Ola Bini: Programming Language Synchronicity

Ruby, Java, Lisp, Io, JRuby. Programming language archeology, creation and discovery.

TORSDAG, MAJ 15, 2008

Dynamically created methods in Ruby

There seems to be some confusion with regards to dynamically defining methods in Ruby. I thought I'd take a look at the three available methods for doing this and just quickly note why you'd use one method in favor of another.

Let's begin by a quick enumeration of the available ways of defining a method after the fact:

- Using a def
- Using define_method
- Using def inside of an eval

There are several things to consider when you dynamically define a method in Ruby. Most importantly you need to consider performance, memory leaks and lexical closure. So, the first, and simplest way of defining a method after the fact is def. You can do a def basically anywhere, but it needs to be qualified if you're not immediately in the context of a module-like object. So say that you want to create a method that returns a lazily initialized value, you can do it like this:

```
class Obj
  def something
  puts "calling simple"
    @abc = 3*42
    def something
    puts "calling memoized"
    @abc
    end
  something
  end
end

o = Obj.new
o.something
o.something
o.something
```

As you can see, we can use the def keyword inside of any context. Something that bites most Ruby programmers at least once - and more than once if they used to be Scheme programmers - is that the second def of "something" will not do a lexically scoped definition inside the scope of the first "something" method. Instead it will define a "something" method on the metaclass of the currently executing self. This means that in the example of the local variable "o", the first call to "something" will first calculate the value and then define a new "something" method on the metaclass of the "o" local variable. This pattern can be highly useful.

Another variation is quite common. In this case you define a new method on a specific object, without that object

being the self. The syntax is simple:

```
def o.something
puts "singleton method"
end
```

This is deceptively simple, but also powerful. It will define a new method on the metaclass of the "o" local variable, constant, or result of method call. You can use the same syntax for defining class methods:

```
def String.something
puts "also singleton method"
end
```

And in fact, this does exactly the same thing, since String is an instance of the Class class, this will define a method "something" on the metaclass of the String object. There are two other idioms you will see. The first one:

```
class << o
  def something
  puts "another singleton method"
  end
end
```

does exactly the same thing as

```
def o.something
puts "another singleton method"
end
```

This idiom is generally preferred in two cases - first, when defining on the metaclass of self. In this case, using this syntax makes what is happening much more explicit. The other common usage of this idiom is when you're defining more than one singleton method. In that case this syntax provide a nice grouping.

The final way of defining methods with def is using module_eval. The main difference here is that module_eval allows you to define new instance methods for a module like object:

```
String.module_eval do
  def something
  puts "instance method something"
  end
end

"foo".something
```

This syntax is more or less equivalent to using the module or class keyword, but the difference is that you can send in a block which gives you some more flexibility. For example, say that you want to define the same method on three different classes. The idiomatic way of doing it would be to define a new module and include that in all the classes. But another alternative would be doing it like this:

```
block = proc do
  def something
  puts "Shared something definition"
  end
end
```

```
String.module_eval &block
Hash.module_eval &block
Binding.module_eval &block
```

The method class_eval is an alias for module_eval - it does exactly the same thing.

OK, so now you know when the def method can be used. Some important notes about it to remember is this: def does _not_ use any enclosing scope. The method defined by def will not be a lexical closure, which means that you can only use instance variables from the enclosing running environment, and even those will be the instance variables of the object executing the method, not the object defining the method. My main rule is this: use def whenever you can. If you don't need lexical closures or a dynamically defined name, def should be your default option. The reason: performance. All the other versions are much harder - and in some cases impossible - for the runtimes to improve. In JRuby, using def instead of define_method will give you a large performance boost. The difference isn't that large with MRI, but that is because MRI doesn't really optimize the performance of general def either, so you get bad performance for both.

Use def unless you can't.

The next version is define_method. It's just a regular method that takes a block that defines that implementation of the method. There are some drawbacks to using define_method - the largest is probably that the defined method can't use blocks, although this is fixed in 1.9. Define_method gives you two important benefits, though. You can use a name that you only know at runtime, and since the method definition is a block this means that it's a closure. That means you can do something like this:

```
class Obj
def something
 puts "calling simple"
 abc = 3*42
 (class <<self; self; end).send :define_method, :something do
   puts "calling memoized"
   abc
 end
 something
end
end

o = Obj.new
o.something
o.something
o.something
```

