Ruby master - Feature #7792

Make symbols and strings the same thing

02/06/2013 10:48 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Status:	Rejected
Priority:	Normal
Assignee:	matz (Yukihiro Matsumoto)
Target version:	3.0

Description

Recently I had to replace several of my symbols to plain strings in my project. Here is what happened:

I generated some results with some class that would add items to an array like this:

results << {id: 1, name: 'abc'}

Then I would store such results in cache using Redis, encoded as a JSON string. But then when I restore the data from cache the hash will be {'id' => 1, 'name' => 'abc'}.

This wasn't a problem until recently because I never used the results directly in the same request before and would always use the value stored on Redis and parsed by JSON.

But recently I had to use the values directly in a view. But then I had a problem because I would have to use symbols in the results for the first time and strings the next times when the result was available on cache. I really don't want to care about memory management in Ruby if possible and symbols forces me to abandon the new sexy hash syntax many times. Now I have to write

results << {'id' => 1, 'name' => 'abc}

when I'd prefer to write

results << {id: 1, name: 'abc}

This is not the first time I had a bad user experience due to symbols being different from strings. And I'm not the only one or ActiveSupport Hash#with_indifferent_access wouldn't be so popular and Rails wouldn't use it all the time internally.

It is really bad when you have to constantly check how you store your keys in your hashes. Am I using symbols or strings as keys? If you use the wrong type on plain hashes you can find a bad time debugging your code. Or you could just use Hash#with_indifferent_access everywhere, thus reducing performance (I guess) and it is pretty inconvenient anyway.

Or if you're comparing the keys of your hash in some "each" closure you have to worry about it being a symbol or a string too.

Ruby is told to be programmers' friendly and it usually is. But symbols are certainly a big exception.

Related issues:

Related to Ruby master - Feature #5964: Make Symbols an Alternate Syntax for ...	**Rejected**

History

#1 - 02/06/2013 11:13 PM - trans (Thomas Sawyer)

I'm not sure that's even possible. If String#hash produced the same number as Symbol#hash, then that would do the trick, but that probably lead to some unforeseen breakages.

I do know one related thing, though. I don't like having to do:

```
if (String === x || Symbol === x)
  x.to_s
```

When all I really want to do is x.to_str, but handle Symbols too.

Oh, btw, I've suggested that Hash add a convert_key procedure which could be used to normalize keys. Very useful, but would obviously mean a bit of speed hit.

#2 - 02/06/2013 11:20 PM - luislavena (Luis Lavena)

rosenfeld: see #5964 for similar discussion.

#3 - 02/06/2013 11:22 PM - shyouhei (Shyouhei Urabe)

Mmm, it sounds too big to me.

rosenfeld (Rodrigo Rosenfeld Rosas) I know your situation. But is it really a right solution for you? How about a hash with indifferent access? Or how about changing {foo:1} to be { 'foo' => 1 }, not { :foo => 1 } ? It seems your frustration can be relaxed in several ways. Letting symbols be strings is not the only one and (seems to be) suboptiomal.

#4 - 02/06/2013 11:23 PM - yorickpeterse (Yorick Peterse)

Symbols and Strings both have different use cases and that's actually a good thing. If you want to be able to use both Strings and Symbols as your Hash keys you can use something like Hashie: https://github.com/intridea/hashie

Yorick

#5 - 02/06/2013 11:29 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Although I'd really prefer that symbols and strings were the same there is an alternative that would satisfy me as well:

Make Hash behave as HashWithIndifferentAccess and create a new class StrictHash to keep the behavior of the existent Hash class. That way this would work:

a = {a: 1}; a[:a] == a['a']

Also any stdlib libraries, such as JSON, should use Hash instead of StrictHash on parse.

That way it wouldn't really matter if {foo: 1} maps to {:foo => 1} or {'foo' => 1}. By the way I'm still curious to know if we'll ever be able to use string interpolation in the hash syntax, like {"#{'foo'}": 1}.

#6 - 02/06/2013 11:33 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

yorickpeterse (Yorick Peterse), your suggestion wouldn't work for my case. The hash is created by JSON.parse where I don't control the hash creation. And I don't like the idea of monkey patching core classes either. Specially in fundamental classes like Hash. If I patch it to make it behave like HashWithIndifferentAccess it could break many libraries/frameworks in unexpected ways.

#7 - 02/06/2013 11:53 PM - yorickpeterse (Yorick Peterse)

You don't need to hijack any code for it, you'd just use it as following:

```
require 'hashie'

parsed = JSON.parse('{"name": "Ruby"}')
hash   = Hashie::Mash.new(parsed)

hash.name     # => "Ruby"
hash['name'] # => "Ruby"
hash[:name]  # => "Ruby"
```

We use Hashie in various production applications and it works quite well for us.

Yorick

#8 - 02/06/2013 11:53 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Em 06-02-2013 12:36, Yorick Peterse escreveu:

> You don't need to hijack any code for it, you'd just use it as following:
>
> ```
> require 'hashie'
>
> parsed = JSON.parse('{"name": "Ruby"}')
> hash   = Hashie::Mash.new(parsed)
>
> hash.name     # => "Ruby"
> hash['name'] # => "Ruby"
> hash[:name]  # => "Ruby"
> ```
>
> We use Hashie in various production applications and it works quite well for us.

So you get a performance hit because you don't want to worry about symbols while symbols are meant to give you better performance, right? How ironic is that?

#9 - 02/06/2013 11:58 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

I'd just like to highlight what performance impact we may be referring to:

https://gist.github.com/rosenfeld/4723061

I'll copy it here:

require 'benchmark'

hashes = []
1_000_000.times { hashes << { some_key_name: 1, 'some_key_name' => 2 }}
Benchmark.bmbm do |x|
x.report { hashes.map{|h| h[:some_key_name]} }
x.report { hashes.map{|h| h['some_key_name']} }
end

Result:

ruby symbols-performance.rb
Rehearsal -----------------------------------
0.540000 0.010000 0.550000 (0.543421)
0.680000 0.020000 0.700000 (0.705931)
------------------------- total: 1.250000sec

```
   user     system      total        real
```

0.240000 0.000000 0.240000 (0.244554)
0.380000 0.000000 0.380000 (0.391517)

Does this small hit in performance worth all the hassle that it is to have to differentiate symbols from strings?

#10 - 02/07/2013 12:29 AM - yorickpeterse (Yorick Peterse)

I don't think I'm following you, can you explain what's supposedly ironic about it? Using Hashie only "slows" things down based on whether you use Symbols, Strings or object attributes. Unless you use it *all* over the place the performance impact is small.

I personally don't fully agree with what Hashie does because I believe people should be competent enough to realize that when they take in external data it's going to be String instances (for keys that is).

Having said that, I think fundamentally changing the way Ruby works when it comes to handling Strings and Symbols because developers can't be bothered fixing the root cause of the problem is flawed. If you're worried about a ddos stop converting everything to Symbols. If you're worried about not remember what key type to use, use a custom object or document it so that people can easily know.

While Ruby is all about making the lifes easier I really don't want it to become a language that spoon feeds programmers because they're too lazy to type 1 extra character *or* convert the output manually. Or better: use a custom object as mention above.

The benchmark you posted is flawed because it does much, much more than benchmarking the time required to create a new Symbol or String instance. Lets take a look at the most basic benchmark of these two data types:

```
require 'benchmark'

amount = 50000000

Benchmark.bmbm(40) do |run|
  run.report 'Symbols' do
    amount.times do
      :foobar
    end
  end
```

```
    run.report 'Strings' do
      amount.times do
        'foobar'
      end
    end
  end
```

On the laptop I'm currently using this results in the following output:

```
  Rehearsal
-----------------------------------------------------------
  Symbols                                  2.310000   0.000000
```

2.310000 (2.311325)
Strings 5.710000 0.000000
5.710000 (5.725365)
---
total: 8.020000sec

```
                                               user     system
```

total real
Symbols 2.670000 0.000000
2.670000 (2.680489)
Strings 6.560000 0.010000
6.570000 (6.584651)

This shows that the use of Strings is roughly 2,5 times slower than Symbols. Now execution time isn't the biggest concern in this case, it's memory usage. For this I used the following basic benchmark:

```
  def get_memory
    return `ps -o rss= #{Process.pid}`.strip.to_f
  end

  def benchmark_memory
    before = get_memory

    yield

    return get_memory - before
  end

  amount = 50000000

  puts "Start memory: #{get_memory} KB"

  symbols = benchmark_memory do
    amount.times do
      :foobar
    end
  end

  strings = benchmark_memory do
    amount.times do
      'foobar'
    end
  end

  puts "Symbols used #{symbols} KB"
  puts "Strings used #{strings} KB"
```

This results in the following:

```
  Start memory: 4876.0 KB
  Symbols used 0.0 KB
  Strings used 112.0 KB
```

Now I wouldn't be too surprised if there's some optimization going on because I'm re-creating the same values over and over again but it already shows a big difference between the two.

To cut a long story short: I can understand what you're trying to get at, both with the two data types being merged and the ddos issue.

However, I feel neither of these issues are an issue directly related to Ruby itself. If Ruby were to automatically convert things to Symbols for you then yes, but in this case frameworks such as Rails are the cause of the problem. Merging the two datatypes would most likely make such a huge different usage/code wise that it would probably be something for Ruby 5.0 (in other words, not in the near future).

Yorick

#11 - 02/07/2013 02:23 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Em 06-02-2013 13:25, Yorick Peterse escreveu:

> I don't think I'm following you, can you explain what's supposedly ironic about it? Using Hashie only "slows" things down based on whether you use Symbols, Strings or object attributes. Unless you use it *all* over the place the performance impact is small.

What I'm trying to say is that the main reason why symbols exist in Ruby in the first place is performance from what I've been told.

But then people don't want to worry if hashes are indexed by strings or symbols so they end up using some kind of HashWithIndifferentAccess or similar techniques. But since the normal Hash class doesn't behave this way you have to loop through all hashes in an object returned by JSON.parse to make them behave as HashWithIndifferentAccess, which is has a huge performance hit when compared to the small gains symbols could add.

> I personally don't fully agree with what Hashie does because I believe people should be competent enough to realize that when they take in external data it's going to be String instances (for keys that is).

It is not a matter of being competent or not. You can't know in advance if a hash returned by some external code is indexed by string or symbols. You have to test by yourself or check the documentation. Or you could just use a HashWithIndifferentAccess class and stop worrying about it. This has a big impact on coding speed and software maintenance, which is the big problem in my opinion.

> Having said that, I think fundamentally changing the way Ruby works when it comes to handling Strings and Symbols because developers can't be bothered fixing the root cause of the problem is flawed.

People reading some Ruby book will notice that it is not particularly designed with performance in mind but it is designed mostly towards programmer's happiness. If that is the case, then worrying about bothered programmers makes sense to a language like Ruby in my opinion.

> If you're worried about a ddos

DDoS is a separate beast that can't be easily prevented no matter what language/framework you use. I'm just talking about DoS exploiting through memory exhaustion due to symbols not being collected. Anyway, this is a separate issue from this one and would be better discussed in that separate thread.

> stop converting everything to Symbols.

I'm not converting anything to symbols. Did you read the feature description use case? I'm just creating regular hashes using the new sexy hash syntax which happens to create symbols instead of strings. Then when I serialize my object to JSON for storing on Redis for caching purpose I'll get a hash indexed by strings instead of symbols. That means that when I'm accessing my hash I have to be worried if the hash has been just generated or if it was loaded from Redis to decide if I should use strings or symbols to get the hash values. There are many more similar situations where this difference between symbols and strings will cause confusion. And I don't see much benefits in keeping them separate things either.

> If you're worried about not remember what key type to use, use a custom object or
> document it so that people can easily know.

This isn't possible when you're serializing/deserializing using some library like JSON or any other. You don't control how hashes are created by such libraries.

> While Ruby is all about making the lifes easier I really don't want it to become a language that spoon feeds programmers because they're too lazy to type 1 extra character *or* convert the output manually. Or better: use a custom object as mention above.

Again, see the ticket description first before assuming things.

