NAME

PCRE - Perl-compatible regular expressions

PCRE DISCUSSION OF STACK USAGE

When you call **pcre[16|32]_exec()**, it makes use of an internal function called **match()**. This calls itself recursively at branch points in the pattern, in order to remember the state of the match so that it can back up and try a different alternative if the first one fails. As matching proceeds deeper and deeper into the tree of possibilities, the recursion depth increases. The **match()** function is also called in other circumstances, for example, whenever a parenthesized sub-pattern is entered, and in certain cases of repetition.

Not all calls of **match()** increase the recursion depth; for an item such as a* it may be called several times at the same level, after matching different numbers of a's. Furthermore, in a number of cases where the result of the recursive call would immediately be passed back as the result of the current call (a "tail recursion"), the function is just restarted instead.

The above comments apply when **pcre[16|32]_exec()** is run in its normal interpretive manner. If the pattern was studied with the PCRE_STUDY_JIT_COMPILE option, and just-in-time compiling was successful, and the options passed to **pcre[16|32]_exec()** were not incompatible, the matching process uses the JIT-compiled code instead of the **match()** function. In this case, the memory requirements are handled entirely differently. See the **pcrejit** documentation for details.

The **pcre[16|32]_dfa_exec()** function operates in an entirely different way, and uses recursion only when there is a regular expression recursion or subroutine call in the pattern. This includes the processing of assertion and "once-only" subpatterns, which are handled like subroutine calls. Normally, these are never very deep, and the limit on the complexity of **pcre[16|32]_dfa_exec()** is controlled by the amount of workspace it is given. However, it is possible to write patterns with runaway infinite recursions; such patterns will cause **pcre[16|32]_dfa_exec()** to run out of stack. At present, there is no protection against this.

The comments that follow do NOT apply to **pcre[16|32]_dfa_exec()**; they are relevant only for **pcre[16|32]_exec()** without the JIT optimization.

Reducing pcre[16|32]_exec()'s stack usage

Each time that **match()** is actually called recursively, it uses memory from the process stack. For certain kinds of pattern and data, very large amounts of stack may be needed, despite the recognition of "tail recursion". You can often reduce the amount of recursion, and therefore the amount of stack used, by modifying the pattern that is being matched. Consider, for example, this pattern:

```
([^<]|<(?!inet))+
```

It matches from wherever it starts until it encounters "<inet" or the end of the data, and is the kind of pattern that might be used when processing an XML file. Each iteration of the outer parentheses matches either one character that is not "<" or a "<" that is not followed by "inet". However, each time a parenthesis is processed, a recursion occurs, so this formulation uses a stack frame for each matched character. For a long string, a lot of stack is required. Consider now this rewritten pattern, which matches exactly the same strings:

```
([^<]++|<(?!inet))+
```

This uses very much less stack, because runs of characters that do not contain "<" are "swallowed" in one item inside the parentheses. Recursion happens only when a "<" character that is not followed by "inet" is encountered (and we assume this is relatively rare). A possessive quantifier is used to stop any backtracking into the runs of non-"<" characters, but that is not related to stack usage.

This example shows that one way of avoiding stack problems when matching long subject strings is to write repeated parenthesized subpatterns to match more than one character whenever possible.

Compiling PCRE to use heap instead of stack for pcre[16|32]_exec()

In environments where stack memory is constrained, you might want to compile PCRE to use heap memory instead of stack for remembering back-up points when **pcre[16|32]_exec()** is running. This makes it run a lot more slowly, however. Details of how to do this are given in the **pcrebuild** documentation. When built in this way, instead of using the stack, PCRE obtains and frees memory by calling the functions that are pointed to by the **pcre[16|32]_stack_malloc** and **pcre[16|32]_stack_free** variables. By default, these point to **malloc()** and **free()**, but you can replace the pointers to cause PCRE to use your own functions. Since the block sizes are always the same, and are always freed in reverse order, it may be possible to implement customized memory handlers that are more efficient than the standard functions.

Limiting pcre[16|32]_exec()'s stack usage

You can set limits on the number of times that **match()** is called, both in total and recursively. If a limit is exceeded, **pcre[16|32]_exec()** returns an error code. Setting suitable limits should prevent it from running out of stack. The default values of the limits are very large, and unlikely ever to operate. They can be changed when PCRE is built, and they can also be set when **pcre[16|32]_exec()** is called. For details of these interfaces, see the **pcrebuild** documentation and the section on extra data for **pcre[16|32]_exec()** in the **pcreapi** documentation.

As a very rough rule of thumb, you should reckon on about 500 bytes per recursion. Thus, if you want to limit your stack usage to 8Mb, you should set the limit at 16000 recursions. A 64Mb stack, on the other hand, can support around 128000 recursions.

In Unix-like environments, the **pcretest** test program has a command line option (**-S**) that can be used to increase the size of its stack. As long as the stack is large enough, another option (**-M**) can be used to find the smallest limits that allow a particular pattern to match a given subject string. This is done by calling **pcre[16|32]_exec()** repeatedly with different limits.

Obtaining an estimate of stack usage

The actual amount of stack used per recursion can vary quite a lot, depending on the compiler that was used to build PCRE and the optimization or debugging options that were set for it. The rule of thumb value of 500 bytes mentioned above may be larger or smaller than what is actually needed. A better approximation can be obtained by running this command:

```
pcretest -m -C
```

The **-C** option causes **pcretest** to output information about the options with which PCRE was compiled. When **-m** is also given (before **-C**), information about stack use is given in a line like this:

```
Match recursion uses stack: approximate frame size = 640 bytes
```

The value is approximate because some recursions need a bit more (up to perhaps 16 more bytes).

If the above command is given when PCRE is compiled to use the heap instead of the stack for recursion, the value that is output is the size of each block that is obtained from the heap.

Changing stack size in Unix-like systems

In Unix-like environments, there is not often a problem with the stack unless very long strings are involved, though the default limit on stack size varies from system to system. Values from 8Mb to 64Mb are common. You can find your default limit by running the command:

```
ulimit -s
```

Unfortunately, the effect of running out of stack is often SIGSEGV, though sometimes a more explicit error message is given. You can normally increase the limit on stack size by code such as this:

```
struct rlimit rlim;
getrlimit(RLIMIT_STACK, &rlim);
rlim.rlim_cur = 100*1024*1024;
setrlimit(RLIMIT_STACK, &rlim);
```

This reads the current limits (soft and hard) using **getrlimit()**, then attempts to increase the soft limit to 100Mb using **setrlimit()**. You must do this before calling **pcre[16|32]_exec()**.

Changing stack size in Mac OS X

Using **setrlimit()**, as described above, should also work on Mac OS X. It is also possible to set a stack size when linking a program. There is a discussion about stack sizes in Mac OS X at this web site: http://developer.apple.com/qa/qa2005/qa1419.html.

AUTHOR

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

REVISION

Last updated: 24 June 2012