NAME

PCRE - Perl-compatible regular expressions

PCRE PERFORMANCE

Two aspects of performance are discussed below: memory usage and processing time. The way you express your pattern as a regular expression can affect both of them.

COMPILED PATTERN MEMORY USAGE

Patterns are compiled by PCRE into a reasonably efficient interpretive code, so that most simple patterns do not use much memory. However, there is one case where the memory usage of a compiled pattern can be unexpectedly large. If a parenthesized subpattern has a quantifier with a minimum greater than 1 and/or a limited maximum, the whole subpattern is repeated in the compiled code. For example, the pattern

```
  (abc|def){2,4}
```

is compiled as if it were

```
  (abc|def)(abc|def)((abc|def)(abc|def)?)?
```

(Technical aside: It is done this way so that backtrack points within each of the repetitions can be independently maintained.)

For regular expressions whose quantifiers use only small numbers, this is not usually a problem. However, if the numbers are large, and particularly if such repetitions are nested, the memory usage can become an embarrassment. For example, the very simple pattern

```
  ((ab){1,1000}c){1,3}
```

uses 51K bytes when compiled using the 8-bit library. When PCRE is compiled with its default internal pointer size of two bytes, the size limit on a compiled pattern is 64K data units, and this is reached with the above pattern if the outer repetition is increased from 3 to 4. PCRE can be compiled to use larger internal pointers and thus handle larger compiled patterns, but it is better to try to rewrite your pattern to use less memory if you can.

One way of reducing the memory usage for such patterns is to make use of PCRE's "subroutine" facility. Re-writing the above pattern as

```
  ((ab)(?2){0,999}c)(?1){0,2}
```

reduces the memory requirements to 18K, and indeed it remains under 20K even with the outer repetition increased to 100. However, this pattern is not exactly equivalent, because the "subroutine" calls are treated as atomic groups into which there can be no backtracking if there is a subsequent matching failure. Therefore, PCRE cannot do this kind of rewriting automatically. Furthermore, there is a noticeable loss of speed when executing the modified pattern. Nevertheless, if the atomic grouping is not a problem and the loss of speed is acceptable, this kind of rewriting will allow you to process patterns that PCRE cannot otherwise handle.

STACK USAGE AT RUN TIME

When **pcre_exec()** or **pcre[16|32]_exec()** is used for matching, certain kinds of pattern can cause it to use large amounts of the process stack. In some environments the default process stack is quite small, and if it runs out the result is often SIGSEGV. This issue is probably the most frequently raised problem with PCRE. Rewriting your pattern can often help. The **pcrestack** documentation discusses this issue in detail.

PROCESSING TIME

Certain items in regular expression patterns are processed more efficiently than others. It is more efficient to use a character class like [aeiou] than a set of single-character alternatives such as (a|e|i|o|u). In general, the simplest construction that provides the required behaviour is usually the most efficient. Jeffrey Friedl's book contains a lot of useful general discussion about optimizing regular expressions for efficient performance. This document contains a few observations about PCRE.

Using Unicode character properties (the \p, \P, and \X escapes) is slow, because PCRE has to use a multistage table lookup whenever it needs a character's property. If you can find an alternative pattern that does not use character properties, it will probably be faster.

By default, the escape sequences \b, \d, \s, and \w, and the POSIX character classes such as [:alpha:] do not use Unicode properties, partly for backwards compatibility, and partly for performance reasons. However, you can set PCRE_UCP if you want Unicode character properties to be used. This can double the matching time for items such as \d, when matched with a traditional matching function; the performance loss is less with a DFA matching function, and in both cases there is not much difference for \b.

When a pattern begins with .* not in parentheses, or in parentheses that are not the subject of a backreference, and the PCRE_DOTALL option is set, the pattern is implicitly anchored by PCRE, since it can match only at the start of a subject string. However, if PCRE_DOTALL is not set, PCRE cannot make this optimization, because the . metacharacter does not then match a newline, and if the subject string contains newlines, the pattern may match from the character immediately following one of them instead of from the very start. For example, the pattern

```
  .*second
```

matches the subject "first\nand second" (where \n stands for a newline character), with the match starting at the seventh character. In order to do this, PCRE has to retry the match starting after every newline in the subject.

If you are using such a pattern with subject strings that do not contain newlines, the best performance is obtained by setting PCRE_DOTALL, or starting the pattern with ^.* or ^.*? to indicate explicit anchoring. That saves PCRE from having to scan along the subject looking for a newline to restart at.

Beware of patterns that contain nested indefinite repeats. These can take a long time to run when applied to a string that does not match. Consider the pattern fragment

```
  ^(a+)*
```

This can match "aaaa" in 16 different ways, and this number increases very rapidly as the string gets longer. (The * repeat can match 0, 1, 2, 3, or 4 times, and for each of those cases other than 0 or 4, the + repeats can match different numbers of times.) When the remainder of the pattern is such that the entire match is going to fail, PCRE has in principle to try every possible variation, and this can take an extremely long time, even for relatively short strings.

An optimization catches some of the more simple cases such as

```
  (a+)*b
```

where a literal character follows. Before embarking on the standard matching procedure, PCRE checks that there is a "b" later in the subject string, and if there is not, it fails the match immediately. However, when there is no following literal this optimization cannot be used. You can see the difference by comparing the behaviour of

```
  (a+)*\d
```

with the pattern above. The former gives a failure almost instantly when applied to a whole line of "a" characters, whereas the latter takes an appreciable time with strings longer than about 20 characters.

In many cases, the solution to this kind of performance issue is to use an atomic group or a possessive quantifier.

AUTHOR

Philip Hazel
University Computing Service
Cambridge CB2 3QH, England.

REVISION

Last updated: 25 August 2012