td>Normal</td>
</tr>
<tr>
<td>Assignee:</td>
<td></td>
</tr>
<tr>
<td>Target version:</td>
<td></td>
</tr>
</tbody>
</table>

### Description

When working on a larger library, with many classes that have both official API methods and internal supporting methods, it can be hard to distinguish between them.

In Rails, for example, we currently do this using `:nodoc:` -- if a method is hidden from the documentation, it is not part of the officially supported API, even if it has public visibility.

This approach can be confusing for users, however, because they can find methods that seem to do what they want, and start calling them, without ever looking at the documentation: either by just guessing a likely method name, or even being guided to it by `did_you_mean`.

Method visibility controls seem like the right solution to this problem: if we make the methods private or protected, users can still choose to call them, but only by first acknowledging that they're using internal API. However, as we have object oriented internals, a lot of our internal API calls are between instances of unrelated classes... and using `send` on all those calls would make our own code very untidy.

I propose that the solution to this problem is to make protected more widely useful, by allowing a module to nominate other modules that are allowed to call its protected methods.

```ruby
class A
  protected def foo
    "secrets"
  end
end

class D
  def call_foo
    A.new.foo
  end
  A.friend D
end

D.new.call_foo # => "secrets"
```

This change is backwards compatible for existing uses of protected: a module is always considered its own friend (so calls that previously worked will continue to do so), and classes have no other friends by default (so calls that were previously disallowed will also continue to do so).

Using a module, a library can easily establish a 'friendship group' of related classes without needing to link them individually, as well as providing a single opt-in for user code that consciously chooses to use unsupported APIs.

```ruby
module MyLib
  module Internals
    end
  end

  class A
    include Internals
    friend Internals

    protected def foo
      "implementation"
    end
  end

class B
```
This change seems in keeping with the ruby philosophy that a method’s visibility is more of a guideline than a strictly enforced rule -- here, we allow the callee to blur the line, instead of leaving it up to the caller to use send.

The implementation is surprisingly simple, and only adds time (searching an array of friends, instead of only looking for the current class) after a method call has already resolved to a protected method.

While I’m personally most interested in how this could be applied in a Rails-sized project (such as.. Rails), I believe it would provide a helpful clarifying tool to any library that has multiple collaborating classes, whose instances are also exposed to user code.

History

#1 - 11/19/2016 01:18 PM - matthewd (Matthew Draper)

Draft implementation:

```c
+VALUE rb_obj_is_friend_of(VALUE obj, VALUE c)
+{
+    VALUE defined_class;
+    c = class_or_module_required(c);
+    if (defined_class) {
+        defined_class = class_search_ancestor(defined_class);
+        rb_obj_is_friend_of_DEFINED = defined_class;
+        rb_obj_is_friend_of(VALUE obj, VALUE c);
+    }
+}
```
defined_class = RB_TYPE_P(c, T_ICLASS) ? RBASIC(c)->klass : rb_class_real(c);
if (rb_obj_is_kind_of(obj, defined_class)) return Qtrue;
while (c) {
    VALUE mod = RB_TYPE_P(c, T_ICLASS) ? RBASIC(c)->klass : c;
    VALUE ary = rb_ivar_get(mod, rb_intern_const("friends"));
    if (RB_TYPE_P(ary, T_ARRAY)) {
        int i;
        long len = RARRAY_LEN(ary);
        for (i=0; i<len; i++) {
            VALUE friend = RARRAY_AREF(ary, i);
            if (rb_obj_is_kind_of(obj, friend)) return Qtrue;
        }
    }
    c = RCLASS_SUPER(c);
}
return Qfalse;

