Ruby trunk - Feature #4801

Shorthand Hash Syntax for Strings

05/30/2011 09:44 AM - wardrop (Tom Wardrop)

Status:	Rejected	
Priority:	Normal	
Assignee:	nobu (Nobuyoshi Nakada)	
Target version:	2.6	

Description

Assuming there's no technical limitation or ambiguities, I suggest that the shorthand syntax for symbol's in the context of an array, be applied to strings also.

E.g. {'key': 'value'}

I don't believe there are any syntax ambiguous that this would give rise to. The only consideration that may need to be made, is if there are plans to support shorthand syntax for quoted symbols, e.g. {'key': 'value'}. If quoted symbols are off the table, then there's no harm in implementing a shorthand hash syntax for strings. This may stem the growing problem of what I like to call 'symbolitis' , where symbol's are selected as the key type purely for their aesthetics and ease of use, even when strings are a more appropriate choice.

Thoughts?

Related issues:

Related to Ruby trunk - Feature #4276: Allow use of quotes in symbol syntacti...	**Closed**	**01/13/2011**

History

#1 - 05/30/2011 09:53 AM - matz (Yukihiro Matsumoto)

Hi,

In message "Re: [ruby-core:36559] [Ruby 1.9 - Feature #4801][Open] Shorthand Hash Syntax for Strings"
on Mon, 30 May 2011 09:44:35 +0900, Tom Wardrop tom@tomwardrop.com writes:

|Assuming there's no technical limitation or ambiguities, I suggest that the shorthand syntax for symbol's in the context of an array, be applied to strings also.
|
|E.g. {'key': 'value'}

Iff {'key': 'value'} means {:key => 'value'} I have no objection.

```
                                                    matz.
```

#2 - 05/30/2011 04:23 PM - yeban (Anurag Priyam)

> Iff Â {'key': 'value'} means {:key

#3 - 05/30/2011 04:23 PM - rkh (Konstantin Haase)

I would find that misleading. +1 for {'a': 'b'}

#4 - 05/30/2011 04:23 PM - matz (Yukihiro Matsumoto)

Hi,

In message "Re: [ruby-core:36571] Re: [Ruby 1.9 - Feature #4801][Open] Shorthand Hash Syntax for Strings"
on Mon, 30 May 2011 16:12:35 +0900, Anurag Priyam anurag08priyam@gmail.com writes:

|> Iff {'key': 'value'} means {:key => 'value'} I have no objection.
|
|Won't that be misleading? I think the OP wants {'key': 'value'} to
|mean {'key' => 'value}.

I don't disagree here. But considering the fact that {key: "value"}

is a shorthand for {:key => "value"}, {"key": "value"} should be a
shorthand for {:"key" => "value"}. Besides that, since it reminds me
JSON so much, making a: and "a": different could cause more confusion
than the above misleading.

```
                            matz.
```

#5 - 05/30/2011 04:53 PM - yeban (Anurag Priyam)

> |> Iff Â {'key': 'value'} means {:key

#6 - 05/30/2011 07:59 PM - Cezary (Cezary Baginski)

On Mon, May 30, 2011 at 04:21:32PM +0900, Yukihiro Matsumoto wrote:

> Hi,
>
> In message "Re: [ruby-core:36571] Re: [Ruby 1.9 - Feature #4801][Open] Shorthand Hash Syntax for Strings"
> on Mon, 30 May 2011 16:12:35 +0900, Anurag Priyam anurag08priyam@gmail.com writes:
> |Won't that be misleading? I think the OP wants {'key': 'value'} to
> |mean {'key' => 'value}.
>
> I don't disagree here. But considering the fact that {key: "value"}
> is a shorthand for {:key => "value"}, {"key": "value"} should be a
> shorthand for {:"key" => "value"}. Besides that, since it reminds me
> JSON so much, making a: and "a": different could cause more confusion
> than the above misleading.
>
> ```
>                             matz.
> ```

Misleading, yes - but I think this is only a symptom of a bigger
problem.

In Rails projects I can never be sure if the Hash object I am working
accepts symbols (usually), strings (sometimes) or is a
HashWithIndifferentAccess and doesn't matter until a plugin builds its
own Hash from the contents.

The current Hash behavior is the most generic and flexible, but
unfortunately both the most surprising for novices and least
practical, IMHO.

My thought: warn when mixing string and symbol access in a Hash,
because it is most likely an error. It becomes too easy to
accidentally mix external data (String based hashes, though not
always) with code/language constructs (usually Symbol based hashes).

With a warning, you won't guess the syntax wrong and not know
immediately.

I am wondering: is having strings as keys instead of symbols in a Hash
actually useful? Aside from obscure string/symbol optimization cases?

Another idea would be a Ruby SymbolHash, StringHash accessible
from C API that raises when used incorrectly. And methods for
conversion between the two would probably clean up a lot of code.

HashWithIndifferentAccess is cool, but it is only a workaround to
reduce surprise, not to prevent the root cause of it.

With the new syntax we create an extra point of confusion for novices
based on implementation details:

1. Will I get a Symbol or a String from this syntax?

2. Should the resulting Hash I pass to an API contain symbols or
   strings or any? What if it changes in the future? How can I get a
   warning or error if I get this wrong?

1) is a problem only because of 2).

Concentrating on getting both correct will cost more productivity than the flexibility would allow. The current behavior also keeps Hash from being more consistent with similar structures in other languages (such as JSON).

These are just thoughts about revising the current Hash behavior to reflect the changes in the syntax - for the same reasons as the syntax additions. I don't feel too competent in this area and I am probably missing a lot of things.

Sorry for the length.

+1 for consistency with :"foo".

--
Cezary Baginski

#7 - 05/30/2011 08:59 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Em 30-05-2011 07:58, Cezary escreveu:

> On Mon, May 30, 2011 at 04:21:32PM +0900, Yukihiro Matsumoto wrote:
>
>> Hi,
>>
>> In message "Re: [ruby-core:36571] Re: [Ruby 1.9 - Feature #4801][Open] Shorthand Hash Syntax for Strings"
>> on Mon, 30 May 2011 16:12:35 +0900, Anurag Priyamanurag08priyam@gmail.com writes:
>> |Won't that be misleading? I think the OP wants {'key': 'value'} to
>> |mean {'key' => 'value}.
>>
>> I don't disagree here. But considering the fact that {key: "value"}
>> is a shorthand for {:key => "value"}, {"key": "value"} should be a
>> shorthand for {:"key" => "value"}. Besides that, since it reminds me
>> JSON so much, making a: and "a": different could cause more confusion
>> than the above misleading.
>>
>> ```
>>                                 matz.
>> ```
>>
>> Misleading, yes - but I think this is only a symptom of a bigger
>> problem.
>
> In Rails projects I can never be sure if the Hash object I am working
> accepts symbols (usually), strings (sometimes) or is a
> HashWithIndifferentAccess and doesn't matter until a plugin builds its
> own Hash from the contents.
>
> The current Hash behavior is the most generic and flexible, but
> unfortunately both the most surprising for novices and least
> practical, IMHO.
>
> My thought: warn when mixing string and symbol access in a Hash,
> because it is most likely an error. It becomes too easy to
> accidentally mix external data (String based hashes, though not
> always) with code/language constructs (usually Symbol based hashes).
>
> With a warning, you won't guess the syntax wrong and not know
> immediately.
>
> I am wondering: is having strings as keys instead of symbols in a Hash
> actually useful? Aside from obscure string/symbol optimization cases?
>
> Another idea would be a Ruby SymbolHash, StringHash accessible
> from C API that raises when used incorrectly. And methods for
> conversion between the two would probably clean up a lot of code.
>
> HashWithIndifferentAccess is cool, but it is only a workaround to
> reduce surprise, not to prevent the root cause of it.
> ...

While on the subject, I really wished that hash constructor ({}) was an instance of HashWithIndiferentAccess from the beginning in Ruby.

Actually, it should still be Hash, but should work like
HashWithIndiferentAccess.

It is very misleading that my_hash[:something] != my_hash['something'].
Also, I never found any useful on allowing that. Most of times an
HashWithIndiferentAccess is what you really want.

Changing back to the subject (kind of), in Groovy, here is what happens:

a = [abc: 'some text']; a['abc'] == 'some text'
a = [1: 'some text']; a.keySet()*.class == [java.lang.Integer]

In Ruby, {1: 'some text'} will raise an error. It would be great if it
allowed numer indexing of hashes with the same syntax.

About {'abc': 'some text'}, it is ok to me to return {:'abc' => 'some
text'}. But I think this should also be possible:

abc = 'cba'
{"#{abc}": 'some text'} == {:'cba' => 'some text'}

This is also possible in Groovy.

Here are my 2 cents.

#8 - 05/30/2011 09:23 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Em 30-05-2011 09:05, Michael Edgar escreveu:

> Since :"#{abc}" is allowed in Ruby, I imagine that any such substitute syntax would preserve that property.
>
> I disagree strongly that Hash, the base class, should special-case the behaviors
> of Strings and Symbols to be equal. It's a hash table, like those encountered in any other language,
> and shouldn't behave unlike typical hash tables. Namely h[a] and h[b] look up the same
> value iff a == b (or a.eql?(b), or whichever equality test you use). Strings and symbols
> are never equal.

Maybe, if we introduced another equality operator for comparing strings
and symbols, let's say ### (I know this is a terrible representation,
but I don't care about it now - just the idea):

:"some-symbol" ### 'some-symbol' => true

Unless one of them is not a symbol, it would work as always:

123 ### '123' => false
123 ### 123 => true

Or maybe this operator should have a similar behavior for comparing
numbers too. For instance, in Perl and other languages, if I remember
correctly, hash[1] == hash['1'], and I think this is a good thing.

After defining how this operator should behave, than the hash
implementation would be typical anyway.

I'm not proposing to change this in Ruby because of current existent
written code that would probably be broken by such a change, but if this
had been the decision from the beginning it would be fantastic!

#9 - 05/30/2011 11:23 PM - Cezary (Cezary Baginski)

On Mon, May 30, 2011 at 09:05:04PM +0900, Michael Edgar wrote:

> Since :"#{abc}" is allowed in Ruby, I imagine that any such
> substitute syntax would preserve that property.
>
> I disagree strongly that Hash, the base class, should special-case
> the behaviors of Strings and Symbols to be equal. It's a hash table,
> like those encountered in any other language, and shouldn't behave
> unlike typical hash tables. Namely h[a] and h[b] look up the same
> value iff a == b (or a.eql?(b), or whichever equality test you use).
> Strings and symbols are never equal.

I though exactly the same thing, until I realized that having keys of
different types in a Hash isn't really part of the general Hash
concept. It is a side effect of Ruby being dynamically typed.

I agree and I wouldn't allow symbols to be equal to strings for keys. I
would take the step further - they shouldn't be both used for keys in
the same Hash - *because* they are two different types. Especially
since they can easily represent one another.

Consider the following:

{ nil =>0, :foo => 1, 'foo' => 2 }

Conceptually, people expect Hash keys to be of the same type, except
maybe for "hacks" like that nil above that can simplify code.

If someone out there in the world actually demands that such a Hash is
valid and that :foo and 'foo' are different keys, you could always wrap
Hash to support that for that single, specialized case.

Otherwise the whole world tries to use HWIA in all the wrong places as
the silver bullet or write complex code to handle "strange" hashes
gracefully. Or use HWIA just to symbolize the keys - "just in case".

In Ruby "foo" + 123 raises a TypeError. Adding a string key to a
symbol-keyed Hash doesn't even show a warning.

I consider hashes with different key types different types of hashes,
that shouldn't even be allowed to merge together without conversion.
This could be useful both in Rails to make the meaning of each HWIA
instance more explicit and for API designers to worry less about how
to process hashes in a robust way.

I think the meaning of symbols and hashes are too similar for such
different types to be allowed as keys in the same Hash instance.

Further more, if the standard Hash didn't allow strings for keys
(another class for current behavior?), the new shorthand syntax would
be even less surprising.

Symbols are recommended in favor of Strings for hashes anyway.

--
Cezary Baginski

#10 - 05/31/2011 12:23 AM - Cezary (Cezary Baginski)

On Mon, May 30, 2011 at 11:24:29PM +0900, Michael Edgar wrote:

> On May 30, 2011, at 10:19 AM, Cezary wrote:
>
> > Symbols are recommended in favor of Strings for hashes anyway.
>
> Only for fixed key sets. Symbols aren't GCd, so if the set of keys for a Hash
> grows with respect to input, then forcing them all to symbols will grow your
> Ruby process's memory usage irreversibly.

Good point.

Because of this, I think it makes ever more sense to have a
specialized Hash for string-keying - the each Hash "type" would have
entirely different applications anyway.

Also, any documentation for conversion between such Hash "types" could
warn about costs and point to alternatives.

--
Cezary Baginski

#11 - 05/31/2011 11:23 PM - Cezary (Cezary Baginski)

On Tue, May 31, 2011 at 05:55:39AM +0900, Piotr Szotkowski wrote:

> Cezary:

First of all, thanks Piotr for taking the time to discuss this.
My original ideas for solving the problem or their descriptions
sucked, but I left your comments because they still apply or provide
good examples.

