Ruby trunk - Feature #5964

Make Symbols an Alternate Syntax for Strings

02/03/2012 08:51 PM - wardrop (Tom Wardrop)

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

Description

Or, to put it another way, make Symbols and Strings one and the same - interchangeable and indistinguishable from the other.

This may seem odd given that the whole point of Symbols is to have a semantically different version of a string that allows you to derive different meaning from the same data, but I think we have to compare the reason why Symbol's exist and the benefits they offer, to how their use has evolved and the problems that has introduced. There are a few main points I wish to make to begin what could be a lengthy discussion.

(1) Most people use Symbols because they're prettier and quicker to type than strings. A classic example of this is demonstrated with the increasing use of the short-hand Hash syntax, e.g. {key: 'value'}. This syntax is not supported for strings, and therefore only further encourages the use of symbols instead of strings, even where semantics and performance are not contributing factors.
(2) While the runtime semantics of Symbols will be lost, such as where the type of an object given (Symbol or String) determines the logical flow, the syntactic semantics will remain. Or in other words, Symbols will still be able to convey programmer intent and assist in code readability, for example, where strings are used for Hash keys and values, one may wish to use the symbolic syntax for the keys, and the traditional quoted syntax for the values.
(3) Runtime semantics are of course beneficial, but the cons are an unavoidable consequence. I mention this because I thought for a brief moment, of the possibility of introducing an alternate means of injecting semantics into strings, but I quickly realised that any such thing would take you back to square one where you'd have to be aware of the context in which a string-like object is used. It goes against the duck-typing philosophy of Ruby. If it walks and quacks like a string, why treat Symbols and Strings any differently.
(4) Performance is the other potential benefit of Symbols. It's important to note that the symbolic String syntax can still be handled internally in the same way as Symbols. You could still compare two strings created with the symbolic syntax by their object ID.

By removing the semantic difference between Strings and Symbols, and making Symbols merely an alternate String literal syntax, you eliminate the headache of having to coerce Strings into Symbols, and vice versa. I'm sure we've all written one of those methods that has about 6 #to_sym and #to_s calls. The runtime semantics that are lost by making Symbols and Strings one and the same, can be compensated for in other ways.

I like to think of the case of the #erb method in the Sinatra web framework, where it assumes a symbol is a file path to the markup, and that a string is the actual markup. This method could either be split into two, e.g. #erb for string, and #erbf for path to file. Or, you could include a string prefix or suffix, e.g. #erb './some_file' to indicate it's a file path. Or as my final example, you could simply go with an options hash, e.g. #erb 'some string' for a string, or #erb file: './some_file'. It all depends on the circumstance.

The whole point is to have only one object for strings, and that's the String object. Making Symbol a subclass of String wouldn't solve the problem, though it may make it more bearable.

I'm hoping those who read this consider the suggestion fairly, and don't automatically defend Symbols on merit.

Related issues:

Related to Ruby trunk - Feature #7792: Make symbols and strings the same thing	**Rejected**

History

#1 - 02/04/2012 02:39 AM - trans (Thomas Sawyer)

If I recall, this was tried for stint early in 1.9 development, but ultimately didn't work out --though I never caught the reason why.

Honestly, I think most all the trouble comes from the simple fact that a String and Symbol are not "hash-key equal". That alone would simplify all the option parameter use cases. Hell, maybe even go so far as to make them case equal (#===), which would get rid of almost all the issues along these lines (albeit require some code refactoring in rare case).

Of course, there is the other simple fact that Symbol doesn't support many of String's string manipulation methods. More support for that would be nice.

My point is, things could be done to greatly improve the issues you mention, without going so far as making Symbol a subclass of String --which technically probably doesn't make much sense.

#2 - 02/04/2012 10:34 AM - wardrop (Tom Wardrop)

I do agree. If String and Symbol were still semantically different (you could have a #sym? method), but they behaved the same (Symbol had all the string methods and was comparable to strings), then it would alleviate a lot of the issues. It wouldn't solve all problems, but at least most of the remaining issues would likely be a result of poorly designed 3rd party API's, which I'd hope would become less prevalent. I'd be happy to draw the line there.

