Ruby - Feature #20444

Kernel#loop: returning the "result" value of StopIteration doesn't work when raised directly

04/22/2024 09:41 PM - esad (Esad Hajdarevic)

Status:	Feedback	
Priority:	Normal	
Assignee:		
Target version:		

Description

There was a https://bugs.ruby-lang.org/issues/11498 a while ago which was merged in, but I was surprised to find out that raising StopIteration in a loop like

loop { raise StopIteration.new(3) }

returns nil and not 3.

History

#1 - 04/23/2024 05:02 AM - nobu (Nobuyoshi Nakada)

- Status changed from Open to Closed

StopIteration.new(3) does not set result, and no way to set it in Ruby level.

```
$ ruby -e 'e = StopIteration.new(3); p e.message, e.result'
"3"
nil
```

#2 - 04/23/2024 06:21 AM - esad (Esad Hajdarevic)

nobu (Nobuyoshi Nakada) wrote in #note-1:

> StopIteration.new(3) does not set result, and no way to set it in Ruby level.
>
> ```
> $ ruby -e 'e = StopIteration.new(3); p e.message, e.result'
> "3"
> nil
> ```

Thanks for the hint. It seems that subclassing StopIteration to provide result works:

```
class MyException < StopIteration
  def result = 5
end

loop { raise MyException } # => 5
```

#3 - 04/24/2024 02:40 AM - nobu (Nobuyoshi Nakada)

I'm curious what your use case is.

Although I don't know the reason why StopIteration#initialize does not have the argument for result, it would be difficult to change it now because of the backward compatibility.

#4 - 04/24/2024 03:33 AM - nobu (Nobuyoshi Nakada)

- Status changed from Closed to Feedback

#5 - 04/24/2024 03:33 AM - nobu (Nobuyoshi Nakada)

- Tracker changed from Bug to Feature

- ruby -v deleted (ruby 3.3.0 (2023-12-25 revision 5124f9ac75) [arm64-darwin20])

- Backport deleted (3.0: UNKNOWN, 3.1: UNKNOWN, 3.2: UNKNOWN, 3.3: UNKNOWN)

#6 - 04/24/2024 09:07 AM - esad (Esad Hajdarevic)

nobu (Nobuyoshi Nakada) wrote in #note-3:

> I'm curious what your use case is.
>
> Although I don't know the reason why StopIteration#initialize does not have the argument for result, it would be difficult to change it now because of the backward compatibility.

I think my use case is a bit of an edge case - I am passing a block into a Ractor where it runs in a loop. This way I can control exit from the loop, but obviously this can be also refactored into a normal result value of the block and "manual" looping control depending on the result.

#7 - 04/24/2024 10:40 AM - ufuk (Ufuk Kayserilioglu)

@esad If you just want to return a result from the loop, you can use break <value> to do that:

```
$ ruby -e "puts loop { break 3 }"
3
```

You shouldn't have to deal with anything low level like StopIteration to do that.

#8 - 04/24/2024 11:00 AM - esad (Esad Hajdarevic)

ufuk (Ufuk Kayserilioglu) wrote in #note-7:

> @esad If you just want to return a result from the loop, you can use break <value> to do that:
>
> ```
> $ ruby -e "puts loop { break 3 }"
> 3
> ```

Calling break from a block passed to a ractor will raise an exception. I think some sample code about my example will be helpful:

Let's make a ractor that just calls all blocks passed to it in a loop:

```
r = Ractor.new do
  block = Ractor.receive
  loop { block.call() }
end
```

Now let's send it a block that raises StopIteration:

```
block = true.instance_eval { proc { raise StopIteration, 3 } }
# instance_eval in true gives us a "shareable" proc
r.send Ractor.make_shareable(block)
r.take # => nil
```

Let's try with a subclass (this works)

```
class MyStop < ::StopIteration
  attr_reader :result
  def initialize(result)
    @result = result
  end
end

block = true.instance_eval { proc { raise MyStop, 3 } }
r.send Ractor.make_shareable(block)
r.take # => 3
```

#9 - 04/26/2024 08:28 AM - nobu (Nobuyoshi Nakada)

That example does not need Ractor.

```
class MyStop < ::StopIteration
  attr_reader :result
  def initialize(result)
    @result = result
  end
end

block = proc {raise StopIteration, 3}
p Thread.start {loop {block.call}}.value #=> nil

block = proc { raise MyStop, 3 }
p Thread.start {loop {block.call}}.value #=> 3
```

#10 - 04/26/2024 10:07 AM - esad (Esad Hajdarevic)

nobu (Nobuyoshi Nakada) wrote in #note-9:

> That example does not need Ractor.

Yes, you are right, it actually doesn't need Thread either, and is simply about calling a block in a loop and how to break the loop from the called block:

proc { break 3}.then { |p| loop { p.call() }} raises exception, and proc { raise StopIteration, 3 }.then { |p| loop { p.call() }} returns nil, etc.