> The benchmark you posted is flawed because it does much, much more than benchmarking the time required to create a new Symbol or String instance. Lets take a look at the most basic benchmark of these two data types:
>
> ```
> require 'benchmark'
>
> amount = 50000000
>
> Benchmark.bmbm(40) do |run|
>   run.report 'Symbols' do
>     amount.times do
>       :foobar
>     end
>   end
>
>   run.report 'Strings' do
>     amount.times do
>       'foobar'
>     end
>   end
> end
> ```
>
> On the laptop I'm currently using this results in the following output:
>
> ```
> Rehearsal
> ```
>
> ---
>
> ```
> Symbols                                  2.310000   0.000000
> ```
>
> 2.310000 (2.311325)
> Strings 5.710000 0.000000
> 5.710000 (5.725365)
>
> ---
>
> total: 8.020000sec
>
> ```
>                                             user     system
> ```
>
> total real
> Symbols 2.670000 0.000000
> 2.670000 (2.680489)
> Strings 6.560000 0.010000
> 6.570000 (6.584651)
>
> This shows that the use of Strings is roughly 2,5 times slower than Symbols. Now execution time isn't the biggest concern in this case, it's memory usage.

Exactly, no real-world software would consist mostly of creating strings/symbols. Even in a simplistic context like my example, it is hard to notice any impact on the overall code caused by string allocation taking more time than symbols.When we get more complete code we'll notice that it really doesn't make any difference if we're using symbols or strings all over our code...

Also, any improvements on threading and parallelizing support are likely

to yield much bigger performance boots than any micro-optimization with symbols instead of strings.

> For this I used the following basic benchmark:
>
> ```
> def get_memory
>   return `ps -o rss= #{Process.pid}`.strip.to_f
> end
>
> def benchmark_memory
>   before = get_memory
>
>   yield
>
>   return get_memory - before
> end
>
> amount = 50000000
>
> puts "Start memory: #{get_memory} KB"
>
> symbols = benchmark_memory do
>   amount.times do
>     :foobar
>   end
> end
>
> strings = benchmark_memory do
>   amount.times do
>     'foobar'
>   end
> end
>
> puts "Symbols used #{symbols} KB"
> puts "Strings used #{strings} KB"
> ```
>
> This results in the following:
>
> ```
> Start memory: 4876.0 KB
> Symbols used 0.0 KB
> Strings used 112.0 KB
> ```
>
> Now I wouldn't be too surprised if there's some optimization going on because I'm re-creating the same values over and over again but it already shows a big difference between the two.

112KB isn't certainly a big difference in my opinion unless you're designing some embedded application. I've worked with embedded devices in the past and although I see some attempts to make a lighter Ruby subset (like mRuby) for such use-case I'd certainly use C or C++ for my embedded apps these days. Did you know that Java initially was supposed to be used by embedded devices from what I've been told? Then it tried to convince people to use it to create multi-platform desktop apps. After that its initial footprint was so big that it wasn't a good idea to try it on embedded devices for most cases. Then they tried to make it work in browsers through applets. Now it seems people want to use Java mostly for web servers (HTTP and other protocols). The result was a big mess in my opinion. I don't think Ruby (the full specification) should be concerned about embedded devices. C is already a good fit for devices with small memory constraints. When you consider using Ruby it is likely that you have more CPU and memory resources than a typical small device would have, so 112KB wouldn't make much difference.

And for embedded devices, it is also recommended that they run some RTOS instead of plain Linux. If they want to keep with Linux, an option would be to patch it with Xenomai patch for instance. But in that case, any real-time task would be implemented in C, not in Ruby or any other language subjected to garbage collected, like Java. So, if we keep the focus on applications running on normal computers, 112KB won't really make any difference, don't you agree?

> To cut a long story short: I can understand what you're trying to get at, both with the two data types being merged and the ddos issue. However, I feel neither of these issues are an issue directly related to

> Ruby itself. If Ruby were to automatically convert things to Symbols for you then yes, but in this case frameworks such as Rails are the cause of the problem.

Rails is not related at all to the use case I pointed out in this ticket description. It happens with regular Ruby classes (JSON, Hash) and with the "redis" gem that is independent from Rails.

> Merging the two datatypes would most likely make such a huge different usage/code wise that it would probably be something for Ruby 5.0 (in other words, not in the near future).

Ruby 3.0 won't happen in a near future. Next Major means Ruby 3.0 if I understand it correctly.

#12 - 02/07/2013 03:23 AM - yorickpeterse (Yorick Peterse)

> What I'm trying to say is that the main reason why symbols exist in Ruby in the first place is performance from what I've been told.

Correct, and your proposed changes would completely nullify those performance benefits (see below).

> People reading some Ruby book will notice that it is not particularly designed with performance in mind but it is designed mostly towards programmer's happiness. If that is the case, then worrying about bothered programmers makes sense to a language like Ruby in my opinion.

So basically what you're saying is "Ruby is written for happiness and not performance, lets make it even more slow!". I'd rather see a world where Ruby is both fast (enough) and easy to use instead of it being easy to use and slower than a sloth.

Regarding the benchmarking information, you're missing a crucial aspect. While the numbers in the specific examples I gave both clearly show that the use of Strings is substantially slower. Yes, it's "only" 112 kb but the difference will keep growing and growing until you hit your memory limit.

This is exactly one of the reasons Symbols exist: to make it easier and faster to use commonly re-used Strings. The best example of this are Hash keys.

> This isn't possible when you're serializing/deserializing using some library like JSON or any other. You don't control how hashes are created by such libraries.

Of course it is. Marshal allows you to store arbitrary Ruby objects (with the exception of a few such as Proc instances), in other cases you can re-create your objects based on the supplied Hash.

If you do not like using raw Hashes the solution in my opinion is not to more or less re-write Ruby (and break everything that exists in the process) but instead solve this on your own application level. Using Hashie is one example but another one, one I consider far better, is to use your own classes. Consider the following:

```
hash = {'id' => '123-abc-456', 'body' => 'hello'}

if hash['id'] and !hash['id'].empty?
  puts "Processing message ID #{hash['id']}"
end

if hash['body'] and !hash['body'].empty?
  do_something(hash['body'])
end
```

This is not a flaw in your proposal in particular but it's one of the reasons why I'm not a big fan of using Hashes all over the place. If in this example the "id" key is suddenly changed to "ID" you now have at least 3 places where you have to modify your code and most likely other places further down the line. This can be solved by doing something as simple as the following:

```
class Message
  attr_reader :id, :body

  def initialize(options)
    @id   = options['id']
    @body = options['body']
  end

  def has_id?
    return @id && !@id.empty?
  end

  def has_body?
    return @body && !@body.empty?
  end
end
```

This allows you to write the following instead:

```
message = Message.new({'id' => '123-abc-456', 'body' => 'hello'})

if message.has_id?
  puts "Processing message ID #{message.id}"
end

if has_body?
  do_something(message.body)
end
```

In this specific example it may seem a bit like an overkill but if that Hash gets used in a dozen places you can save yourself tremendous amounts of time by just wrapping a class around it.

Yorick

#13 - 02/07/2013 03:53 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Em 06-02-2013 16:22, Yorick Peterse escreveu:

> > What I'm trying to say is that the main reason why symbols exist in Ruby in the first place is performance from what I've been told.
> > Correct, and your proposed changes would completely nullify those performance benefits (see below).
> >
> > People reading some Ruby book will notice that it is not particularly designed with performance in mind but it is designed mostly towards programmer's happiness. If that is the case, then worrying about bothered programmers makes sense to a language like Ruby in my opinion.
> > So basically what you're saying is "Ruby is written for happiness and not performance, lets make it even more slow!". I'd rather see a world where Ruby is both fast (enough) and easy to use instead of it being easy to use and slower than a sloth.
>
> Regarding the benchmarking information, you're missing a crucial aspect. While the numbers in the specific examples I gave both clearly show that the use of Strings is substantially slower. Yes, it's "only" 112 kb but the difference will keep growing and growing until you hit your memory limit.

Man, you're instantiating 50 millions strings and it only increased the memory in 112KB. If your application creates so many strings that won't be garbage collected then it is unlikely that symbols would help as a replacement.

And "growing until you hit your memory limit" is actually only valid for

symbols, not for strings that are garbage collected already. Unless you have some leak in your code that prevent those strings from being collected by GC.

> This is exactly one of the reasons Symbols exist: to make it easier and faster to use commonly re-used Strings. The best example of this are Hash keys.

Most of the programming languages don't support the concept of symbols like Ruby. And you won't see C or C++ programmers complaining about this neither.

> > This isn't possible when you're serializing/deserializing using some library like JSON or any other. You don't control how hashes are created by such libraries.
> > Of course it is. Marshal allows you to store arbitrary Ruby objects (with the exception of a few such as Proc instances), in other cases you can re-create your objects based on the supplied Hash.

Marshal is not portable across multiple languages (I use both Groovy and Ruby in my overall application interacting with Redis). I'm talking about JSON here. You don't have to find an alternative to JSON. Just try to understand the issue I'm talking about.,

> If you do not like using raw Hashes the solution in my opinion is not to more or less re-write Ruby (and break everything that exists in the process)

Like what?

> but instead solve this on your own application level. Using Hashie is one example but another one, one I consider far better, is to use your own classes. Consider the following:

Ok, I won't repeat myself. Please give an example for the Redis + JSON serialization use case presented in the ticket description.

Otherwise you cleared missed the point.

#14 - 02/07/2013 04:23 AM - yorickpeterse (Yorick Peterse)

> And "growing until you hit your memory limit" is actually only valid for symbols, not for strings that are garbage collected already. Unless you have some leak in your code that prevent those strings from being collected by GC.

Since existing code (and developers) assume that Symbols are only created once they are generally used all over the place without any side effects. The moment you start garbage collecting them there's an increased chance of the GC kicking in right in the middle of (say) an HTTP request. Yes, you may now be able to use both Symbols and Strings as Hash keys but you now have to deal with increased GC activity.

Note that this of course depends on the code you're using. If you use carefully written code that doesn't use Symbols this is not going to be an issue. However, pretty every single Gem out there uses them and a lot of them also use them quite heavily.

> Most of the programming languages don't support the concept of symbols like Ruby. And you won't see C or C++ programmers complaining about this neither

C has a goto operator, does that mean Ruby should have one too (I'm aware it's already there, it's not just enabled unless you specify some compiler flag)? Just because one language has feature X it doesn't mean all the others should have it too.

> Marshal is not portable across multiple languages (I use both Groovy and Ruby in my overall application interacting with Redis). I'm talking about JSON here. You don't have to find an alternative to JSON. Just try to understand the issue I'm talking about.,

It wasn't suggested as an alternative, merely an example that there is a way of serializing arbitrary Ruby data.

> Ok, I won't repeat myself. Please give an example for the Redis + JSON serialization use case presented in the ticket description.
>
> Otherwise you cleared missed the point.

Take a closer look at my previous Email, there's a fairly big example at the bottom of it that you can't really miss. However, just in case:

```
require 'redis'
require 'json'

client = Redis.new

client.set('user', JSON({'id' => 1, 'name' => 'John Doe'}))

# This would happen somewhere else (e.g. an external process)
hash = JSON(client.get('user'))
user = User.new(hash)

# instead of user['id'] you can now just do `user.id` which means
# that if the key name ever changes you only have to change it in
# one place.
if user.id
  # ...
end
```

Another benefit is that this lets you attach your own methods to the object without having to monkeypatch existing classes or using helper methods (which feels very much like procedural programming).

Yorick

#15 - 02/07/2013 05:19 AM - drbrain (Eric Hodel)

- Status changed from Open to Rejected

=begin
This proposal has no description of how to overlay the functionality of strings (mutable) with symbols (immutable).

This was previously tried during 1.9 which had such a plan but was ultimately rejected.

Due to a previous attempt and failure along with the lack of a concrete plan in this feature request I will reject this.

As to symbols or strings as hash keys, you should almost always use strings. This is the current community best-practice consensus.

You should only use symbols as keys in a Hash if you have a small, fixed set of keys and do not use user input to look up items in the hash.

