NAME

PCRE2 - Perl-compatible regular expressions

PARTIAL MATCHING IN PCRE2

In normal use of PCRE2, if there is a match up to the end of a subject string, but more characters are needed to match the entire pattern, PCRE2_ERROR_NOMATCH is returned, just like any other failing match. There are circumstances where it might be helpful to distinguish this "partial match" case.

One example is an application where the subject string is very long, and not all available at once. The requirement here is to be able to do the matching segment by segment, but special action is needed when a matched substring spans the boundary between two segments.

Another example is checking a user input string as it is typed, to ensure that it conforms to a required format. Invalid characters can be immediately diagnosed and rejected, giving instant feedback.

Partial matching is a PCRE2-specific feature; it is not Perl-compatible. It is requested by setting one of the PCRE2_PARTIAL_HARD or PCRE2_PARTIAL_SOFT options when calling a matching function. The difference between the two options is whether or not a partial match is preferred to an alternative complete match, though the details differ between the two types of matching function. If both options are set, PCRE2_PARTIAL_HARD takes precedence.

If you want to use partial matching with just-in-time optimized code, as well as setting a partial match option for the matching function, you must also call **pcre2_jit_compile()** with one or both of these options:

```
PCRE2_JIT_PARTIAL_HARD
PCRE2_JIT_PARTIAL_SOFT
```

PCRE2_JIT_COMPLETE should also be set if you are going to run non-partial matches on the same pattern. Separate code is compiled for each mode. If the appropriate JIT mode has not been compiled, interpretive matching code is used.

Setting a partial matching option disables two of PCRE2's standard optimization hints. PCRE2 remembers the last literal code unit in a pattern, and abandons matching immediately if it is not present in the subject string. This optimization cannot be used for a subject string that might match only partially. PCRE2 also remembers a minimum length of a matching string, and does not bother to run the matching function on shorter strings. This optimization is also disabled for partial matching.

REQUIREMENTS FOR A PARTIAL MATCH

A possible partial match occurs during matching when the end of the subject string is reached successfully, but either more characters are needed to complete the match, or the addition of more characters might change what is matched.

Example 1: if the pattern is /abc/ and the subject is "ab", more characters are definitely needed to complete a match. In this case both hard and soft matching options yield a partial match.

Example 2: if the pattern is /ab+/ and the subject is "ab", a complete match can be found, but the addition of more characters might change what is matched. In this case, only PCRE2_PARTIAL_HARD returns a partial match; PCRE2_PARTIAL_SOFT returns the complete match.

On reaching the end of the subject, when PCRE2_PARTIAL_HARD is set, if the next pattern item is \z, \Z, \b, \B, or $ there is always a partial match. Otherwise, for both options, the next pattern item must be one that inspects a character, and at least one of the following must be true:

(1) At least one character has already been inspected. An inspected character need not form part of the final matched string; lookbehind assertions and the \K escape sequence provide ways of inspecting characters before the start of a matched string.

(2) The pattern contains one or more lookbehind assertions. This condition exists in case there is a lookbehind that inspects characters before the start of the match.

(3) There is a special case when the whole pattern can match an empty string. When the starting point is at the end of the subject, the empty string match is a possibility, and if PCRE2_PARTIAL_SOFT is set and neither of the above conditions is true, it is returned. However, because adding more characters might result in a non-empty match, PCRE2_PARTIAL_HARD returns a partial match, which in this case means "there is going to be a match at this point, but until some more characters are added, we do not know if it will be an empty string or something longer".

PARTIAL MATCHING USING pcre2_match()

When a partial matching option is set, the result of calling **pcre2_match()** can be one of the following:

A successful match
A complete match has been found, starting and ending within this subject.

PCRE2_ERROR_NOMATCH
No match can start anywhere in this subject.

PCRE2_ERROR_PARTIAL
Adding more characters may result in a complete match that uses one or more characters from the end of this subject.

When a partial match is returned, the first two elements in the ovector point to the portion of the subject that was matched, but the values in the rest of the ovector are undefined. The appearance of \K in the pattern has no effect for a partial match. Consider this pattern:

```
/abc\K123/
```

If it is matched against "456abc123xyz" the result is a complete match, and the ovector defines the matched string as "123", because \K resets the "start of match" point. However, if a partial match is requested and the subject string is "456abc12", a partial match is found for the string "abc12", because all these characters are needed for a subsequent re-match with additional characters.

If there is more than one partial match, the first one that was found provides the data that is returned. Consider this pattern:

```
/123\w+X|dogY/
```

If this is matched against the subject string "abc123dog", both alternatives fail to match, but the end of the subject is reached during matching, so PCRE2_ERROR_PARTIAL is returned. The offsets are set to 3 and 9, identifying "123dog" as the first partial match. (In this example, there are two partial matches, because "dog" on its own partially matches the second alternative.)