I'm trying to get an idea of how the implementation decisions behind
hashes affect the general use of hashes in Ruby and if something could
be slightly changed in favor improving the user's experience with the
language without too much sacrifice in other areas.

I believe Hash was designed with efficiency and speed in mind and the
recent Hash syntax changes suggest that all the current ways people use
Hash in Ruby is way beyond scope of the original concept.

Refinements may minimize the need for changes here, but even still, I
think this is a good time to consider what Hash is used for and how
syntax changes can help users better express their ideas instead of
just being able to choose only between an array, a very, very general
associative array or 3rd party gems that have no syntax support.

I hope I am not going overboard with this topic. I have serious doubts
that the slight changes in Hash behavior presented won't cause
problems, but I cannot think of any serious downsides, especially if
only a warning is emitted. And with such a usability upside, I must be
missing a big flaw in the idea or a big gain from the current
behavior.

If this topic does not contribute to Ruby from the user's perspective
I am ready to drop the subject entirely.

>> I though exactly the same thing, until I realized
>> that having keys of different types in a Hash
>> isn't really part of the general Hash concept.

> Why? [citation needed]

My wording isn't correct.

First, a Hash in ruby is an associative array that I read about here:

http://en.wikipedia.org/wiki/Associative_array

And from this:

"From the perspective of a computer programmer, an associative array
can be viewed as a generalization of an array. While a regular array
maps an integer key (index) to a value of arbitrary data type, an
associative array's keys can also be arbitrarily typed. In some
programming languages, such as Python, the keys of an associative
array do not even need to be of the same type."

The type of the key can be anything. Keys can even be different types
with a single instance. The latter is not a requirement of every
possible associative array implementation and this is what I meant.

It can be implementation specific, for example - an rbtree requires
ordering of keys. In this specific case, you cannot have a symbol and
string in such an associative array, because you cannot compare them.

But since Hash uses a hash table, it is possible to have a wider range
of key types, including both symbol and string together. The
implementation allows it, but my question is: is it *that* useful in
the real world? Or does it cause more harm than good?

>> { nil => 0, :foo => 1, 'foo' => 2 }
>>
>> Conceptually, people expect Hash keys to be of the same type,
>> except maybe for "hacks" like that nil above that can simplify code.

> Well, they either do or don’t, then. :)

Right. What I wrote isn't correct. I think people expect hash keys to
match a given domain to consider them valid. Just like every variable
should have a value within bounds or raise at the first possible
opportunity. Unless the cause of a problem is otherwise trivial to
find and fix.

I don't recommend the example with nil above. Better alternatives IMHO:

{ :'' => 0, :foo => 1 }[some_key || :'']

or

{ :foo => 1 }[some_key] || 0

or set the default in Hash

Hash.new(0).merge(:foo => 1)[some_key]

That is why I called it a hack - using a Hash key to get default
values.

> Hm, IMHO ‘any object can be a key, just as any object can be
> a value’ is the general case, and ‘I want my Strings and Symbols
> to be treated the same when they’re similar, oh, and maybe with
> the nil handled separately for convenience’ is the specialised case.

Exactly. The specialized case is obviously bad. But the general case
turned out not to be too great. I am thinking about third solution:
generic, but within a specified domain - ideally were the differences
between string and symbol stop them from unintentionally being in the
same Hash without being too specialized. And without subclassing.

Even by just a warning that is emitted when a Hash becomes unsortable,
we are not breaking the association array concept while *still*
supporting 99% or more actual real world use cases. And not making any
type-specific assumptions you presented.

As a side effect, if a user writes {'foo': 123}.merge('foo' => 456),
they will get a warning instead of just a hash with two pairs.

Such a warning most likely will help find design flaws and make
difficult to debug errors less often when refactoring. And hopefully
encourage a better design or just think a little more about the
current one.

> > In Ruby "foo" + 123 raises a TypeError. Adding a string
> > key to a symbol-keyed Hash doesn't even show a warning.
>
> I don’t see why it should – as long as it still
> responds to #hash and #eql?, it’s a valid Hash key.

Both methods are specific to Ruby's association array's internals
which uses a hash table. Users generally care only about their
string->symbol problems until they realize that using strings for keys
is generally not a good thing because of problems and debugging time.

Implementation wise I think Hash is great. However, the flexibility
along with symbol/string similarities and more ingenious uses of Hash
will probably cause only more problems over time.

Example:

Python doesn't have symbols and has named arguments. In Ruby we use a
symbol keyed Hash to simulate the latter which is great, but if the
hash is not symbol key based, there is no quick, standard way to
handle that. Sure, you can ignore or raise or convert, but why handle
something you should be able to prevent?

Ignoring keys you don't know seems like a good idea, but the result is not very helpful in debugging obscure error messages. And lets face it: most of the Ruby code people work on is not their own.

The only people who don't need to care are the experts who already have the right habits and understanding that allows them to avoid problems without too much thought. The rest have to learn the hard way.

> Hashes in Ruby serve a lot of purposes (they even maintain insertion order); if you want to limit their functionality, feel free to subclass.

Why do I have to subclass Hash to get a useful named arguments equivalent in Ruby? Why would I want object instances for argument names? Why can't I choose *not* to have them in a simple way?

The overhead and effort required to maintain and use a subclass becomes a good enough reason to give up on writing robust code.

Which is probably what most rubists do.

We have RBTree and HashWithIndifferentAccess. Neither really helps in creating good APIs for many of the wrong reasons:

- HWIA is for Rails specific cases but is usually abused to avoid costly string/symbol mistakes
- RBTree is a gem most people don't know about and stick with Hash anyway. It adds an ordering requirement but that seems like a side effect. It was proposed to be added in Ruby 1.9, but I don't remember why it ultimately didn't
- the {} notation is too convenient to lose in the case of subclassing, especially when Hash is used for method parameters
- in practice, you can only use the subclass in your own code

> There's nothing preventing you from subclassing Hash to create StringKeyHash, SymbolKeyHash or even MonoKeyHash that would limit the keys' class to the first one defined.

I thought about that exactly to avoid subclassing: by having an alternative to the current Hash already as a standard Ruby collection.

But now it think the idea is too limiting to be practical. From the user's perspective, having Hash restrict its behavior the way RBTree does would save people a lot of grief.

If Hash changed its behavior in the way described, most of the existing code would work as usual. Manually replacing {} with a subclass in a large project is a waste of time. Hashes are used too often to even consider subclassing.

