shortcut

Addison-Wesley Professional Ruby Series

Writing Efficient Ruby Code

Dr. Stefan Kaes

Addison-Wesley
Pearson Education

www.informIT.com/Ruby

What This Short Cut Covers

Introduction

Patterns

About the Author

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 848-7047

ISBN-13: 978-0-321-54003-4
ISBN-10: 0-321-54003-4

Second release, February 2008

```
Benchmark.bm(10) do |x|
   x.report("each .id"){ n.times{ articles.each {|a| a.id }}}
   x.report("for .id"){ n.times{ for a in articles; a.id; end }}
   x.report("each 1"){ n.times{ articles.each {|a| 1 }}}
   x.report("for 1"){ n.times{ for a in articles; 1; end }}
end
```

shows that the `for` loop is indeed faster (see Table 3).

TABLE 3 `For` **Loop Benchmark Results for Ruby 1.8.5**

	User	System	Total	Real
each .id	2.960000	0.000000	2.960000	(2.961801)
For .id	2.550000	0.000000	2.550000	(2.562234)
each 1	2.120000	0.000000	2.120000	(2.118094)
for 1	1.660000	0.010000	1.670000	(1.660997)

Well, at least in the 1.8 series of Ruby. Ruby 1.9 penalizes `for` loop users severely (see Table 4).

```
 def submit_to_remote(name, value, options = {})
   options[:with] ||= 'Form.serialize(this.form)'
   options[:html] ||= {}
   options[:html][:type] = 'button'
   options[:html][:onclick] = "#{remote_function(options)}; return false;"
   options[:html][:name] = name
   options[:html][:value] = value tag("input", options[:html], false)
end
```

TABLE 4 `For` Loop Benchmark Results for Ruby 1.9

	User	System	Total	Real
each .id	1.970000	0.010000	1.980000	(1.974855)
for .id	6.800000	0.010000	6.810000	(6.806600)
each 1	1.240000	0.000000	1.240000	(1.247908)
for 1	5.790000	0.010000	5.800000	(5.801756)

However, the Ruby 1.9 implementation isn't finished yet, and we can hope that this obvious deficiency will be eliminated for the final release.

Make Decisions at Load Time

When the Ruby interpreter loads a source file containing Ruby code, the source is first parsed and then evaluated. A module or class definition is just a piece of Ruby code that gets evaluated to produce a class/module object inside the interpreter. You can use this dynamicity of Ruby to select alternative implementations based on application configuration options. Example:

```
class C
   def foo(bar)
      if APP_CONTAINER == 'Mongrel'
         # do stuff specific to mongrel
      else # FCGI
         # do something specific to FCGI
      end
   end
end
```

is better written as:

```
class C
   if APP_CONTAINER == 'Mongrel'
      def foo(bar)
         # do stuff specific to mongrel
      end
   else
      def foo(bar)
         # do something specific to FCGI
       end
   end
end
```

The second version is better because the condition gets evaluated only once during the evaluation of the class definition. Additionally, the abstract syntax tree is smaller and therefore saves some CPU cycles during garbage collection.

Self-Modifying Code

Ruby makes it easy to optimize code on the fly by generating or redefining methods at runtime (a.k.a. *self-modifying code*). In Rails core, for example, this happens when you access attributes of `ActiveRecord` objects via their name.

Let's take a look at the previous example: If we assume that the application server type can be determined only dynamically, that is, when method `foo` is called the first time, we can still optimize our code by redefining `foo` at runtime:

```
class C
   def foo(bar)
      if retrieve_app_container == 'Mongrel'
            class_eval <<-end_eval
                  def foo(bar)
                        # do stuff specific to Mongrel
                  end
            end_eval
      else
            class_eval <<-end_eval
                  def foo(bar)
                        # do something specific to FCGI
                  end
            end_eval
      end
      send :foo, bar
   end
end
```

After Ruby has evaluated the class definition, the name `foo` will refer to a method which will alter this association when the method is called. When the interpreter arrives at line 16, the evaluation of one of the `class_eval` code pieces has already replaced the association to refer to the new definition of `foo`. Thus, we can invoke the new code by calling `foo` again.[5]

It should be obvious from the example that your code will get a bit more complicated to look at and understand. Another downside is that it can confuse debuggers and profiling tools. Again, you have to weigh the performance gains against the added complexity for each individual case.

[5] *Note that the old code has become garbage after the first call to `foo` and will be freed upon the next garbage collection.*

An alternative way to write self-modifying code, which is a bit more readable and less confusing to analysis tools, uses access to singleton classes and Ruby's `alias_method` and `remove_method` methods:

```
class C
   def mongrel_foo(bar)
      # do stuff specific to Mongrel
   end
   def fcgi_foo(bar)
      # do something specific to FCGI
   end
   def foo(bar)
      al, rm = :fcgi_foo,:mongrel_foo
      al, rm = rm, al if retrieve_app_container == 'Mongrel'
      singleton = class << self; self; end
      singleton.send :alias_method, :foo, aa
      singleton.send :remove_method, rm
      foo(bar)
   end
 end
```

Test Most Frequent Case First

When writing conditional code, using either if or case expressions, make sure to test in order of expected case frequency. Sometimes this means you need to add additional code, as the following example from Rails shows: