

Ruby - Feature #17323

Ractor::LVar to provide ractor-local storage

11/12/2020 04:55 PM - ko1 (Koichi Sasada)

Status:	Closed	
Priority:	Normal	
Assignee:		
Target version:		
Description Ruby supports thread and fiber local storage: • Thread#[sym] provides <i>Fiber</i> local storage • Thread#thread_variable_get(sym) These APIs can access other threads/fibers like that: <pre>th = Thread.new{ Thread.current.thread_variable_set(:a, 10) } th.join # access from main thread to child thread p th.thread_variable_get(:a)</pre> To make Ractor local storage, this kind of feature should not be allowed to protect isolation. This ticket propose alternative API Ractor::LVar that allows to provide Ractor local variable. <pre>LV1 = Ractor::LVar.new p LV1.value #=> nil # default value LV1.value = 'hello' # can set unshareable objects because LVar is ractor local. Ractor.new do LV1.value = 'world' # set Ractor local variable end.take p LV1.value #=> 'hello' # Lvar.new can accept default_proc which should be isolated Proc. LV2 = Ractor::LVar.new{ "x" * 4 } p LV2.value #=> "xxxx" LV2.value = 'yyy' Ractor.new do p LV2.value #=> 'xxx' end p LV2.value #=> 'yyy'</pre> This API doesn't support accessing from other ractors. Ractor::LVar is from Ractor::TVar, but I have no strong opinion about it. For example, Ractor::LocalVariable is longer and clearer. Implementation: https://github.com/ruby/ruby/pull/3762		

History

#1 - 11/12/2020 05:07 PM - ko1 (Koichi Sasada)

Another advantages compare with current API is,

- we don't need to care about variable name.
- we can make ractor-local constants and instance variables like that:

```
class Fib
  attr_reader :cache

  def initialize
    @cache = Ractor::LVar.new{ {} }
  end

  private def _fib n
    if n < 2
      1
    else
      fib(n-1) + fib(n-2)
    end
  end

  def fib n
    if v = @cache.value[n]
      v
    else
      ans = _fib(n)
      @cache.value[n] = ans
    end
  end
end

fiboner = Fib.new
p fiboner.fib(10) #=> 89
pp fiboner.cache.value
#=> {1=>1, 0=>1, 2=>2, 3=>3, 4=>5, 5=>8, 6=>13, 7=>21, 8=>34, 9=>55, 10=>89}

Ractor.new fiboner do |f2|
  p f2.fib(5) #=> 8
  p f2.cache.value #=> {1=>1, 0=>1, 2=>2, 3=>3, 4=>5, 5=>8}
end.take
```

#2 - 11/12/2020 06:24 PM - marcandre (Marc-Andre Lafortune)

I'm curious for better use cases.

The above example is not very convincing:

- there's no much use in sending this object to a Ractor to start with
- deep-copying the cache is probably faster than having to recalculate part of it
- more importantly, this would probably be the wrong solution if we had SharedHash:

```
class SharedHash
  def initialize(initial = {})
    @ractor = Ractor.new(initial) do |hash|
      loop do
        case Ractor.receive
        in [:read, key, default, ractor] then
          ractor << hash.fetch(key, default)
        in [:write, key, value] then
          hash[key] = value
        in [:inspect | :to_s | :to_h => cmd, ractor] then
          ractor << hash.send(cmd)
        else raise ArgumentError
        end
      end
    end
  end

  def [](key)
    @ractor << [:read, key, nil, Ractor.current]
    Ractor.receive
  end

  def []=(key, value)
    @ractor << [:write, key, value]
    value
  end
end
```

```

def inspect
  @ractor << [:inspect, Ractor.current]
  Ractor.receive
end
end

class Fib
  attr_reader :cache

  def initialize
    @cache = SharedHash.new
  end

  private def _fib n
    if n < 2
      1
    else
      fib(n-1) + fib(n-2)
    end
  end

  def fib n
    @cache[n] ||= _fib(n)
  end
end

fiboner = Fib.new
p fiboner.fib(10) #=> 89
pp fiboner.cache
#=> {1=>1, 0=>1, 2=>2, 3=>3, 4=>5, 5=>8, 6=>13, 7=>21, 8=>34, 9=>55, 10=>89}

Ractor.new fiboner do |f2|
  p f2.fib(5) #=> 8 already cached!gi
  p f2.cache #=> {1=>1, 0=>1, 2=>2, 3=>3, 4=>5, 5=>8, 6=>13, 7=>21, 8=>34, 9=>55, 10=>89}
end.take