Consider regular expressions: you can specify options to a regexp, defining its behavior. Having the same for hashes could be cool:

```
{'a' => 3, :a => 3}/so  # s = strict, o = ordered
```

As examples, we could also have:

r = uses RBTree for the Hash (and so implies 's')

i = indifferent access, but not recommended (actually, I personally wouldn't want this as an option)

> How would you treat subclasses? Let's say I have a Hash with keys being instances of People, Employees and Volunteers (with Employees ans Volunteers being subclasses of People). Should they all be allowed as keys in a single MonoKeyHash or not?

Good example of using a Hash to associate values with (even random) objects!

Since having keys orderable already answers the part about allowing
into the Hash, I'll concentrate on the case where items are of
different types.

How about an array of objects and a hash of object id's instead?

```
[ person1, person2, ...]
{ person1.object_id => some_value, ... }
```

Or just use the results of #hash as the keys if it is about object
contents. This makes your intention more explicit.

```
{ person1.hash => some_value, ... }
```

If you really need different types as a way of associating values with
random objects, you could create a Hash of types and each type would
have object instances:

```
{
Fixnum => { 1 => "one", 2 => "two" },
String => { "1" => "one", "2" => "two" },
}
```

Then you can use hash[some_key.class][some_key] for access if you
really need the current behavior.

Not much harder to handle, but you have much more control over the hash
contents. You probably need to know about used types in the structure
anyway to handle its contents (domain).

> What about String-only keys, but with different
> keys having their own different singleton methods?
>
> (For discussion's sake: what about if a couple of the Strings
> had redefined #hash and #eql? methods, on an instance level?)

That's relying heavily on implementation specific details - like
counting on Ruby hashes preserving order or not. That changed
actually, yes. I don't really remember what was the main reason
though.

#hash and #eql? are called by Hash internally - if there is a good
reason for redefining these, there is probably a good way to do it
without relying on Hash internals.

If for some fictional reason Ruby used an rbtree internally for Hash,
#<=> would be used instead of #hash + #eql. Everything else would be
the same except for allowed key values.

>> I think the meaning of symbols and hashes are too similar for such
>> different types to be allowed as keys in the same Hash instance.
>
> But that would introduce a huge exception in the current
> very simple model. Ruby is complicated enough; IMHO we
> should strive to make it less complicated, not more.

Novice users find symbols, strings and Hashes complicated and confusing.
Changing this is my focus here. A complex model that is easily
discoverable is probably better than a simple model that requires
complex solutions from the users to do a great job.

I know it takes hard work and countless hours to keep Ruby a fun
and great language as it is and I think it pays off, nevertheless.
If the goal was to create a simple language with a simple
implementation, we would might have had another Java instead.

Even if it results in an overly complex parser and implementation, I
think only good will come from going out of one's way to make Ruby
users lives easier.

Which is why I really appreciate your input and for giving me the
motivation to understand the topic and Ruby internals better.

Thanks!

--
Cezary Baginski

#12 - 06/02/2011 10:23 AM - cjheath (Clifford Heath)

On 02/06/2011, at 10:28 AM, Kurt Stephens wrote:

> A String used as inline or static nmenomic in real code is pinned
>
> down anyway and will not be GCed.

Yes it will. What's pinned down is a call to a constructor (with a

pinned value,
but there is a fixed and finite number of such values), so that every

time
that code executes, a new String is created, and that will get GC'd.

Symbols
avoid the overhead of constructing and GC'ing new Strings, but the

problem
Cezary is talking about is where code that creates Symbols dynamically

from
Strings, that can create a potentially unbounded number of Symbols,

none of
which can be GC'd.

Clifford Heath.

#13 - 06/02/2011 12:23 PM - spatulasnout (B Kelly)

Clifford Heath wrote:

> ... the problem
> Cezary is talking about is where code that creates Symbols dynamically
>
> from Strings, that can create a potentially unbounded number of Symbols,
>
> none of which can be GC'd.

Agreed.

If a concrete example would help, my objects are receiving
RPC messages from untrusted clients, and I check for valid
messages by Hash lookup.

Ideally the Hash keys would be Symbols, but if I then
convert the untrusted messages to Symbols to perform the
Hash lookup, I've opened my server to a memory leak DoS
exploit.

(On a related note, the RPC protocol supports all Ruby
data types including Symbol, and the untrusted message names
actually *arrive* at the protocol level as Symbols; but by
default, any Symbols are deserialized as Strings when they
reach the remote, because of the same DoS potential.)

Regards,

Bill

#14 - 06/02/2011 01:53 PM - cjheath (Clifford Heath)

On 02/06/2011, at 1:29 PM, Kurt Stephens wrote:

> Good point. However, if the internal symbol table used weak

> references to Symbol objects,
> all dynamic Symbols that are not pinned down by code could be
>
> garbage collected.

You'd lose some performance though, because it makes the GC graph

traversal bigger.
Possibly you could activate Symbol sweeping infrequently, and only if

Symbols represent
a significant percentage of all objects. Not sure how such an option

would play in the GC
code however.

Clifford Heath.

#15 - 06/02/2011 05:59 PM - Cezary (Cezary Baginski)

On Thu, Jun 02, 2011 at 01:47:30PM +0900, Clifford Heath wrote:

> On 02/06/2011, at 1:29 PM, Kurt Stephens wrote:
>
>> Good point. However, if the internal symbol table used weak references to
>> Symbol objects,
>> all dynamic Symbols that are not pinned down by code could be garbage
>> collected.
>
> You'd lose some performance though, because it makes the GC graph
> traversal bigger. Possibly you could activate Symbol sweeping
> infrequently, and only if Symbols represent a significant percentage
> of all objects. Not sure how such an option would play in the GC
> code however.

I wonder if it could be possible to internally delay the conversion
from string to symbol until it actually can save memory?

--
Cezary Baginski

#16 - 06/05/2011 05:53 AM - Cezary (Cezary Baginski)

On Sat, Jun 04, 2011 at 02:17:28AM +0900, Piotr Szotkowski wrote:

> // Apologies for the delayed reply – it takes
> // a bit to digest such a detailed response! :)

Oh, don't apologize - my fault for being way too elaborate and taking
so much of your time.

The topic got me really thinking on some concepts.

Here is an overview:

1. Regarding coding issues only, still I don't see the difference between Hash and RBTree as feature. I don't see #hash +#eql? as being superior in this regard than #<=>.

Hash API is YAGNI category for users, if you ask me.

RBTree is a good reference on how a hash can work with #<=>.

(RBTree wasn't included because it wasn't mature enough at the
time).

1. I patched Ruby to warn about cases where key type mixing takes place. The cases that popped up didn't justify the need for Hash's generic behavior (though I only checked a few things).

The result of this "experiment" however convinced me that adding
"Ruby best practice" warnings is both valuable and easy. If only it
were easier to turn them on and off in code...

