Ruby master - Bug #17159

extend `define_method` for Ractor

09/07/2020 01:49 AM - ko1 (Koichi Sasada)

Status:	Open		
Priority:	Normal		
Assignee:			
Target version:			
ruby -v:		**Backport:**	2.5: DONTNEED, 2.6: DONTNEED, 2.7: DONTNEED

Description

Ractor prohibits use of non-isolated Procs.

Non-isolated example is here:

```
s = "foo"
pr = Proc.new{ p s }
```

This Proc pr can not be shared among ractors because outer variable s can contain an unshareable object. Also outer binding is a mutable object. Sharing it can lead race conditions.

Because of these reasons, define_method is also a problem on a multi-Ractor program.
(current implementation allows it just because check is not implemented, and it leads BUG).

I think there are several patterns when define_method is needed.

(1) To choose method names on-the-fly

```
name = ...
define_method(name){ nil }
```

(2) To embed variables to the code

```
10.times{|i|
  define_method("foo#{i}"){ i }
}
```

(3) To use global state by local variables

```
cnt = 0
define_method("inc"){ cnt += 1 }
```

(4) Others I can't imagine

---

(1) is easy. We can allow define_method(name, &Proc{nil}.isolate).

(3) can never be OK. It introduces data races/race conditions. For this purpose one need to use shared hash.

```
STATE = SharedHash.new(cnt: 0)
define_method("inc"){ STATE.transaction{ STATE[:cnt] += 1 }}
```

I think there are many (2) patterns that should be saved.
To help (2) pattern, the easiest way is to use eval.

```
10.times{|i|
  eval("def foo#{i} #{i}; end")
}
```

However, eval has several issues (it has huge freedom to explode the program, editor's syntax highlighting and so on).

Another approach is to embed the current value to the code, like this:

```
i = 0
define_method("foo", ractorise: true){ i }
#=> equivalent to:
#   define_method("foo"){ 0 }
# so that if outer scope's i changed, not affected.
i = 1
foo #=> 0

s = ""
define_method("bar", ractorise: true){ s }
#=> equivalent to:
#   define_method("bar"){ "" }
# so that if outer scope's s or s's value, it doesn't affect
s << "x"
bar #=> ""
```

However, it is very differenct from current Proc semantics.

Another idea is to specify embedding value like this:

```
i = 0
define_method("foo", i: i){ i }
#=> equivalent to:
#   define_method("foo"){ 0 }
# so that if outer scope's i changed, not affected.
i = 1
foo #=> 0

s = ""
define_method("bar", s: s){ s }
#=> equivalent to:
#   define_method("bar"){ "" }
# so that if outer scope's s or s's value, it doesn't affect
s << "x"
bar #=> ""
```

i: i and s: s are redundant. However, if there are no outer variable i or s, the i and s in blocks are compiled to send(:i) or send(:s). But I agree these method invocation should be replaced is another idea.

Thoughts?

Thanks,
Koichi

Related issues:

Related to Ruby master - Feature #17100: Ractor: a proposal for a new concurr... **Closed**

History

#1 - 09/07/2020 02:39 AM - matz (Yukihiro Matsumoto)

Just a comment. In general,

```
i = 0
define_method("foo#{i}"){ i }
```

and

```
i = 0
eval("def foo#{i} #{i}; end")
```

behave differently. The former returns the current value of i in the closure, the latter embed the value of i when the method is defined.
In the above example, they behave same because i is a block local variable and not shared by other closures

Matz.

#2 - 09/07/2020 04:36 AM - ko1 (Koichi Sasada)

Yes, I mean most of case (2) can be replaced with eval and proposed changes.

#3 - 09/13/2020 10:01 AM - Eregon (Benoit Daloze)

Copying captured variables seems a nice feature to have, also for optimizations.
In fact, this already exists in TruffleRuby internally:
https://github.com/oracle/truffleruby/blob/574d6bd2425caa856707ffd713fdb8ffc87be1e1/src/main/java/org/truffleruby/extra/TruffleGraalNodes.java#L113-L120
It seems known as fixTemps in Smalltalk.

So I'd suggest to add this as a keyword argument, not sure about the name but ractorize doesn't explain what it does.
copy_captured_locals: true, copy_captured_variables: true, capture_scope: false maybe?