```

If the "start from a clear cache" is actually the right idea, than a solution could look like:

```

class Fib
  attr_reader :cache

  def initialize
    @cache = {}
  end

  def initialize_copy(_)
    @cache = {}
  end

  private def _fib n
    if n < 2
      1
    else
      fib(n-1) + fib(n-2)
    end
  end

  def fib n
    @cache[n] ||= _fib(n)
  end
end

fiboner = Fib.new
p fiboner.fib(10) #=> 89
pp fiboner.cache
#=> {1=>1, 0=>1, 2=>2, 3=>3, 4=>5, 5=>8, 6=>13, 7=>21, 8=>34, 9=>55, 10=>89}

Ractor.new fiboner.dup, move: true do |f2|
  p f2.fib(5) #=> 8
  p f2.cache.value #=> {1=>1, 0=>1, 2=>2, 3=>3, 4=>5, 5=>8}
end.take

```

Above does not work yet, depends on [#17286](#)...

It seems to me that having a way to define how to deep-copy an object might be important.

#3 - 11/12/2020 06:44 PM - ko1 (Koichi Sasada)

marcandre (Marc-Andre Lafortune) wrote in [#note-2](#):

- deep-copying the cache is probably faster than having to recalculate part of it
- more importantly, this would probably be the wrong solution if we had SharedHash:

it depends on the problem. per-ractor cache is faster to access because there are no synchronization overhead.

Anyway, cache is not a good example.

I wrote it in GH thread <https://github.com/ruby/ruby/pull/3762#issuecomment-726227262>

I'm not sure it is a feasible example, but we can set separate configuration between ractors.

Another idea is to provide unshareable, but similar to global variables, such as Random::DEFAULT which is discussed on <https://bugs.ruby-lang.org/issues/17322>.
We can use LVar to implement Ractor::default.

Maybe we can study with the usage of Thread#thread_variable_get(sym).

but not so many <https://gist.github.com/ko1/c00020a2c06dceaf9fd5d930e721651e>

#4 - 11/13/2020 07:53 PM - Dan0042 (Daniel DeLorme)

Would it be possible to somehow have ractor-local variables that are automatically dereferenced instead of having to append .value everywhere?

I'm thinking of cases like this where a class-level mutable constant (or classvar) makes it hard to make this code compatible with ractors.

```
class X
  # original, not compatible with Ractor, so how do we fix?
  CACHE = {}

  # like this, except now we need to append .value to every CACHE access
  CACHE = Ractor::LVar.new{ {} }

  def initialize(value)
    @value = value
  end

  def analyzed
    # here, lookup of CACHE constant could automatically return the
    # ractor-local Hash inside the LVar instead of the LVar itself
    CACHE[@value] ||= analyze(@value)
  end
end
```

The problem with the example above is that it's too magical to have a variable or constant return an object different from what was assigned. So what I'm saying is that I'd like *something like* this, that achieves the same effect.

I have the feeling that a different syntax would be needed, to differentiate it from assignment. Maybe CONST: expr could be used to lazily evaluate expr once per ractor; then it would make sense for CONST to return this value rather than the LVar used behind the scenes. The example above would become CACHE: {}. But of course introducing new syntax is not something to be done lightly.

#5 - 11/16/2020 07:59 AM - ko1 (Koichi Sasada)

Dan0042 (Daniel DeLorme) wrote in [#note-4](#):

The problem with the example above is that it's too magical to have a variable or constant return an object different from what was assigned.

Absolutely. By ractor's design, there is a possibility to provide "fork" model, which dup all constants at ractor creation. But we didn't choose (at least now).

So what I'm saying is that I'd like *something like* this, that achieves the same effect.

I understand the motivation. But not sure it is easy to use.

Adding .value is, it is clear that it is not access constants.

But not so clear it is ractor-local value.
It is disadvantage compare with traditional `Ractor.current[:sym]` approach.

(and I agree adding `.value` for each access is not easy)

#6 - 11/20/2020 08:39 AM - ko1 (Koichi Sasada)

- Description updated

#7 - 12/06/2020 11:20 PM - marcandre (Marc-Andre Lafortune)

I'm having difficulties because we don't have Ractor-local storage...