Converting a string to a symbol you look up in a hash is not recommended. You have created two hash lookups out of one (the first for string to symbol mapping, the second for the lookup in your hash) and you risk a DoS as symbols are not garbage collected.

Consider this benchmark:

require 'benchmark'

N = 10_000_000

Benchmark.bmbm do |bm|
bm.report 'NULL' do
h = { 'foo' => 'bar'}

```
  N.times do
    # null
  end
end
```

```
bm.report 'string key, string lookup' do
  h = { 'foo' => 'bar' }

  N.times do
    h['foo']
  end
end

bm.report 'symbol key, symbol lookup' do
  h = { :foo => 'bar' }

  N.times do
    h[:foo]
  end
end

bm.report 'symbol key, string intern lookup' do
  h = { :foo => 'bar' }

  N.times do
    h['foo'.intern]
  end
end
```

end

Here are the results:

```
Rehearsal ----------------------------------------------------------------
NULL                          0.440000   0.000000   0.440000 (  0.448186)
string key, string lookup     1.870000   0.000000   1.870000 (  1.866935)
symbol key, symbol lookup     0.660000   0.000000   0.660000 (  0.661466)
symbol key, string intern lookup   2.230000   0.000000   2.230000 (  2.222772)
------------------------------------------------------------ total: 5.200000sec

                                        user     system      total        real

NULL                          0.430000   0.000000   0.430000 (  0.434306)
string key, string lookup     1.860000   0.000000   1.860000 (  1.862942)
symbol key, symbol lookup     0.680000   0.000000   0.680000 (  0.681767)
symbol key, string intern lookup   2.270000   0.010000   2.280000 (  2.264888)
```

While symbol key and symbol lookup is 2.7 times faster than string key and string lookup, it is also 1.2 times faster than interning a string for a symbol lookup.

=end

#16 - 02/07/2013 08:01 AM - phluid61 (Matthew Kerwin)

There's another issue here, which has been overlooked because it's philosophical rather than technical:

symbols aren't strings.

A string's value is a sequence of characters, which can be iterated over and operated on.

A symbol's only value is itself, is used as a token, and only supports comparison and casting operations. In MRI it's actually implemented as an integer, which can be looked up by its name (like a C enum).

I strongly believe there is already too much conflation between the two (completely unrelated) types. We can never benefit by blurring them any further.

#17 - 02/07/2013 08:51 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

drbrain (Eric Hodel), that means I'm unable to do things like

results << {id: id, name: name}

and have to use the old syntax all the time (and worrying about the differences all the time):

results << {'id' => id, 'name' => 'name'}

I still believe MRI should try to optimize frozen strings instead of symbols and just make symbols behave like frozen strings. :symbol would be a shortcut to 'symbol'.freeze.

I don't really think any of those micro-benchmarks would justify all the hassle and unhappiness that symbols bring to Ruby programmers.

#18 - 02/07/2013 08:53 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Em 06-02-2013 17:12, Yorick Peterse escreveu:

> And "growing until you hit your memory limit" is actually only valid
> for symbols, not for strings that are garbage collected already.
> Unless you have some leak in your code that prevent those strings from
> being collected by GC.
> Since existing code (and developers) assume that Symbols are only
> created once they are generally used all over the place without any side
> effects. The moment you start garbage collecting them there's an
> increased chance of the GC kicking in right in the middle of (say) an
> HTTP request. Yes, you may now be able to use both Symbols and Strings
> as Hash keys but you now have to deal with increased GC activity.

You currently already don't control when GC activity begins, so I don't understand how this could be considered any disadvantage...

> Note that this of course depends on the code you're using. If you use
> carefully written code that doesn't use Symbols this is not going to be
> an issue. However, pretty every single Gem out there uses them and a lot
> of them also use them quite heavily.
>
> > Most of the programming languages don't support the concept of symbols
> > like Ruby. And you won't see C or C++ programmers complaining about
> > this neither
> > C has a goto operator, does that mean Ruby should have one too (I'm
> > aware it's already there, it's not just enabled unless you specify some
> > compiler flag)? Just because one language has feature X it doesn't mean
> > all the others should have it too.

I didn't mean to say that Ruby should take inspiration on other languages. My only intent was to show that symbols are not really required. But then I remembered that both C, C++ and Java support constant strings. In that sense Ruby could just optimize symbols to behave like frozen strings and try to optimize that just like the other languages optimize their constants.

> > Marshal is not portable across multiple languages (I use both Groovy
> > and Ruby in my overall application interacting with Redis). I'm
> > talking about JSON here. You don't have to find an alternative to
> > JSON. Just try to understand the issue I'm talking about.,
> > It wasn't suggested as an alternative, merely an example that there is a
> > way of serializing arbitrary Ruby data.

I know there are marshaling libraries in Ruby. I just don't understand how that is relevant to this ticket.

> > Ok, I won't repeat myself. Please give an example for the Redis +
> > JSON serialization use case presented in the ticket description.
> >
> > Otherwise you cleared missed the point.
> > Take a closer look at my previous Email, there's a fairly big example at
> > the bottom of it that you can't really miss.

I didn't miss it. I just didn't keep it in the reply because it is not relevant to my use case.

> However, just in case:

```
require 'redis'
require 'json'

client = Redis.new

client.set('user', JSON({'id' =>  1, 'name' =>  'John Doe'}))
```

```
# This would happen somewhere else (e.g. an external process)
hash = JSON(client.get('user'))
user = User.new(hash)

# instead of user['id'] you can now just do `user.id` which means
# that if the key name ever changes you only have to change it in
# one place.
if user.id
  # ...
end
```

Ok, you missed the point. Let me show you a complete example as I think it will help you understanding my concerns (I thought I made myself clear in the ticket description, but I hope this code example will help you understand what I meant):

```
require 'redis'
require 'json'
require 'sequel'

cache = Redis.new
users = cache['users'] || begin
db = Sequel.connect('postgres://user:password@localhost/mydb')
cache['users'] = db[:users].select(:id, :name).map{|r| id: r[:id],
name: r[:name]} # or just select(:id, :name).all
end

p users.first[:id]
```

What will be the output of the code above?

Exactly! It depends! If the results were cached it will print "nil", otherwise it will print "1" (or whatever the first id is).

This is the problem that symbols cause because they don't behave like strings.

Best,
Rodrigo.

#19 - 02/07/2013 09:59 AM - david_macmahon (David MacMahon)

Hi, Rodrigo,

FWIW, I sympathize with your symbols-vs-strings as keys frustration, but I think it's not so trivial to have the best of both worlds (flexibility and performance). Here is a simplification of your example:

```
require 'redis'
cache = Redis.new
value = :value
key = 'key'

v1 = cache[key] || (cache[key] = value)
v2 = cache[key] || (cache[key] = value)

p v1              # :value or "value"
p v2              # always "value"
p cache[key] == value  # always false
```

IMHO, the crux of the problem here is that Redis converts all Symbols to Strings, but assigning "v1 = cache[key] = value" does not store a String in v1 if value is a Symbol. This could be addressed by doing:

```
v1 = cache[key] || (cache[key] = value; cache[key])
```

which will store a String in v1 if value is a Symbol even if key does not yet exist in cache. Yes, it is an extra cache fetch.

The same kind of thing happens if value is a Fixnum. Surely you wouldn't want Fixnums and Strings to be the same thing! :-)

Thanks for starting this fascinating thread,
Dave

On Feb 6, 2013, at 3:45 PM, Rodrigo Rosenfeld Rosas wrote:

> Ok, you missed the point. Let me show you a complete example as I think it will help you understanding my concerns (I thought I made myself

> clear in the ticket description, but I hope this code example will help you understand what I meant):
>
> ```
> require 'redis'
> require 'json'
> require 'sequel'
>
> cache = Redis.new
> users = cache['users'] || begin
> db = Sequel.connect('postgres://user:password@localhost/mydb')
> cache['users'] = db[:users].select(:id, :name).map{|r| id: r[:id], name: r[:name]} # or just select(:id, :name).all
> end
>
> p users.first[:id]
> ```
>
> What will be the output of the code above?
>
> Exactly! It depends! If the results were cached it will print "nil", otherwise it will print "1" (or whatever the first id is).
>
> This is the problem that symbols cause because they don't behave like strings.
>
> Best,
> Rodrigo.

#20 - 02/07/2013 10:18 AM - phluid61 (Matthew Kerwin)

rosenfeld (Rodrigo Rosenfeld Rosas) wrote:

> drbrain (Eric Hodel), that means I'm unable to do things like
>
> ```
> results << {id: id, name: name}
> ```
>
> and have to use the old syntax all the time (and worrying about the differences all the time):
>
> ```
> results << {'id' => id, 'name' => 'name'}
> ```

You say "old syntax," maybe you should think of it as the "general syntax." Or, as I think of it, the "real syntax." The new {id: id} syntax is special case sugar for Hashes keyed on Symbols, which we've already determined is not a great thing to be doing in most cases.

> I still believe MRI should try to optimize frozen strings instead of symbols and just make symbols behave like frozen strings. :symbol would be a shortcut to 'symbol'.freeze.

But symbols *aren't* strings. You can't do any of the string-specific things on them, except by first casting them to strings. Why create a string, then use it to generate an (unrelated) type of object? Just create the symbol in the first place (as already happens).

> I don't really think any of those micro-benchmarks would justify all the hassle and unhappiness that symbols bring to Ruby programmers.

A simple solution presents itself: stop using them. Or I should say, stop using them except when you absolutely must.

If we stop thinking of them as special/frozen/whatever strings, and stop using them as "strings with construction optimisation," this whole issue becomes irrelevant. Then we can also stop using the symbol-specific {a: a} syntax, except when it really makes sense. And voila! everything is gravy.

Earlier you also wrote:

> You can't know in advance if a hash returned by some external code is indexed by string or symbols. You have to test by yourself or check the documentation.

Indeed. Similarly you can't know in advanced if it's keyed on integers, or CustomMagicClass instances. Not a symbol-specific issue. If the library authors cleared up their misconception that symbols are magic strings, you wouldn't have a problem. Solution: work around their shortcomings, or find/write a better library.

#21 - 02/07/2013 07:46 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

phluid61 (Matthew Kerwin) wrote:

> You say "old syntax," maybe you should think of it as the "general syntax." Or, as I think of it, the "real syntax." The new {id: id} syntax is special case sugar for Hashes keyed on Symbols, which we've already determined is not a great thing to be doing in most cases.

I agree that a string is what I want in all cases. That is exactly why I don't feel the need for symbols. If symbols are just really required as a fundamental implementation detail of the MRI implementation, then I don't think it is a good reason to justify keeping them in the language level. Just find other ways to optimize methods/etc lookup in the internal MRI code. This should be a separate discussion from the language design itself.

I'd really prefer you to think if symbols are really a good thing to have in the design of the Ruby language if you forget about all performance impacts it might have on the MRI implementation details. Then, still forgetting about performance and internal implementation details, try to reason why :symbol != 'symbol' is useful in Ruby just like a[:a] != a['a']. I've been using Ruby for several years now and I can tell you for sure that people often want them to behave the same and they don't want to worry about performance impact either. People just don't know when to use symbols and strings.

Take the Sequel library for instance.

DB[:user].select(:id, :name).all will return [{id: 1, name: 'Rodrigo'}] as opposed to [{'id' => 1}, {'name' => 'Rodrigo'}]

A similar case happens all around and people simply are confused whether they should be using symbols or strings in their code. This is specially misleading in hashes, specially because it is not uncommon that you can intermix strings and symbols as hash keys causing frequent bugs. It doesn't make sense to argue vs String vs Integer or Symbol vs Integer because this kind of bug doesn't really happen. People don't confuse Integer with Strings or Symbols. They confuse Symbols with Strings only. And each library will use a different criteria to decide if symbols or strings should be used and that forces each programmer to worry about those differences between libraries.

> I still believe MRI should try to optimize frozen strings instead of symbols and just make symbols behave like frozen strings. :symbol would be a shortcut to 'symbol'.freeze.

But symbols *aren't* strings.

I know they aren't. That is why I'm asking to change this behavior!

> You can't do any of the string-specific things on them, except by first casting them to strings.