How a partial match is processed by pcre2_match()

What happens when a partial match is identified depends on which of the two partial matching options is set.

If PCRE2_PARTIAL_HARD is set, PCRE2_ERROR_PARTIAL is returned as soon as a partial match is found, without continuing to search for possible complete matches. This option is "hard" because it prefers an earlier partial match over a later complete match. For this reason, the assumption is made that the end of the supplied subject string is not the true end of the available data, which is why \z, \Z, \b, \B, and $ always give a partial match.

If PCRE2_PARTIAL_SOFT is set, the partial match is remembered, but matching continues as normal, and other alternatives in the pattern are tried. If no complete match can be found, PCRE2_ERROR_PARTIAL is returned instead of PCRE2_ERROR_NOMATCH. This option is "soft" because it prefers a complete match over a partial match. All the various matching items in a pattern behave as if the subject string is potentially complete; \z, \Z, and $ match at the end of the subject, as normal, and for \b and \B the end of the subject is treated as a non-alphanumeric.

The difference between the two partial matching options can be illustrated by a pattern such as:

```
/dog(sbody)?/
```

This matches either "dog" or "dogsbody", greedily (that is, it prefers the longer string if possible). If it is matched against the string "dog" with PCRE2_PARTIAL_SOFT, it yields a complete match for "dog". However, if PCRE2_PARTIAL_HARD is set, the result is PCRE2_ERROR_PARTIAL. On the other hand, if the pattern is made ungreedy the result is different:

```
/dog(sbody)??/
```

In this case the result is always a complete match because that is found first, and matching never continues after finding a complete match. It might be easier to follow this explanation by thinking of the two patterns like this:

```
/dog(sbody)?/    is the same as  /dogsbody|dog/
/dog(sbody)??/   is the same as  /dog|dogsbody/
```

The second pattern will never match "dogsbody", because it will always find the shorter match first.

Example of partial matching using pcre2test

The **pcre2test** data modifiers **partial_hard** (or **ph**) and **partial_soft** (or **ps**) set PCRE2_PARTIAL_HARD and PCRE2_PARTIAL_SOFT, respectively, when calling **pcre2_match()**. Here is a run of **pcre2test** using a pattern that matches the whole subject in the form of a date:

```
  re> /^\d?\d(jan|feb|mar|apr|may|jun|jul|aug|sep|oct|nov|dec)\d\d$/
data> 25dec3\=ph
Partial match: 23dec3
data> 3ju\=ph
Partial match: 3ju
data> 3juj\=ph
No match
```

This example gives the same results for both hard and soft partial matching options. Here is an example where there is a difference:

```
  re> /^\d?\d(jan|feb|mar|apr|may|jun|jul|aug|sep|oct|nov|dec)\d\d$/
data> 25jun04\=ps
 0: 25jun04
 1: jun
data> 25jun04\=ph
Partial match: 25jun04
```

With PCRE2_PARTIAL_SOFT, the subject is matched completely. For PCRE2_PARTIAL_HARD, however, the subject is assumed not to be complete, so there is only a partial match.

MULTI-SEGMENT MATCHING WITH pcre2_match()

PCRE was not originally designed with multi-segment matching in mind. However, over time, features (including partial matching) that make multi-segment matching possible have been added. A very long string can be searched segment by segment by calling **pcre2_match()** repeatedly, with the aim of achieving the same results that would happen if the entire string was available for searching all the time. Normally, the strings that are being sought are much shorter than each individual segment, and are in the middle of very long strings, so the pattern is normally not anchored.

Special logic must be implemented to handle a matched substring that spans a segment boundary. PCRE2_PARTIAL_HARD should be used, because it returns a partial match at the end of a segment whenever there is the possibility of changing the match by adding more characters. The PCRE2_NOTBOL option should also be set for all but the first segment.

When a partial match occurs, the next segment must be added to the current subject and the match re-run, using the *startoffset* argument of **pcre2_match()** to begin at the point where the partial match started. For example:

```
  re> /\d?\d(jan|feb|mar|apr|may|jun|jul|aug|sep|oct|nov|dec)\d\d/
data> ...the date is 23ja\=ph
Partial match: 23ja
data> ...the date is 23jan19 and on that day...\=offset=15
 0: 23jan19
 1: jan
```

Note the use of the **offset** modifier to start the new match where the partial match was found. In this example, the next segment was added to the one in which the partial match was found. This is the most straightforward approach, typically using a memory buffer that is twice the size of each segment. After a partial match, the first half of the buffer is discarded, the second half is moved to the start of the buffer, and a new segment is added before repeating the match as in the example above. After a no match, the entire buffer can be discarded.