The interesting conclusion is that such changes don't have to be in
the standard MRI to be useful.

There could even be a patched "lint" version of Ruby, that could
warn about not using the short hand hash syntax.

1. Just for reference: maybe I didn't make myself clear, but the last thing I want is to have symbols compared to strings. HWIA handles a Rails specific case for convenience, so it doesn't count.

The idea that '3' + 3 doesn't work is something I find very useful.
Likewise, if an integer keyed hash didn't merge with a string keyed
hash I would find such a case very similar.

The PHP way would be to discover that "Array is a special case of
Hash (implementation aside), with integers as keys, so not why
create a general array/hash class, call it array and have one class
less for novices?" That didn't turn out nice IMHO.

The PHP way would be to drop symbols because they are too difficult
to grasp - or make them coerce to one another, which is probably
worse.

As an analogy, my approach with making Hash more strict seems to be
like making the hurdles more difficult to jump over, but pasting on
them instructions about what you need to learn to clear them.

Hardly the PHP approach if you ask me.

So, from my point of view, slightly putting the generalized behavior
"out of view" (but not out of reach) would get people to sit back
and think more about design, and not just reach for what they know.

Much better conditions for learning than debugging.

> I'd argue it is useful in that it's a very simple model

By contrast, RBTree also seems simple - at least to me. Although the
name doesn't suggest how similar it is to Hash.

> Also, Ruby is not known for treating the 'does it cause more harm
> than good' question as a benchmark

True. With so many interesting languages popping up, syntax will
probably become more important for Ruby's success in the future.

We already have two very successful Rubies: 1.8.7 and 1.9.2. With such
a long history already, it is now easier to make harder decisions
about the language and syntax with less risk.

> Yes, but the domain is usually specific, and I don't think
> enforcing any parts of it on all Hashes is a good idea.

I thought so too and I'm not saying it definitely is - but I still
cannot think of practical reasons why.

> (maybe that's what you want? 'abc'.hash == :abc.hash when used
> in certain contexts? but that'd be even bigger a hack, IMHO).

No, that is the case I would like to prevent from occurring! I'm
guessing a lack of #<=> could be worked around by using #hash, #eql?
and using object_id to determine order predictably.

But I didn't really think this through and I haven't looked that
deeply into RBTree.

> (...) as I can undefine #<=> on any Hash key at a whim.

Not sure what you mean. You can undefine #hash also. Breaking things
is ok, as long as fixing them is quick and simple IMHO. Unit tests are
for great from keeping broken things from leaving one's file system.

> Why are you against subclassing Hash and coming up with a NameHash
> (or MonoKeyHash)?

It still seems like treating just the symptom. And suggests too much
duplication - the differences are just slight behavior differences.
And it feels to Java'ish for Ruby.
Maybe I feel like subclassing Hash is more work than it should be.

Consider the following as alternatives from a design perspective:

filter (ignore garbage) + sort, convert from array
Hash[{z:0, a:1, 'b' =>2}.select {|x| x.is_a?(Symbol)}.sort]
=> {:a=>1, :z=>0}

filter (validate) + sort, convert from array
Hash[{z:0, a:1, 'b' =>2}.each {|k,_|
raise ArgumentError unless k.is_a?(Symbol); k}.sort]
#=> ArgumentError

And the following:

no filtering, always sorted, no invalid state, remains RBTree
RBTree[{z:0, a:1, 'b' =>2}] #=> ArgumentError

I'd probably prefer mixins that can be included in Hash. But I'm
unsure how that would turn out. Again, refinements come to mind, but I
wonder if the current API is easily ... "refinable".

And maybe allow for optimizations.

Here is an example of what I mean:

a = {}.add_option(inserting: {order: :sort_key, duplicates: :raise})
.set_option(default: 'X')
.add_option(inserting: {|a| !a.is_a? Symbol} => :raise)

a.merge(z: 3, a: 1) #=> {a:1, z:3}
a.merge(z: 3, a: 1).sort #=> {a:1, z:3} (no sorting required)

a.keys.to_a.sort #=> :a, :z

a[:foo] #=> 'X', works like block given to Hash
a['foo'] #=> raises an ArgumentError

[].add_option(inserting: {duplicates: :merge}) #=> effectively a Set

Refinements would minimize the need for this.

The only problem I can see now with Ruby API is that people want to
override behavior and not methods - this makes subclassing more
difficult than it should be IMHO.

For example #[], #[]= and merge can add items, but you cannot just
override 'add_item' (st_insert() I believe).

> Again, while agreeing with both of the above, I still
> think coming up with NameHash is a much better solution
> than trying to make Hash outsmart the programmer.

If I could do {a: 3}.to_symhash I guess that would work out ok.

> I agree that in 99% of the cases all Strings share
> the same methods, but changing fundamental classes (like Hash)
> unfortunately is all about handling the edge cases.

If I had more control over what can be in a hash, I have a lot less
edge cases to worry about. Same with other types.

> The discussion about warning a sloppy developer is similar to
> whether '1' + 2 should work, and if so, whether it should be
> 3 or '12'.

These examples are obvious errors. For Hash compatibility I proposed
just a warning or make Hash mixing deprecated. But that assumes
restricting Hash is actually valuable - which I am unsure of.

> Note that Rails monkey-patches NilClass to make the errors on
> nil. more obvious; maybe that's the way to go?

It is why I preferred to hack rb_hash instead of subclassing. Simple
task and handles internal calls to rb_hash as well.

> The Ruby approach in this case is to have enough test coverage
> (ideally: upfront) so that the problem is quite obvious. ;)

Aggressive TDD is how I learned root cause analysis (I hope). Adding a
touch of Design by Contract may help reduce some unnecessary edge
cases without resorting to too much intelligence.

> I understand what you mean by the 'experts' remark, but I'm not sure
> that this case falls on the 'expert' side of the border; understanding
> how Hashes work is quite crucial

Sure, but not necessarily on the first page of a Ruby tutorial. With
disciplined TDD you get actually quite far IMHO without understanding
details. Refactoring is actually a good time for learning such things.

And warnings are a good way to focus more deeply on a given subject.

> Well, you want a particular kind of a Hash

More like just a particular behavior, but I'm otherwise nodding my
head reading your comments.

> But are the mistakes really that common? It should be doable
> to add guarding code to Hash#initialize if it's really needed.
> You could also argue for getting Rails' HWIA into Ruby core
> (I'm not sure whether it was proposed before or not).

Yes it was. The reasoning behind arguments for including suggested
exactly that - that mistakes are common.

> Then make Hash#initialize smarter if you need.