#3 - 02/04/2012 12:06 PM - jballanc (Joshua Ballanco)

It is better to think of Symbols as numbers with labels attached, rather than Strings with slightly different semantics. "String.to_sym" is the equivalent of looking in a hash table of these labeled numbers for one with a label that matches the String, and if none exists than allocating a new number to attach this label to.

But leaving all of that aside, you have to consider that Symbols are *never* collected. This is required for their semantics. If you were to make Strings and Symbols equivalent, how would you decide when something could or couldn't be collected? How do you prevent memory from blowing up?

#4 - 02/05/2012 03:25 PM - kstephens (Kurt Stephens)

Joshua Ballanco wrote:

> But leaving all of that aside, you have to consider that Symbols are *never* collected. This is required for their semantics.

True, CRuby Symbols are not collected. However, in general, this is not required for every implementation of "symbols". There is an open bug to make CRuby Symbols collectable, but it will require C API changes. What semantics prevent Ruby Symbols from being collected?

Why should Strings and Symbols be distinct? Try adding the following "convenience" and watch what happens to performance and related semantics; I have seen this code in the wild:

class Symbol; def ==(other); self.to_s == other.to_s; end

Could Symbols behave more like Strings? Sure. I wish Symbol#<=>(Symbol) existed, and for some String-related operations, Symbols on the right-hand side might be automatically coerced.

But to conflate the identity properties of the Symbols and Strings would be a mistake. Those who think Symbols and Strings should be one-and-the-same may not understand the benefits of their differences: a Symbol's representation is its identity and its representation is immutable; a String does not have these properties.

Symbols represent concepts, Strings represent data. Their differences in identity helps maintain that distinction and have important performance and semantic implications.

#5 - 02/05/2012 04:52 PM - wardrop (Tom Wardrop)

In reply to "Symbols represent concepts, Strings represent data". That's true, and so I agree, but it's also a bit of an ideology. How strings and symbol's are used in actual code (i.e. open-source ruby libraries), does not always reflect that.

As I mentioned earlier, the decision to use String and Symbols can often come down to which is "nicer" from the perspective of reading and writing code. The relatively subtle semantic differences between Strings and Symbols is itself a problem. The two concepts are too similar. A symbol is simply an immutable string, which is globally unique. That's pretty much it. The problem I experience a lot, as a human writing code (and let's not forget that), is an assumption I make that parsing a String to a bit of code that expects a Symbol (i.e calling hash['key'] instead of hash[:key]) will work. It's not something you usually do when writing a String or Symbol literal, but it happens to me often when I've got a variable that represents a string (whether it be a String or a Symbol), where I pass the value of that variable to some method that errors out because it got the wrong "type" of string.

My point is that in a lot of cases, the tendency for human error out-weighs (I think) the benefits of the subtle semantic differences. When designing an API, it's rarely a straight-forward process deciding on whether to use a String or Symbol for a particular task - it's not like it's always obvious where a String should be used, and where a Symbol should be used, as it normally depends on the context from which your API is being called, which is hard to predict.

So it's clear that the two main areas that Symbols differ from Strings, is in their semantics and performance. Which of those two are people most concerned about losing? If everyone replaced all the Symbols in their code with Strings, what would you miss the most?

Just to re-iterate, I acknowledge that Symbols are useful. This discussion really comes down to whether you think the disadvantages (as a human reading and writing code), out-weigh the benefits?

#6 - 02/07/2012 05:56 AM - kstephens (Kurt Stephens)

Tom Wardrop wrote:

> When designing an API, it's rarely a straight-forward process deciding on whether to use a String or Symbol for a particular task - it's not like it's always obvious where a String should be used, and where a Symbol should be used, as it normally depends on the context from which your API is being called, which is hard to predict.

Using an API requires the ability to read code/documentation, fluency of the language, etc. If an API is hard to use or understand, don't blame the language. Good API design requires thinking, mastery of the language, knowledge of prior-art and convention, anticipation of unforeseen usage, documentation, etc.

