Ruby master - Bug #17030

Enumerable#grep{_v} should be optimized for Regexp

07/13/2020 08:26 PM - marcandre (Marc-Andre Lafortune)

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

Description

Currently,

```
array.select { |e| e.match?(REGEXP) }
```

is about three times faster and six times more memory efficient than

```
array.grep(REGEXP)
```

This is because grep calls Regexp#===, which creates useless MatchData.

History

#1 - 07/13/2020 08:27 PM - marcandre (Marc-Andre Lafortune)

Code to reproduce by fatkodima (Dima Fatko):

```
require 'benchmark-ips'
require 'benchmark-memory'

arr = %w[foobar foobaz bazquux hello world just making this array longer]

REGEXP = /o/

def select_match(arr)
  arr.select { |e| e.match?(REGEXP) }
end

def grep(arr)
  arr.grep(REGEXP)
end

Benchmark.ips do |x|
  x.report("select.match?")  { select_match(arr) }
  x.report("grep")           { grep(arr) }
  x.compare!
end

puts "********* MEMORY *********"

Benchmark.memory do |x|
  x.report("select.match?")  { select_match(arr) }
  x.report("grep")           { grep(arr) }
  x.compare!
end
```

#2 - 07/14/2020 01:24 AM - shyouhei (Shyouhei Urabe)

Yes but E#grep's allocating MatchData is by spec. You can observe $& etc. by passing a block to it.

```
p %w[q w e r].grep(/./) { $~ }
```

So this is at least a breaking change.

#3 - 07/14/2020 03:37 AM - marcandre (Marc-Andre Lafortune)

You are right for grep with a block, we can't necessarily optimize, but we should optimize grep without a block, no?

#4 - 07/14/2020 10:25 AM - nobu (Nobuyoshi Nakada)

Even without a block, grep sets $~ to the last match result.

#5 - 07/14/2020 01:28 PM - Eregon (Benoit Daloze)

nobu (Nobuyoshi Nakada) wrote in #note-4:

> Even without a block, grep sets $~ to the last match result.

I guess cases using $~ after the call to grep are very rare (notably because only the last match of the Enumerable would be accessible).

So I would suggest not setting $~ for grep without block.

#6 - 07/14/2020 01:46 PM - marcandre (Marc-Andre Lafortune)

nobu (Nobuyoshi Nakada) wrote in #note-4:

> Even without a block, grep sets $~ to the last match result.

I agree with Eregon (Benoit Daloze), doesn't seem like it makes much sense to use that.

There's also no valid reason it should set $~ for all the matches tested before the last one.

#7 - 07/21/2020 11:27 AM - scivola20 (sciv ola)

I have an idea to solve it without any compatibility problem.

[1] Introduce such a Regexp object that === method is same as match?.
[2] Introduce regexp literal option that makes the Regexp object as [1].

If the option is 'f', we can write as /o/f, and grep(/o/f) is faster than grep(/o/).

This speed up not only grep but also all?, any?, case and so on.

Many people have written like this:

```
IO.foreach("foo.txt") do |line|
  case line
  when /^#/
    # do nothing
  when /^(\d+)/
    # using $1
  when /xxx/
    # using $&
  when /yyy/
    # not using $&
  else
    # ...
  end
end
```

This is slow because of the above mentioned problem.
Replacing /^#/ with /^#/f, and /yyy/ with /yyy/f will make it faster.

#8 - 08/25/2020 05:23 PM - fatkodima (Dima Fatko)

I like scivola20 (sciv ola) 's idea and would like to try to implement it.
Before starting, nobu (Nobuyoshi Nakada), wdyt, is this something that will be merged?

#9 - 08/25/2020 10:09 PM - fatkodima (Dima Fatko)

I have implemented a simple PoC - https://github.com/ruby/ruby/pull/3455.

I got the following results.

