

Jay Fields' Thoughts

experiences in software development

Monday, April 07, 2008

Alternatives for redefining methods

Ruby's open classes allow you define and redefine behavior pretty much at will; unfortunately, almost every option comes with caveats.

The example below is a gateway class that defines a process method. For the purposes of the example, assume that we need to redefine the process method on Gateway itself and call the original process method.* Also, assume that Gateway is not our class, so we cannot easily alter the original definition of process.

```
class Gateway
  def process(document)
    p "gateway processed document: #{document}"
  end
end

Gateway.new.process("hello world")
# >> "gateway processed document: hello world"
```

Solution 1: alias

The following example uses alias to redefine the process method.

```
class Gateway
  alias old_process process
  def process(document)
    p "do something else"
    old_process(document)
  end
end

Gateway.new.process("hello world")
# >> "do something else"
# >> "gateway processed document: hello world"
```

The example above creates an alias (old_process) for the process method. With an alias in place you can redefine the process method to anything you want and call the old_process method using the alias. This is probably the easiest solution and the most commonly used solution.

Unfortunately, it's not without problem. First of all, if someone redefines `old_process` you will get unexpected behavior. Second of all, the `old_process` method is really nothing more than an artifact of the fact that you have no other way to refer to the original method definition. Lastly, if the code is loaded twice, an infinite loop is created that causes the always painful to see 'stack level too deep' error.

Solution 2: `alias_method_chain`

Like I said, solution 1 is the most popular way to redefine a method. In fact, it's so popular Rails defines the `alias_method_chain` method to encapsulate the pattern. From the Rails source above `alias_method_chain`:

Encapsulates the common pattern of:

```
#
# alias_method :foo_without_feature, :foo
# alias_method :foo, :foo_with_feature
#
# With this, you simply do:
#
# alias_method_chain :foo, :feature
#
# And both aliases are set up for you.
#
# Query and bang methods (foo?, foo!) keep the same punctuation:
#
# alias_method_chain :foo?, :feature
#
# is equivalent to
#
# alias_method :foo_without_feature?, :foo?
# alias_method :foo?, :foo_with_feature?
#
# so you can safely chain foo, foo?, and foo! with the same feature.
```

Using `alias_method_chain` we can define our Gateway as the example below.

```
class Gateway
  def process_with_logging(document)
    p "do something else"
    process_without_logging(document)
  end
  alias_method_chain :process, :logging
end

Gateway.new.process("hello world")
# >> "do something else"
# >> "gateway processed document: hello world"
```

Using `alias_method_chain` is nice because it's something familiar to many Rails developers. Unfortunately, it also suffers from the same problems as using `alias` on your own.

Solution 3: Close on an unbound method

The following code uses the class method `"instance_method"` to assign the `"process"` method (as an unbound method) to a local variable. The `"process_method"` local variable is in scope of the closure used to define the new `process` method, so it can be used within the `process` definition. Calling an unbound method is as simple as binding it to any instance of the class that it was unbound from and then using the `call` method.

```
class Gateway
  process_method = instance_method(:process)
  define_method :process do |document|
    p "do something else"
    process_method.bind(self).call(document)
  end
end

Gateway.new.process("hello world")
# >> "do something else"
# >> "gateway processed document: hello world"
```

I've always preferred this solution because it doesn't rely on artifact methods that may or may not collide with other method definitions. Also, if the code is loaded multiple times the behavior is altered multiple times, but I find that easier to diagnose than when my only clue is "stack level too deep".

Unfortunately, this solution is not without flaws. Firstly, it relies on the fact that `define_method` uses a closure and has access to the unbound method. Of course this also implies that you have a handle on anything else defined in the same context. As with any closure, it's possible to accidentally create a memory leak. Also, (in MRI) I'm told that `define_method` takes 3 times as long to execute when compared to `def`.

Solution 4: Extend a module that redefines the method and uses `super`

This solution relies on creating a module with the new behavior and extending an instance with the module. Since the module is extended from the instance it will be checked first for the method definition when `"process"` is called (because it's the first ancestor). Since the module is the first ancestor it can use `super` to execute the `process` method defined in `Gateway` (the second ancestor).

```
module ProcessLogging
  def process(document)
    p "do something else"
    super
  end
end

Gateway.new.extend(ProcessLogging).instance_eval("class << self; self;
```

```
end").ancestors
# => [ProcessLogging, Gateway, Object, Kernel]
Gateway.new.extend(ProcessLogging).process("hello world")
# >> "do something else"
# >> "gateway processed document: hello world"
```


This solution is my favorite because I can use `def` and `super` and never worry about creating any memory leaks or artifact methods.

Of course, it assumes that you get the opportunity to extend instances of the class. However, I haven't found that to be a problematic requirement.

* There are generally other options such as delegation, defining hooks, etc. Often I find these to be cleaner solutions and try that route first. But, sometimes redefining a method cannot be avoided.

Labels: [alias](#), [def](#), [define_method](#), [extend](#)

Share: [Email](#) | [Del.icio.us](#) | [Digg](#) | [Reddit](#) | [Shadow](#) | [Rojo](#)

posted by Jay Fields : 1:41 PM 
[general focus: [Ruby](#) - [Ruby on Rails](#) - [Ruby Rake](#)]

Comments:

Useful information -- thanks. It's good to know and understand the options.

[Ara T Howard suggested a way of doing this](#) which is not unrelated to your third technique.

posted by  Andy Stewart : 6:03 AM

"Of course, it assumes that you get the opportunity to extend instances of the class."

So when is this *not* the case?

```
module ProcessLoggingExtender
  def new
    super().extend(ProcessLogging)
  end
end
```

```
Gateway.extend(ProcessLoggingExtender)
```

posted by  Aristotle : 2:40 PM

Hi Aristotle,

I haven't thought too deeply about it, but yeah, that sounds like it would do the trick.

Thanks for the comment. Jay