/*
 * call-seq:
 * obj.tap{|x|...} -> obj
@@ -1529,6 +1564,61 @@
    return rb_str_dup(rb_class_name(klass));
 }
 */
+* call-seq:
+* mod.friend(module, ...) -> mod
+* +
+* Makes the given modules friends of <i>mod</i>. Instances of friends
+* + are permitted to call protected methods on instances of <i>mod</i>.
+* */
+static VALUE
+rb_mod_friend(int argc, const VALUE *argv, VALUE mod)
+{
+    VALUE ary;
+    int i;
+    ID id = rb_intern_const("friends");
+    ary = rb_ivar_get(mod, id);
+    if (!RB_TYPE_P(ary, T_ARRAY)) ary = rb_ary_new();
+    for (i = 0; i < argc; i++) {
+        rb_ary_push(ary, class_or_module_required(argv[i]));
+    }
+    rb_ivar_set(mod, id, ary);
+    return Qnil;
+}
+/*
+ * call-seq:
+ * mod.friends -> array
+ * +
+ * Returns the list of modules and classes whose instances are
+ * + permitted to call protected methods on <i>mod</i>. The returned list
+ * + does not include <i>mod</i>, which is always considered a friend of
+ * + itself.
+ */
+static VALUE
+rb_mod_friends(VALUE c)
+{
+    VALUE result = rb_ary_new();
+    c = class_or_module_required(c);
+    while (c) {
+        VALUE mod = RB_TYPE_P(c, T_ICLASS) ? RBASIC(c)->klass : c;
+    }
+```
```c
VALUE ary = rb_ivar_get(mod, rb_intern_const("friends"));
if (RB_TYPE_P(ary, T_ARRAY)) {
    rb_ary_concat(result, ary);
}
c = RCLASS_SUPER(c);
return result;
}

void rb_mod_friends(rb_module_t* module, rb_ary_t* ary) {
    VALUE mod = module;
    VALUE ary = rb_ivar_get(mod, rb_intern_const("friends"));
    if (RB_TYPE_P(ary, T_ARRAY)) {
        rb_ary_concat(result, ary);
    }
    c = RCLASS_SUPER(c);
    result = rb_ary_steal(result);
    return result;
}
```

```c
/*
*   call-seq:
*       mod.freeze       -> mod
@@ -3512,6 +3602,8 @@ InitVM_Object(void)
         rb_define_global_const("NIL", Qnil);
         rb_deprecate_constant(rb_cObject, "NIL");
     + rb_define_method(rb_cModule, "friend", rb_mod_friend, -1);
     + rb_define_method(rb_cModule, "friends", rb_mod_friends, 0);
     rb_define_method(rb_cModule, "freeze", rb_mod_freeze, 1);
     rb_define_method(rb_cModule, "===", rb_mod_eqq, 1);
     rb_define_method(rb_cModule, "==", rb_obj_equal, 1);
```

```c
diff --git a/vm_eval.c b/vm_eval.c
index ea398e0..48ee2df 100644
--- a/vm_eval.c
+++ b/vm_eval.c
@@ -593,13 +593,7 @@
     case METHOD_VISI_PROTECTED:
         if (visi == METHOD_VISI_PROTECTED && scope == CALL_PUBLIC) {
             VALUE defined_class = klass;
-            if (RB_TYPE_P(defined_class, T_ICLASS)) {
-                defined_class = RBASIC(defined_class)->klass;
-            }
-            if (self == Qundef || !rb_obj_is_kind_of(self, defined_class)) {
-                return MISSING_PROTECTED;
-            }
+            if (self == Qundef || !rb_obj_is_friend_of(self, klass)) {
+                return MISSING_PROTECTED;
+            }
         }
```

```c
diff --git a/vm_insnhelper.c b/vm_insnhelper.c
index 43db728..cdbcfd1 100644
--- a/vm_insnhelper.c
+++ b/vm_insnhelper.c
@@ -2274,7 +2274,7 @@
                     }
     case METHOD_VISI_PROTECTED:
         if (self == Qundef || !rb_obj_is_kind_of(self, klass)) {
-            return MISSING_PROTECTED;
+            return MISSING_PROTECTED;
         } else {
             return VM_CALL_METHOD_MISSING;
         }
```

```c
#2 - 11/19/2016 05:34 PM - shevegen (Robert A. Heiler)
The terminology is a bit peculiar - friendly modules? Do we have unfriendly modules as well? Is that a new terminology altogether? I never read friend-methods before.

However had, leaving aside the choice of names, I do not like syntax constructs such as:
"protected def foo"
Looks fairly Java-ish.

This is also probably only a minor issue because e.g. "private" keyword identifier allows all subsequent methods be defined as private, so I suppose the same would apply for protected.

I agree with you in one regards - using .send() as workarounds. Not on your comment that it would make it "untidy", I love .send(), but I agree that it is a bit strange that there is also .public_send() altogether and also why sometimes method calls work via .send() but not otherwise. I am just not fully sure that the above proposal adds a lot; then again your mileage may vary and thankfully I don't have to make any decisions that impact anyone else really other than in my gems.

The code does look sorta alien to me, perhaps it is less alien for rails people.

#3 - 11/19/2016 11:09 PM - matthewd (Matthew Draper)

Robert A. Heiler wrote:

The terminology is a bit peculiar - friendly modules? Do we have unfriendly modules as well?
Is that a new terminology altogether? I never read friend-methods before.

It has slightly different semantics (more in line, IMO, with ruby's existing definitions of private & protected), but I didn't invent the term: https://en.wikipedia.org/wiki/Friend_class

However had, leaving aside the choice of names, I do not like syntax constructs such as:

"protected def foo"

That's existing ruby syntax. I used it here mostly because it's a line shorter.

To be clear, the change introduces no new syntax, and two new methods: Module#friend(*modules), and Module#friends. Beyond that, it's about making some method calls succeed where they would previously have raised a NoMethodError due to visibility.

#4 - 01/20/2017 03:29 AM - shyouhei (Shyouhei Urabe)

We looked at this issue yesterday at developers meeting.

While I understand the needs to distinguish official API and internal ones, (ab)using protected for that purpose was not recommended by the attendees. If you could isolate the "friendship"-ness into a single file, maybe refinements can solve the issue.

#5 - 01/20/2017 09:54 AM - dsferreira (Daniel Ferreira)

My proposal of Internal interfaces comes in line with this proposal.
I would love to have this functionality in place.
Would make code integrity so much better.

Maybe the use of the new internal access modifier would make things more clear?

#6 - 01/20/2017 10:10 AM - dsferreira (Daniel Ferreira)

The use of :nodoc: for these situations it is not a valid option in my opinion.
The reason being because Object#public_methods will not be inline with the documentation.
From that fact the developer cannot anymore trust the code.

Code should be 100% reliable in the feedback it gives to the developer.

#7 - 02/14/2022 02:23 PM - Eregon (Benoit Daloze)

Do you have an example in Rails?

This sounds like it can be solved by moving such private-but-shared method to Internals or some other module.

Calling "private/protected" methods on another object feels rather wrong to me.

Maybe private_constant can also be leveraged to e.g. have a private class/module but all methods on it public.

#8 - 02/14/2022 03:02 PM - matthewd (Matthew Draper)

Calling "private/protected" methods on another object feels rather wrong to me.
The whole point of protected is that it allows you to call methods on another object. But I assume you meant another object that is not an instance of the caller's class.

A quick grep for :nodoc: turned up this example:
https://github.com/ruby/ruby/blob/26187a8520b806645206a2064c11a7ab86a89845/lib/net/http/response.rb#L163

In my experience it's just not unusual for two collaborating objects to need to talk to each other in more detail than their "user"-facing public API intends to expose. In a typed OO language, this is where the objects communicate using a private concrete class, while only "publishing" a more focused interface.

As in that Net::HTTP example, a separate module to stash the methods is not really viable, because we're talking about instance methods that need access to ivars -- that's why it's a method on the current target object, and not a private method on the caller.

#9 - 02/14/2022 03:30 PM - Eregon (Benoit Daloze)
For this Net::HTTP example IMHO it'd be better to use send, that would make it a private method and be explicit that code is calling into partly- internals.

IMHO friend-like visibility is too magic and the linear search could be a significant overhead on such calls (+ rb_obj_is_kind_of, so could be O(n*m)). Also the additional state per module/class seems to add non-trivial complexity.

send can be easily optimized OTOH (especially with a literal Symbol as name).

#10 - 02/15/2022 01:25 PM - matthewd (Matthew Draper)
The trouble with using send in this [general] situation is that it makes it too equally-easy to reach into all internals: you want access to methods that are suitable for a collaborator but not a downstream user, but you can also immediately access truly-private methods that cause the object to violate its intended invariants. In a large project, you still want some guard-rails for your collaborators, even as you give them more abilities than usual.

Widespread use of send is also just unpleasant to read and maintain. In practice, people don't like writing code like that: I can see additional uses of :nodoc: for this purpose in stdlib (the chosen example was just particularly illustrative because of its comment), yet send(: turns up remarkably little. Given the currently-available options, it seems that more than just Rails chooses "secretly public" methods over send+private.