Enumerable#grep

```
ARR = %w[foobar foobaz bazquux hello world just making this array longer]

REGEXP = /o/
FAST_REGEXP = /o/f
```

```
Benchmark.ips do |x|
  x.report("select.match?")  { ARR.select { |e| e.match?(REGEXP) } }
  x.report("grep")           { ARR.grep(REGEXP) }
  x.report("fast_grep")      { ARR.grep(FAST_REGEXP) }
  x.compare!
end

puts "********* MEMORY *********\n"

Benchmark.memory do |x|
  x.report("select.match?")  { ARR.select { |e| e.match?(REGEXP) } }
  x.report("grep")           { ARR.grep(REGEXP) }
  x.report("fast_grep")      { ARR.grep(FAST_REGEXP) }
  x.compare!
end
```

```
Warming up --------------------------------------
       select.match?    57.956k i/100ms
                grep    22.715k i/100ms
           fast_grep    59.434k i/100ms
Calculating -------------------------------------
       select.match?    580.339k (± 0.5%) i/s -      2.956M in   5.093260s
                grep    225.854k (± 0.6%) i/s -      1.136M in   5.028890s
           fast_grep    532.658k (± 9.0%) i/s -      2.675M in   5.067008s

Comparison:
       select.match?:   580338.8 i/s
           fast_grep:   532658.1 i/s - same-ish: difference falls within error
                grep:   225853.7 i/s - 2.57x  (± 0.00) slower

********* MEMORY *********
Calculating -------------------------------------
       select.match?   120.000  memsize (     0.000  retained)
                         1.000  objects (     0.000  retained)
                         0.000  strings (     0.000  retained)
                grep   536.000  memsize (   168.000  retained)
                         3.000  objects (     1.000  retained)
                         0.000  strings (     0.000  retained)
           fast_grep   200.000  memsize (     0.000  retained)
                         1.000  objects (     0.000  retained)
                         0.000  strings (     0.000  retained)

Comparison:
       select.match?:        120 allocated
           fast_grep:        200 allocated - 1.67x more
                grep:        536 allocated - 4.47x more
```

case-when

```
REGEXP = /z/
FAST_REGEXP = /z/f

def case_when(str)
  case str
  when REGEXP
    true
  end
end

def fast_case_when(str)
  case str
  when FAST_REGEXP
    true
  end
end

STR = 'foobarbaz'

Benchmark.ips do |x|
  x.report("case_when")       { case_when(STR) }
  x.report("fast_case_when")  { fast_case_when(STR) }
  x.compare!
end

puts "********* MEMORY *********\n"
```

```
Benchmark.memory do |x|
  x.report("case_when")       { case_when(STR) }
  x.report("fast_case_when")  { fast_case_when(STR) }
  x.compare!
end

Warming up --------------------------------------
           case_when    95.463k i/100ms
      fast_case_when   456.981k i/100ms
Calculating --------------------------------------
           case_when    964.438k (± 0.8%) i/s -      4.869M in   5.048469s
      fast_case_when      4.571M (± 0.6%) i/s -     23.306M in   5.098414s

Comparison:
      fast_case_when:  4571379.8 i/s
           case_when:   964438.3 i/s - 4.74x  (± 0.00) slower

********* MEMORY *********
Calculating --------------------------------------
           case_when   168.000  memsize (     0.000  retained)
                         1.000  objects (     0.000  retained)
                         0.000  strings (     0.000  retained)
      fast_case_when     0.000  memsize (     0.000  retained)
                         0.000  objects (     0.000  retained)
                         0.000  strings (     0.000  retained)

Comparison:
      fast_case_when:          0 allocated
           case_when:        168 allocated - Infx more
```

