Ruby master - Feature #12962

Feature Proposal: Extend 'protected' to support module friendship

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

Status:	Open
Priority:	Normal
Assignee:	
Target version:	

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.

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

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

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.

```
module MyLib
  module Internals
  end

  class A
    include Internals
    friend Internals

    protected def foo
      "implementation"
    end
  end

  class B
```

```
    include Internals
    friend Internals

    protected def bar
      A.new.foo
    end
  end
end

class UserCode
  def call_things
    [MyLib::A.new.foo, MyLib::B.new.bar]
  end
end

class FriendlyUserCode
  include MyLib::Internals

  def call_things
    [MyLib::A.new.foo, MyLib::B.new.bar]
  end
end

UserCode.new.call_things # !> NoMethodError: protected method `foo'..
FriendlyUserCode.new.call_things # => ["implementation", "implementation"]
```

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:

```
diff --git a/include/ruby/intern.h b/include/ruby/intern.h
index 8776a59..fa5b1dc 100644
--- a/include/ruby/intern.h
+++ b/include/ruby/intern.h
@@ -588,6 +588,7 @@ VALUE rb_any_to_s(VALUE);
 VALUE rb_inspect(VALUE);
 VALUE rb_obj_is_instance_of(VALUE, VALUE);
 VALUE rb_obj_is_kind_of(VALUE, VALUE);
+VALUE rb_obj_is_friend_of(VALUE, VALUE);
 VALUE rb_obj_alloc(VALUE);
 VALUE rb_obj_clone(VALUE);
 VALUE rb_obj_dup(VALUE);
diff --git a/object.c b/object.c
index 05bef4d..257db8a 100644
--- a/object.c
+++ b/object.c
@@ -698,6 +698,41 @@ rb_class_search_ancestor(VALUE cl, VALUE c)
     return class_search_ancestor(cl, RCLASS_ORIGIN(c));
 }

+/*
+ *
+ */
+
+VALUE
+rb_obj_is_friend_of(VALUE obj, VALUE c)
+{
+    VALUE defined_class;
+
+    c = class_or_module_required(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 @@ rb_mod_to_s(VALUE klass)
     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;
```

```
+        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;
+}
+
 /*
  *  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, 0);
     rb_define_method(rb_cModule, "===", rb_mod_eqq, 1);
     rb_define_method(rb_cModule, "==", rb_obj_equal, 1);
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 @@ rb_method_call_status(rb_thread_t *th, const rb_callable_method_entry_t *me, cal

             /* self must be kind of a specified form for protected method */
             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)) {
+                if (self == Qundef || !rb_obj_is_friend_of(self, klass)) {
                     return MISSING_PROTECTED;
                 }
             }
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 @@ vm_call_method(rb_thread_t *th, rb_control_frame_t *cfp, struct rb_calling_info

           case METHOD_VISI_PROTECTED:
             if (!(ci->flag & VM_CALL_OPT_SEND)) {
-                if (!rb_obj_is_kind_of(cfp->self, cc->me->defined_class)) {
+                if (!rb_obj_is_friend_of(cfp->self, cc->me->defined_class)) {
                     cc->aux.method_missing_reason = MISSING_PROTECTED;
                     return vm_call_method_missing(th, cfp, calling, ci, cc);
                 }
@@ -2776,7 +2776,7 @@ vm_defined(rb_thread_t *th, rb_control_frame_t *reg_cfp, rb_num_t op_type, VALUE
               case METHOD_VISI_PRIVATE:
                 break;
               case METHOD_VISI_PROTECTED:
-                if (!rb_obj_is_kind_of(GET_SELF(), rb_class_real(klass))) {
+                if (!rb_obj_is_friend_of(GET_SELF(), klass)) {
                     break;
                 }
               case METHOD_VISI_PUBLIC:
```

#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 intefaces 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.