Ruby master - Feature #14580

Hash#store accepts a block

03/06/2018 03:42 PM - Soilent (Konstantin x)

Status:	Open	
Priority:	Normal	
Assignee:		
Target version:		

Description

Given a hash

```
hash = { a: 2 }
```

I want to update a single value in the hash:

```
hash[:a] = hash[:a] + 42
hash[:a] #=> 44
```

But instead, I would like to have a method that yields the current value for a given key and associates the block result with the key (similar to Hash#update). I think that Hash#store can be extended to support a block arg.

```
hash.store(:a) { |val| val + 42 }
hash[:a] #=> 44
```

Or it can be something like this:

```
hash.transform_values(:a, :b) { |val| val + 42 }
hash[:a] #=> 44
```

History

#1 - 03/06/2018 04:09 PM - Soilent (Konstantin x)

- Description updated

#2 - 03/07/2018 03:39 PM - shevegen (Robert A. Heiler)

If I understood your proposal correctly then you want an
additional way to update an existing value in a hash, correct?

So the comparable syntax parts would be:

```
hash[:a] = hash[:a] + 42
```

versus

```
hash.store(:a) { |val| val + 42 }
```

right?

So, if this is correct, then as I understand it, the major
point of your proposal, and benefit, is that you omit querying
the old value in the sedond variant, since you just operate
on the block variable called "val" in your case.

If this is indeed the case, and that is your proposal, then I
think I understand what you mean, feature-wise. In this case
you skip the step where you query the old value explicitely
and just tap into the block value for making a modification.

I think this is ok.

It follows my own "philosophy" of "do not make me think" or
"make me think less". :D

I have no idea how matz feels about it; perhaps someone could
suggest it in the upcoming ruby developer meeting in ~a

week or so.

The current documentation for Hash#store can be found at:

https://docs.ruby-lang.org/en/2.5.0/Hash.html#method-i-store

#3 - 03/07/2018 03:44 PM - Eregon (Benoit Daloze)

What should happen if the given key doesn't exist in Hash?
This looks like a compute-if-present operation.

#4 - 03/07/2018 04:19 PM - Hanmac (Hans Mackowiak)

```
hash.transform_values(:a, :b) { |val| val + 42 }
hash[:a] #=> 44
```

what about the b key? should it:
a) throw exception
b) gives nil to the block ? which your code would be an NoMethod + for nil
c) will be skipped

#5 - 03/07/2018 04:21 PM - Soilent (Konstantin x)

Hi Robert,

Thank you for your reply. You understood everything correctly.

Also, I might be wrong, but it seems to me that in the following case

```
hash[:a] = hash[:a] + 42
```

Ruby VM will look up the key twice. The proposed method should eliminate the second lookup in this case.

#6 - 03/07/2018 04:24 PM - zverok (Victor Shepelev)

Maybe a bit off-topic, but I experimented with same ideas in hm gem. It allows code like this:

```
Hm(hash)
  .transform_values(:a) { |val| val + 42 }
  .to_h
```

After trying several approaches in production, the design decision I've made about not found keys is simply ignore them.

#7 - 03/07/2018 04:25 PM - Soilent (Konstantin x)

Eregon (Benoit Daloze) wrote:

> What should happen if the given key doesn't exist in Hash?
> This looks like a compute-if-present operation.

Good question, thank you. I think, the result of default_proc or the default value should be yielded.

#8 - 03/07/2018 04:40 PM - Soilent (Konstantin x)

Hanmac (Hans Mackowiak) wrote:

> ```
> hash.transform_values(:a, :b) { |val| val + 42 }
> hash[:a] #=> 44
> ```
>
> what about the b key? should it:
> a) throw exception
> b) gives nil to the block ? which your code would be an NoMethod + for nil
> c) will be skipped

Thanks for the question.

I think that hash.store(:b) should yield the default value if the key does not exist, i.e. option b.
But in case of hash.transform_values(:a, :b), when we want to update several keys, it is best to skip non-existent keys (option c)

#9 - 03/08/2018 04:45 AM - sawa (Tsuyoshi Sawada)

Why not write hash[:a]+= 42?

#10 - 03/08/2018 08:44 AM - Soilent (Konstantin x)

sawa (Tsuyoshi Sawada) wrote:

> Why not write hash[:a]+= 42?

Good point, but this works only for arithmetic operators (and also does 2 key lookups). Consider another example hash.store(:time) { |ts| Time.parse(ts) }

#11 - 03/08/2018 09:31 AM - mame (Yusuke Endoh)

I think it is not so simple to optimize the double lookup by this API. Consider:

```
hash.store(:a) {|val| 10000.times {|n| hash[n] = true }; val + 42 }
```

or:

```
hash.store(:a) {|val| hash.rehash; val + 42 }
```

We need to keep a flag if rehash occurred or not during the block executed.

#12 - 03/08/2018 10:24 AM - Eregon (Benoit Daloze)

Soilent (Konstantin x) wrote:

> Consider another example hash.store(:time) { |ts| Time.parse(ts) }

That looks weird to me.
Either the Hash is caching String to Time, and then it should use

```
Hash.new { |h,k| h[k] = Time.parse(k) }
```

or it contains other data and then there seems to be little reason to first store a String for key :time and then only later parse it to a Time instance.

#13 - 03/08/2018 10:29 AM - Eregon (Benoit Daloze)

mame (Yusuke Endoh) wrote:

> We need to keep a flag if rehash occurred or not during the block executed.

Also, what should happen with:

```
hash.store(:a) { |v| hash.delete(:a); v + 42 }
```

"store" starts to feel to me like the wrong name, it sounds more like an "update" of an existing key (but Hash#update is an alias of Hash#merge!).

#14 - 03/08/2018 11:05 AM - Soilent (Konstantin x)

mame (Yusuke Endoh) wrote:

> I think it is not so simple to optimize the double lookup by this API. Consider:
>
> ```
> hash.store(:a) {|val| 10000.times {|n| hash[n] = true }; val + 42 }
> ```
>
> or:
>
> ```
> hash.store(:a) {|val| hash.rehash; val + 42 }
> ```
>
> We need to keep a flag if rehash occurred or not during the block executed.

I think that an exception should be thrown if the block modifies the hash.

#15 - 03/08/2018 11:59 AM - Soilent (Konstantin x)

Eregon (Benoit Daloze) wrote:

> Soilent (Konstantin x) wrote:
>
> > Consider another example hash.store(:time) { |ts| Time.parse(ts) }
>
> That looks weird to me.

> Either the Hash is caching String to Time, and then it should use
>
> ```
> Hash.new { |h,k| h[k] = Time.parse(k) }
> ```
>
> or it contains other data and then there seems to be little reason to first store a String for key :time and then only later parse it to a Time instance.

I see your point, but the example was not about String to Time caching. Let's say you receive an HTTP POST request with the body timestamp=2018-03-08T11:24:44Z&temperature=27. You might want to validate the request and store it in a database:

```
begin
  params.store(:temperature) { |tm| Integer(tm) }
  params.store(:timestamp) { |ts| Time.parse(ts) }
rescue ArgumentError => err
  # Handle invalid request
end

# Do something with `params`

db[:events].insert(params)
```

I think that Hash#store with a block arg looks quite natural with the rest of the methods from the Hash API.