Enumerable#any?

```
REGEXP = /longer/
FAST_REGEXP = /longer/f
ARR = %w[foobar foobaz bazquux hello world just making this array longer]

Benchmark.ips do |x|
  x.report("any?")        { ARR.any?(REGEXP) }
  x.report("fast_any?")   { ARR.any?(FAST_REGEXP) }
  x.compare!
end

puts "********* MEMORY *********\n"

Benchmark.memory do |x|
  x.report("any?")        { ARR.any?(REGEXP) }
  x.report("fast_any?")   { ARR.any?(FAST_REGEXP) }
  x.compare!
end

Warming up --------------------------------------
                any?    25.840k i/100ms
           fast_any?    95.381k i/100ms
Calculating --------------------------------------
                any?    261.095k (± 1.0%) i/s -      1.318M in   5.047859s
           fast_any?    893.676k (±13.2%) i/s -      4.388M in   5.070820s

Comparison:
           fast_any?:   893675.9 i/s
                any?:   261095.0 i/s - 3.42x  (± 0.00) slower

********* MEMORY *********
Calculating --------------------------------------
                any?   168.000  memsize (   168.000  retained)
                         1.000  objects (     1.000  retained)
                         0.000  strings (     0.000  retained)
           fast_any?     0.000  memsize (     0.000  retained)
                         0.000  objects (     0.000  retained)
                         0.000  strings (     0.000  retained)

Comparison:
           fast_any?:          0 allocated
                any?:        168 allocated - Infx more
```

If that seems OK, I will update and finish my PR with tests/docs/etc.

#10 - 08/25/2020 11:37 PM - sawa (Tsuyoshi Sawada)

I feel scivola20 (sciv ola)'s idea promising, but have a concern that it is going to introduce the same kind of mess as when "string"f notation was introduced (same "f" used due to frozen and fast, but this is just coincidental). People are going to need to write /regex/f all over the place.

Just by analogy from the situation with strings, what about introducing the following pragma, which will make all regex literals on that page fast regex literals (i.e., === becomes match?)?

```
# boolean_regex_literal: true
```

And perhaps in the long run, Matz might want to make all regexes work like that, or simply change the definition of Regexp#===.

#11 - 08/26/2020 08:31 AM - fatkodima (Dima Fatko)

sawa (Tsuyoshi Sawada) wrote in #note-10:

> I feel scivola20 (sciv ola)'s idea promising, but have a concern that it is going to introduce the same kind of mess as when "string"f notation was introduced (same "f" used due to frozen and fast, but this is just coincidental). People are going to need to write /regex/f all over the place.
>
> Just by analogy from the situation with strings, what about introducing the following pragma, which will make all regex literals on that page fast regex literals (i.e., === becomes match?)?
>
> ```
> # boolean_regex_literal: true
> ```
>
> And perhaps in the long run, Matz might want to make all regexes work like that, or simply change the definition of Regexp#===.

Yes, seems like this will solve the problem of typing regex/f all over. However, imo, this is not as big problem as for strings, considering regexes to strings amount ratio.
Does it also mean that we then should have something like in frozen string world (String#@+) to manually change to the old behavior where we need it, like

```
# boolean_regex_literal: true

case var
when /foo/
  # does not set $~, etc
when +/bar/
  # sets $~, etc
end
```

?

#12 - 08/26/2020 03:01 PM - marcandre (Marc-Andre Lafortune)

Couldn't static analysis of the code determine in most cases if match data need be generated or not?

This is Ruby, so I can think of some corner cases where things like const_get(:Regexp).last_match would be impacted (in theory), what other limitations would this have?

Maybe it would be best to start a different thread as none of these proposals have a relation to grep{_v} without block not being optimized.

#13 - 08/26/2020 05:01 PM - fatkodima (Dima Fatko)

marcandre (Marc-Andre Lafortune) wrote in #note-12:

> Maybe it would be best to start a different thread as none of these proposals have a relation to grep{_v} without block not being optimized.

I have already implemented a patch to make grep{_v} faster and right before submitting the Create Pull Request button, I realized (with the help of scivola20's comment), that this case can be generalized. Because we already have, at least, Enumerable#{all?,any?,none?} and many future methods to be added (like grep), which can benefit from this generalized solution.

> This is Ruby, so I can think of some corner cases where things like const_get(:Regexp).last_match would be impacted (in theory), what other limitations would this have?

case-when?

> Couldn't static analysis of the code determine in most cases if match data need be generated or not?

