Ruby master - Feature #18934
Proposal: Introduce method results memoization API in the core
07/21/2022 05:01 PM - zverok (Victor Shepelev)

<table>
<thead>
<tr>
<th>Status:</th>
<th>Rejected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority:</td>
<td>Normal</td>
</tr>
<tr>
<td>Assignee:</td>
<td></td>
</tr>
<tr>
<td>Target version:</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

**Abstract:** I propose to introduce a simple core API for memoizing argument-less method return values.

```ruby
class Test
def foo
  puts "call!"
  5
end

  memoized :foo
end
```

```ruby
o = Test.new
o.foo # prints "call!", returns 5
o.foo # returns 5 immediately
```

The full proposal is below.

**Intro**

For further reasoning, I'll be using the following class. It is simplified/invented for demonstrative purposes, so I'd prefer to discuss problems/solutions described in general and not focus on “you could rewrite this class that way”.

```ruby
class Sentence
  attr_reader :text

  def initialize(text) = @text = text

  def tokens() = SomeTokenizer.new(text).call

  def size() = tokens.size
  def empty?() = tokens.all?(&:whitespace?)

  def words() = tokens.select(&:word?)
  def service?() = words.empty?
end
```

**The problem**

The class above is nice, clean, and easy to read. The problem with it is efficiency: if we imagine that SomeTokenizer#call is not cheap (it probably isn't), then creating a few sentences and then processing them with some algorithm will be—with the demonstrated definition of the class—much less efficient then it could be. Every statement like...

```ruby
many_sentences.reject(&:empty?).select { _1.words.include?('Ruby') }.map { _1.words.count / _1.tokens.count }
```

...is much less efficient than it "intuitively" should be because tokenization happens again and again.

Caching just tokens would probably not be enough for a complex algorithm working with some notable amounts of data: whitespace? and word? might also be non-trivial; but even trivial methods like select and empty? when needlessly repeated thousands of times, would be visible in profiling.

So, can we stop recalculating them constantly?
Existing solutions

Just pre-calculate everything in initialize

We could create a lot of instance variables in initialize:

```ruby
def initialize(text)
  @text = text
  @tokens = SomeTokenizer.new(@text).call
  @size = @tokens.size
  @empty = @tokens.all?(&:whitespace?)
  @words = @tokens.select(&:word?)
  @service = @words.empty?
end
```

It will work, of course, but it loses nice visibility of what's objects main data and what is derivative; and it is more code (now we need to define attr_readers and predicate methods for all of that!). And adding every new small service method (like def question?() = tokens.last&.text == '?') would require rethinking "Is it efficient enough to be a method, or should I add one more instance var"?

||= idiom

The common idiom for caching is ||=:

```ruby
def words() = @words ||= tokens.select(&:word?)
```

It has its drawbacks, though:

1. doesn't suit methods that can return false or nil (like service?)
2. harder to use with methods that need several statements to calculate the end result
3. it mixes the concerns of "how it is calculated" and "it is memoized" (looking at the method's code, you don't immediately know if the variable is used only for memoization, or it could've been set elsewhere, and here we just providing default value)
4. it pollutes the object's data representation

1-2 is typically advised to solve with a less elegant but futureproof solution (which also makes it impossible to define in one-line methods, even if the main code is short):

```ruby
def empty?
  return @empty if defined?(@empty)
  @empty = tokens.all?(&:whitespace?)
end
```

About 4: while using this solution, we'll have a lot of instance vars (that are derivative and logically not the part of object's state) now visible in default #inspect and serialization:

```ruby
s = Sentence.new('Ruby is cool')
p s
# #<Sentence:0x0000000000000000 @text="Ruby is cool">
puts s.to_yaml
  # --- !ruby/object:Sentence
  # text: Ruby is cool
p s.empty?
  # false
p s
  # #<Sentence:0x0000000000000000 @text="Ruby is cool", @empty=false>
puts s.to_yaml
  # --- !ruby/object:Sentence
  # text: Ruby is cool
  # empty: false
```

Existing memoization libraries

There are several well-known memoization libraries out there, to name a couple: old and reliable memoist, new and efficient memo_wise.

They solve problems 1-3 of ||=, and also add several cool features (like argument-dependent memoization) with a macro (this is
memo_wise, memoist behaves the same, just different macro name):

```ruby
class Sentence
  prepend MemoWise
  # ...
  memo_wise def empty?() = tokens.all?(&:whitespace?)
end
```

Now we have a nice declarative and decoupled statement "it is memoized", which also supports booleans and nils and multi-statement methods.

The problem of "detail leaking" isn't solved, though:

```ruby
p s.empty?
# false
p s
# #<Sentence:0x00007f0f474eb418 @_memo_wise={:empty?=>false}, @text="Ruby is cool">
puts s.to_yaml
# --- !ruby/object:Sentence
#   _memo_wise:
#     :empty?: false
#   :text: Ruby is cool
```

Also, using third-party gems introduces a few new problems:

1. Performance penalty. However well it is optimized, Ruby-land "redefine method, then check it is there, then calculate" has not zero overhead.
2. Dependency penalty. If the memoizing gem is not in the project yet, it is a decision whether to introduce it or not and for small no-dependencies gems or for the strictly-controlled codebase, it might be a problem. Also, doing prepend MemoWise (or extend Memoist) is another point where the question "should I introduce this dependency?" arises (in a small class with exactly one method to memoize, for example!)

**Feature proposal & Design decisions**

I propose to introduce the Module#memoized(*symbols) method in the core, implemented in C.

1. Name: memoize is a typical name that the community is used to. I want the new method to look uniform with other existing "macros" that have wording suitable for the phrase "this method is {word}": private or module_function; that's why I propose the name memoized
2. I believe that the memoisation should be fully opaque: not visible on #inspect or serialization; no settings or API to interact with the internal state of memoization.
3. Only argument-less methods are memoizable, memoize def foo(any, args) should raise an exception
4. (Not sure about that one) "Memoised" state of the method should be inheritable. Probably we might need a symmetric unmemoized :foo to overwrite that in descendants.

**Non-features**

There are several more features typically seen in memoization gems considered unsuitable for core functionality:

- No arguments-dependent memoization. I believe it is a “business logic” concern: how exactly the arguments should be stored, cache growth control (with too many argument-result pairs memoized), cache cleanup, etc. Third-party libraries can handle that.
- No cache presetting/resetting API. If "it is memoized in general, but sometimes reset", it is again a business-layer concern and shouldn't be solved by a language-level declaration. Third-party libraries can handle that.
- No extra API to memoize class methods, like we don't have a specific API for making class methods private.

**History**

#1 - 08/17/2022 01:31 PM - Dan0042 (Daniel DeLorme)

The limitations of @var ||= memoization are brought up from time to time (e.g. #17316, #6023) so I think that's an indication of a need for something like this.

I like that the memoized info is automatically kept separate from inspect and serialization. Very clean and convenient.

I'm also excited by the possibility this could allow to memoize methods on frozen/shareable objects, which is impossible with instance variables. If you declare a class with 3 memoized methods, it should be possible to pre-allocate a 3-slot vector for each instance, which can then store the memoized info regardless of frozen status.
In general I agree the API should be as simple as possible and leave advanced features to gems, but at the same time it should be flexible enough to re-implement the various memoization gems. If memoist/memo_wise can't be refactored to use this API, then it's just not flexible enough. This should be a good building block, not just an inferior alternative to existing memoization gems. And in order to achieve that I think it's necessary to have some way to reset a memoized value.

#2 - 08/17/2022 02:55 PM - jeremyevans0 (Jeremy Evans)

Dan0042 (Daniel DeLorme) wrote in #note-1:

I'm also excited by the possibility this could allow to memoize methods on frozen/shareable objects, which is impossible with instance variables. If you declare a class with 3 memoized methods, it should be possible to pre-allocate a 3-slot vector for each instance, which can then store the memoized info regardless of frozen status.

This type of optimization is not safe unless the entire class hierarchy is frozen. Otherwise, additional memoized methods could be added to any ancestor at any time.

zverok (Victor Shepelev) wrote:

- No extra API to memoize class methods, like we don't have a specific API for making class methods private.

This may not be a good example, as Class#private_class_method is the specific API for making class methods private.

IMO, one issue with memoization in general is that it is only safe when the memoization depends on immutable state. If the results of the method can change by modifying the object's state, then memoization can result in incorrect behavior. So I'm not sure if this is something that should be available in Object/Kernel. It would be safer to introduce this only for immutable objects, such as the class discussed in #16122.

One thing missing from this proposal is an implementation approach. Are the memoized results stored in a hash? Are they stored in instance variables? How will the memoization approach interact with object shapes (#18776)?

#3 - 08/17/2022 03:17 PM - Dan0042 (Daniel DeLorme)

jeremyevans0 (Jeremy Evans) wrote in #note-2:

This type of optimization is not safe unless the entire class hierarchy is frozen. Otherwise, additional memoized methods could be added to any ancestor at any time.

Yes indeed, there are caveats and special situations to handle when a memoized method is added after instances have been created. But it's not like it's impossible to handle.

How will the memoization approach interact with object shapes (#18776)?

Yes! Using object shapes, great idea! That's actually a good way to handle the abovementioned issue with additional memoized methods.

#4 - 08/18/2022 06:27 AM - matz (Yukihiro Matsumoto)

- Status changed from Open to Rejected

I reject this proposal to make the feature built-in for several reasons:

- I still think it should be done in library (gem)
- The term memoize in this proposal is misused. The canonical memoize process record the function result according to arguments. But in this proposal, it's restricted to functions (methods) without arguments. Since it differs from the canonical definition, it is hard to make it built-in.

Matz.