The "Symbol vs. String" argument could be "String vs. Numeric", "XML vs. JSON", "eval vs. define_method" -- these types of arguments lead to important design decisions -- which *are* the job of a programmer, regardless of syntax or semantics.

> My point is that in a lot of cases, the tendency for human error out-weighs (I think) the benefits of the subtle semantic differences. ...

Humans can learn after making an error. Unfortunately, it's difficult for software to learn: "this String represents a concept" and optimize automatically. It's the programmer's job to use their tools effectively, sometimes that means catering to semantics that encode subtle computing concepts (like object identity). To take away a useful tool is limiting those who can.

> So it's clear that the two main areas that Symbols differ from Strings, is in their semantics and performance. Which of those two are people most concerned about losing? If everyone replaced all the Symbols in their code with Strings, what would you miss the most?

Symbols are used "under the hood" in CRuby -- Ruby 1.9 exposes and reinforces it more than 1.8 did. Internal and external performance would suffer if Symbols didn't maintain their current identity semantics and behavior. This change would effect *all* Ruby code, *all* Ruby implementations, whether or not a programmer decides to use Strings vs. Symbols based on stylistic reasons or by accident.

> Just to re-iterate, I acknowledge that Symbols are useful. This discussion really comes down to whether you think the disadvantages (as a human reading and writing code), out-weigh the benefits?

When I read:

{ :foo => "bar" }

I see a mapping of a concept to data or vice versa. When I read:

{ "foo" => "bar" }

I see a mapping of data to data. To me, they immediately express very different ideas and performance characteristics, at the same time.

Ruby's expressive power comes, in part, from a rich set of primitives -- take useful parts away and we end up with a dumbed-down, non-performant language.

Many programmers left other languages for Ruby because of its richness. What would Ruby be without eval, lambda, and blocks?

#7 - 02/12/2012 10:53 AM - wardrop (Tom Wardrop)

I do agree Kurt (for the most part), though I still maintain that Ruby should be more forgiving when it comes to Strings and Symbols. It's not always clear whether something is "data" as you put it, or a pointer/concept to data. What makes this even less clear cut, is that data can become a concept at runtime. In the ideal world, it would be very obvious where to use a String and where to use a Symbol, but it is not in this world. I think at the least, Ruby should provide a simpler means to compare Strings and Symbols where the programmer doesn't care what type of string data hits their API. That should be the default. If you explicitly care about the type of String data you have received, then that should be the exception, and thus you should have to be perform a strict comparison.

So in summary, String == Symbol should be by default I think. To do a strict comparison where you care about type, you should have to do a ===. That's one of the fews things I use to like about PHP. "==" was a lose comparison, "===" was a type-sensitive strict comparison, and it was consistent across the board.

#8 - 02/13/2012 09:53 AM - matz (Yukihiro Matsumoto)

Hi,

Two points:

(1) Currently symbols and strings are very different in several aspect. Making them "compatible" would break backward compatibility.

(2) "==" and "===" have different roles from PHP. So your logic is incomplete. I suggest to learn "Ruby Way", before proposing language changes to Ruby.

So in summary, this proposal is half-baked. Need to be more concrete.

```
                          matz.
```

In message "Re: [ruby-core:42509] [ruby-trunk - Feature #5964] Make Symbols an Alternate Syntax for Strings"
on Mon, 13 Feb 2012 09:10:21 +0900, Tom Wardrop tom@tomwardrop.com writes:

|I do agree Kurt, though I still maintain that Ruby should be more forgiving when it comes to Strings and Symbols. It's not always clear whether something is "data" as you put it, or a pointer/concept to data. What makes this even less clear cut, is that data can become a concept at runtime. In the ideal world, it would be very obvious where to use a String and where to use a Symbol, but it is not in this world. I think at the least, Ruby should provide a simpler means to compare Strings and Symbols where the programmer doesn't care what type of string data hits their API. That should be

the default. If you explicitly care about the type of String data you have received, then that should be the exception, and thus you should have to be perform a strict comparison.
|
|So in summary, String == Symbol should be by default I think. To do a strict comparison where you care about type, you should have to do a ===. That's one of the fews things I use to like about PHP. "==" was a lose comparison, "===" was a type-sensitive strict comparison, and it was consistent across the board.