In many cases, probably yes, but again, case-when, when arguments/consts/etc instead of local vars are used - it is hard to tell if them are regexes or

not.

#14 - 08/26/2020 05:30 PM - marcandre (Marc-Andre Lafortune)

fatkodima (Dima Fatko) wrote in #note-13:

> In many cases, probably yes, but again, case-when, when arguments/consts/etc instead of local vars are used - it is hard to tell if them are regexes or not.

That's not really what I'm proposing. I'm proposing something like an internal Regexp.needs_last_match? that would return true or false depending on the Ruby code, and that could be used to optimize methods. It would return true if any subsequent code could be impacted by $~ and al.

```
def foo
  /x/ =~ 'x' # needs_last_match? # => false
  case method
  when /(foo)/ # needs_last_match? # => false
    do_something
  when /(bar)/ # needs_last_match? # => true
    puts $2
    # ... # needs_last_match? # => false
  end
end

def bar
  # ... # needs_last_match? # => true
  case x
  when /(foo)/ # needs_last_match? # => true
    do_something
  end
  Regexp.last_match
  # ... # needs_last_match? # => false (false negative)
  Regexp.send :last_match
  # ... # needs_last_match? # => false (false negative)
  const_get(:Regexp).last_match
end
```

#15 - 08/26/2020 07:17 PM - Dan0042 (Daniel DeLorme)

> Couldn't static analysis of the code determine in most cases if match data need be generated or not?

scivola20 had a good idea, but this is even better. We can automatically get the best performance without having to manually optimize each case.

But static analysis has other limits besides const_get(:Regexp).last_match

```
def foo(v)
  /x/ =~ 'x' # needs_last_match? depends on whether 'v' is regexp
  case method
  when v
    $1
  end
end
```

So a simpler approach would be to check if the match operation's scope (in this case the method body) contains any of the regexp-related pseudo-globals.

#16 - 08/26/2020 07:31 PM - marcandre (Marc-Andre Lafortune)

Dan0042 (Daniel DeLorme) wrote in #note-15:

> But static analysis has other limits

Good example but it is easily resolved: assume v isn't a Regexp and we may get a false positive, which is not a big issue. There will be other false positives: str.gsub(regexp, &block). That's not a real issue, simply assume that block will want access to Regexp.last_match. I'm really only worried about false negatives... Any other example comes to mind?

#17 - 08/26/2020 08:43 PM - Dan0042 (Daniel DeLorme)

What about this?

```
2.times do
```

```
  p $~  # depends on match *below*
  rx =~ str
end
```

Now imagine if 2.times is replaced by foo; a priori we can't know if or how many times the block will be executed. So what I was trying to say is that flow control can lead to all kinds of code paths where it's extremely difficult to know which matching operations a pseudo-global may depend on. Maybe not impossible, but personally I wouldn't want to code that kind of analysis when a simple approach is enough for >90% of cases, and guaranteed to be bug-free.

> There will be other false positives: str.gsub(regexp, &block). That's not a real issue, simply assume that block will want access to Regexp.last_match.

Actually... block does not have access to Regexp.last_match (unless you created the block in the same scope as the gsub operation, but that would be unusual)

#18 - 08/27/2020 01:37 AM - sawa (Tsuyoshi Sawada)

- Description updated

#19 - 08/27/2020 02:10 AM - marcandre (Marc-Andre Lafortune)

Dan0042 (Daniel DeLorme) wrote in #note-17:

> Maybe not impossible, but personally I wouldn't want to code that kind of analysis when a simple approach is enough for >90% of cases, and guaranteed to be bug-free.

Agreed

> > There will be other false positives: str.gsub(regexp, &block). That's not a real issue, simply assume that block will want access to Regexp.last_match.
>
> Actually... block does not have access to Regexp.last_match (unless you created the block in the same scope as the gsub operation, but that would be unusual)

You are right, my bad.