Hash already has a block for default values. If I could define a block
called for every implicitly added item and have the block working with
#merge, it might be a good solution.

```
a = Hash.new {|_, key| raise unless key.is_a?(Symbol)}
a['a']  #=> RuntimeError
a.merge(3 => 4) #=> {3=>4} (no error)
```

> Well, if you come up with a practical NameHash
> then I think there's a chance it'll end up in core.

It becomes more practical once it is in core ;)
Chicken and egg problem.

Proving it *is* practical may be a problem. Proving it wouldn't be the
easiest - if I knew how. And it would result in much shorter threads
on ruby-core...

> As for the {} syntax: (a) NameHash could have its own
> syntax sugar

I has too much in common with Hash - the {} syntax is one of the
reasons I started considering replacing Hash. As for alternatives,

what is left? ('a' => 3), %h{a: 3}, ... ?

> or (b) as I wrote above, you can try abusing Hash#initialize to
> create NameHash if all the keys are Strings/Symbols (but I can see
> this blowing up eventually).

Actually blowing up (if I understand you correctly) is better than
silent failure and long hours of debugging through a great big
metaprogramming jungle.

I don't want the extreme of making Ruby interpret code a mind bending
puzzle challenge (those experienced in strongly typed languages may
find this familiar), but on the other side - I don't think Ruby has
reached the sweet spot yet.

> > {'a' => 3, :a => 3}/so # s = strict, o = ordered

> Hm, I'd rather have Hash#/ as a valid method (say, for
> splitting Hashes into shards?) than a syntax construct.

Sure. That was just random brainstorming - but I don't really like it
myself. It is too specific. But then again - hashes and arrays won't
change dramatically over time.

> Interestingly, regular expressions reminded me of that fact
> that it might be convenient to have both Regexps and Strings
> as keys in the same Hash (for matching purposes). :)

That sounds crazy but you have a point. I wonder if after 10 years of
abusing hashes people will reinvent LISP as a result. Or everyone will
be configuring their favorite Ruby syntax upon installation.

> > > Let's say I have a Hash with keys being instances of People,
> > > Employees and Volunteers (with Employees and Volunteers being
> > > subclasses of People). Should they all be allowed as keys in a
> > > single MonoKeyHash or not?

I'm not sure about the actual use case, so try it with RBTree and see
for yourself.

> I'm still not sure what you mean by that (and what happens if I
> remove #<=> from a random key).

Not sure here too. Try it with RBTree.

> Say, I have a graph with nodes being various subclasses of Person
> and various subclasses of Event and I want the graph to track
> Person/Person and Person/Event relations – I really want to be able
> to use the various People and Event subclasses as keys in my Hash.

Yes, but why in the same hash? Why not two different hashes? What is
the common behavior between Event and Person? You are probably going
to iterate the graph in order to ... ?

You can always add a level of indirection, then your graph will become
more generic and reusable.

And effectively, you are hashing object contents - I'm not sure that
is really what you want. My intuition tells me such the case you
describe is refactorable.

> Hm, I think I totally disagree – #hash and #eql? are
> the public interface of Hash, and the contract is that
> anything that implements these can be used as a Hash key.

True, but I was referring to a higher level of abstraction of an assoc
array, which Hash is intended for (but not limited to):

a[b] = x   (association)
ab

At this level, both Hash, RBTree and even Array are identical.

#hash and #eql? are assoc array implementation specific. RBTree
doesn't use hashing, but serves the same purpose. The difference is
the implementation restricts the items available for keys.

> I strongly disagree here; a simple model which, in addition, is fairly
> easy to explain (it's only #hash and #eql?, really), is much better
> than a complex model carried around only for the sake of novices.

Could you say what exactly is complex? I always thought an RBTree was
simpler to understand than "Hash", which to me initially worked
"magically" and sometimes I still get hashes and identities mixed up.

By analogy, an even more stricter "hash" - Array - is even less
confusing:

a = [1,2,3]
a[0] # => 1

>> a[nil] # => TypeError (!)

And it is limited to integers specifically. And I don't have to run
'ri' or look into array.c to work it out.

We could discuss if a[nil] fails for common sense reasons or
implementation reasons. Maybe in the same way I'm not getting that in
practice, association arrays are always hash based and it is obvious
for everyone but me.

> I see where PHP ended up with 'novice-friendly' approach and
> it's awful

I totally agree.

> – and I strongly believe a simple and consistent model is actually
> more novice-friendly in the long run than wondering why '0' == false
> and false == null but '0' != null.

Handling these cases says it all (isset, isnull, etc). For me PHP is
both incredibly difficult to learn and even take. The only hope for
PHP at this point would be to start undoing "novice helping" and start
generating errors and warnings.

In case of double, a clear common syntax is the best criteria if you
ask me. With an error or warning you at least have a question to
start with.

I'm not sure what you mean by model and how in what way it is
novice-friendly. Do you mean easier to understand implementation? If
personally think that clear syntax wins in the long run. How intent
maps to code.

The underlying model can change many time and be as complicated as
possible and I wouldn't really care. Probably because I spend more
time in Ruby and none in C.

>> Even if it results in an overly complex parser and implementation,
>> I think only good will come from going out of one's way to make
>> Ruby users lives easier.

> Definitely – it's just we disagree on what is easier (in the long
> run).

I'll be quick to correct myself: easier for users to become productive
and happy (and rich?) experts delivering valuable software.

> I agree it might be useful to have NameHash for name → object
> mappings,

I would just stick with SymbolHash and have Hash for strings.

> MonoKeyHash that keeps the keys in check

This would probably be a copy of rb_hash implementation, where type
checking is both simple and cheap. Not sure how to handle objects
though.

> and/or a #<=>-based ComparisonHash

meaning basically to help RBTree get adopted - maybe with a nicer,
less scary name

> and I encourage you to implement them and push for them
> to be included in the core

I still think I lack the necessary understanding, so I'll spend some
more time researching actual hash usage along with external libraries
in this area (AS, facets, extlib).

> I'm simply very grateful for the extremely well though-out and
> versatile Hash we have now and I'd rather it's not made more
> complicated (or dumbed-down) for the sake of a (granted, popular)
> single use-case.

Could you explain that using Array and RBTree as examples?
Is Array a dumbed-down hash? Is RBTree overcomplicated?

> I really like this discussion as well! Thanks for bringing this up.

Thanks for your time. Thanks to you I made a lot of new distinctions
between Ruby core concepts! Initially I wanted to contribute, but I
ended with just increasing my own knowledge for now.

If I find some interesting patterns how Hash (or Ruby in general) is
(ab)used, I'll post them as a new thread with possible ways Ruby could
help simplify/fix things.

P.S. Looking at my reply ... I'm sure even the mail server deserves a
break after this.