On performance, protected already performs a linear rb_obj_is_kind_of search. As you note, the total search becomes O(n*m) here... but the m is the number of friends (so only becomes > 0 when friendship is in use), and is only even considered when the target method is protected (or, in an alternate variant of this proposal, some specific fourth method visibility level).

I would expect this check to be easier to optimize than send, for monomorphic call-sites, by leaning on existing call-site caching. (If a friendship change bumps the class serial, there's no need to recheck the cached call.)

Even setting aside the syntactic noise of repeated send(:sym), I'm not aware of any current optimization effort that could make it competitive.

#11 - 02/15/2022 02:53 PM - p8 (Petrik de Heus)
I can imagine having a protected_send just like we have a public_send and send (which is basically private_send). Although that probably is a lot more difficult to optimize.

#12 - 02/15/2022 03:14 PM - Eregon (Benoit Daloze)
matthewd (Matthew Draper) wrote in #note-10:

The trouble with using send in this [general] situation is that it makes it too equally-easy to reach into all internals: you want access to methods that are suitable for a collaborator but not a downstream user, but you can also immediately access truly-private methods that cause the object to violate its intended invariants. In a large project, you still want some guard-rails for your collaborators, even as you give them more abilities than usual.

I see. Such semi-private methods could have a comment explaining they are also used by other parts of the gem. Or even an alias of private to indicate that but be clearer on the intention.

Widespread use of send is also just unpleasant to read and maintain. In practice, people don't like writing code like that: I can see additional uses of :nodoc: for this purpose in stdlib (the chosen example was just particularly illustrative because of its comment), yet send(: turns up remarkably little. Given the currently-available options, it seems that more than just Rails chooses "secretly public" methods over send+private.

It might also be written as __send__(::).

There are not many occurrences in stdlib, but stdlibs are also fairly small compared to larger gems, and probably don't need this much.

I would expect this check to be easier to optimize than send, for monomorphic call-sites, by leaning on existing call-site caching. (If a friendship change bumps the class serial, there's no need to recheck the cached call.)

Right, at least this can be cached for a call site since it's add-only.
It would likely be very difficult to cache when trying to persist JITed code though, as it depends on lots of live values.

I'm not aware of any current optimization effort that could make it competitive.
There is [#17291](#17291), but it would need a redefinition check to be semantically correct. TruffleRuby already optimizes `send` and `__send__`, in JITed code there is no difference and only a small overhead in interpreter (when using a constant method name).

**#13 - 02/15/2022 03:19 PM - Eregon (Benoit Daloze)**

p8 (Petrik de Heus) wrote in [#note-11](#note-11):

I can imagine having a `protected_send` just like we have a `public_send` and `send` (which is basically `private_send`).

Yeah, I was also thinking to something like:

```ruby
friend/internal/protected def my_method
...
end

def foo
  obj.send_friend/send_internal/send_protected(:my_method)
end
```

and such method could only be called by `send_friend/send_internal/send_protected`, and that would be very efficient and simple to check.

I don't think it's hard to optimize `send` and similar, e.g. there could be an extra call cache when parsing `send/__send__/etc`, and that could be used to cache the second lookup too.

**#14 - 02/17/2022 12:11 PM - matz (Yukihiro Matsumoto)**

- Status changed from Open to Rejected

I revisited this issue at the developer meeting and concluded to reject this proposal. I understand OP's need for the friend visibility but considering the dynamic nature of the language (plus current usage of protected visibility), this visibility provides the value far less than the pain of complexity. I'd rather like to remove protected visibility, if possible (I know it's too late).

Thus this is rejected. Sorry.

Matz.