## Ruby master - Feature #8110
### Regex methods not changing global variables

03/18/2013 06:26 PM - prijutme4ty (Ilya Vorontsov)

<table>
<thead>
<tr>
<th>Status:</th>
<th>Closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority:</td>
<td>Normal</td>
</tr>
<tr>
<td>Assignee:</td>
<td>matz (Yukihiro Matsumoto)</td>
</tr>
<tr>
<td>Target version:</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

It is useful to have methods allowing pattern matching without setting global variables. It can be very hard to understand where the problem is when you for example insert a string like `puts pat === my_str` and your program fails in a place which is far-far away from inserted place. This can happen due to replacing global variables of previous pattern match. I caught to this when placed pattern-match inside case-statement and shadowed global vars which were initially filled by match in when-statement. For now one can extract pattern matching into another method thus defining method-scope for that variables. But sometimes it looks like an overkill. May be simple method like `#match_globalsafe` can prevent that kind of errors. At least when a programmer see such a method in a list of methods, he's warned that usual match can cause such problems.

**Related issues:**

- Related to Ruby master - Feature #8206: Should Ruby core implement String#bla... Open
- Related to Ruby master - Feature #12306: Implement String #blank? #present? a... Open
- Related to Ruby master - Bug #14695: [2.5.1] `===` is 1.77x slower than `match?` Closed

**Associated revisions**

Revision 780c056d - 05/19/2016 02:37 AM - nobu (Nobuyoshi Nakada)

re.c: `match?` should return nil if no match

- re.c (rb_reg_match_m_p): should return nil if no match, as the document says. [Feature #8110]

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@55067 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 55067 - 05/19/2016 02:37 AM - nobu (Nobuyoshi Nakada)

re.c: `match?` should return nil if no match

- re.c (rb_reg_match_m_p): should return nil if no match, as the document says. [Feature #8110]

Revision 55067 - 05/19/2016 02:37 AM - nobu (Nobuyoshi Nakada)

re.c: `match?` should return nil if no match

- re.c (rb_reg_match_m_p): should return nil if no match, as the document says. [Feature #8110]

Revision 55067 - 05/19/2016 02:37 AM - nobu (Nobuyoshi Nakada)

re.c: `match?` should return nil if no match

- re.c (rb_reg_match_m_p): should return nil if no match, as the document says. [Feature #8110]

Revision 55067 - 05/19/2016 02:37 AM - nobu (Nobuyoshi Nakada)

re.c: `match?` should return nil if no match

- re.c (rb_reg_match_m_p): should return nil if no match, as the document says. [Feature #8110]

Revision 55067 - 05/19/2016 02:37 AM - nobu (Nobuyoshi Nakada)

re.c: `match?` should return nil if no match

- re.c (rb_reg_match_m_p): should return nil if no match, as the document says. [Feature #8110]

**History**

- #1 - 03/22/2013 05:38 PM - naruse (Yui NARUSE)
  - Category set to core
  - Status changed from Open to Assigned
  - Assignee set to matz (Yukihiro Matsumoto)
  - Target version set to 2.6

It sounds reasonable.
The API can be some of following:

- new API like Regexp#match_without_backref
- new option for Regexp like Regexp.new("foo", Regexp::NO_BACKREF) or /foo/B
- new syntax

#2 - 04/03/2013 09:13 AM - sam.saffron (Sam Saffron)

naruse (Yui NARUSE)

There is a perf implication that really needs addressing here that would help elsewhere:

in re.c, there is a whole bunch of work that can be avoided when NO_BACKREF is passed in for the match:

In particular:

```c
match = match_alloc(rb_cMatch);
onig_region_copy(RMATCH_REGS(match), regs);
else {
    if (rb_safe_level() >= 3)
        OBJ_TAINT(match);
    else
        FL_UNSET(match, FL_TAINT);
}
RMATCH(match)->str = rb_str_new4(str);
RMATCH(match)->regexp = re;
RMATCH(match)->rmatch->char_offset_updated = 0;
rb_backref_set(match);
OBJ_INFECT(match, re);
OBJ_INFECT(match, str);
```

This in turn should improve the performance of regex matching with the /B option quite a lot.

I have been looking at this recently due to some performance issues I noticed on Active Supports String#blank?

The c implementation of:

```ruby
def blank?
  self !~ /[\^[:space:]]/ 
end
```

is the somewhat crazy:

https://github.com/SamSaffron/fast_blank/blob/master/ext/fast_blank/fast_blank.c#L16-L55

This implementation is 5 to 8x faster.