We all know how symbols are different from strings, it doesn't help repeating it all the way. I'd prefer that you focus on explaining why you think keeping symbols a separate beast is of any usefulness (ignore any performance concerns for now, I'd like to make sure performance is the only factor involved here first).

> Why create a string, then use it to generate an (unrelated) type of object? Just create the symbol in the first place (as already happens).

This is what I do but I don't control other libraries. Anyway this makes the new sexy hash syntax almost unuseful to me since strings is what I want most of the times. And I really do hate the general syntax for hashes. The new one is more compact and takes me much less type to type and it is also similar to what most languages do (JavaScript, Groovy, etc). The difference is that in the other languages a string is used since they don't have the symbols concept.

#22 - 02/07/2013 07:50 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Also, so that you stop arguing that the differences between symbols and strings are just like the differences between strings and integers (non-sense), notice that HashWithIndifferentAccess makes this distinction:

```
h = HashWithIndifferentAccess.new({1 => 'a', a: 'a'})
h['1'] == nil
h[1] == 'a'
h['a'] == 'a'
h[:a] == 'a'
```

Since you don't see any popular hash implementation that will consider h[1] == h'1', you could take the conclusion that people only really care that string behave differently than symbols.

#23 - 02/07/2013 07:59 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

rosenfeld (Rodrigo Rosenfeld Rosas) wrote:

```
cache = Redis.new
users = cache['users'] || begin
db = Sequel.connect('postgres://user:password@localhost/mydb')
cache['users'] = db[:users].select(:id, :name).map{|r| id: r[:id],
name: r[:name]} # or just select(:id, :name).all
```

Sorry, I forgot the JSON conversion in the example above:

```
cache = Redis.new
db = Sequel.connect('postgres://user:password@localhost/mydb')
users = if cached = cache['users']
JSON.parse cached
else
```

db[:users].select(:id, :name).all.tap{|u| cache['users'] = JSON.unparse u}
end

I know the code above could be simplified, but I'm avoiding the parse -> unparse operation when it is not required (although it would make the code always work in this case):

users = JSON.parse (cache['users'] ||= JSON.unparse db[:users].select(:id, :name).all)

#24 - 02/07/2013 08:05 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

david_macmahon (David MacMahon) wrote:

> Hi, Rodrigo,
>
> FWIW, I sympathize with your symbols-vs-strings as keys frustration, but I think it's not so trivial to have the best of both worlds (flexibility and performance).

I'm not asking for performance nor flexibility. The only reason I didn't suggest to simply remove symbols at all is because I know it would be promptly rejected since all Ruby code would have to be changed to accomplish a change like that. And I don't really believe that symbols help the overall performance of any Ruby program. I feel it is more likely to reduce performance because people are often using conversions between symbols and strings making the overall performance slower, not faster.

> ...
>
> IMHO, the crux of the problem here is that Redis converts all Symbols to Strings...

Sorry, my example was wrong, I forgot about the JSON serialization. I don't really know what the Redis library does if I try to store a non-string object so I never tried to do so. I'm trying to avoid an unnecessary JSON.parse operation when I don't need it. See my comments above.

#25 - 02/07/2013 09:23 PM - phluid61 (Matthew Kerwin)

On 7 February 2013 20:46, rosenfeld (Rodrigo Rosenfeld Rosas) wrote:

> I agree that a string is what I want in all cases. That is exactly why I don't feel the need for symbols. If symbols are just really required as a fundamental implementation detail of the MRI implementation, then I don't think it is a good reason to justify keeping them in the language level. Just find other ways to optimize methods/etc lookup in the internal MRI code. This should be a separate discussion from the language design itself.
>
> I'd really prefer you to think if symbols are really a good thing to have in the design of the Ruby language if you forget about all performance impacts it might have on the MRI implementation details.

Ok, methods. They have a bucket of queriable information (a Method instance), and they have a symbolic representation (a Symbol). I don't want to have to instantiate an entire Method object (or a whole bunch of them) every time I want to talk to an object abouts its methods; I just want a single, simple, universal token that represents that (or those) method(s).

Sorry, that's a performance optimisation detail. Ok, I don't want to have to instantiate a Method object that potentially doesn't have a corresponding method. That could be confusing.

You will now argue that I could as easily use a String as a Symbol, and yes, ignoring performance and implementation details that is true. But I don't want to write code that performs poorly. If, in thise case, exposing implementation details make my code easier and better, then by the gods, let me use it. It is then up to me not to misuse it. Similarly: why have any numeric class that isn't Rational?

And for the record: "I don't ever want to use ClassX so let's remove it" is, frankly, silly.

> Then, still forgetting about performance and internal implementation details, try to reason why :symbol != 'symbol' is useful in Ruby just like a[:a] != a['a']. I've been using Ruby for several years now and I can tell you for sure that people often want them to behave the same and they don't want to worry about performance impact either.

Ok, completely philosphically, without any reference to performance or implementation details, why is a Java enum not equivalent to (or auto-cast to and from) a Java string? An enum is just a token, yes? It looks like a string; it often spells a word that people might like to read, even capitalise. But it's not a string. It's something else. A Symbol is exactly like that enum.

I, too, have been using Ruby for several years now; and I, too, have seen a lot of people wanting Symbol and String to behave the same. Hells, at times even I have wanted that. But the simple fact is: those people (myself included) are wrong. If they want a String, use a String. If they want to force a Symbol-shaped peg into a String-shaped hole, then they'll have to do whatever hoop-jumping is required; exactly as if you want a Java enum to support implicit coercion to and from a string.

> People just don't know when to use symbols and strings.

Bingo. Your solution is: hide Symbols from those people. My solution is: don't change anything; maybe eventually enough people will learn that the two classes are, in fact, different.

> Take the Sequel library for instance.

No thanks, apparently the authors don't know the difference between Symbols and Strings.

> We all know how symbols are different from strings,

Well apparently not, otherwise this would be a non-issue.

> it doesn't help repeating it all the way.

Perhaps I believe that if I say it enough times, in enough places, people might actually notice. And maybe even listen.

> I'd prefer that you focus on explaining why you think keeping symbols a separate beast is of any usefulness

I'll choose to interpret that as "... why I think keeping symbols at all ...". Simply: because they're already here. Relegating them to an implementation detail and hiding them from the language will only break 100% of existing code. Some of that code is good code. Is it worth breaking everything so a bunch of people can't accidentally use ClassX when they should be using ClassY?

-- I'll inject this later comment here, because it's topical:

> Also, so that you stop arguing that the differences between symbols and strings are just like the differences between strings and integers (non-sense), notice that HashWithIndifferentAccess makes this distinction:
> [...]
> Since you don't see any popular hash implementation that will consider h[1] == h'1', you could take the conclusion that
> people only really care that string behave differently than symbols.

Yes, but those are people who don't know the difference between Symbols and Strings. Just because they don't know it, doesn't make it untrue. Personally I've never used HashWithIndifferentAccess, or needed to.

Incidentally those people don't want a Hash at all. They want an associative array, one that uses something like <=> or === to compare keys (instead of #hash and #eql?). If RBTree was more mature, HashWithIndifferentAccess wouldn't be needed. Shall we repeat this discussion, but this time about Hash and assoc.array instead of Symbol and String?
--

> This is what I do but I don't control other libraries.

This is true of any issue in a library. If you think the library's benefits outweigh its costs, then you use the library. If the fact that the authors erroneously conflate Symbols and Strings is outweighed by the fact that it's otherwise extremely useful, it's up to you to work around the shortcomings. Just like if some otherwise brilliant library uses 0 instead of nil, or something.

> Anyway this makes the new sexy hash syntax almost unuseful to me since strings is what I want most of the times.

So, like I said before, just don't use it.

> And I really do hate the general syntax for hashes. The new one is more compact and takes me much less type to type and it is also similar to what most languages do (JavaScript, Groovy, etc).

The general syntax served us well enough through 1.8 and 1.9. Personally I preferred being able to use : at the end of if/when/etc. statements. If I want to use javascript syntax, there's always node.js

> The difference is that in the other languages a string is used since they don't have the symbols concept.

That's a good point. I'd love to be able to change the new syntax so {a:1} meant {'a'=>1}, but that's not going to happen. As such, in your eyes and mine, the new syntax is useless for most intents and purposes, so we might as well keep on writing Ruby code the way we always have (with => tokens).

P.S. sorry to anyone else who is sick of this conversation, but I think it needs to be had. Let us know if we should take it offline somewhere.

#26 - 02/07/2013 10:23 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Em 07-02-2013 10:04, Matthew Kerwin escreveu:

> On 7 February 2013 20:46, rosenfeld (Rodrigo Rosenfeld Rosas) wrote:
>
> > I agree that a string is what I want in all cases. That is exactly why I don't feel the need for symbols. If symbols are just really required as a fundamental implementation detail of the MRI implementation, then I don't think it is a good reason to justify keeping them in the language level. Just find other ways to optimize methods/etc lookup in the internal MRI code. This should be a separate discussion from the language
> > design itself.
> >
> > I'd really prefer you to think if symbols are really a good thing to have in the design of the Ruby language if you forget about all performance impacts it might have on the MRI implementation details.
>
> Ok, methods. They have a bucket of queriable information (a Method instance), and they have a symbolic representation (a Symbol). I don't want to have to instantiate an entire Method object (or a whole bunch of them) every time I want to talk to an object abouts its methods; I just want a single, simple, universal token that represents that (or those) method(s).

Like a string?

> Sorry, that's a performance optimisation detail.

Before I continue with my arguments and could focus solely on the performance issue I'd like to confirm that there is no other reason why symbols do exist. If performance is the sole reason and if I can make the point that symbols actually degrades most Ruby programs rather than improve the overall performance then I can still maintain my hopes for this ticket.

> ...
> And for the record: "I don't ever want to use ClassX so let's remove it" is, frankly, silly.

This is clearly not the reason here. That's why I'm asking: what Symbols are useful for? Is it performance the only reason why symbols exist?

Symbols cause lots of confusion as I showed in previous examples. That's why I want to remove it, but I didn't ask to remove it in this ticket anyway. Just to make it behave exactly like strings.

> ...Ok, completely philosphically, without any reference to performance or implementation details, why is a Java enum not equivalent to (or auto-cast to and from) a Java string?

Java, C and C++ have different goals then Ruby. They aim at the best possible performance given their constraints. They are also statically typed. Enums have two goals in such languages. Improving performance and reducing memory footprint is one of them. The other one is to help the compiler to find errors at compile time by restricting the input type in some functions/methods and variables. I don't really understand how this is relevant to this discussion.

> I, too, have been using Ruby for several years now; and I, too, have seen a lot of people wanting Symbol and String to behave the same.
>
> Hells, at times even I have wanted that. But the simple fact is:
>
> those people (myself included) are wrong. If they want a String, use a String. If they want to force a Symbol-shaped peg into a String-shaped hole, then they'll have to do whatever hoop-jumping is required; exactly as if you want a Java enum to support implicit coercion to and from a string.

I don't want that for Java enums and I don't really understand how Java enums relate to the string vs symbols debate in Ruby.

> > People just don't know when to use symbols and strings.
>
> Bingo. Your solution is: hide Symbols from those people.

Yes!

> My solution is: don't change anything; maybe eventually enough people will learn that the two classes are, in fact, different.

They won't.

> > Take the Sequel library for instance.
>
> No thanks, apparently the authors don't know the difference between Symbols and Strings.

But I really love the library. Should I really stop using it just because it returns an array of hashes indexed by symbols? And Sequel is not the only library doing so. Should I stop using all gems out there because the authors don't understand they should be always using strings instead of symbols in such cases?

You should ask yourself: why are authors so confusing about whether to use strings or symbols? Are they all just stupid? Isn't it clear in all Ruby books? No, it isn't! It is just really confusing. I'm yet to read some book that does a strong argument whether you should be using symbols or strings. They just say that symbols perform better than strings so authors think: "hey, then I'll just use symbols everywhere and my gems will perform the best possible way!". But this thinking is plain wrong because you'll need extra steps for conversions among those

types very often. The fact is that most authors don't really care about symbols or strings at all. They don't spend their time thinking about whether they should be using symbols or strings. They don't WANT to worry about it! And they're not wrong! Since they don't want someone looking at their code and telling them that their gem could perform better if they used symbols instead of strings they will just use symbols everywhere! This reality won't change. That is why I think programmers shouldn't have to worry about any performance difference that might exist between using symbols or strings in their code.