#9 - 02/13/2012 11:34 AM - trans (Thomas Sawyer)

@tom For Ruby you want #eql? method. In Ruby #=== is more loosely defined and can mean different things for different class of object.

I think there are two parts to this proposal.

1) Make string and symbol more interchangeable, e.g. :s == "s" and hash[:s] == hash['s']. This would of course break backward compatibility. I imagine around half of the reason to want this is for dealing with named parameters in method interfaces. Since Ruby 2.0 is dedicated support for named parameters, and they will always be symbols (right?) then a good chunk of this issue will get resolved (though there is still the whole HashWithIndifferentAccess suckiness).

2) Make Symbol support a larger variety of manipulation methods, e.g. :this!.chomp('!') => :this. It doesn't have to support every such method, but cover some of the most fundamental ones would at least be nice. This doesn't so much effect backward compatibility.

#10 - 02/16/2012 08:34 AM - wardrop (Tom Wardrop)

matz (Yukihiro Matsumoto), the === proposal was more for example. Indeed at the moment, #eql? would be the method for type-sensitive string comparison, where as == would be for value comparison of string-like objects (Strings and Symbols). I'd also like to clarify that any such change should be part of a major release where backwards compatibility is expected to be broken to some extent, e.g. Ruby 2.0.

The crux of my suggestion is to make it easier to work with Strings and Symbols and less explicit (e.g. to_s and to_sym is a pain; it should be implied). Even some kind of directive for automatic type casting would be handy. I still believe that by default, a String should equal a Symbol, and vice versa. Right now you need to be explicit to get that behaviour, e.g. value.to_s == other_value.to_s, where it should be implicit, e.g. :something == 'something'. After all, as has been said, #eql? is the type-sensitive comparison operator. #== should be comparing on value. So technically, you could say the #== method of String and Symbol is breaking a Ruby idiom. Of course, if you were to make Symbol == String, then you would need most, if not all of the String methods to be attached to Symbol class, otherwise you'd be back to doing explicit type checking which were somewhat defeat the benefits of making Symbol == String.

By the way, I make these suggestions as both a user and designer of API's. The problem for the designer of the API is to try and meet user expectations, and that expectation is normally that Strings and Symbols be interchangeable (the user ideally shouldn't need to care; the user should only care whether it has string-like data). As the user of the API, you need to be aware of whether the API is type sensitive to String and Symbols - some methods may be, some may not. The whole point of raising this issue is to resolve this outstanding issue of caring about what type of string you have (String or Symbol), when you should only need care about whether you have string-data or not. The whole point of this of course is to make Ruby an even nicer programming language.

#11 - 02/16/2012 09:08 AM - drbrain (Eric Hodel)

- Category set to core

=begin
This may break RubyGems which depends on the difference between Symbol and String keys in ((%~/.gemrc%)).

Symbol keys set RubyGems options and String keys set ((%gem%)) command default arguments.

I'm sure RubyGems isn't the only code that depends upon this difference between String and Symbol
=end

#12 - 02/16/2012 09:47 AM - wardrop (Tom Wardrop)

No, there's quite a lot of such code. Though, if they're explicitly checking for the class type, as I'd imagine most of them would be, then they shouldn't break.

#13 - 02/16/2012 10:29 AM - headius (Charles Nutter)

On Sun, Feb 5, 2012 at 12:25 AM, Kurt Stephens redmine@ruby-lang.org wrote:

> True, CRuby Symbols are not collected. However, in general, this is not required for every implementation of "symbols". There is an open bug to make CRuby Symbols collectable, but it will require C API changes. What semantics prevent Ruby Symbols from being collected?

The only thing that has prevented me from making symbols GCable in JRuby is the concern that someone might rely on the same symbol having the same ID forever. They're implied to always be the same object, forever and ever, and allowing them to GC would break that.

If they could be "the same object, as long as someone's referencing it" and the implications of a symbol possibly going away and being reborn as a new object had no negative implications for Ruby

applications, making them GCable would be fine. I don't know that
that's the case.