I vote for:

- new option for Regexp like Regexp.new("foo", Regexp::NO_BACKREF) AND /foo/B

You can then feature detect if its available by looking for Regexp::NO_BACKREF

I do wonder how much faster this will be for my micro benchmark vs the native c implementation, when you are done can you ping me so I can bench it? (at sam.saffron@gmail.com)

#3 - 04/03/2013 09:18 AM - sam.saffron (Sam Saffron)

A slight concern here is naming, since:

```
rb_define_virtual_variable("$-", match_getter, match_setter);
rb_define_virtual_variable("$&", last_match_getter, 0);
rb_define_virtual_variable("$`", prematch_getter, 0);
rb_define_virtual_variable("$'", postmatch_getter, 0);
rb_define_virtual_variable("$+", last_paren_match_getter, 0);
rb_define_virtual_variable("$=" , ignorecase_getter, ignorecase_setter);
rb_define_virtual_variable("$KCODE", kcode_getter, kcode_setter);
rb_define_virtual_variable("$-K", kcode_getter, kcode_setter);
```

even though core uses the term backref quite extensively, often people can confuse it with:
"round brackets also create a "backreference". A backreference stores the part of the string matched by the part of the regular expression inside the parentheses."

see: http://www.regular-expressions.info/brackets.html

I wonder if a different term all together should leak out Regexp::SKIP_GLOBALS and /foo/S , this is far more explicit and clear to explain.

#4 - 04/03/2013 09:37 AM - sam.saffron (Sam Saffron)
another slight note, I wonder how far this can stretch into onigaruma itself, can it be smart enough to avoid uneeded allocations when in a no backref mode?

#5 - 04/04/2013 03:53 PM - naruse (Yui NARUSE)
sam.saffron (Sam Saffron) wrote:
    another slight note, I wonder how far this can stretch into onigaruma itself, can it be smart enough to avoid uneeded allocations when in a no backref mode?

Oniguruma supplies the way to regexp search without backref.
see also my patch in https://bugs.ruby-lang.org/issues/8206#note-4

#6 - 04/05/2013 01:45 PM - funny_falcon (Yura Sokolov)
+1 for skip globals: when String#match used, there is no need to set globals, but no way to avoid it. Equally String[]. And even sometime with =~ and ===. So that /S and Regexp::SKIP_GLOBALS will be very useful.

#7 - 04/12/2013 12:06 PM - sam.saffron (Sam Saffron)
Has anyone given any thought at how to make this friendly with older versions of Ruby ... say I have

```ruby
def is_foo?(val)
    val =~ /foo/
end
```

And now I want this code to work in both 1.9.3 and master.

```ruby
# ugly and slow
def is_foo?(val)
    if defined? Regexp::SKIP_GLOBALS
        val =~ /foo/G
    else
        val =~ /foo/
    end
end
```

```ruby
# will not work on 1.9.3
def is_foo?(val)
    val =~ /foo/G
end
```

```ruby
# could work, risky perf
def is_foo?(val)
    val =~ _G(/foo/)
end
```

```ruby
# least horribly imho
IS_FOO = _G(/foo/)
def is_foo?(val)
    val =~ IS_FOO
end
```

So I wonder, is the plan to backport this? Are there any other ways to keep code compatible and clean?

#8 - 04/12/2013 01:23 PM - phluid61 (Matthew Kerwin)
=begin
sam.saffron (Sam Saffron) wrote:
    Has anyone given any thought at how to make this friendly with older versions of Ruby ... say I have
=end
def is_foo?(val)
  val =~ /foo/
end

And now I want this code to work in both 1.9.3 and master.

ugly and slow

def is_foo?(val)
  if defined? Regexp::SKIP_GLOBALS
    val =~ /foo/G
  else
    val =~ /foo/
  end
end

[snip]

least horribly imho

IS_FOO = _G(/foo/)
def is_foo?(val)
  val =~ IS_FOO
end

So I wonder, is the plan to backport this? Are there any other ways to keep code compatible and clean?

Defining a ({_G}) method in 1.9.* is no more or less a backport than allowing (and possibly ignoring) a /G modifier, and is pretty ugly to boot. I see no harm in adding some amount of /G support to 1.9.x and 2.0.0, once (if) the functionality is added to trunk, however I also think it is reasonable to expect developers to either

1) target 2.0.1 by using language features only supported by 2.0.1, or
2) target <=2.0.0 and 2.0.1 by only using language features that haven't changed, or
3) go to lengths to explicitly polyfill the older language versions.

Similar things happened with parser changes from 1.8 to 1.9 when adding the new ({{a: 1}}) hash syntax (which makes new code not work in old ruby) and removing the ([if x: y; end]) syntax (which makes old code not work in new ruby). At least adding a new pattern modifier doesn't break old code. In fact, it doesn't change the behaviour of the old code at all.