--
Cezary Baginski

#17 - 08/15/2011 11:54 AM - wardrop (Tom Wardrop)

Sounds like everyone wants to get down to the root causes here, which is great, so I'll jump on the band wagon. As I see it, all these problems stem from the fact that symbols and strings are too conceptually similar, in fact they're the same when you really think about. A symbol IS a string. You could say that Strings ARE string data, where as symbols REPRESENT string data, in the same way that "some string" is the string object, and lvar in (lvar = "some string") represents that string object. The difference is that 'some string' == lvar is true, where as 'some string' == :'some string' is not. Symbols are like variables, except that the data the symbol (variable) references is also the name of the variable, thus in my opinion, 'some string' is equal :'some string'. I see no reason to make such a rigid distinction between strings and symbols.

In Ruby, the only differences between strings and symbols are their semantic meaning, and how they're handled internally in regards to memory. The fact remains though that they're both strings. This is why people get caught out with Hashes, and why HWIA is so handy. It's not that people don't understand the differences, but it's just that their difference are not significant enough, thus one is prone to interchange the two which is what causes many of those hard to track down bugs. This is especially the case when accessing hases with dynamic keys, such as from a configuration file or some other external data source such as a database or user input.

Thus as I see it, the simplest and most natural solution would be to make strings and symbols equal. They can both remain as different Ruby classes, in fact they're implementations can remain exactly the same, you'd only have to change methods such as #eql? on the String and Symbol classes to fix the majority of String/Symbol related problems (so 'some string' == :'some string'). Though to do it properly and with respect to the principal of least surprise, the Symbol class would have to support all string methods, and symbols would have to be implicitly cast to strings when used in the context of a string - if you didn't do this you'd probably have more problems, e.g.

def say_hello(name)
"Hello #{name.upcase}"
end
say_hello :John #=> "Hello JOHN"

That result of that example is completely desireable. Can you imagine how stress-free Ruby programming would be if Symbols were treated more like Strings?

Can anyone tell me what the advantage is of making such a rigid distiction between Symbols and Strings? Can anyone think of an example where making 'some string' == :'some string' would cause undesireable behaviour in code, assuming the Symbol class implements all the methods of the String class as well as supporting implicit conversion?

#18 - 08/15/2011 11:59 AM - wardrop (Tom Wardrop)

So basically what I proposed in my previous message, is to some extent, get rid of Symbols. You'd still have the :symbol syntax, and symbols would still be handled differently by the garbage collector (as they currently are), but they would otherwise be a literal named string, like a shorthand for mystring = 'mystring'.

Going back to the original feature suggestion regarding this hash synax: {'key': 'value'}, with my proposal it wouldn't matter whether 'key' was a symbol or a string. The decision would come down to what makes the most sense in terms of memory and the garbage collector.

Side note: I was meant to add to my previous message that maybe Symbol should sub-class String if my proposal were to be implemented?

#19 - 08/15/2011 12:10 PM - cwgem (Chris White)

The problem is that symbols are used for more than just keys in hashes. They're also a key player in meta programming, used for cases like referring to methods and instance / class variables. I'm a bit worried that isolating symbols to the Hash use case will create overhead in their usage as method / variable references (to see the potential impact, check Symbol.all_symbols on a simple program with just a puts call).

#20 - 08/15/2011 01:18 PM - wardrop (Tom Wardrop)

Chris, I'm assuming you're responding to what I wrote, in which case I'd like to reassure you that what I propose preserves symbols, so Symbol.all_symbols will still show the same list of symbols as it would without my proposed changes. All legacy code should still work. All I'm suggesting is to take away the barriers that divide symbols from strings to make them interchangeable in-code, though still semantically different. You'll still have symbols and you'll still have strings, but symbol's will regarded as a type of string (either as a sub-class of String, or where String and Symbol both sub-class a GenericString or CharacterSequence); like an immutable version of String who's name double's as the data it represents. You can't change the value without changing the name.

#21 - 08/15/2011 11:25 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Tom, I agree with you that :something == 'something' returning true would be a good thing in a "clean code" context, but since I don't know about the internal implementation details that originated symbols in the first place, I don't know if :something == 'something' would be as fast as current implementation if it's changed to allow symbols to be compared to strings seamlessly. That's why I suggested the new ### operator for doing this kind of comparison, but maybe you're right and it would be just much simpler if :something == 'something'.

#22 - 08/16/2011 03:18 AM - drbrain (Eric Hodel)

- File bm.rb added

How do you determine (or who determines) "what makes the most sense in terms of memory and the garbage collector"?

For a Hash with keys that are all the same class (String or Symbol) the best performance comes from matching the Hash's keys to the input type's key. Attached is a benchmark illustrating this, here is the output: http://paste.segment7.net/mu.html

More simply, if you have String input (say from a user, or as parameters from an HTTP request) your Hash should have String keys. If you have internal fixed values you're looking up you should use Symbols for both.

By using the simple guideline of "make the Hash keys match the key's class" we don't need any changes to Ruby to do what makes the most sense.

PS: This discussion seems to be a bit off-topic from the one in the subject line.

PPS: If you think Ruby should determine "what makes the most sense in terms of memory and the garbage collector" I think you've got a PhD thesis on your hands.

#23 - 08/16/2011 07:07 AM - wardrop (Tom Wardrop)

=begin
More simply, if you have String input (say from a user, or as parameters from an HTTP request) your Hash should have String keys. If you have internal fixed values you're looking up you should use Symbols for both.
=end

It's rarely that simple though. I just have to think of one example where using the Sinatra frameworks template helper, #erb. It accepts an optional hash of local variables. It expects symbols for variable names which isn't such a bad decision, but in this particular circumstance I was calling templates dynamically based on URL parameters (doing string manipulation at the same time). Of course this lead to me wondering why the local variables I set weren't coming though; of course I was using Strings instead of Symbols, but it highlights the expectation of users for Symbol's and String's to be somewhat interchangeable when being used in-code.

=begin
How do you determine (or who determines) "what makes the most sense in terms of memory and the garbage collector"?
=end

If Ruby could somehow do that, then great, but otherwise what we have now with the differences between how strings and symbols are handled internally, is good enough I think.

#24 - 08/16/2011 07:07 AM - wardrop (Tom Wardrop)

> More simply, if you have String input (say from a user, or as parameters from an HTTP request) your Hash should have String keys. If you have internal fixed values you're looking up you > should use Symbols for both.