If there are memory constraints, you may want to discard text that precedes a partial match before adding the next segment. Unfortunately, this is not at present straightforward. In cases such as the above, where the pattern does not contain any lookbehinds, it is sufficient to retain only the partially matched substring. However, if the pattern contains a lookbehind assertion, characters that precede the start of the partial match may have been inspected during the matching process. When **pcre2test** displays a partial match, it indicates these characters with '<' if the **allusedtext** modifier is set:

```
  re> "(?<=123)abc"
data> xx123ab\=ph,allusedtext
Partial match: 123ab
          <<<
```

However, the **allusedtext** modifier is not available for JIT matching, because JIT matching does not record the first (or last) consulted characters. For this reason, this information is not available via the API. It is therefore not possible in general to obtain the exact number of characters that must be retained in order to get the right match result. If you cannot retain the entire segment, you must find some heuristic way of choosing.

If you know the approximate length of the matching substrings, you can use that to decide how much text to retain. The only lookbehind information that is currently available via the API is the length of the longest individual lookbehind in a pattern, but this can be misleading if there are nested lookbehinds. The value returned by calling **pcre2_pattern_info()** with the PCRE2_INFO_MAXLOOKBEHIND option is the maximum number of characters (not code units) that any individual lookbehind moves back when it is processed. A pattern such as "(?<=(?<!b)a)" has a maximum lookbehind value of one, but inspects two characters before its starting point.

In a non-UTF or a 32-bit case, moving back is just a subtraction, but in UTF-8 or UTF-16 you have to count characters while moving back through the code units.

PARTIAL MATCHING USING pcre2_dfa_match()

The DFA function moves along the subject string character by character, without backtracking, searching for all possible matches simultaneously. If the end of the subject is reached before the end of the pattern, there is the possibility of a partial match.

When PCRE2_PARTIAL_SOFT is set, PCRE2_ERROR_PARTIAL is returned only if there have been no complete matches. Otherwise, the complete matches are returned. If PCRE2_PARTIAL_HARD is set, a partial match takes precedence over any complete matches. The portion of the string that was matched when the longest partial match was found is set as the first matching string.

Because the DFA function always searches for all possible matches, and there is no difference between greedy and ungreedy repetition, its behaviour is different from the **pcre2_match()**. Consider the string "dog" matched against this ungreedy pattern:

```
/dog(sbody)??/
```

Whereas the standard function stops as soon as it finds the complete match for "dog", the DFA function also finds the partial match for "dogsbody", and so returns that when PCRE2_PARTIAL_HARD is set.

MULTI-SEGMENT MATCHING WITH pcre2_dfa_match()

When a partial match has been found using the DFA matching function, it is possible to continue the match by providing additional subject data and calling the function again with the same compiled regular expression, this time setting the PCRE2_DFA_RESTART option. You must pass the same working space as before, because this is where details of the previous partial match are stored. You can set the PCRE2_PARTIAL_SOFT or PCRE2_PARTIAL_HARD options with PCRE2_DFA_RESTART to continue partial matching over multiple segments. Here is an example using **pcre2test**:

```
  re> /^\d?\d(jan|feb|mar|apr|may|jun|jul|aug|sep|oct|nov|dec)\d\d$/
data> 23ja\=dfa,ps
Partial match: 23ja
data> n05\=dfa,dfa_restart
 0: n05
```

The first call has "23ja" as the subject, and requests partial matching; the second call has "n05" as the subject for the continued (restarted) match. Notice that when the match is complete, only the last part is shown; PCRE2 does not retain the previously partially-matched string. It is up to the calling program to do that if it needs to. This means that, for an unanchored pattern, if a continued match fails, it is not possible to try again at a new starting point. All this facility is capable of doing is continuing with the previous match attempt. For example, consider this pattern:

```
1234|3789
```

If the first part of the subject is "ABC123", a partial match of the first alternative is found at offset 3. There is no partial match for the second alternative, because such a match does not start at the same point in the subject string. Attempting to continue with the string "7890" does not yield a match because only those alternatives that match at one point in the subject are remembered. Depending on the application, this may or may not be what you want.

If you do want to allow for starting again at the next character, one way of doing it is to retain some or all of the segment and try a new complete match, as described for **pcre2_match()** above. Another possibility is to work with two buffers. If a partial match at offset *n* in the first buffer is followed by "no match" when PCRE2_DFA_RESTART is used on the second buffer, you can then try a new match starting at offset *n+1* in the first buffer.

AUTHOR

Philip Hazel
University Computing Service
Cambridge, England.

REVISION

Last updated: 04 September 2019