OK, let this code sample sink in for a while. It's actually several things rolled into one. They are all necessary though. First, note that abc is no longer an instance variable. It's instead a local variable to the first "something" method. Secondly, the funky looking thing(class <<self; self; end) is the easiest way to get the metaclass of the current object. Unlike def, define_method will not implicitly define something on the metaclass if you don't specify where to put it. Instead you need to do it manually, so the syntax to get the metaclass is necessary. Third, define_method happens to be a private method on Module, so we need to use send to get around this. But wait, why don't we just open up the metaclass and call define_method inside of that? Like this:

```
class Obj
def something
 puts "calling simple"
```

```
 abc = 3*42
 class << self
   define_method :something do
     puts "calling memoized"
     abc
   end
 end
 something
end
end

o = Obj.new
o.something
o.something
o.something
```

Well, it's a good thought. The problem is that it won't work. See, there are a few keywords in Ruby that kills lexical closure. The class, module and def keywords are the most obvious ones. So, the reference to abc inside of the define_method block will actually not be a lexical closure to the abc defined outside, but instead actually cause a runtime error since there is no such local variable in scope. This means that using define_method in this way is a bit cumbersome in places, but there are situations where you really need it.

The second feature of define_method is less interesting - it allows you to have any name for the method you define, including something random you come up with at runtime. This can be useful too, of course.

Let's summarize. The method define_method is a private method so it's a bit problematic to call, but it allows you to define methods that are real closures, thus providing some needed functionality. You can use whatever name you want for the method, but this shouldn't be the deciding reason to use it.

There are two problems with define_method. The first one is performance. It's extremely hard to generally optimize the performance of invocation of a define_method method. Specifically, define_method invocations will usually be a bit slower than activating a block, since define_method also needs to change the self for the block in question. Since it's a closure it is harder to optimize for other reasons too, namely we can never be exactly sure about what local variables are referred to inside of the block. We can of course guess and hope and do optimistic improvements based on that, but you can never get define_method invocations are fast as invoking a regular Ruby method.

Since the block sent to define_method is a closure, it means it might be a potential memory leak, as I documented in an older blog post. It's important to note that most Ruby implementations keep around the original self of the block definition, as well as the lexical context, even though the original self is never accessible inside the block, and thus shouldn't be part of the closed environment. Basically, this means that methods defined with define_method could potentially leak much more than you'd expect.

The final way of defining a method dynamically in Ruby is using def or define_method inside of an eval. There are actually interesting reasons for doing both. In the first case, doing a def inside of an eval allows you to dynamically determine the name of the method, it allows you to insert any code before or after the actual functioning code, and most importantly, defining a method with def inside of eval will usually have all the same performance characteristics as a regular def method. This applies for invocation of the method, not definition of it. Obviously eval is slower than just using def directly. The reason that def inside of an eval can be made fast is that at runtime it will be represented in exactly the same way as a regular def-method. There is no real difference as far as the Ruby runtime sees it. In fact, if you want to, you can model the whole Ruby file as running inside of an eval. Not much

difference there. In particular, JRuby will JIT compile the method if it's defined like that. And actually, this is exactly how Rails handles potentially slow code that needs to be dynamically defined. Take a look at the rendering of compiled views in ActionPack, or the route recognition. Both of these places uses this trick, for good reasons.

The other one I haven't actually seen, and to be fair I just made it up. =) That's using define_method inside of an eval. The one thing you would gain from doing such a thing is that you have perfect control over the closure inside of the method defined. That means you could do something like this:

```
class BinderCreator
def get
  abc = 123
  binding
end
end

eval(<<EV, BinderCreator.new.get)
  Object.send :define_method, :something do
  abc
end
EV
```

In this code we create a new method "something" on Object. This method is actually a closure, but it's an extremely controller closure since we create a specific binding where we want it, and then use that binding as the context in which the define_method runs. That means we can return the value of abc from inside of the block. This solution will have the same performance problems as regular define_method methods, but it will let you control how much you close over at least.

So what's the lesson? Defining methods can be complicated in Ruby, and you absolutely need to know when to use which one of these variations. Try to avoid define_method unless you absolutely have to, and remember that def is available in more places than you might think.

Upplagd av Ola Bini kl. 2:35:00 PM

Etiketter: metaprogramming, methods, ruby

11 kommentarer:

Piers Cawley sa...

Hmm... wrapping define_method up in a string eval certainly has possibilities.

Admittedly, not necessarily the sort of possibilities I want to think about too hard, but still...

5:37:00 PM

austinfromboston sa...

Wow, amazingly useful article. Thanks for laying this all out so clearly. One of your paragraphs is getting mis-parsed in the browser. It starts with "Ok, let this code sample sink in..." and ends with a giant _self_ tag.

6:21:00 PM

Sobe sa...

Really, an excellent and very useful article ! Bravo !