Ruby - Feature #19104

Introduce the cache-based optimization for Regexp matching

11/05/2022 01:30 PM - make_now_just (Hiroya Fujinami)

Status:	Closed	
Priority:	Normal	
Assignee:		
Target version:		

Description

Regexp matching causes a time-complexity explosion problem as known as ReDoS (https://en.wikipedia.org/wiki/ReDoS). ReDoS has become serious vulnerability in many places in recent years, and Ruby is no exception. The following is the incomplete list of such vulnerability reports:

- https://github.com/sinatra/sinatra/pull/1823
- https://github.com/lostisland/faraday-net_http/pull/27

These reports have been addressed by fixing the library/software implementation. But, if the language's Regexp implementation becomes safe, the vulnerability is fundamentally archived.

For a few months, Ruby has implemented a Regexp matching timeout (https://bugs.ruby-lang.org/issues/17837). It is one of the useful methods for preventing ReDoS vulnerability, but it is a problem that setting a correct timeout value is hard. This value is depending on input length, environment, network status, system load, etc. When the value is too small, a system may be broken, but when the value is too large, it is not useful for preventing ReDoS.

Therefore, as a new way to prevent ReDoS, we propose to introduce cache-based optimization for Regexp matching. As CS fundamental knowledge, automaton matching result depends on the position of input and state. In addition, matching time explosion is caused for repeating to arrive at the same position and state many times. Then, ReDoS can be prevented when pairs of position, and state arrived once is recorded (cached). In fact, under such an optimization, it is known as the matching time complexity is linear against input size [1].

[1]: Davis, James C., Francisco Servant, and Dongyoon Lee. "Using selective memoization to defeat regular expression denial of service (ReDoS)." *2021 IEEE symposium on security and privacy (SP)*. IEEE, 2021.
https://www.computer.org/csdl/proceedings-article/sp/2021/893400a543/1oak988ThvO

See the following example.

```
$ ruby --version
ruby 3.2.0preview2 (2022-09-09 master 35cfc9a3bb) [arm64-darwin21]

$ time ruby -e '/^(a|a)*$/ =~ "a" * 28 + "b"'
ruby -e '/^(a|a)*$/ =~ "a" * 28 + "b"'  8.49s user 0.04s system 98% cpu 8.663 total

$ ./miniruby --version
ruby 3.2.0dev (2022-10-27T13:39:56Z recache bc59b7cc1e) [arm64-darwin21]

$ time ./miniruby -e '/^(a|a)*$/ =~ "a" * 28 + "b"'
./miniruby -e '/^(a|a)*$/ =~ "a" * 28 + "b"'  0.01s user 0.01s system 8% cpu 0.310 total
```

In this example, using ruby v3.2.0-preview2, matching /^(a|a)*$/ against "a" * 28 + "b" takes 8.6 seconds because matching time of this Regexp takes exponentially against "a" * N + "b" form string. But, using the patched version of ruby, it takes 0.01 seconds. Incredibly it is 860 times faster because matching is done in linear time.

By this optimization, the matching time is linear to the input size. It sounds secure and good. Unfortunately, when Regexp uses some extension (e.g. look-around, back-reference, subexpression call), the optimization is not applied. Also, the optimization needs a bit more memory for caching. However, we have already investigated that they are not so the major problems (See the "Limitation" section).

Implementation

The basic cache implementation is complete at this time and can be found in the following Pull Request.

https://github.com/ruby/ruby/pull/6486

Some tests seem to be failed, but it is no problem! Because the failed tests are for Regexp timeout, optimization works correctly and so they failed as expected. Of course, we need to fix these tests before merging.

Implementation notes:

- A null-check on loop causes non-linear behavior, so the field to index the latest null-check item on the stack is added to OnigStackType. (https://github.com/ruby/ruby/pull/6486/files#diff-4347460e379cd970ba0b88b4acb215ea60cbae308124675de8811eed377367aaR831)
- When the loop is null and this loop has a capture, matching behaves as a monomaniac. To reproduce this behavior, caches in the loop is cleared as necessary. (https://github.com/ruby/ruby/pull/6486/files#diff-c3cfe19efff0cc5181384741386067b81eadb2505b8b9b3f980d10f815037395R3433)
  Like a flip-flop operator, we hope to drop this if possible. But it still remains backward compatibility.

