Ruby - Feature #11882

Map or NamedMap

12/26/2015 08:28 PM - hcatlin (Hampton Catlin)

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

Description

Hash is one of the best features of Ruby. I remember being so pleased when I first learned Ruby to find out that *anything* could be a key and that you could do some really clever things with scripts, if you key of non-traditional elements.

However, like many people, 10 years into using Ruby, I still am writing code to cast around symbols to strings and strings to symbols, just to use Hash as a more traditional dictionary, keyed with string-like values. And, explaining to a junior programmers why they broke the code by using a key of a different type... it's not the most elegant thing to have to explain over and over.

Several proposals exist for how to deal with this, and all have been rejected... however it doesn't seem like it's for essential reasons, more technical or syntactic issues. Coming up with syntax is something I quite enjoy (Sass/Haml), so I thought I'd make a pitch for it.

Requirements:

1. Doesn't break existing code
2. Doesn't totally destroy the parser
3. Seems simple to differentiate
4. Clear upgrade path

My proposal is to introduce an entirely different type of Hash, called a Map (or NamedMap, if that's too ambiguous), that requires a string-like key. There are no other types of keys allowed on this Hash, other than either strings or symbols. Internally, each key would be considered a symbol only.

```
map = Map.new(a: 2, b: 3)
map["a"] #=> 2
map[:a] #=> 2
```

Already, we're better than HashWithIndifferentAccess, as it's clearly a bit easier to type. ;)

What about a literal syntax?

```
map = {{a: 2}}
empty_map = {{}}
```

As far as I can tell in the Ruby-syntax style, this should be pretty easy to distinguish syntactically from both a regular hash literal and a block. Further, as almost every method's option hash is string-keyed, you could easily define this.

```
def my _method(arg1, options = {{}})
end
```

Immediately, you could deal with your options hash, and not have to stress about if the end user has passed in strings or symbols into the options.

It would be trivial to create a Map polyfill for most libraries to start using the non-literal version right away, as it would basically be HashWithIndifferentAccess, except we need to guarantee that the keys are string-like.

So, to sum up, we avoid the 'breaking other people's existing code' by introducing a new data-type, the literal syntax (I think) should be fairly easy to implement, and it makes a very natural keyed data object (e.g. Javascript Objects) and brings that to Ruby.

History

#1 - 12/26/2015 08:57 PM - hcatlin (Hampton Catlin)

Sorry, forgot to add that the literal syntax would support all currently supported string-key variations that Hash does...

```
{{"a" => true}}
{{a: true}}
```

```
{{:a => true}}
```

Map.new would merge in standard Hashes string keys, using a sort of overwrite last-in system, thanks to the ordered nature of hashes.

```
m = Map.new({a: false, "a" => true})
m[:a] #=> true
```

An attempt to use same keys in the literal syntax would do the same thing. Optionally, you could raise a warning, as it works now with hashes with the same key.

```
{{a: 1, a: 2}} # => {{a: 2}}
## warning: key :a is duplicated and overwritten on line 1
```

As you can see in my examples, I'm using the post-fixed colon in the inspected versions, which I think is what should be done, but invalid symbols can be shown as the following

```
{{"a" => 2, "-" => 3}} #=> {{a: 2, "-": 3}}
```

Very similar to how we handle things currently with Hashes and those keys, but tending towards displaying them with the post-fix, giving a visual clue to whether it's a Hash or a Map.

Also, Maps should inherit from Hash.

```
{{}}.is_a? Hash #=> true
{}.is_a? Map #=> false
{{}}.to_h.is_a? Map #=> false


# the following two are equivalent
{}.to_map.is_a? Map #=> true
Map.new({}).is_a? Map #=> true
```

That's all I can think of for the moment.