If there is a real boost using symbols internally in MRI then this should be an implementation detail only, not exposed to Ruby programs. That is why I suggested the optimizations to care if the string is frozen or not (like other compilers will optimize constants) instead of creating a new concept (symbols) just for that. They could keep the :symbol syntax as an alias to 'symbol'.freeze.

> ...
>
> > I'd prefer that you focus on explaining why you think keeping symbols a separate beast is of any usefulness
>
> I'll choose to interpret that as "... why I think keeping symbols at all ...". Simply: because they're already here.

This is not a good argument in my opinion. If you want to keep the syntax :name as an alias to 'name'.freeze I believe most current Ruby programs wouldn't be affected by such change.

> ... Personally I've never used HashWithIndifferentAccess, or needed to.

Me neither. But for different reasons. I need the behavior but I don't think it worths the extra dependencies in my code (ActiveSupport) nor the cumbersome of writing such a big class name everytime I want my hash to behave like HWIA. I prefer to take some time to investigate if all my hash keys are really strings than to just instantiate HWIA all over the places.

> Incidentally those people don't want a Hash at all. They want an associative array, one that uses something like <=> or === to compare keys (instead of #hash and #eql?).

:a === 'a' is not true so I don't understand how such suggested associative array would help here.

> ...
> This is true of any issue in a library. If you think the library's benefits outweigh its costs, then you use the library. If the fact that the authors erroneously conflate Symbols and Strings is outweighed by the fact that it's otherwise extremely useful, it's up to you to work around the shortcomings. Just like if some otherwise brilliant library uses 0 instead of nil, or something.

Again, please don't pretend that the confusion between strings and symbols are similar to confusions between 0 and nil.

> The general syntax served us well enough through 1.8 and 1.9.

Actually I never liked writing hashes as {key => value} in all years I've been working with Ruby. But I won't stop using Ruby just because I don't like its hash declaration syntax just the way I won't replace Sequel just because they return hashes indexed by symbols instead of strings.

> ...
>
> > The difference is that in the other languages a string is used since they don't have the symbols concept.

> That's a good point. I'd love to be able to change the new syntax so {a:1} meant {'a'=>1}, but that's not going to happen.

I agree it is unlike to happen. What about another syntax: {{a: 1}} => {'a' => 1}? Maybe it would worth trying to ask for some syntax change like this one. We could even add interpolation to it: {{"value #{computed}": 1}}.

#27 - 02/08/2013 02:01 AM - jeremyevans0 (Jeremy Evans)

phluid61 (Matthew Kerwin) wrote:

> > Take the Sequel library for instance.
>
> No thanks, apparently the authors don't know the difference between Symbols and Strings.

Sequel uses symbol keys instead of string keys intentionally. Sequel maps SQL features directly to ruby objects, mapping SQL identifiers (columns/tables/aliases) to ruby symbols and SQL strings to ruby strings. SQL query results can be thought of as a mapping of SQL identifiers to the values for each identifier in the query, thus Sequel uses a hash with symbol keys.

Ruby uses symbols in a very similar way to how SQL uses identifiers, with symbols basically acting as an identifier. The fact that ruby uses symbols as identifiers should be obvious to anyone who has looked at MRI's implementation, where a symbol is simply an alternate representation of an ID (the internal identifier type that MRI uses to map names to values):

```
#define ID2SYM(x) (((VALUE)(x)<<RUBY_SPECIAL_SHIFT)|SYMBOL_FLAG)
#define SYM2ID(x) RSHIFT((unsigned long)(x),RUBY_SPECIAL_SHIFT)
```

The basic philosophical difference between a symbol and a string is a ruby symbol is an identifier, while a ruby string just represents arbitrary data. You should convert a string to a symbol if you know that the data the string contains represents an identifier you would like to use. You should convert a symbol to a string if you are using name of the identifier as data.

#28 - 02/08/2013 02:53 AM - david_macmahon (David MacMahon)

On Feb 7, 2013, at 3:00 AM, rosenfeld (Rodrigo Rosenfeld Rosas) wrote:

> Issue #7792 has been updated by rosenfeld (Rodrigo Rosenfeld Rosas).
>
> rosenfeld (Rodrigo Rosenfeld Rosas) wrote:
>
> > cache = Redis.new
> > users = cache['users'] || begin
> > db = Sequel.connect('postgres://user:password@localhost/mydb')
> > cache['users'] = db[:users].select(:id, :name).map{|r| id: r[:id],
> > name: r[:name]} # or just select(:id, :name).all
>
> Sorry, I forgot the JSON conversion in the example above:
>
> ```
> cache = Redis.new
> db = Sequel.connect('postgres://user:password@localhost/mydb')
> users = if cached = cache['users']
> JSON.parse cached
> else
> db[:users].select(:id, :name).all.tap{|u| cache['users'] = JSON.unparse u}
> end
> ```

I still think the fundamental issue is that Sequel is returning something that contains symbols, but JSON.parse(JSON.unparse(x)) will never return anything containing a symbol even if x does. Because of this the "users" variable is assigned something that contains symbols in one case and something that contains no symbols in all other cases. IMHO, the example shows not a deficiency in the Ruby language itself but rather a (subtle, easy to fall victim to) coding pitfall of not properly managing/reconciling the incompatible return types from the two libraries.

> I know the code above could be simplified, but I'm avoid the parse -> unparse operation when it is not required (although it would make the code always work in this case):
>
> users = JSON.parse (cache['users'] ||= JSON.unparse db[:users].select(:id, :name).all)

This seems an excellent solution IMHO as it clearly results in "users" being assigned the return value of JSON.parse every time and only results in one extra JSON.parse call only on the first time through. That seems a pretty small price to pay for reconciling the two libraries' incompatible return

return types.

Dave

#29 - 02/08/2013 03:23 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

david_macmahon (David MacMahon) wrote:

> I still think the fundamental issue is that Sequel is returning something that contains symbols, but JSON.parse(JSON.unparse(x)) will never return anything containing a symbol even if x does. Because of this the "users" variable is assigned something that contains symbols in one case and something that contains no symbols in all other cases. IMHO, the example shows not a deficiency in the Ruby language itself but rather a (subtle, easy to fall victim to) coding pitfall of not properly managing/reconciling the incompatible return types from the two libraries.

But don't you agree that we wouldn't have this kind of problem if Symbols behave just like Strings? Or if Symbols didn't exist at all? Or even if hashes always behaved as HWIA? This is what I'm talking about. But the biggest problem is that I can't really see any real advantage on symbols behaving differently than strings, except maybe for internal usage of symbols.

> > I know the code above could be simplified, but I'm avoid the parse -> unparse operation when it is not required (although it would make the code always work in this case):
> >
> > users = JSON.parse (cache['users'] ||= JSON.unparse db[:users].select(:id, :name).all)
>
> This seems an excellent solution IMHO as it clearly results in "users" being assigned the return value of JSON.parse every time and only results in one extra JSON.parse call only on the first time through. That seems a pretty small price to pay for reconciling the two libraries' incompatible return return types.

You seem to be underestimating how slow JSON.parse can be on big lists. I've spent a lot of time in the past doing benchmarks for several JSON parsers implementations available for Ruby because most of the time spent on some requests were caused by JSON#unparse. JSON#unparse was the bottleneck of some requests. I don't think JSON#parse would be much faster. An extra parse operation on a big list could easily add 100ms to the request timing. That is why I want to avoid it.

Also, it doesn't happen only the first time but on each request.

#30 - 02/08/2013 03:53 AM - david_macmahon (David MacMahon)

On Feb 7, 2013, at 10:23 AM, rosenfeld (Rodrigo Rosenfeld Rosas) wrote:

> Issue #7792 has been updated by rosenfeld (Rodrigo Rosenfeld Rosas).
>
> david_macmahon (David MacMahon) wrote:
>
> > I still think the fundamental issue is that Sequel is returning something that contains symbols, but JSON.parse(JSON.unparse(x)) will never return anything containing a symbol even if x does. Because of this the "users" variable is assigned something that contains symbols in one case and something that contains no symbols in all other cases. IMHO, the example shows not a deficiency in the Ruby language itself but rather a (subtle, easy to fall victim to) coding pitfall of not properly managing/reconciling the incompatible return types from the two libraries.
>
> But don't you agree that we wouldn't have this kind of problem if Symbols behave just like Strings? Or if Symbols didn't exist at all? Or even if hashes always behaved as HWIA?

Yes, I agree that we wouldn't have this kind of problem if any of those alternatives existed. I'm just not (yet) convinced that any of those alternatives are desirable just to avoid this kind of problem. I do somewhat like the idea of having Hash behave as HWIA. I think the number of real world uses of something like:

{:a => 0, 'a' => 1}

is likely to be very small.

> > I know the code above could be simplified, but I'm avoid the parse -> unparse operation when it is not required (although it would make the code always work in this case):
> >
> > users = JSON.parse (cache['users'] ||= JSON.unparse db[:users].select(:id, :name).all)
>
> This seems an excellent solution IMHO as it clearly results in "users" being assigned the return value of JSON.parse every time and only results in one extra JSON.parse call only on the first time through. That seems a pretty small price to pay for reconciling the two libraries' incompatible return return types.
>
> An extra parse operation on a big list could easily add 100ms to the request timing.

So use explicit initialization instead of lazy initialization.

> Also, it doesn't happen only the first time but on each request.

In the original example, JSON.parse was used on every call except the first one. The modified example uses JSON.parse on every call including the first one. That's why I said the modified example has only one extra JSON.parse call (i.e. the one extra one on the first call). If that's too much overhead, either don't use lazy initialization or explicitly invoke the method once at startup to force lazy initialization so it doesn't impact the (un?)lucky first user.

Dave

#31 - 02/08/2013 03:58 AM - trans (Thomas Sawyer)

I think the best thing to do about this is simply to ask that a HWIA class be added to Ruby's core. That's as far as this issue can possibly go at this time. And be very happy if it happens b/c 1) its hard to get such things to happen and 2) it's a nice big leap in the right direction. Asking for Hash to become a HWIA, is simply asking for too much backward compatibility breakage for a non-sufficient reason. Keeping on about it will do no good and simply fall on deaf ears, and rather then encourage the inclusion of a HWIA, may well do the opposite.

#32 - 02/08/2013 04:23 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Em 07-02-2013 16:43, David MacMahon escreveu:

> > ...An extra parse operation on a big list could easily add 100ms to the request timing.
> > So use explicit initialization instead of lazy initialization.

Sorry, didn't get. Could you please show some sample code?

> > Also, it doesn't happen only the first time but on each request.
> > In the original example, JSON.parse was used on every call except the first one. The modified example uses JSON.parse on every call including the first one. That's why I said the modified example has only one extra JSON.parse call (i.e. the one extra one on the first call). If that's too much overhead, either don't use lazy initialization or explicitly invoke the method once at startup to force lazy initialization so it doesn't impact the (un?)lucky first user.

I see the confusion. I simplified the code in that example. Here is how
it would look in a real Rails controller:

class MyController

```
def my_action
   @users = if cached = CacheStore.fetch('users')
     JSON.parse cached
   else
     DB[:users].select(:id, :name).all.tap{|u| CacheStore.store
```

'users', JSON.unparse u}
end
end
end

Of course I don't cache the users list, this is just a small example.
The real query is much more complex and could take up to a second when
lots (20) of fields are searched using the user interface. Usually the
query would take about 100ms or less (up to 5 fields usually) but then
the user may want to print the results or export to Excel or changing to
another page and caching would help a lot in that case.

#33 - 02/08/2013 04:28 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Thomas, if you think we could get some constructor for HWIA, like say, {{a: 'a'}}, then it would already help a lot. But I don't see typing such a long name (HashWithIndifferentAccess) whenever I need such behavior being it included on core or not.