This can also be useful on Proc in general, so it might be better to have a method on Proc for that.
Having it as a keyword argument for define_method as a shortcut seems good, since on the implementation side it's useful to combine both operations into one to avoid intermediate Procs, bytecode, etc.

For Ractor there is a big limitation though: only shareable objects can be copied that way.
Otherwise it would lead to shared state and segv.

#4 - 09/13/2020 10:04 AM - Eregon (Benoit Daloze)

- Related to Feature #17100: Ractor: a proposal for a new concurrent abstraction without thread-safety issues added

#5 - 09/13/2020 10:07 AM - Eregon (Benoit Daloze)

> #eval (...) has huge freedom to explode the program

I think this will happen too with the new feature proposed here.
One needs new bytecode/etc for the Proc/method where i is replaced by a literal value, isn't it?
Or do you alternatively copy the captured frame(s)?

Regarding implementation, this also needs to handle nested blocks inside define_method.

#6 - 09/14/2020 01:29 AM - shyouhei (Shyouhei Urabe)

- Description updated

The reason I use #define_method (4)

I sometimes use it to alias a part of a module, like this:

```
class Foo
  %i[sin cos tan].each do |sym|
    define_method(sym, Math.instance_method(sym))
  end
end
p Foo.new.sin(3.14)
```

There seems be no reason to reject such usages.

Capturing local variables

C++ since C++11 have had lambdas. In the language you can explicitly specify how you want to capture a variable each time when you create a lambda. As of C++20 there are 12 different specifier. Some of them exist for template metaprogramming (we can ignore such things), but I think there are several interesting cases.

```
#include <cstdio>

int main() {
    int x, y, z;

    x = y = z = 1;

    auto f = [x, &y, &z]() mutable {
        auto g = [x, y, &z]() mutable {
            printf("#1: x, y, z = %d, %d, %d\n", x, y, z);
            x = y = z = 4;
        };

        x = y = z = 3;
        g();
    };
    x = y = z = 2;
    f();

    printf("#2: x, y, z = %d, %d, %d\n", x, y, z);
}
```

This program outputs

```
#1: x, y, z = 1, 2, 3
#2: x, y, z = 2, 3, 4
```

Compilicated? But the s: s proposal is very much like this. You can mix call-by-reference and call-by-value. I agree this gives us maximum freedom, but at a cost of complexity.

C++ also has simpler specifier which has no such headaches:

```
#include <cstdio>

int main() {
    int x, y, z;

    x = y = z = 1;

    auto f = [=]() mutable {
        auto g = [=]() mutable {
            printf("#1: x, y, z = %d, %d, %d\n", x, y, z);
            x = y = z = 4;
        };

        x = y = z = 3;
        g();
    };

    x = y = z = 2;
    f();

    printf("#2: x, y, z = %d, %d, %d\n", x, y, z);
}
```

The above should print:

```
#1: x, y, z = 1, 1, 1
#2: x, y, z = 2, 2, 2
```

And I think this behaviour is much more understandable. The ractorise: true proposal is on this line. I'd push this way.

#7 - 09/25/2020 08:09 AM - nobu (Nobuyoshi Nakada)

- Backport changed from 2.5: UNKNOWN, 2.6: UNKNOWN, 2.7: UNKNOWN to 2.5: DONTNEED, 2.6: DONTNEED, 2.7: DONTNEED

- Description updated

#8 - 10/25/2020 08:08 PM - marcandre (Marc-Andre Lafortune)

How about:

```
define_method(:name, make_shareable: true) { ... }
# equivalent to:
define_method(:name, &Ractor.make_shareable(Proc.new{...}))`
```

With make_shareable as making accessed external variables shareable and not reassignable (semantics c))

Or define_method(:name, deep_freeze: true) { ... }

#9 - 10/29/2020 04:06 PM - ko1 (Koichi Sasada)

marcandre (Marc-Andre Lafortune) wrote in #note-8:

> How about:
>
> ```
> define_method(:name, make_shareable: true) { ... }
> # equivalent to:
> define_method(:name, &Ractor.make_shareable(Proc.new{...}))`
> ```

Matz, how about this proposal?