CONTACT@REACTIVE.IO (MAILTO:CONTACT@REACTIVE.IO) | (646) 820-4411 (TEL:646-820-4411)

The Official Blog of Reactive.IO

```
attr_accessor :password

ENCRYPT = Digest::SHA256

has_many :logins, :dependent => :destroy

belongs_to :site, :counter_cache => true

validates_uniqueness_of :email_address, :scope => 'site_id',
                        :message => 'must be unique'

validates_format_of :email_address, :with => /[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,4}/i,
                    :message => "must be valid (e.g. name@domain.com)"

validates_format_of :password, :with => /^([\x20-\x7E]){6,}$/,
                    :unless => :skip_password_encryption_and_validation?
```

The Difference Between Ruby Symbols and Strings (/tips/2009/01/11/the-difference-between-ruby-symbols-and-strings)

January 11, 2009 Robert Sosinski Ruby

Symbols are quite an interesting, and often ill-understood, facet of Ruby. Used extensively throughout Rails and many other Ruby libraries, Symbols are a common sight. However, their rational and purpose is something of a mystery to many Rubyists. This misunderstanding can probably be attributed to many methods throughout Ruby using Symbols and Strings interchangeably. Hopefully, this tutorial will show the value of Symbols and why they are a very useful attribute of Ruby.

Symbols are Strings, Sort Of

The truth of the matter is that Symbols are Strings, just with an important difference, Symbols are immutable. Mutable objects can be changed after assignment while immutable objects can only be overwritten. Ruby is quite unique in offering mutable Strings, which adds greatly to its expressiveness. However mutable Strings can have their share of issues in terms of creating unexpected results and reduced performance. It is for this reason Ruby also offers programmers the choice of Symbols.

So, how flexible are Symbols in terms of representing data? The answer is just as flexible as Strings. Lets take a look at some valid Strings and their Symbol equivalents.

```
"hello"
:hello

"hello world"
:"hello world"

bang = "!"

"hello world#{bang}" # => "hello world!"
:"hello world#{bang}" # => :"hello world!"
```

Many Ruby programmers think that Symbols can only contain alphanumeric characters, however by using quotes you can not only use the same characters as you would a String, but also interpolate Symbols as well. The only thing that a Symbol needs syntactically is a colon `:` prepended. Symbols are actually so similar to Strings that converting between the two is very simple and consistent.

```
:"hello world".to_s # => "hello world"
"hello world".intern # => :"hello world"
```

This being the case, what can't Symbols do? Well, they can't change.

```
puts "hello" << " world"
puts :hello << :" world"

# => hello world
# => *.rb:4: undefined method `<<' for :hello:Symbol (NoMethodError)
```

In this example, we tried to insert `world` into the end of both a String and a Symbol. While the mutable String updated itself, the immutable Symbol gave us an error. From this, we can determine that all the receiver changing methods (e.g. upcase!, delete!, gsub!) we have grown to depend on with Strings will not be with us in Symbols. This being the case, why would Ruby even have Symbols as Strings seem to be quite a flexible upgrade? The reason is that while mutability is very useful, it can also cause quite a few problems.

Mutability Gone Wrong

Strings often perform double duty in an program by both storing data and driving operation. For holding data, mutability offers us a slew of expressive options. However, a programs operation (especially in critical applications) should be a bit more ridged. Take the following example:

```
status = "peace"

buggy_logger = status

print "Status: "
print buggy_logger << "\n" # <- This insertion is the bug.

def launch_nukes?(status)
  unless status == 'peace'
    return true
  else
    return false
  end
end

print "Nukes Launched: #{launch_nukes?(status)}\n"

# => Status: peace
# => Nukes Launched: true
```

In this example, a script silently watches the world for aggression and danger. Once things go wrong, it launches the nukes. However, by having a mutable String control the program's operation, a buggy logger disintegrates the entire world. Now, the above script does make some mistakes (I would have cloned the status String instead of just assigning it for example). However, we can see that relying on something that can change during run-time can cause some unexpected results.

Frozen Strings

There are two ways to handle the mutability issue above. The first is to freeze the String and thus making it immutable. Once a String is frozen it cannot be changed.

```
example = "hello world"
example.upcase!

puts example

example.freeze
example.downcase!

# => "HELLO WORLD"
# => *.rb:7:in `downcase!': can't modify frozen string (TypeError)
```

The second way would be to make your program use Symbols. But if frozen Strings work just as Symbols, we are back to wondering the true value of Symbols and why they should be used.

String and Symbol Performance

Because Strings are mutable, the Ruby interpreter never knows what that String may hold in terms of data. As such, every

String needs to have its own place in memory. We can see this by creating some Strings and printing their object id.

```
puts "hello world".object_id
puts "hello world".object_id
puts "hello world".object_id
puts "hello world".object_id
puts "hello world".object_id

# => 3102960
# => 3098410
# => 3093860
# => 3089330
# => 3084800
```

NOTE: Your object id's will be different then the ones above.

What we see above might not seem like a big issue, but behind the scenes is some heavy waste. To understand why, we first have to understand what is going on under the hood. An abridged explanation is as follows:

1. First, a new String object is instantiated with the value of `hello world`.
2. The Ruby interpreter needs to look at the heap (your computers memory), find a place to put the new string and keep track of it via its object id.
3. The String is passed to the puts method, and output to the screen.
4. The Ruby interpreter sees that the String will not be used again as it is not assigned to a variable, and marks it for destruction.
5. Back to step one four more times.