Limitation

Cache-based optimization is not applied in the following cases:

1. Regexp using some extensions (back-reference and subexpression call, look-around, atomic, absent operators) is not optimized because it is impossible or hard. However, it may be possible for look-around operators.
2. A bounded or fixed times repetition nesting in another repetition (e.g. /(a{2,3})*/). It is an implementation issue entirely, but we believe it is hard to support this case correctly.
3. Bounded or fixed times repetition is too large (e.g. /(a|b){100000,200000}/). The cache table size is proportional to the product of the number of cache points of regex and input size. In this case, since the number of cache points becomes too large, the optimization cannot be applied.

Experiments were conducted to investigate how these limitations are problematic in practice. We used ObjectSpace to collect Regexps and investigate whether they could be optimized and the number of cache points. Regexps were collected from the standard library, Webrick, and Rails. See the following gist for the details (https://gist.github.com/makenowjust/83e1e75a2d7de8b956e93bdac004a06b).

The experiments result is shown in the following table.

Collected from	# Regexp	# non-optimizable	Maximum number of cache points
stdlib	1009	86 (8.52%)	81
Webrick	356	44 (12.36%)	20
Rails	759	74 (7.75%)	27
Total (Duplications are reduced)	1506	118 (7.84%)	81

This result shows that the percentage of non-optimizable Regexp is less than 10%, and the amount of memory used for optimization is about 10 times the length of the string (81/8, for a bit array) at worst in this case. It is considered that a sufficient number of Regexp can be optimized in practice.

Specification

The detailed specification has been fixed yet. We have some ideas and we would like to discuss them.

- When is optimization enabled? Currently, it turns on when the backtrack causes as many as the input length.
- How the number of cache points is allowed, and how memory can be allocated? It is not determined for now.
- If the above parameters can be specified by users, how are they specified? (via command-line flags, or static / instance parameters like Regexp#.timeout= and Regexp#timeout=)
- Unless the input size is too large, the availability of optimization can be determined on compile-time. So, we would like to add a new flag to Regexp to determine whether a cache is available. It becomes one of the criteria for whether Regexp is efficiently executable or not. We believe it helps users. Thus, which letter is preferred for this purpose? l (linear) or r (regular) sounds good, but I am not sure which is the best.

Thank you.

Associated revisions

Revision f093b619a4863be96e6ebfa2fd58c77f4a360eae - 12/12/2022 04:56 AM - nobu (Nobuyoshi Nakada)

[DOC] NEWS about [Feature #19104]

Revision f093b619a4863be96e6ebfa2fd58c77f4a360eae - 12/12/2022 04:56 AM - nobu (Nobuyoshi Nakada)

[DOC] NEWS about [Feature #19104]

Revision f093b619 - 12/12/2022 04:56 AM - nobu (Nobuyoshi Nakada)

[DOC] NEWS about [Feature #19104]

History

#1 - 11/07/2022 02:32 AM - matz (Yukihiro Matsumoto)

It sounds good. My only concern is memory consumption, but I don't think guessing game won't work for this case.
Merge this PR and experiment with 3.1 previews.

Matz.

#2 - 11/07/2022 02:43 AM - nobu (Nobuyoshi Nakada)

- Description updated

#3 - 11/07/2022 06:26 PM - Eregon (Benoit Daloze)

From https://bugs.ruby-lang.org/issues/19074#note-6 from @mame (Yusuke Endoh):

> ... (2) should we provide a way to tell users if a given regexp is optimizable or not (e.g., a warning at unoptimizable regexp creation, a new API like Regexp#optimizable?, or a new opt-in Regexp flag /foo/r to raise a SyntaxError when it is not optimizable, etc.)

I think a command-line flag or a new warning category is best for this.
For instance I envisioned --warn-slow-regexps in https://eregon.me/blog/assets/research/just-in-time-compiling-ruby-regexps-on-truffleruby.pdf slide 18.
A warning category is nice because it can be used to e.g. raise a exception on such slow Regexps and so ensure an application doesn't use any such Regexp.