Also note that your "ugly and slow" code won't work because the parser still attempts (and fails) to parse ({{foo/G}}) even if the condition is false.

To make your code fully backwards-compatible you'd probably use something like:

IS_FOO = Regexp.new('foo', defined?(Regexp::SKIP_GLOBALS) ? Regexp::SKIP_GLOBALS : 0)
def is_foo?(val)
  val =~ IS_FOO
end

I also note you've changed /S to /G in your examples.

=end

#9 - 04/12/2013 01:40 PM - sam.saffron (Sam Saffron)
sorry, I really did not mean to say the language should ship a crazy _G macro it was just a simple polyfill in the app. even with the polyfill it is way verbose.

#app code not ruby
def _G(re)
  Regexp.new(re.to_s, re.options | defined?(Regexp::SKIP_GLOBALS) ? Regexp::SKIP_GLOBALS : 0)
end

#10 - 04/12/2013 01:53 PM - phluid61 (Matthew Kerwin)
sorry, I really did not mean to say the language should ship a crazy _G macro it was just a simple polyfill in the app. even with the polyfill it is way verbose.
def _G(re)
    Regexp.new(re.to_s, re.options | defined?(Regexp::SKIP_GLOBALS) ? Regexp::SKIP_GLOBALS : 0)
end

Why not just do the following (evil, wicked, untenable) hack?

class Regexp; SKIP_GLOBALS = 0 unless defined? SKIP_GLOBALS; end

You still can't use ((lfw)) or ((%r(lw)) literally, but it means you can use
Regexp.new("foo|bar(baz)?", Regexp::SKIP_GLOBALS)
without fear.

#11 - 04/12/2013 04:45 PM - headius (Charles Nutter)

Crazy idea: what if in the future you needed to set $~ to nil in order for it to be settable by downstream calls (e.g. regexp match)? It would eliminate a
great deal of magic and treat those calls the same way we treat closures: if the variable has not been instantiated outside the nested scope/call, it's
not available to be set.

e.g.

def foo(regexp)
    "Hello, world?" =~ regexp # does not set backtrace
    $~ = nil
    "Hello, world?" =~ regexp # does set backtrace, same as a closure setting a non-local variable.
end

Is this unreasonable?

#12 - 04/14/2013 11:19 PM - naruse (Yui NARUSE)

headius (Charles Nutter) wrote:

Crazy idea: what if in the future you needed to set $~ to nil in order for it to be settable by downstream calls (e.g. regexp match)? It would
eliminate a great deal of magic and treat those calls the same way we treat closures: if the variable has not been instantiated outside the nested
scope/call, it's not available to be set.

e.g.

def foo(regexp)
    "Hello, world?" =~ regexp # does not set backtrace
    $~ = nil
    "Hello, world?" =~ regexp # does set backtrace, same as a closure setting a non-local variable.
end

Is this unreasonable?

It breaks compatibility like following code:

def foo
    regexp =~ "foo"
    p $&
end

#13 - 04/15/2013 01:26 AM - naruse (Yui NARUSE)

You may misunderstand, unlike Perl, Ruby's setting global variable cost is small.
Ruby only set a MatchData object to its scope.
$~ (Regexp.last_match) gets it.
The implementation of $& (Regexp.last_match[0]), $' (Regexp.last_match.pre_match), and $' (Regexp.last_match.post_match)
are get $~ and call [0], pre_match, or post_match.
So setting cost is very small (0.2 second for 1,000,000 times).

And if it doesn't set global variable, it means that it can't recycle previous MatchData object.
So it allocates new MatchData object each time, it costs both allocation and GC.
On following case, its cost is beyond the setting cost.
r = Regexp.new("foo", Regexp::SKIP_GLOBALS); 1000000.times{|r =~ "foo"|

Therefore if you want speed up, you must remove making MatchData object.
String#match won't speed up so much because its API need creating MatchData object.
(meroever its current implementation uses $~)
should this be moved to ruby common?

You may misunderstand, unlike Perl, Ruby's setting global variable cost is small. Ruby only set a MatchData object to its scope.

That ignores the fact that without $~, the scope wouldn't need to be allocated either. In JRuby, when we know there's no $~ use, we allocate no scope; JVM can then inline our methods and avoid all allocation, putting locals in registers and speeding things up tremendously.

As an example of how much it helps... MRI 2.0.0 was changed to not allocate frames for core class methods, a change we made a couple years ago for JRuby 1.6. This had a massive impact on performance. If MRI could do this for Ruby methods as well, it would improve things further, but $~ and its implicit nature prevent that from being feasible right now.