#34 - 02/08/2013 04:29 AM - david_macmahon (David MacMahon)

On Feb 7, 2013, at 11:04 AM, Rodrigo Rosenfeld Rosas wrote:

> Em 07-02-2013 16:43, David MacMahon escreveu:
>
> > ...An extra parse operation on a big list could easily add 100ms to the request timing.
> > So use explicit initialization instead of lazy initialization.

> Sorry, didn't get. Could you please show some sample code?

I was referring to the example, but now I see that the real-world variation is not simply lazy initialization, but rather caching of a result derived from user input that may or may not be used again. In the latter case, the result cannot be pre-initialized and an extra JSON.parse would be significant.

Sorry for veering off topic,
Dave

#35 - 02/08/2013 04:55 AM - trans (Thomas Sawyer)

rosenfeld (Rodrigo Rosenfeld Rosas) Well I would hope for a much shorter name myself. e.g. Map, Index, Dict are some viable candidates.

#36 - 02/08/2013 05:08 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

I could see myself using "Map(a: 1, b: 2)" instead of "{a: 1, b: 2}" when I want HWIA. Sounds good to you? Should I create a ticket like this one?

#37 - 02/08/2013 06:23 AM - phluid61 (Matthew Kerwin)

On 7 February 2013 23:09, Rodrigo Rosenfeld Rosas wrote:

> Enums have two goals in such languages. Improving performance and reducing memory footprint is one of them. The other one is to help the compiler to find errors at compile time by restricting the input type in some functions/methods and variables. I don't really understand how this is relevant to this discussion.

No, no no no. Enums exist because they are identifiers. They are symbolic representations of a concept that either does not necessarily otherwise exist in the code, or does not have to be fully instantiated in order to discuss it. That is exactly what Symbols are.

> They don't spend their time thinking about whether they should be using symbols or strings. They don't WANT to worry about it!

Your overarching goal is to coddle developers who write code without understanding either the language, or the concepts behind their own code. There is a massive philosophical disjunct here between you and I, and I think it will never be overcome.

> I agree it is unlike to happen. What about another syntax: {{a: 1}} => {'a' => 1}? Maybe it would worth trying to ask for some syntax change like this one. We could even add interpolation to it: {{"value #{computed}": 1}}.

You'd probably be more likely to succeed with a new %string-style notation, like %h{a:1, b:2}. Although then again, possibly not.

#38 - 02/08/2013 06:23 AM - phluid61 (Matthew Kerwin)

On 8 February 2013 03:01, jeremyevans0 (Jeremy Evans) <merch-redmine@jeremyevans.net> wrote:

> phluid61 (Matthew Kerwin) wrote:
>
> > Take the Sequel library for instance.
>
> No thanks, apparently the authors don't know the difference between Symbols
> and Strings.

> Sequel uses symbol keys instead of string keys intentionally. Sequel maps SQL features directly to ruby objects, mapping SQL identifiers (columns/tables/aliases) to ruby symbols and SQL strings to ruby strings. SQL query results can be thought of as a mapping of SQL identifiers to the values for each identifier in the query, thus Sequel uses a hash with symbol keys.

Sorry, I was being glib.

Interestingly, this is actually an example of Symbols being used correctly
(or at least, not just out-right incorrectly). It's a pity people see this
is a problem to be worked around, rather than a feature.

#39 - 02/08/2013 07:23 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Em 07-02-2013 19:11, Matthew Kerwin escreveu:

> On 7 February 2013 23:09, Rodrigo Rosenfeld Rosas wrote:
>
> > Enums have two goals in such languages. Improving performance and
> > reducing memory footprint is one of them. The other one is to help the
> > compiler to find errors at compile time by restricting the input type in
> > some functions/methods and variables. I don't really understand how this
> > is relevant to this discussion.
>
> No, no no no. Enums exist because they are identifiers. They are
> symbolic representations of a concept that either does not necessarily
> otherwise exist in the code, or does not have to be fully instantiated
> in order to discuss it. That is exactly what Symbols are.

If you really believe symbols are similar to enums I guess you haven't
done much C, C++ or Java programming and used enums. Here is the main
reason why enums exist. First let me notice that C and Java implement
enums in different ways:

C example:

```
typedef enum {HEAD, TAIL} coin_side;
coin_side my_coin = HEAD; // my_coin = 0 would also work here

if you try to create another typedef like this, the compiler will
complain that HEAD is already declared:
typedef enum {HEAD, TAIL} alternate_coin_side;

if you do something like:

typedef enum {PAPER, ROCK, SCISORS} game;
coin_side my_coin = PAPER;
```

The C compiler won't complain. But Java takes a different approach when
it comes to enums:

```
class MyClass {
enum Game {PAPER, ROCK, SCISORS};
enum CoinSide {HEAD, TAIL};
void test(){
Game a = Game.PAPER;
Game b = CoinSide.HEAD; // won't compile!
}
}
```

In that sense, if you write a method accepting a coin side you can only
pass in a CoinSize enum. Not a number. Not another enum type. This is
what I understand by enums. Not symbols related at all in my opinion.

> > They don't spend their time thinking about whether they should be using
> > symbols or strings. They don't WANT to worry about it!
>
> Your overarching goal is to coddle developers who write code without
> understanding either the language, or the concepts behind their own
> code. There is a massive philosophical disjunct here between you and
> I, and I think it will never be overcome.

It is a matter of choosing the right tool. If you're really concerned
about some really small performance improvements you might get by using
symbols instead of strings I would question if Ruby is really the right
language for you.

I'd never consider Ruby or Java to write hard-real-time applications the same way I wouldn't consider Windows or plain Linux for such task. I'd most likely use C and Linux + Xenomai patch (if building for desktops) instead.

When I'm writing Ruby code I don't want to worry about micro performance improvements. The minimal amount of time I would probably care in optimizing would be 100ms instead of micro-seconds when I'm writing C programs that must complete some complex tasks in very short times when writing real-time tasks.

> I agree it is unlike to happen. What about another syntax: {{a: 1}} => {'a' => 1}? Maybe it would worth trying to ask for some syntax change like this one. We could even add interpolation to it: {{"value #{computed}": 1}}.

> You'd probably be more likely to succeed with a new %string-style notation, like %h{a:1, b:2}. Although then again, possibly not.

That's an idea, yes, but I guess I prefer Thomas' suggestion of using Map(a: 1, b: 2).

#40 - 02/08/2013 07:30 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Em 07-02-2013 19:15, Matthew Kerwin escreveu:

> On 8 February 2013 03:01, jeremyevans0 (Jeremy Evans)
> >
> wrote:
>
> ```
> phluid61 (Matthew Kerwin) wrote:
> > > Take the Sequel library for instance.
> >
> >  No thanks, apparently the authors don't know the difference
> between Symbols
> >  and Strings.
>
> Sequel uses symbol keys instead of string keys intentionally.
>  Sequel maps SQL features directly to ruby objects, mapping SQL
> identifiers (columns/tables/aliases) to ruby symbols and SQL
> strings to ruby strings. SQL query results can be thought of as a
> mapping of SQL identifiers to the values for each identifier in
> the query, thus Sequel uses a hash with symbol keys.
> ```
>
> Sorry, I was being glib.
>
> Interestingly, this is actually an example of Symbols being used correctly (or at least, not just out-right incorrectly). It's a pity people see this is a problem to be worked around, rather than a feature.

I've never complained about Sequel returning symbols. I'm complaining about symbols not behaving just like regular strings when they're used as keys in hashes among other usage examples. That's why I never asked Jeremy to change Sequel's current behavior.

#41 - 02/08/2013 08:23 AM - phluid61 (Matthew Kerwin)

Sent from my phone, so excuse the typos.
On Feb 8, 2013 8:50 AM, "Rodrigo Rosenfeld Rosas" rr.rosas@gmail.com wrote:

> If you really believe symbols are similar to enums I guess you haven't done much C, C++ or Java programming and used enums. Here is the main reason why enums exist.

Yes, the reason they are called 'enums' and not 'arbitrary identifiers' is because they define an explicit, finite, enumerated set of identifiers; as such an 'enum' is a set of things-that-are-like-symbols. But ´head´ is not, and should never be, equal to 0 or 1 or whatever. The fact that C reveals its implementation in this way is an artifact of C, not the concept of enums in general.

#42 - 02/08/2013 08:59 AM - Anonymous

Hi,

During early stage of 1.9 development, I tried to make symbols and strings behave same (at least similar), and it had broken too many programs. I understand your problem but it's not worth raising huge compatibility issues.

```
matz.
```

#43 - 02/08/2013 09:29 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Em 07-02-2013 21:58, Yukihiro Matsumoto escreveu:

> Hi,
>
> During early stage of 1.9 development, I tried to make symbols and strings behave same (at least similar), and it had broken too many programs. I understand your problem but it's not worth raising huge compatibility issues.
>
> ```
> matz.
> ```

Thank you for your feedback, Matz. I'd really never suspect such a change could break so many programs...

I'm really curious what kind of programs rely on symbols behaving differently from strings. But I guess you won't remember which programs would be them, will you?

#44 - 02/08/2013 10:23 AM - drbrain (Eric Hodel)

On Feb 7, 2013, at 10:43, David MacMahon davidm@astro.berkeley.edu wrote:

> I think the number of real world uses of something like:
>
> {:a => 0, 'a' => 1}
>
> is likely to be very small.

Every time you run gem you use a hash that contains both symbol and string keys used for different purposes.

In ~/.gemrc symbols are used for configuration options while strings are used for command default arguments.

#45 - 02/08/2013 10:23 AM - Anonymous

Hi,

In message "Re: [ruby-core:52017] Re: [ruby-trunk - Feature #7792] Make symbols and strings the same thing"
on Fri, 8 Feb 2013 09:26:38 +0900, Rodrigo Rosenfeld Rosas rr.rosas@gmail.com writes:

|Thank you for your feedback, Matz. I'd really never suspect such a
|change could break so many programs...
|
|I'm really curious what kind of programs rely on symbols behaving
|differently from strings. But I guess you won't remember which programs
|would be them, will you?

It was long long ago. But some programs distinguished symbols and strings as hash keys, for example.

Symbols are taken from Lisp symbols, and they has been totally different beast from strings. They are not nicer (and faster) representation of strings. But as Ruby stepped forward its own way, the difference between symbols and strings has been less recognized by users.

```
matz.
```

#46 - 02/08/2013 10:53 AM - david_macmahon (David MacMahon)

On Feb 7, 2013, at 5:15 PM, Eric Hodel wrote:

> On Feb 7, 2013, at 10:43, David MacMahon davidm@astro.berkeley.edu wrote:
>
> > I think the number of real world uses of something like:
> >
> > {:a => 0, 'a' => 1}
> >
> > is likely to be very small.
>
> Every time you run gem you use a hash that contains both symbol and string keys used for different purposes.
>
> In ~/.gemrc symbols are used for configuration options while strings are used for command default arguments.

Well, that's one! In my defense, one is a very small number even if that one is of very large consequence. :-)

Just to play Devil's advocate, could that not be separated into two different hashes: one for config options and one for command default arguments? You have given semantic meaning to (the class of) the key. Isn't that akin to treating the primary key of a database record in a non-opaque manner?

Dave

#47 - 02/08/2013 10:59 AM - spatulasnout (B Kelly)

Rodrigo Rosenfeld Rosas wrote:

> I'm really curious what kind of programs rely on symbols behaving
> differently from strings.

One example:

Since symbols aren't garbage collected, my Ruby-based RPC system
is designed by default to convert symbols in incoming messages to
strings instead. (PacketBuf.preserve_symbols_on_extract = false)

This has led to sending certain internal messages with names
consisting of symbols instead of strings. These messages cannot
be spoofed by an external source, as there's no way for a remote
sender to produce messages locally containing symbols.

(If not for the garbage collection problem, I would have designed
the system to preserve externally generated symbols, and so would
have then required a different approach to distinguish internal
vs. external message names. So current design is arbitrary in
that sense, but nevertheless, the symbol/string distinction is
being put to use.)

Regards,

Bill

#48 - 02/08/2013 04:23 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Em 07-02-2013 23:57, Bill Kelly escreveu:

> ...but nevertheless, the symbol/string distinction is being put to use.)

Thanks for letting me know, Bill.