"a new API like Regexp#optimizable?" seems useless to me, the value of this is checking all Regexps in the system can be efficiently executed.
Same problem with the opt-in flag, to be useful it needs to apply to all Regexps.

We need to keep in mind that the set of "slow regexps" can change per Ruby/Regexp implementation and over time.
Although it seems it's very very similar features being problematic between this approach and
https://eregon.me/blog/assets/research/just-in-time-compiling-ruby-regexps-on-truffleruby.pdf slide 11.

I'll summarize it here.

Not supported on both:

- back-reference \1, \k<name> in the Regexp (not in replacement strings: #gsub)
- recursive subexpression call (?<sqbr>[\g<sqbr>*])
- negative lookahead (?!) and lookbehind (?<!)
- atomic groups (?>)
- absent operator (?~)
- possessive quantifiers *+, ++, ?+, {n,m}+
- too large bounded or fixed times repetition (e.g. /(a|b){100000,200000}/)

Not supported on CRuby, but supported on TruffleRuby:

- non-recursive subexpression call
- positive lookahead and lookbehind

More might be supported in the future.

#4 - 11/16/2022 07:37 AM - byroot (Jean Boussier)

I believe this change may have introduced a weird bug which is causing the sass gem to fail in unpredictable ways. I was able to produce a short reproduction script:

```
module Sass
  H        = /[0-9a-fA-F]/
  UNICODE  = /\\#{H}{1,6}[ \t\r\n\f]?/
  s = '\u{80}-\u{D7FF}\u{E000}-\u{FFFD}\u{10000}-\u{10FFFF}'
  NONASCII = /[#{s}]/
  ESCAPE   = /#{UNICODE}|\\[^0-9a-fA-F\r\n\f]/
```

```
  NMSTART  = /[_a-zA-Z]|#{NONASCII}|#{ESCAPE}/
  NMCHAR   = /[a-zA-Z0-9_-]|#{NONASCII}|#{ESCAPE}/
  VALID_UNIT = /#{NMSTART}#{NMCHAR}|%*/

  100_000.times do
    print '.'
    raise "WTF?" if "%" !~ VALID_UNIT
  end
end
```

The above regexp will sometime not match after a random number of iterations.

cc @mame (Yusuke Endoh)

#5 - 11/16/2022 02:09 PM - byroot (Jean Boussier)

Not sure if helpful, but after working around this bug, we realized that it wasn't just returning the wrong value.

Avoiding this codepath also caused a lot of weird crashes to vanish (e.g. Segmentation fault at 0x0000000000000000 and [BUG] Tried to mark T_NONE). So I heavily suspect that one of the side effects of this bug is to overwrite parts of the heap with 0s.

#6 - 11/16/2022 02:47 PM - make_now_just (Hiroya Fujinami)

I created a pull request for the report. Thanks @byroot (Jean Boussier).

https://github.com/ruby/ruby/pull/6744

#7 - 12/10/2022 06:41 AM - mmizutani (Minoru Mizutani)

Regex fuzzing encountered an edge-case regression:

```
$ ruby --version
ruby 3.1.2p20 (2022-04-12 revision 4491bb740a) [x86_64-linux]

$ time ruby -e 'puts /(?~)/.match?("")'
true
0.04s user 0.04s system 74% cpu 0.118 total

$ ruby --version
ruby 3.2.0rc1 (2022-12-10) [x86_64-linux]

$ time ruby -e 'puts /(?~)/.match?("")'
# This regex matching involving an absent operator and non-ASCII characters runs indefinitely and consumes gig
abytes of memory, eventually killed by the OS.
```

#8 - 12/10/2022 07:25 PM - mame (Yusuke Endoh)

mmizutani (Minoru Mizutani) wrote in #note-7:

> Regex fuzzing encountered an edge-case regression:

Thanks. This has nothing to do with this ticket. It's a different issue caused by my change 1d2d25dcadda0764f303183ac091d0c87b432566 .

#9 - 12/12/2022 04:56 AM - nobu (Nobuyoshi Nakada)

- Status changed from Open to Closed

Applied in changeset git|f093b619a4863be96e6ebfa2fd58c77f4a360eae.

[DOC] NEWS about [Feature #19104]