Jay Fields' Thoughts

experiences in software development

Monday, February 25, 2008

Ruby: Dynamically Define Method

Defining Methods Dynamically

You have methods that can be defined more concisely if defined dynamically.

```
def failure
 self.state = :failure
end

def error
 self.state = :error
end
```

becomes

```
def_each :failure, :error do |method_name|
 self.state = method_name
end
```

Motivation

I use Dynamically Define Method quite frequently. Of course, I default to defining methods explicitly, but at the point when duplication begins to appear I quickly move to the dynamic definitions.

Dynamically defined methods can help guard against method definition mistakes, since adding another method usually means adding one more argument; however, this is not the primary reason for Dynamically Define Method.

The primary goal for Dynamically Define Method is to more concisely express the method definition in a readable and maintainable format.

Mechanics

- Dynamically define one of the similar methods
- Test

- Convert the additional similar methods to use the dynamic definition
- Test

Example: Using def_each do define similar methods.

Defining several similar methods is verbose and often unnecessary. For example, each of the following methods is simply changing the value of the instance variable state.

```
def failure
 self.state = :failure
end

def error
 self.state = :error
end

def success
 self.state = :success
end
```

The above code executes perfectly well, but it's too similar to justify 11 lines in our source file. The following example could be a first step to removing the duplication.

```
[:failure, :error, :success].each do |method|
 define_method method do
    self.state = method
 end
end
```

Dynamically defining methods in a loop creates a more concise definition, but it's not a particularly readable one. To address this issue I define the def_each method. The motivation for defining a def_each method is that it is easy to notice and understand while scanning a source file.

```
class Class
 def def_each(*method_names, &block)
   method_names.each do |method_name|
     define_method method_name do
        instance_exec method_name, &block
     end
   end
 end
end
```

The instance_exec method

Ruby 1.9 includes instance_exec by default; however, Ruby 1.8 has no such feature. To address this limitation I generally include the following code created

> by Mauricio Fernandez.
>
> ```
> class Object
>   module InstanceExecHelper; end
>  include InstanceExecHelper
>  def instance_exec(*args, &block)
>    begin
>      old_critical, Thread.critical = Thread.critical, true
>      n = 0
>      n += 1 while respond_to?(mname="__instance_exec#{n}")
>      InstanceExecHelper.module_eval{ define_method(mname, &block) }
>    ensure
>      Thread.critical = old_critical
>    end
>    begin
>      ret = send(mname, *args)
>    ensure
>      InstanceExecHelper.module_eval{ remove_method(mname) } rescue nil
>    end
>    ret
>  end
> end
> ```

With def_each now available I can define the methods similar to the example below.

```
def_each :failure, :error, :success do |method_name|
 self.state = method_name
end
```

Example: Defining instance methods with a class method.

The def_each method is a great tool for defining several similar methods, but often the similar methods represent a concept that can be used within code to make the code itself more descriptive.

For example, the previous method definitions were all about setting the state of the class. Instead of using def_each you could define a states class method that would generate the state setting methods. Defining a states class method helps create more expressive code.

```
def error
 self.state = :error
end

def failure
 self.state = :failure
end

def success
 self.state = :success
```

```
end
```

becomes

```
class Post
 def self.states(*args)
   args.each do |arg|
     define_method arg do
        self.state = arg
     end
   end
 end

 states :failure, :error, :success
end
```

Example: Defining methods by extending a dynamically defined module

Sometimes you have an object and you simply want to delegate method calls to another object. For example, you might want your object to decorate a hash so that you can get values by calling methods that match keys of that hash.

As long as you know what keys to expect, you could define the decorator explicitly.

```
class PostData
 def initialize(post_data)
   @post_data = post_data
 end

 def params
   post_data[:params]
 end

 def session
   post_data[:session]
 end
end
```

While this works, it's truly unnecessary in Ruby. Additionally, it's a headache if you want to add new delegation methods. You could define method_missing to delegate directly to the hash, but I find debugging method_missing problematic and avoid it when possible. I'm going to skip straight to defining the methods dynamically from the keys of the hash. Let's also assume that the PostData instances can be passed different hashes, thus we'll need to define the methods on individual instances of PostData instead of defining the methods on the class itself.

```
class PostData
 def initialize(post_data)
   (class << self; self; end).class_eval do
```

```
      post_data.each_pair do |key, value|
       define_method key.to_sym do
          value
       end
     end
   end
 end
end
```

The above code works perfectly well, but it suffers from readability pain. In cases like these I like to take a step back and look at what I'm trying to accomplish.

What I'm looking for is the keys of the hash to become methods and the values of the hash be returned by those respective methods. The two ways to add methods to an instance are to define methods on the metaclass and to extend a module.

Luckily for me, Ruby allows me to define anonymous modules. I have a hash and a decorator, but what I want is a way to define methods of the decorator by extending a hash, so I simply need to convert the hash to a module.

The following code converts a hash to a module with a method for each key that returns the associated value.

```
class Hash
  def to_mod
    hash = self
    Module.new do
      hash.each_pair do |key, value|
        define_method key do
          value
        end
      end
    end
  end
end
```

With the above code in place, it's possible to define the PostData class like the example below.

```
class PostData
  def initialize(post_data)
    self.extend post_data.to_mod
  end
end
```

Labels: define_method, metaprogramming, refactoring

Share: Email | Del.icio.us | Digg | Reddit | Shadow | Rojo

# posted by Jay Fields : 10:29 AM
[general focus: Ruby - Ruby on Rails - Ruby Rake]

Comments:

These seems like bad ideas.
# posted by Brennan : 2:36 PM

In some cases, you would do the same with tests/specs covering refactored code. Often, it works backward - dynamically defined examples or example groups makes you thinking of corresponding changes to the code tested.
# posted by Lq : 3:41 PM

one big downside to all of this define_method hackery is the performance. Methods defined with define_method are over 3 times slower to call then normal def methods. There are also memory leak concerns as define_method closes over scope and can result in unintentional memory leaks as references to objects never get let go.

So please think twice before you use define_method, it will bit you in the end more often then not.
# posted by ezmobius : 4:23 PM

ezmobius, deciding not to use define_method because it is 3 times slower than def seems like premature optimisation to me. I can't help but feel that in (say) a web application, the effect that define_method will have on performance will be negligible, and that areas to focus on with respect to performance will probably be the interaction with the database, and caching of commonly used content. I think it's important to operate in two modes when programming - one where you focus on readability (and ignore performance) and another where you focus solely on performance increases. That way, you can fix those areas of the code that have a large detrimental effect on performance, and don't unnecessarily sacrifice readability in other areas.
# posted by Shane Harvie : 9:46 AM

I must be missing something -- how is this "code generation" (so to speak) at all preferable to simple refactoring, e.g. changing the earlier example into one method, 'def set_state(state); self.state = state; end'?
# posted by Cyrus : 6:10 PM

I've used this approach when I had an array of, basically, enums, and wanted to have 5 or 6 methods of the form is_yellow?(), is_red?(), etc.

I iterated over the values in the array and constructed appropriate funcs programatically - a huge win, because when you add a new enum, all the necessary methods pop up "for free"
# posted by Travis from SmartFlix : 10:09 AM

Post a Comment