#49 - 02/09/2013 09:53 AM - drbrain (Eric Hodel)

On Feb 7, 2013, at 17:37, David MacMahon davidm@astro.berkeley.edu wrote:

> Just to play Devil's advocate, could that not be separated into two different hashes: one for config options and one for command default arguments? You have given semantic meaning to (the class of) the key. Isn't that akin to treating the primary key of a database record in a non-opaque manner?

Two hashes would not be backwards compatible with previous versions of RubyGems. I'm hesitant to change the format of a configuration file and break downgrades between RubyGems versions.

It would be more confusing to new users. Now we can tell users "paste '…' into ~/.gemrc to do …" and it will work. They don't need to know anything about ruby or yaml to do this.

If matz decides that we should break backwards compatibility in this manner I will change my mind.

#50 - 02/12/2013 09:23 AM - david_macmahon (David MacMahon)

On Feb 8, 2013, at 4:50 PM, Eric Hodel wrote:

> On Feb 7, 2013, at 17:37, David MacMahon davidm@astro.berkeley.edu wrote:
>
> > Just to play Devil's advocate, could that not be separated into two different hashes: one for config options and one for command default arguments? You have given semantic meaning to (the class of) the key. Isn't that akin to treating the primary key of a database record in a non-opaque manner?
>
> Two hashes would not be backwards compatible with previous versions of RubyGems. I'm hesitant to change the format of a configuration file and break downgrades between RubyGems versions.
>
> It would be more confusing to new users. Now we can tell users "paste '…' into ~/.gemrc to do …" and it will work. They don't need to know anything about ruby or yaml to do this.
>
> If matz decides that we should break backwards compatibility in this manner I will change my mind.

I was not trying to induce change. I think my follow-on query would have been more appropriate for ruby-talk rather than ruby-core. I will (try to) keep my ruby-core postings more on topic.

Thanks for the explanation,
Dave

P.S. Sorry if this gets out more than once. I'm in the midst of an involuntary email "transition".

#51 - 03/21/2013 12:02 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Just another confusion caused by different behavior of strings vs symbols: https://github.com/blog/1440-today-s-email-incident

#52 - 03/21/2013 12:09 PM - Student (Nathan Zook)

The problem is not "confusion", it is that there is an apparent effort to rework core features in a massive framework in little pieces. These things don't fail in isolation--it is an error to attempt to fix them in isolation.

#53 - 03/21/2013 10:01 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

What I meant is that most people only use symbols as an identifier that could be a just a string as well. If you look at the code semanthic you'll notice that they don't really care about symbol vs strings differences. They just want an identifier but then they have to be concerned about their differences all over the code even when they don't want this because different pieces will have different opinions about if symbols or strings should be used for that case. Sometimes they only happen to use symbols because of the convenience of the new hash better looking syntax. I'm just trying to convince Matz to evaluate once again the possibility of making symbols behave like string.freeze! or simply like regular strings even if it would break backward compatibility... I still believe all the confusion created by their differences will keep happening and breaking compatibility for good in this case makes sense to me. Please reconsider this ticket...

#54 - 03/21/2013 10:04 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Also, as a nice side effect we wouldn't even have to worry about garbage collecting symbols to avoid DoS as they would be automatically collected just like strings. If MRI wants to keep the original notion of symbols as an internal implementation detail, that's fine, but this shouldn't leak to end users...

#55 - 03/26/2013 12:14 AM - Student (Nathan Zook)

Actually, rails trusts symbols at times in ways that it does not trust strings. This is the source of a recent security issue or two, as it was assumed that symbols creation would not be directed from outside data.

Better would have been to have run in safe mode 1, or to have watched the tainted property directly.

Without studying this most recent issue in detail, it looks like the rails devs have opted for rapid, "mostly right" instead of deeply-analysed, thoroughly-vetted solutions. Given the nature of some of these issues, that was almost certainly correct as an initial response.

#56 - 03/27/2013 05:46 AM - alexeymuranov (Alexey Muranov)

Student (Nathan Zook) wrote:

> Actually, rails trusts symbols at times in ways that it does not trust strings. This is the source of a recent security issue or two, as it was assumed that symbols creation would not be directed from outside data.

In Rails, the Hash#symbolize_keys! method looks somewhat unacceptable to me.

#57 - 03/30/2013 02:34 AM - Student (Nathan Zook)

Yep. If to_sym threw or nilled on previously unknown symbols, or at least on tainted data, this would be okay. Otherwise, it is a serious security problem.

#58 - 04/03/2013 07:06 PM - wardrop (Tom Wardrop)

=begin
= To Summarise...

== Symbol Differences

=== Immutable
A symbol is an alias for a numeric ID. The symbol representation of that numeric ID can never change.

=== Not garbage collected
An extension of the previous point. Garbage collecting symbol's would break the fact that symbol's are an alias for a specific numeric ID.

=== Semantically different
The type difference between a String and a Symbol can be used in program logic.

== The Problem
((*And yes, there is clearly a problem! You can argue the solution.*))

=== Inconsistance use
It's virtually impossible to reliably determine when to use a Symbol or a String. Today, Symbol might be the logical choice for scenario X, but tomorrow you're looking up those Symbols with user-entered data, or reading them in from a file, etc. Programs change, requirements change, everything changes.

=== Library incompatibility
As a consequence of the previous point, combined with different levels of programmer experience and plain personal opinion, libraries will always contain incompatibility.

=== Unignorable (((*It's a real word, I swear*)))
As a consequence of the previous two, even if you only use one or the other, unless you want to avoid using 3rd party code, stdlib, or even ruby-core, there's no way to ignore the fact that Symbol's are different and completely incompatible with String's. Pita.

== The Solution
Or more to the point, what criteria does a solution need to meet. I don't believe any solution can resolve all the problems, while maintaing full backwards compatibility. Therefore, I see there being a short-term solution, i.e. minor version release of 2.x, and a long term solution, i.e 3.0.

I think the Ruby project needs to maintain a list of desired backwards-incompatible changes, so these can be rolled up into one big 3.0 release. Ruby can't be backwards compatible forever, otherwise it'll slowly fade into insignificance... hey Microsoft.

=== Short Term
I don't think there is one. HashWithIndifferentAccess will not solve the problem with hash keys. You'll just go from coercing Symbols to String and vice versa, to coercing Hash to HashWithIndifferentAccess, and back again, and there's no way we can make the differences ignorable to those libraries that don't care, while still satisfying those libraries that do care. Anything we change will probably only help to dig us all a deeper hole.

The short term solution is to accept there's a problem, and that anything backwards-compatible isn't worth changing.

=== Long Term
I propose the remainder of this thread should focus on a long term solution that solves ALL the problems without worrying about full backwards-compatibility, and for everyone to accept that there is indeed a problem with the existence and/or current implementation of Strings and Symbols. Getting rid of Symbols, making Symbols a sub-class of String, whatever. Let's talk about that now, please. It's clear that even Matz is unsatisfied with the current situation.

=end

#59 - 04/03/2013 07:44 PM - phluid61 (Matthew Kerwin)

=begin
In that case, I suggest that the long term solution is education.

By the way: "((*A symbol is an alias for a numeric ID.*))" Not entirely true (otherwise we could argue for Symbol#to_i or even Symbol#to_int) A symbol represents an immutable concept, analogous to a number (i.e. 3 is always 3, even outside of Ruby; so :to_s is always :to_s). The numeric ID is an implementation detail.

Also, it's spelled "Summarise" with an 'a' (or "Summarize" in America.) ;)
=end

#60 - 04/03/2013 09:01 PM - wardrop (Tom Wardrop)

No that's spelt correctly. I sadly couldn't embed any pictures of palm trees and coconuts ;). Fixed, thanks.

#61 - 04/03/2013 10:29 PM - spatulasnout (B Kelly)

wardrop (Tom Wardrop) wrote:

> === Immutable
> A symbol is an alias for a numeric ID. The symbol representation of that
> numeric ID can never change.

As Matthew Kerwin noted, the numeric ID is an implementation detail; and indeed, the IDs already do change:

```
$ ruby -e "p [:foo.object_id, :bar.object_id]"
[143928, 144008]

$ ruby -e "p [:bar.object_id, :foo.object_id]"
[143928, 144008]
```

> === Not garbage collected
> An extension of the previous point. Garbage collecting symbol's would break
> the fact that symbol's are an alias for a specific numeric ID.

Presumably this could only affect programs which retain the object_id of a symbol past the point where the symbol itself is no longer referenced by the program, and which then expect to be able to compare that object_id with the object_id of the equivalent symbol after it has been GC'd and re-interned.

Assuming the rest of the GC-related problems have been solved, such as marking any symbols interned by ruby extensions as non-GC-able, then we should only need wonder how much pure ruby code exists that, during the lifetime of a given ruby interpreter process, is storing and comparing symbol object_ids rather than the symbols themselves.

If much such code indeed exists in production environments, then presumably symbol GC could be introduced as a VM startup option, eventually becoming the default after the legacy behavior has gone through a deprecation phase.

It's not evident to me that a solution to the symbol GC issue needs to be bundled with solution(s) for symbol/string equivalence issues, however.

Regards,

Bill

#62 - 04/04/2013 07:42 AM - wardrop (Tom Wardrop)

spatulasnout (B Kelly) Yes on your first point. One should always compare :symbol with :symbol rather than by their internal ID, so I'd say that any code using the internal ID's deserves to break. I imagine it would be hardly any, other than the interpreter itself.

So on that note, I think you're right. Garbage collecting symbol's could be implemented while maintaining backwards compatibility. That may address the technical implications such as the potential for DoS attacks. As you said, this could safely be implemented as a startup option, or potentially even a runtime option.

#63 - 07/15/2014 03:39 PM - Ajedi32 (Andrew M)

So now that Symbol GC is implemented (#9634), does that change anything about the viability of this proposal?

#64 - 08/08/2014 08:02 PM - eloyesp (Eloy Esp)

I think the proposal is goood, if the change is better to new programmers it means it is better for all programmers, (because eventually the old experienced ruby programmer will die) and we will be adapting the people to the code instead of the other way around (that is not a good practice).

As I understand, the proposed change can be read as:

- Currently there is an exception in the language for strings as hash keys (they are duped and frozen).
- This behaviour is unexpected for new rubysts and for not so new (I learned that yesterday, and I worked with ruby for 4+ years).
- Using strings as symbols for hash keys is a headache for many programmers. (there are a lot of libraries to make this easier).
- There is a lot of libraries written with the traditional way in mind. Thus all the gems that stores in the same hash strings and symbols could break if we change the exception.

Given the change will break things, but will make things better, why not plan it for ruby 3.0?

The only use-case that will break things is:

```
hash = {}
hash['key'] = value
hash[:key] = other_value
```

We can add a warning when you do something like this and make the change a year after that.

#65 - 02/16/2017 01:22 AM - subtileos (Daniel Ferreira)

Hi Bill

> One example:
>
> Since symbols aren't garbage collected, my Ruby-based RPC system
> is designed by default to convert symbols in incoming messages to
> strings instead. (PacketBuf.preserve_symbols_on_extract = false)
>
> This has led to sending certain internal messages with names
> consisting of symbols instead of strings. These messages cannot
> be spoofed by an external source, as there's no way for a remote
> sender to produce messages locally containing symbols.
>
> (If not for the garbage collection problem, I would have designed
> the system to preserve externally generated symbols, and so would
> have then required a different approach to distinguish internal
> vs. external message names. So current design is arbitrary in
> that sense, but nevertheless, the symbol/string distinction is
> being put to use.)

With symbols being garbage collected now is this functionality broken or are you still relying on symbols over strings for your solution?

Thanks,

Daniel

#66 - 02/16/2017 02:21 AM - spatulasnout (B Kelly)

Hi Daniel,

Daniel Ferreira wrote:

> > (If not for the garbage collection problem, I would have designed
> > the system to preserve externally generated symbols, and so would
> > have then required a different approach to distinguish internal
> > vs. external message names. So current design is arbitrary in
> > that sense, but nevertheless, the symbol/string distinction is
> > being put to use.)
>
> With symbols being garbage collected now is this functionality broken
> or are you still relying on symbols over strings for your solution?