#2 - 12/27/2015 10:13 PM - hcatlin (Hampton Catlin)

It was pointed out to me, that there is an ambiguity in the following case.

```
array.method {{a: 1}}
# could be
array.method { {a: 1} }
#or
#array.method(Map.new({a: 1}))
```

To be fair, I feel that this would be an extremely, extremely, extremely rare occurrence, though it is possible. The conflict won't happen if there are *any* named arguments passed into the block, aka...

```
array.map { |arg| {v: arg} }
```

The case where you are yielding a block and it's single-line and the value is a hash that is non-dynamic, and no spaces were placed... seems highly unlikely, though, again, possible. I can't think of any examples where you would implement a yielding block that would be a simple hash coming back, only. The cases where this might be present in code would be something like an options hash, but I've never seen that be implemented as a block, when Ruby has first-class support for named arguments as an option hash in the method signature. Aka...

```
configure(timeout: @timeout) # passing config via an option
configure {{timeout: @timeout}} # passing in config via a block with a simple value
```

In these edge cases, I would say that we interpret it as a Map, and any errors that occur would be fairly obvious (wrong number of arguments, etc). It does mean that technically it's not 100% backwards compatible, but if this edge case is the one sacrifice for healing one of Ruby's most frustrating code idioms, it seems a fair trade to me.

#3 - 12/28/2015 01:42 AM - jeremyevans0 (Jeremy Evans)

Hampton Catlin wrote:

> Requirements:
>
> 1. Doesn't break existing code
> 2. Doesn't totally destroy the parser
> 3. Seems simple to differentiate
> 4. Clear upgrade path

The proposal to treat meth {{k: v}} as meth(Map.new(k: v)) instead of the current behavior of meth do {k: v} end clearly breaks existing code, and a lot more of it than you probably expect, which definitely does not fulfill your first requirement.

I think a Map/NamedMap/SymbolHash data structure is useful in some cases, but I don't think it belongs in core, or that it should have syntax support. Symbols and strings are different and should be used for different things. This data structure may appear helpful to newcomers that don't understand the difference between symbols and strings, but recommending this data structure to them is a short term gain with long term costs, as it will take them longer to understand the difference between the two.

#4 - 12/28/2015 03:30 AM - hcatlin (Hampton Catlin)

That's such a kind of.... odd opinion to me. I've been doing ruby for 10 years, and I still have to write fairly obvious handling code when building programming interfaces and libraries... over and over again. I guess I just see 95% of Hash usage being made up of instances where symbols and strings are treated equally, and compensating code is required at every step to ensure that. I find it odd that this is considered an edge-case.

#5 - 12/28/2015 03:56 AM - shevegen (Robert A. Heiler)

I concur with Hampton Catlin for the most part (save for the
name "Map", that is not a good name IMO).

I assume that exposing Symbols was never directly the primary
design goal to be had; it came to be, and please feel free to
correct me if I am incorrect, mostly because using Symbols would be
faster than Strings and I assume that especially rails had pushed
for it due to having found other problems lateron with their big
code base. (The rails users also originated HashWithIndifferentAccess
as far as I am aware, but that is an abomination to type. I would
never use such a long name, but I mentioned it because it also
taps into the question of "strings versus symbols" situation,
so this is a recurring theme.)

I do not have any particular route to recommend or partake in,
so I can't pick from the above selection nor can I really add
any myself, but I would like to say that I concur with Hampton
Catlin even if I am coming from a somewhat different angle.

To me, as a newbie (I still am a newbie really even after years,
but I am like a somewhat "competent" newbie by now), I don't really
want to have to think about whether I need to use Strings or
Symbols at all as identifiers. Don't get me wrong there, I actually
love Symbols for a completely odd reason - I can use :foo rather
than 'foo' so I can save one character! (I could also use ?foo
but I find ? ugly and we have method names with ? and also
ternary, I don't want to use so many ?; thankfully we have the
lonely-person-stares-at-dot operator rather than another ?
again)

But, from a conceptual point of view, when I am writing some
ruby scripts, having to wonder or ponder about whether to use
Symbols or Strings, I feel it adds a small layer of complexity
that should not really exist in the first place (from the
"human user perspective" that is; it's fine as an internal
part of ruby).