Ruby's garbage collector, or GC, is of the mark and sweep variety. This means that objects to be destroyed are marked during the programs operation. The GC then sweeps up all the marked objects on the heap whenever it has some spare time. However, above we are creating a new object, storing and ultimately destroying it when reusing the same object would be much more efficient. Lets do the same thing but with Symbols instead.

```
puts :"hello world".object_id
puts :"hello world".object_id
puts :"hello world".object_id
puts :"hello world".object_id
puts :"hello world".object_id

# => 239518
# => 239518
# => 239518
# => 239518
# => 239518
```

This time, every Symbol shares the same object id, and as such the same space on the heap. To further increase performance, Ruby will not mark Symbols for destruction, allowing you to reuse them again and again. As Symbols stay in memory throughout the programs operation, we can quickly snag them from memory instead of instantiating a new copy every time. In fact, Symbols are not only stored in memory, they are also keep track of via an optimized Symbols dictionary. You can see it by running the next example.

```
puts Symbol.all_symbols.inspect

# => A long list of Symbols...
```

The `all_symbols` class method will return an Array of every Symbol currently used in your program. Every time you create a new Symbol, it is also put here. When you use a Symbol, Ruby will check the dictionary first, and if available it will be fetched and used. If the Symbol cannot be found in the dictionary, only then will the interpreter instantiate a new Symbol and put it in the heap. We can easily see this process by dropping into IRB and typing the following:

```
>> Symbol.all_symbols.collect{|sym| sym.to_s}.include?("new_symbol")
=> false
>> :new_symbol
=> :new_symbol
>> Symbol.all_symbols.collect{|sym| sym.to_s}.include?("new_symbol")
=> true
```

So how much faster are Symbols compared to Strings? Well, lets build a script to benchmark the difference.

```
require 'benchmark'

str = Benchmark.measure do
  10_000_000.times do
    "test"
  end
end.total

sym = Benchmark.measure do
  10_000_000.times do
    :test
  end
end.total

puts "String: " + str.to_s
puts "Symbol: " + sym.to_s
puts
```

In this example, we are creating 10 million new Strings and then 10 million new Symbols. With the Benchmark library, we can find out how long each activity takes and compare. After running this script three times, I got the following results (yours will most likely be different).

```
$ ruby benchmark.rb
String: 2.24
Symbol: 1.32

$ ruby benchmark.rb
String: 2.25
Symbol: 1.32

$ ruby benchmark.rb
String: 2.24
Symbol: 1.33
```

On average, we are getting a 40% increase solely by using Symbols. However, Symbols are not just faster then Strings in how they are stored and used, but also in how they are compared. As Symbols of the same text share the same space on the heap, testing for equality is as quick as comparing the object id's. Testing equality for Strings however, is a bit more computationally expensive. As every String has its own place in memory, the interpreter has to compare the actual data making up the String. A good example to visualize this type of check would be the following:

```
a = "test".split(//)
b = "test".split(//)
e = true

number = (a.length > b.length) ? a.length : b.length

number.times do |index|
  if a[index] == b[index]
    puts "#{a[index]} is equal to #{b[index]}"
  else
    puts "#{a[index]} is not equal to #{b[index]}"
    e = false
    break
  end
end

puts "#{a.join} equal to #{b.join}: #{e}"

# => t is equal to t
# => e is equal to e
# => s is equal to s
# => t is equal to t
# => test equal to test: true
```

So, back to benchmarking. How much faster is comparing Symbols then to Strings? Lets find out:

```
require 'benchmark'

str = Benchmark.measure do
  10_000_000.times do
    "test" == "test"
  end
end.total

sym = Benchmark.measure do
  10_000_000.times do
    :test == :test
  end
end.total

puts "String: " + str.to_s
puts "Symbol: " + sym.to_s
puts
```

Just like before, I ran this script three times and got the following results.

```
$ ruby benchmark.rb
String: 4.48
Symbol: 2.39

$ ruby benchmark.rb
String: 4.47
Symbol: 2.39

$ ruby benchmark.rb
String: 4.47
Symbol: 2.38
```

And just like before, we are close to a cutting our processing time by half. Not so bad.

Summary

Overall, we went though quite a bit in this tutorial. First, we learned about mutability and how Strings and Symbols are different in this regard. We then jumped into when to use each and finally looked at the performance characteristics of both. Now that you have a better idea of this interesting aspect of Ruby, you can start using Symbols to keep your applications running faster and more consistently.

About Reactive.IO

Based in NYC, Reactive.IO builds your mission-critical software right the first time.

Get Reactive.TODAY

Subscribe to Reactive.TODAY, the newsletter that keeps you ahead of the competition.

Email Address Subscribe

New Reactive.TIPS

Parallel Computing Made Easy With Scala and Akka (/tips/2014/04/08/parallel-computing-made-easy-with-scala-and-akka)

Getting Started with Actor Based Programming Using Scala and Akka (/tips/2014/03/28/getting-started-with-actor-based-programming-using-scala-and-akka)

Binding Scope in JavaScript (/tips/2009/04/28/binding-scope-in-javascript)

Get Connected

222 Broadway
New York, NY 10038
(https://www.google.com/maps/place/222+Broadway/@40.710968,-74.0084713,18z
/data=!3m1!4b1!4m2!3m1!1s0x89c25a18456b1dc7:0xa07b4b73fc1baa2d)

(646) 820-4411 (tel:646-820-4411)
contact@reactive.io (mailto:contact@reactive.io)

Stay Connected

in (https://www.linkedin.com/company/reactive-io) (https://github.com/reactive-io) (https://angel.co/reactive-io) </> (https://coderwall.com/team/reactive-io) (https://twitter.com/reactive_io)