I'd like to implement a Ractor compatible SharedQueue.

I'd like to do it without going through a bridge Ractor, so I'm trying to refine Ractor to add channels as in [#17365](#).

To do that, I need a Ractor-local "saved messages queue", but there is currently no API for that.

I guess I'll have to roll my own using `Thread.main.thread_variable_get` and wrap the set inside a global Mutex...

#8 - 12/07/2020 04:26 AM - marcandre (Marc-Andre Lafortune)

In the meantime, I create the gem `ractor-local_variable`.

Code here: https://github.com/ractor-tools/ractor-local_variable/

Of course, Mutex is Ractor local... I used a Ractor as a basic mutex, there's no other way, right?

#9 - 12/10/2020 07:06 AM - matz (Yukihiro Matsumoto)

I prefer Thread-like Ractor local storage, e.g. `Ractor.current[key]`.

Matz.

#10 - 12/10/2020 10:50 AM - marcandre (Marc-Andre Lafortune)

Will there be a thread-safe way to initialize this storage (other than from the Ractor block)?

Because we can't use a Mutex to synchronize as there is no way to initialize the Mutex safely either...

Example: A gem needs Ractor-local storage to function. How can they initialize it in a thread-safe manner (other than by asking users of the gem to call `MyLib.initialize_ractor` from the ractor creation block)?

#11 - 12/10/2020 11:43 AM - Eregon (Benoit Daloze)

marcandre (Marc-Andre Lafortune) wrote in [#note-10](#):

Will there be a thread-safe way to initialize this storage (other than from the Ractor block)?

One possibility, when using `Ractor::LVar.new` would be:

```
LVar = Ractor::LVar.new do
  initial value
end
```

(not unlike Java's `ThreadLocal.withInitial(() -> { ... })`)

For a library, the `Ractor.current[key]` form seems inconvenient as it would need to generate some key, instead of referencing an `Ractor::LVar.new` object.

#12 - 12/10/2020 12:20 PM - marcandre (Marc-Andre Lafortune)

Eregon (Benoit Daloze) wrote in [#note-11](#):

For a library, the `Ractor.current[key]` form seems inconvenient as it would need to generate some key, instead of referencing an `Ractor::LVar.new` object.

I agree using a key does not sound that convenient, but it is a well known API. It's also not too hard to find a key. Instead of storing a LVar in `MyGem::Something::REGISTRY`, one can use `:'MyGem::Something::REGISTRY'` as key, there's not much of a difference except it needs to be fully scoped. I don't see other uses than globals for ractor-local storage, right?

The thread-safety is imo a bigger issue: if there's an easy way (here use `||=` and ignore thread safety) that is almost safe, and a really hard way that is

safe, I fear that the easy way will be taken most of the time.

#13 - 12/21/2020 08:58 PM - ko1 (Koichi Sasada)

- *Status changed from Open to Closed*

Traditional style Ractor#[]/#[]= are introduced.
35471a948739ca13b85fe900871e081d553f68e6

#14 - 01/06/2021 03:09 PM - dsisnero (Dominic Sisneros)

I don't like the name because it shadows - implies MVAR, TVAR which are known in the functional programming world.

#15 - 01/06/2021 03:11 PM - chrisseaton (Chris Seaton)

it shadows - implies MVAR, TVAR

I think that's the point isn't it? We have TVar (transactional), MVar (mutable), LVar (local), and matches ivar (instance.)

LVar is a bit overloaded that's true - left-value in terms of assignment - but most short names are.

#16 - 01/06/2021 04:22 PM - dsisnero (Dominic Sisneros)

LVAR is not used much but it is used. <http://lambda-the-ultimate.org/node/4823>,
<http://composition.al/blog/2013/09/22/some-example-mvar-ivar-and-lvar-programs-in-haskell/> and it is not
local variable it is a lattice based monotonic container

#17 - 01/06/2021 04:25 PM - chrisseaton (Chris Seaton)

I don't think there's a massive overlap there, and there's no many letters we could use.

#18 - 01/07/2021 06:33 AM - ko1 (Koichi Sasada)

Ractor::LocalVariable ?

#19 - 01/07/2021 01:11 PM - Eregon (Benoit Daloze)

I was thinking Ractor::Local.new would be fine too, (e.g., Java has new ThreadLocal()).