The name "Map" is a bit weird though. Reason I dislike that
name is because we already have .map and my brain refers
to .map in a completely different way (aka, "apply on each
element" mostly). And if I were to see Map.map I would want
to not like it.

> I guess I just see 95% of Hash usage being made up of instances
> where symbols and strings are treated equally, and compensating
> code is required at every step to ensure that. I find it odd
> that this is considered an edge-case.

Oh I know how you feel there. :)

I remember that in my cookbooks project, I am saving every program
as a yaml file, so all programs will be described via yaml. So
this is a "yaml database", ok flat files.

Then, when I wanted to access the giant hash that is created
by loading all yaml files, I had to wonder whether I want to
keep them saved as a string or as a symbol (the key identifiers,
e. g. "vim" versus :vim to access all information pertaining
to vim).

I do not remember which way I went offhand but whatever it was,

I decided that I will convert input to the target format anyway
(so either I am doing a .to_s or a .to_sym but it's an extra
step of thinking required that really does not help me a lot;
on the other hand, it also isn't something that is a huge
issue to me either, it's more a peculiarity).

Ruby 3.0 is in the pipeline!

#6 - 12/28/2015 07:02 AM - phluid61 (Matthew Kerwin)

I used to have a strong opinion about the difference between String and Symbol, but ever since Symbol GC (and the resulting distinction between GC-able Symbols and "real" Symbols) I don't care so much. Now people can easily create what they call a 'Symbol' but think of as a less-useful, interned String; and it's safe because it can be GC'd. This fits what they've done all along, even though it used to be "wrong."

That said, I think the {{}} syntax doesn't work. Perhaps, borrowing from numeric 'r' and 'i' suffixes, a suffix on the Hash literal could work? Like: {}m (mnemonic: "map"), or {}s (mnemonic: "safe")

#7 - 12/28/2015 06:16 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

For a long time I've been saying Symbols are the biggest disadvantage of Ruby, so you can count on me with any efforts in trying to make symbols and strings interchangeable even if only for hashes.

I've tried many suggestions, like making the symbols syntax generate frozen strings, or making their comparison work string-wise and even tried to make Hash work the way you described without the need to create new classes for that. I've also wanted that the short-style syntax for hashes used strings as keys instead of symbols. This symbol vs string thing is a source of lots of bugs and takes us a long time to search for other use cases to understand whether we should use symbols or strings as keys while we only care about the name of the keys...

If it's not possible to make this transparent out of the box, at least I'd like Ruby to support some short syntax for them, like {}i (i as a shortcut to indifferent here), but it means we wouldn't be able to call methods like this:

do_something at: Date.today + 1

Even worse, if 'at' is a named argument, than if a hash is passed to do_something, then the keys have to be symbols (and the argument must be a Hash).

As I said before, I'd prefer a more general and transparent option even if it's not backwards-compatible (although very unlikely to introduce real problems) and cleaner than something like what has been proposed here. But I still prefer a solution like this than having to worry about the symbols vs strings dilema when I only care about named identifiers.

#8 - 01/30/2016 04:08 AM - avit (Andrew Vit)

Robert A. Heiler wrote:

> I concur with Hampton Catlin for the most part (save for the
> name "Map", that is not a good name IMO).

"Dict" is another possible name.

How would this proposal handle other key types? Should obj[1] = 'a' convert using to_s or raise an error?

#9 - 01/17/2021 01:31 PM - hcatlin (Hampton Catlin)

I agree that { {a: 1} } and {{a: 1}} in my suggested syntax is difficult to disambiguate. The magic inside the Ruby parser that can separate the differences as they are today must be very mind bogglingly difficult. If it were possible, it would be pretty magical though.

Another option for a syntax here is to have modifiers on the literal hash syntax:

```
my_map = ^{host: "http://google.com}

def my_method(options = ^{})
end
```

I do think to make this really useful for developers you have to have a way to define a method signature's trailing-options-hash syntax, as then you can remove any coercion or type checking from your methods and better define your expectations.

#10 - 04/03/2024 03:50 AM - hsbt (Hiroshi SHIBATA)

- Status changed from Open to Assigned