- Charlie

#14 - 02/16/2012 11:59 AM - cjheath (Clifford Heath)

On 16/02/2012, at 12:28 PM, Charles Oliver Nutter wrote:

> On Sun, Feb 5, 2012 at 12:25 AM, Kurt Stephens redmine@ruby-lang.org wrote:
>
> > True, CRuby Symbols are not collected. However, in general, this is not required for every implementation of "symbols". There is an open bug to make CRuby Symbols collectable, but it will require C API changes. What semantics prevent Ruby Symbols from being collected?
>
> The only thing that has prevented me from making symbols GCable in
> JRuby is the concern that someone might rely on the same symbol having
> the same ID forever.

That's clearly not the case if you define "forever" to mean "across program instances".
So if they save the object_id of a symbol somewhere, without still having a reference
to the symbol, they'll be broken if "somewhere" persists across instances, but not
if its persistence is only within the same instance.

I say do it, and let them whinge. My bet is they'll be too embarrassed to have done
such a grubby thing and won't even complain.

Clifford Heath.

#15 - 02/18/2012 03:09 AM - kstephens (Kurt Stephens)

On 16/02/2012, at 12:28 PM, Charles Oliver Nutter wrote:

> ... someone might rely on the same symbol having
> the same ID forever.

The only "safe" object_id is a monotonic one that is not tied to a memory address; by definition, a collector will reuse it. :)
I resort to this:

```
(a_cache[obj.object_id] ||= [ obj.some_cacheable_value, obj ]).first
```

#16 - 03/30/2012 12:45 AM - mame (Yusuke Endoh)

- Status changed from Open to Assigned

- Assignee set to matz (Yukihiro Matsumoto)

#17 - 03/30/2012 04:20 PM - matz (Yukihiro Matsumoto)

- Status changed from Assigned to Rejected

In Ruby, Symbols annd Strings are different in both semantics and behavior (and implementation).
Unifying them cause a lot of problems. I guess it's not worth changing.

Matz.

#18 - 10/05/2017 06:21 PM - sheerun (Adam Stankiewicz)

matz (Yukihiro Matsumoto) wrote:

> In Ruby, Symbols annd Strings are different in both semantics and behavior (and implementation).
> Unifying them cause a lot of problems. I guess it's not worth changing.
>
> Matz.

Did it change since ruby got frozen string? Is there much difference between frozen string and symbol?

http://blog.arkency.com/could-we-drop-symbols-from-ruby/

#19 - 10/05/2017 08:00 PM - avit (Andrew Vit)

sheerun (Adam Stankiewicz) wrote:

> Did it change since ruby got frozen string? Is there much difference between frozen string and symbol?

They are still semantically different objects: methods like :a + :b or :c.upcase make no sense.

#20 - 10/05/2017 09:26 PM - Ajedi32 (Andrew M)

avit (Andrew Vit) wrote:

> methods like :a + :b or :c.upcase make no sense.

They do make sense though. In fact, I can easily tell exactly what those methods should return just by their names. Are you saying you can't?

I can even imagine a reasonable use-case for both of those methods. For example, converting the name of an attribute reader method to the name of its corresponding setter (:name + '='), or grabbing the value of a constant based on a symbol passed to method_missing (mod.const_get(symbol.upcase)).

#21 - 10/06/2017 12:55 AM - nobu (Nobuyoshi Nakada)

Ajedi32 (Andrew M) wrote:

> I can even imagine a reasonable use-case for both of those methods. For example, converting the name of an attribute reader method to the name of its corresponding setter (:name + '='), or grabbing the value of a constant based on a symbol passed to method_missing (mod.const_get(symbol.upcase)).

The setter case is possible by :"#{name}=".
And Symbol#upcase and so on are defined.

#22 - 10/06/2017 10:35 AM - avit (Andrew Vit)

My only point was that symbols are not meant to be operated on the same as strings, because they do serve different purposes. They can serve as a kind of lightweight type-safety when you are dealing with an internal identifier, but in practice there is so much overlap and conversion between them (HashWithIndifferentAccess etc.) that this is mostly irrelevant.

I completely forgot that a few string methods were added to Symbol in 1.9... I can see the point, and these blur the line even more.