It's rarely that simple though. I just have to think of one example where using the Sinatra frameworks template helper, #erb. It accepts an optional hash of local variables. It expects symbols for variable names which isn't such a bad decision, but in this particular circumstance I was calling templates dynamically based on URL parameters (doing string manipulation at the same time). Of course this lead to me wondering why the local variables I set weren't coming though; of course I was using Strings instead of Symbols, but it highlights the expectation of users for Symbol's and String's to be somewhat interchangeable when being used in-code.

> How do you determine (or who determines) "what makes the most sense in terms of memory and the garbage collector"?

If Ruby could somehow do that, then great, but otherwise what we have now with the differences between how strings and symbols are handled internally, is good enough I think.

As for your benchmark, if you're trying to highlight the performance penalties that come as a result of my suggestion (which makes Strings and Symbol interchangeable), then I would not imagine any type casting would be required. Though I don't know for sure, I would imagine Symbol's are stored internally as strings, thus the Ruby runtime could optimise the comparison operator for these two types by having it look at the actual string data already in memory, instead of type casting which I would imagine copies and moves data in memory.

#25 - 08/16/2011 05:53 PM - nobu (Nobuyoshi Nakada)

Hi,

At Mon, 15 Aug 2011 11:59:34 +0900,
Tom Wardrop wrote in [ruby-core:38957]:

> So basically what I proposed in my previous message, is to
> some extent, get rid of Symbols. You'd still have the :symbol
> syntax, and symbols would still be handled differently by the
> garbage collector (as they currently are), but they would
> otherwise be a literal named string, like a shorthand for
> mystring = 'mystring'.

First of all, you shouldn't mix different proposals. It's one of the
fastest way to confuse the discussion and make all unacceptable.

Second, that proposal, "Symbol as a subclass of String", is an
abolished feature. Though I can't remember the reason now,
there should be the rationale to thrown it away. You need to
disprove it to make your proposal accepted.

--
Nobu Nakada

#26 - 08/16/2011 06:53 PM - sdsykes (Stephen Sykes)

For info, look at
http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-core/9251 and
the surrounding thread.

On Tue, Aug 16, 2011 at 11:43 AM, Nobuyoshi Nakada nobu@ruby-lang.org wrote:

> Second, that proposal, "Symbol as a subclass of String", is an
> abolished feature. Â Though I can't remember the reason now,
> there should be the rationale to thrown it away. Â You need to
> disprove it to make your proposal accepted.

--
Nobu Nakada

#27 - 08/16/2011 09:20 PM - kstephens (Kurt Stephens)

The most important distinction between Symbols and Strings: a Symbol has the property where its identity *is* its lexical equality where as a String's identity is distinct from its equality. The difference has important performance and semantic benefits. Altering Symbol#==(String) to return true where :'foo'.to_s == 'foo' removes this difference.

As its namesake implies, a Symbol is an atomic placeholder for a concept, a mnemonic device, where as String is a mutable sequence of characters. Symbols represent concepts; Strings hold data.

I agree that Symbols and Strings should cooperate, but altering Symbol#== is not appropriate. Perhaps String#+(Symbol) and String#<<(Symbol) should behave as one might expect.

#28 - 08/17/2011 11:49 AM - wardrop (Tom Wardrop)

@Nobuyoshi Nakada, it does seem that my original suggestion for the Hash syntax cannot be implemented due to the String/Symbol incompatibilities, thus how this discussion has panned out is quite natural, though I think it would be better if this discussion continue in another thread.

Does anyone think it's worth opening up such a discussion on whether Strings and Symbols should be more compatible? If the idea of Symbol inheriting from String has been discussed before, that's ok, because that's something implementation specific, where as what I'm talking about is more of a general interoperability concept for working with Strings and Symbols.

#29 - 03/25/2012 04:02 PM - mame (Yusuke Endoh)

- Status changed from Open to Assigned

- Assignee set to nobu (Nobuyoshi Nakada)

#30 - 08/22/2012 04:05 AM - alexch (Alex Chaffee)

Symbols *should* be frozen strings. Any performance problems can be solved by the interpreter. The reason the feature was "abolished" (rolled back) had to do with existing libraries, especially those using "case" statements to compare classes.

But the case statement gotcha can be solved by always putting "when Symbol" above "when String". This is no worse than some other workarounds for otherwise useful language features, such as "always use self when calling a setter on the current instance".

See more discussion (including a link to Matz' 2006 comments) at http://stackoverflow.com/questions/11085564/why-are-symbols-not-frozen-strings

If "class Symbol < String" then a HUGE source of confusion and errors will be removed from the language, without sacrificing any of the semantic power of symbols.

#31 - 08/23/2012 04:17 PM - nobu (Nobuyoshi Nakada)

Symbols are not strings.
Symbols are symbols, unique objects with fixed names.

#32 - 08/27/2012 03:25 AM - alexch (Alex Chaffee)

nobu, your statement is true, and if symbols are *implemented* as strings then it will still be true. I promise! We can have "class Symbol < String" and they will still be unique objects with fixed names.

Implementing symbols as frozen strings works for Smalltalk, so why couldn't it work for Ruby? From the Smalltalk documentation, "Symbols are Strings that are represented uniquely." That doesn't sound very controversial, does it?

Matz argued the case here -- http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-core/9192 -- and what he calls a "side effect" is actually solving a major problem with the language. It is very surprising to newcomers (and annoying to experts) that string keys in a hash do not match symbol keys, and that JSON and XML import/export don't work right for symbols, and that some libraries expect strings while others expect symbols, and that basically Ruby makes you think really hard about whether this *particular* series of characters is and/or should be a symbol or a string.

I make this case, and refute some objections, in this StackOverflow question:
http://stackoverflow.com/questions/11085564/why-are-symbols-not-frozen-strings

#33 - 08/27/2012 06:59 AM - trans (Thomas Sawyer)

=begin
Silly mixin name aside...

```
module Stringy
# ...
end
```

class String
include Stringy
end

class Symbol
include Stringy
end
=end

#34 - 11/20/2012 11:24 PM - mame (Yusuke Endoh)

- Target version set to 2.6

#35 - 11/20/2012 11:40 PM - matz (Yukihiro Matsumoto)

- Status changed from Assigned to Rejected

The discussion has gone away in the wind without making any consensus.
So I marked this 'rejected'. My point is clearly stated in the first comment.

Besides that, we have already tried making Symbols a subclass of String, or making Symbols behave like Strings, and it didn't work out.

Matz.

#36 - 11/22/2012 05:50 PM - judofyr (Magnus Holm)

matz (Yukihiro Matsumoto) wrote:

> Besides that, we have already tried making Symbols a subclass of String, or making Symbols behave like Strings, and it didn't work out.

Do you remember why it didn't work out to have Symbols behave like Strings? Any links?

Files

bm.rb	2.54 KB	08/16/2011	drbrain (Eric Hodel)