$~ (Regexp.last_match) gets it. The implementation of $& (Regexp.last_match[0]), $' (Regexp.last_match.pre_match), and $' (Regexp.last_match.post_match) are get $~ and call [0], pre_match, or post_match. So setting cost is very small (0.2 second for 1,000,000 times).

The scope cost is the hidden cost.

And if it doesn't set global variable, it means that it can't recycle previous MatchData object. So it allocates new MatchData object each time, it costs both allocation and GC.

There are other ways to reduce the cost of allocating MatchData. In the end the MatchData object isn't as big as the matcher structures from the regexp engine anyway, right?

On following case, its cost is beyond the setting cost.

```
$ = Regexp.new('foo', Regexp::SKIP_GLOBALS); 1000000.times{$='foo'}
```

The cost here is as much the closure binding as it is the setting of $~. If $= did not set $~, no binding at all would be required for the closure and it would boil down to just the cost of calling =~ and creating the literal string.

Therefore if you want speed up, you must remove making MatchData object. String#match won't speed up so much because its API need creating MatchData object. (moreover its current implementation uses $~)

String#match would be known to not need $~, and implementations could avoid allocating the memory used to store $~ (not the MatchData but the method scope).

I will grant that since MRI does not have a JIT compiler, you need artificial scopes/frames anyway, but for implementations with optimizing JITs (JRuby, Rubinius) $~ is one of the biggest barriers to optimization.

should this be moved to ruby common?

If CommonRuby officially becomes the host project for features...yes, it should. I'm not sure we've had a final decision yet.
Without it JRuby would optimize to skip globals.

As an example of how much it helps... MRI 2.0.0 was changed to not allocate frames for core class methods, a change we made a couple years ago for JRuby 1.6. This had a massive impact on performance. If MRI could do this for Ruby methods as well, it would improve things further, but $~ and its implicit nature prevent that from being feasible right now.

If so, ko1 should implement some way to treat $~ without frames.

```ruby
$~ (Regexp.last_match) gets it.
The implementation of $& (Regexp.last_match[0]), $` (Regexp.last_match.pre_match), and $' (Regexp.last_match.post_match)
are get $~ and call [0], pre_match, or post_match.
So setting cost is very small (0.2 second for 1,000,000 times).
```

The scope cost is the hidden cost.

This doesn't include the cost to create a new scope because they are the same scope on $~ context.

And if it doesn't set global variable, it means that it can't recycle previous MatchData object.
So it allocates new MatchData object each time, it costs both allocation and GC.

There are other ways to reduce the cost of allocating MatchData. In the end the MatchData object isn't as big as the matcher structures from the regexp engine anyway, right?

My String#include?(regexp) patch in Feature #8206 is an example.

```ruby
On following case, its cost is beyond the setting cost.
r = Regexp.new("foo", Regexp::SKIP_GLOBALS); 1000000.times{r=~"foo"}
```

The cost here is as much the closure binding as it is the setting of $~. If =~ did not set $~, no binding at all would be required for the closure and it would boil down to just the cost of calling =~ and creating the literal string.

block doesn't make scope for $~.
And the time I show is compared between original and simply commented out rb_backref_set version.

Therefore if you want speed up, you must remove making MatchData object.
String#match won't speed up so much because its API need creating MatchData object.
(moreover its current implementation uses $~)

String#match would be known to not need $~, and implementations could avoid allocating the memory used to store $~ (not the MatchData but the method scope).

Setting $~ itself doesn't cause memory allocation because it is only setting the same object to VM.

I will grant that since MRI does not have a JIT compiler, you need artificial scopes/frames anyway, but for implementations with optimizing JITs (JRuby, Rubinius) $~ is one of the biggest barriers to optimization.

I know $~ prevents optimization because I used Perl before.
But it doesn't relate to Regexp::SKIP_GLOBALS because $~ is still here even if you usually use Regexp::SKIP_GLOBALS.

```
#18 - 05/17/2016 08:44 AM - mrkn (Kenta Murata)
- Related to Feature #12306: Implement String #blank? #present? and improve #strip and family to handle unicode added
```

```
#19 - 05/17/2016 08:52 AM - akr (Akira Tanaka)
How about Regexp#match? to return bool without MatchData?
```

```
#20 - 05/17/2016 08:55 AM - matz (Yukihiro Matsumoto)
+1 for Regexp#match?
```

```
Matz.
```

```
#21 - 05/18/2016 10:51 AM - naruse (Yui NARUSE)
```
- Status changed from Assigned to Closed

Implemented at r55061, thanks!

#22 - 04/18/2018 07:57 AM - shyouhei (Shyouhei Urabe)

- Related to Bug #14695: [2.5.1] 
  `===` is 1.77x slower than `match?` added