#20 - 08/27/2020 08:03 AM - fatkodima (Dima Fatko)

Dan0042 (Daniel DeLorme) wrote in #note-15:

> So a simpler approach would be to check if the match operation's scope (in this case the method body) contains any of the regexp-related pseudo-globals.

I didn't quite get it. So, to summarize, how this new approach should work? Can you elaborate in few more sentences?

Does ruby already do some kind of static analysis that you can point me to?

#21 - 08/27/2020 04:55 PM - Dan0042 (Daniel DeLorme)

Yeah ok, that sentence wasn't very clear, sorry.

The first thing is that when compiling a method to an iseq, you have to set a flag on the iseq if the method contains any of the "last_match" pseudo-globals ($~, $&, $1, Regexp.last_match, ...)

Then in rb_reg_match (aka Regexp#=~), you check if the current iseq has the flag set. This is similar to how rb_backref_get gets the last_match object from execution context > control frame > normal control frame > ep > svar > backref. If the flag is not set it means you can use a variant of reg_match_pos that only returns the position without using rb_reg_search to set the last_match, in the same vein as rb_reg_match_m_p (aka Regexp#match?).

But I may be missing a few details here, as I don't have a full understanding of the VM.

#22 - 08/27/2020 05:16 PM - jeremyevans0 (Jeremy Evans)

Dan0042 (Daniel DeLorme) wrote in #note-21:

> Yeah ok, that sentence wasn't very clear, sorry.
>
> The first thing is that when compiling a method to an iseq, you have to set a flag on the iseq if the method contains any of the "last_match" pseudo-globals ($~, $&, $1, Regexp.last_match, ...)

Then in rb_reg_match (aka Regexp#=~), you check if the current iseq has the flag set. This is similar to how rb_backref_get gets the last_match object from execution context > control frame > normal control frame > ep > svar > backref. If the flag is not set it means you can use a variant of reg_match_pos that only returns the position without using rb_reg_search to set the last_match, in the same vein as rb_reg_match_m_p (aka Regexp#match?).

But I may be missing a few details here, as I don't have a full understanding of the VM.

Unfortunately, you can't take this approach for VM optimizations without breaking backwards compatibility unless you also have a deoptimization approach that will handle code such as:

```
def a; /(a)/ =~ 'a'; binding; end; a.eval('$1')

def a; /(a)/ =~ 'a'; proc{}; end; a.binding.eval('$1')

def a(c, m); /(a)/ =~ 'a'; c.send(m); end; a(Regexp, :last_match)
```

#23 - 08/28/2020 03:01 PM - Eregon (Benoit Daloze)

This is something that a JIT with inlining and escape analysis can optimize and always be correct.
Static analysis doesn't cut it for Ruby.

On TruffleRuby (master + a fix I'll merge soon) for the benchmark above:

```
Calculating -------------------------------------
       select.match?      2.503M (± 2.9%) i/s -     12.677M in   5.068796s
                grep      2.502M (± 2.8%) i/s -     12.558M in   5.022837s
       select.match?      2.519M (± 2.6%) i/s -     12.677M in   5.036105s
                grep      2.498M (± 2.2%) i/s -     12.558M in   5.030485s

Comparison:
       select.match?:  2518943.6 i/s
                grep:  2497618.0 i/s - same-ish: difference falls within error
```

MRI 2.6 for comparison:

```
Calculating -------------------------------------
       select.match?    943.017k (± 0.6%) i/s -      4.770M in   5.058962s
                grep    470.844k (± 0.8%) i/s -      2.389M in   5.074917s
       select.match?    944.326k (± 0.7%) i/s -      4.770M in   5.052020s
                grep    471.122k (± 2.5%) i/s -      2.389M in   5.074969s

Comparison:
       select.match?:   944325.5 i/s
                grep:   471122.3 i/s - 2.00x  (± 0.00) slower
```

#24 - 08/28/2020 03:30 PM - Eregon (Benoit Daloze)

I'm surprised at the performance difference on MRI though. Just allocating the MatchData shouldn't be so expensive. Maybe Regexp#match? has additional optimizations?