I've continued using the aforementioned RPC system without enabling
the setting that would allow it to recreate symbols on the receiving
end.

So it continues to work as it did before (if the remote sends symbols,
they appear only as strings locally.)

I will say however that on the balance, in the time I've been using
this system, it can feel sub-optimal to have to keep track of which
parameters are symbols and which are strings. I'd say my primary
motivation for using producing symbols on the sending end is Ruby's
convenient hash syntax:

result = some.rpc_method(foo:123, bar:"baz")

I find that i like the syntax enough, that I'll put up with having
to think about when those symbols will have been turned into strings
down the line.

But it would be nice to not have to keep track of that detail.

Regards,

Bill

#67 - 02/16/2017 02:33 AM - subtileos (Daniel Ferreira)

> result = some.rpc_method(foo:123, bar:"baz")
>
> I find that i like the syntax enough, that I'll put up with having
> to think about when those symbols will have been turned into strings
> down the line.

I believe it is time now with the advent of Ruby 3 to revisit this feature request.
Although maybe not quite like it is stated right now.

What are your feelings about it?

#68 - 10/05/2017 06:19 PM - sheerun (Adam Stankiewicz)

shyouhei (Shyouhei Urabe)matz (Yukihiro Matsumoto) Could this be revisited as ruby now has frozen strings? They behave pretty much the same as symbols.

Here's an article worth read: http://blog.arkency.com/could-we-drop-symbols-from-ruby/

#69 - 10/07/2017 04:15 AM - shevegen (Robert A. Heiler)

I found the old issue here too from the blog article. :)

However had, I do not think that frozen strings would be the same (semantic-wise) as Symbols so I am not sure that this would lead to a reconsideration of the String/Symbol distinction.

I personally am fine with the way how Symbols behave as-is; I do also understand that other people think of the distinction somewhat annoying - the best example may be HashWithIndifferentAccess :D

I mean, that is so long to write out, I wonder how people can use it over "Hash"... but I understand the problem situation. People want to squeeze out whatever is faster, and also not have to think "when to use strings and when to use symbols" - it's like an obsession and somewhat understandable the larger (and perhaps slower) a codebase becomes + amount of users of said codebase. The mental "cognitive load" will remain as long as ruby has a string-versus-symbols distinction, but I myself have come to like symbols. :cat and :dog versus 'cats' and 'dogs'.

My only minor concern with the distinction was whether I would be able to treat Symbols as String-like for some string-like operations, such as via :foobar[1,2] - but that is already possible. Walks like a duck, quacks like a duck, is a symbolized duck. Actually, an even more radical solution would be to get rid of Symbols altogether, since that would simplify the usage instantly. Are frozen strings and symbols really as fast? I have no benchmarks but I think that symbols may still be faster, since they may do more - but since I have no benchmark, this is just idle speculation. I mention this because the old proposal here wrote "make symbols and strings the same thing" but you could also have a proposal that would "remove symbols" ... :P

Matz explained the origin of Symbols back then here:

http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-core/52019

I think Symbols became more popular after the Dave-book stating that they are much faster in Hashes than string-object, so since people like speed, they'd use Symbols a lot. I like symbols a lot too, but mostly because ... it's less to type! I can also enable some specific behaviour depending on input arguments given to methods. This may be a very akward usage example but I like it.

Perhaps you may have a chance to see the situation be revisited for ruby 3.x but I myself think that the Symbols will stay as they are for other reasons, including inertia. Another factor to be considered is how ruby "views" the its world - and I think that the original idea never was to expose symbols as-is and instead may have users focus only on strings primarily.

Btw here is a short discussion at stackoverflow about symbols in lisp:

https://stackoverflow.com/questions/8846628/what-exactly-is-a-symbol-in-lisp-scheme

What I gather there is that the primary use case is indeed due to speed (or "light-weight execution") compared to alternatives.

#70 - 12/26/2017 08:36 AM - dsferreira (Daniel Ferreira)

If we are considering breaking backwards compatibility in ruby 3.0 with removal of API like IO.read which is all over let's fix this symbol vs string issue once and for all shall we?
Ruby programmers will be much happier.
Why don't we start a new issue maybe in common ruby to come up with a broad solution?

#71 - 12/26/2017 08:43 AM - dsferreira (Daniel Ferreira)

If we do want to keep symbols as part of ruby API what about considering a new syntax for symbols and leave the current symbol syntax to be identified as string literals or just strings?

#72 - 12/26/2017 09:08 AM - normalperson (Eric Wong)

danieldasilvaferreira@gmail.com wrote:

> If we are considering breaking backwards compatibility in ruby
> 3.0 with removal of API like IO.read which is all over let's

Removal of IO.read? I hope you're not misinterpreting my [ruby-core:84461]
response to https://bugs.ruby-lang.org/issues/14239
I only want to warn on the IO.read("| command") case,
not remove the whole thing.

Anyways, I do not have much opinion on symbols/strings;
but I hate breaking compatibility.

#73 - 12/26/2017 09:24 AM - dsferreira (Daniel Ferreira)

It seems that what is in question to be removed in ruby 3.0 is the pipe feature associated to Kernel#open and related API not the all method.

Still breaking backwards compatibility though.

Sorry for misreading it and thanks Eric for pointing me out the error.

https://bugs.ruby-lang.org/issues/14239

#74 - 12/26/2017 09:36 AM - normalperson (Eric Wong)

Daniel Ferreira subtileos@gmail.com wrote:

> Why isn't your reply on the issue tracker?

It should be, I see it. Might take some minutes for mail to propagate.

#75 - 12/26/2017 02:33 PM - ko1 (Koichi Sasada)

> It seems that what is in question to be removed in ruby 3.0 is the pipe feature associated to Kernel#open and related API not the all method.
>
> Still breaking backwards compatibility though.

Important point is, we can know the breaking compatibility with exceptions. If we remove some method or feature (like IO.open('|...') we (the application programmer) can understand it was obsolete.

However, this case, program breaks silently and difficult to debug.

Personally I like this idea and if I have a time machine, I wanted to suggest this idea at 199x.
If we need to introduce breaking compatibility, we need to make transition path.

#76 - 12/26/2017 03:11 PM - dsferreira (Daniel Ferreira)

> If we need to breaking compatibility, we need to make transition path

Thanks Koichi.
I totally agree with you and Eric about worries on breaking backwards compatibility.
I was wondering if it wouldn't be possible to use some kind of transitional flag like:

frozen_string_literal: true

Or
--enable-frozen-string-literal

#77 - 12/26/2017 03:43 PM - ko1 (Koichi Sasada)

> I was wondering if it wouldn't be possible to use some kind of transitional flag like:

Please consider transition path for users who are using symbol and string difference like:

```
key = ...
...
when key
case String
  ...
case Symbol
  ...
end
```

How to find out such programs?

frozen-strign-literal, we can know issues by seeing FrozenError (and also debug_frozen_string_literal is supported to find out the problem).

If you (or someone) find out any good transition path, we think we can consider again.

#78 - 12/26/2017 07:30 PM - dsferreira (Daniel Ferreira)

Hi Koichi.

I've been thinking about the challenge you put me.
What if we add as a transition path the method:
String#old_symbol.

Then we could use it in the Symbol::===(var)
and we would warn if var.is_a?(String) && var.old_symbol == true

What do you think?
Something missing?

#79 - 12/26/2017 09:45 PM - jeremyevans0 (Jeremy Evans)

I'm 100% against merging Symbols and Strings. Symbols and Strings have always represented different concepts. Symbols are a simple way to get a unique named identifier, and Strings represent arbitrary data/text. In some cases, you need to convert from one or the other, generating an identifier from the content of data/text (String->Symbol), or treating the name of the identifier as data/text (Symbol->String). But Strings and Symbols represent different concepts and should be treated differently.

Merging Symbols and Strings would break libraries that do treat them differently in ways that would be almost impossible to fix without making APIs very ugly (and obviously breaking backwards compatibility).

The main argument I've heard for combining Strings and Symbols is to save some characters when typing, as shown in the initial post for this issue. I don't consider this a major problem, but if other people really want to save those characters, I think it would be better to add a terser form of string literals, as opposed to completely breaking symbols.

#80 - 12/26/2017 10:11 PM - dsferreira (Daniel Ferreira)

> But Strings and Symbols represent different concepts and should be treated differently.

Jeremy I agree with you on this and that is one of the reasons, possibly the biggest reason, why I think it will make a huge difference if we stop using symbols as if they were plain simple strings like I see scattered all over the place.

Let's put symbols where they belong shall we?

I ask again: Shall we open a new issue under common ruby with a broader purpose?

The reality is that converting current symbol's syntax to be used as yet another string's syntax will keep intact the great majority of the code currently in production.

From this discussions we don't see many examples of proper use of symbols as such.

I believe we can fix this and keep Symbols and Strings as part of ruby API in a much better way.

A new syntax for symbols is the first thing that comes to my mind or a plain Symbol.new instantiation.

#81 - 12/27/2017 05:35 PM - jeremyevans0 (Jeremy Evans)

dsferreira (Daniel Ferreira) wrote:

> > But Strings and Symbols represent different concepts and should be treated differently.
>
> Jeremy I agree with you on this and that is one of the reasons, possibly the biggest reason, why I think it will make a huge difference if we stop using symbols as if they were plain simple strings like I see scattered all over the place.

I think it's far more common to use symbols correctly than incorrectly. The only time I see confusion is when using things like HashWithIndifferentAccess. That more than anything else has probably contributed to some ruby programmers not understanding what the difference is.

> I ask again: Shall we open a new issue under common ruby with a broader purpose?

You are free to open an issue. However, considering this has already been rejected, I don't see the point.

> The reality is that converting current symbol's syntax to be used as yet another string's syntax will keep intact the great majority of the code currently in production.

"Extraordinary claims require extraordinary evidence"

> From this discussions we don't see many examples of proper use of symbols as such.

I'm not sure if this discussion is reflective of symbol usage in most ruby code. In most cases I see symbols used correctly, as named identifiers. In most cases I see strings used correctly, as text/data. There are a few cases where I see them used incorrectly, mostly at the borders of programs, where input is always provided as strings. However, that's a security sensitive area and care should be taken before converting any strings to symbols.

> I believe we can fix this and keep Symbols and Strings as part of ruby API in a much better way.
>
> A new syntax for symbols is the first thing that comes to my mind or a plain Symbol.new instantiation.

This does not make sense to me. If you introduce a new syntax for symbols, most people won't switch to it unless they don't care about backwards compatibility at all. If by some miracle the community does switch to it, you'll end up the same situation we currently have eventually.

You've already recognized that strings and symbols are different concepts and have different purposes. Therefore the only reason make the current symbol syntax represent strings if you want a terser syntax for strings. In which case instead of breaking backwards compatibility, it would make more sense to introduce a new separate terser syntax for strings.

#82 - 12/27/2017 07:52 PM - dsferreira (Daniel Ferreira)

> I think it's far more common to use symbols correctly than incorrectly. The only time I see confusion is when using things like HashWithIndifferentAccess.

Let me give you just this example to show you how things are out of control:

```
alias_method :foo, :bar
alias_method "foo", "bar"
```

Both usages are allowed.

This kind of interface is present in many places.
You have it in ruby core and you have it even more in gems and even more in applications.
And there will be no way you can change that line of thought scattered throughout the community.
We are using strings and symbols interchangeably and this is an evidence.

#83 - 12/27/2017 08:28 PM - Ajedi32 (Andrew M)

Leaving aside the issue of semantics for a moment, can anyone provide a real, concrete example of a situation where symbols could not trivially be replaced with strings without negatively impacting the functionality of the surrounding code?

I'm having trouble thinking of such a scenario that's not obviously contrived.

#84 - 12/27/2017 08:39 PM - dsferreira (Daniel Ferreira)

> Leaving aside the issue of semantics for a moment, can anyone provide a real, concrete example of a situation where symbols could not trivially be replaced with strings without negatively impacting the functionality of the surrounding code?

Koichi's challenge for a transition path is the example we have to support so far:

```
key = ...
...
when key
case String
  ...
case Symbol
  ...
end
```