NAME

PCRE2 - Perl-compatible regular expressions (revised API)

SAVING AND RE-USING PRECOMPILED PCRE2 PATTERNS

int32_t pcre2_serialize_decode(pcre2_code *****codes*,**
 int32_t *number_of_codes***, const uint32_t *****bytes***,**
 pcre2_general_context *****gcontext*);**

int32_t pcre2_serialize_encode(pcre2_code *****codes*,**
 int32_t *number_of_codes***, uint32_t *****serialized_bytes***,**
 PCRE2_SIZE *****serialized_size*, pcre2_general_context *****gcontext***);**

void pcre2_serialize_free(uint8_t *****bytes*);**

int32_t pcre2_serialize_get_number_of_codes(const uint8_t *****bytes*);**

If you are running an application that uses a large number of regular expression patterns, it may be useful to store them in a precompiled form instead of having to compile them every time the application is run. However, if you are using the just-in-time optimization feature, it is not possible to save and reload the JIT data, because it is position-dependent. The host on which the patterns are reloaded must be running the same version of PCRE2, with the same code unit width, and must also have the same endianness, pointer width and PCRE2_SIZE type. For example, patterns compiled on a 32-bit system using PCRE2's 16-bit library cannot be reloaded on a 64-bit system, nor can they be reloaded using the 8-bit library.

Note that "serialization" in PCRE2 does not convert compiled patterns to an abstract format like Java or .NET serialization. The serialized output is really just a bytecode dump, which is why it can only be reloaded in the same environment as the one that created it. Hence the restrictions mentioned above. Applications that are not statically linked with a fixed version of PCRE2 must be prepared to recompile patterns from their sources, in order to be immune to PCRE2 upgrades.

SECURITY CONCERNS

The facility for saving and restoring compiled patterns is intended for use within individual applications. As such, the data supplied to **pcre2_serialize_decode()** is expected to be trusted data, not data from arbitrary external sources. There is only some simple consistency checking, not complete validation of what is being re-loaded. Corrupted data may cause undefined results. For example, if the length field of a pattern in the serialized data is corrupted, the deserializing code may read beyond the end of the byte stream that is passed to it.

SAVING COMPILED PATTERNS

Before compiled patterns can be saved they must be serialized, which in PCRE2 means converting the pattern to a stream of bytes. A single byte stream may contain any number of compiled patterns, but they must all use the same character tables. A single copy of the tables is included in the byte stream (its size is 1088 bytes). For more details of character tables, see the section on locale support in the **pcre2api** documentation.

The function **pcre2_serialize_encode()** creates a serialized byte stream from a list of compiled patterns. Its first two arguments specify the list, being a pointer to a vector of pointers to compiled patterns, and the length of the vector. The third and fourth arguments point to variables which are set to point to the created byte stream and its length, respectively. The final argument is a pointer to a general context, which can be used to specify custom memory mangagement functions. If this argument is NULL, **malloc()** is used to obtain memory for the byte stream. The yield of the function is the number of serialized patterns, or one of the following negative error codes:

```
  PCRE2_ERROR_BADDATA      the number of patterns is zero or less
```

PCRE2_ERROR_BADMAGIC	mismatch of id bytes in one of the patterns
PCRE2_ERROR_MEMORY	memory allocation failed
PCRE2_ERROR_MIXEDTABLES	the patterns do not all use the same tables
PCRE2_ERROR_NULL	the 1st, 3rd, or 4th argument is NULL

PCRE2_ERROR_BADMAGIC means either that a pattern's code has been corrupted, or that a slot in the vector does not point to a compiled pattern.

Once a set of patterns has been serialized you can save the data in any appropriate manner. Here is sample code that compiles two patterns and writes them to a file. It assumes that the variable *fd* refers to a file that is open for output. The error checking that should be present in a real application has been omitted for simplicity.

```
int errorcode;
uint8_t *bytes;
PCRE2_SIZE erroroffset;
PCRE2_SIZE bytescount;
pcre2_code *list_of_codes[2];
list_of_codes[0] = pcre2_compile("first pattern",
  PCRE2_ZERO_TERMINATED, 0, &errorcode, &erroroffset, NULL);
list_of_codes[1] = pcre2_compile("second pattern",
  PCRE2_ZERO_TERMINATED, 0, &errorcode, &erroroffset, NULL);
errorcode = pcre2_serialize_encode(list_of_codes, 2, &bytes,
  &bytescount, NULL);
errorcode = fwrite(bytes, 1, bytescount, fd);
```

Note that the serialized data is binary data that may contain any of the 256 possible byte values. On systems that make a distinction between binary and non-binary data, be sure that the file is opened for binary output.

Serializing a set of patterns leaves the original data untouched, so they can still be used for matching. Their memory must eventually be freed in the usual way by calling **pcre2_code_free()**. When you have finished with the byte stream, it too must be freed by calling **pcre2_serialize_free()**. If this function is called with a NULL argument, it returns immediately without doing anything.

RE-USING PRECOMPILED PATTERNS

In order to re-use a set of saved patterns you must first make the serialized byte stream available in main memory (for example, by reading from a file). The management of this memory block is up to the application. You can use the **pcre2_serialize_get_number_of_codes()** function to find out how many compiled patterns are in the serialized data without actually decoding the patterns:

```
uint8_t *bytes = <serialized data>;
int32_t number_of_codes = pcre2_serialize_get_number_of_codes(bytes);
```

The **pcre2_serialize_decode()** function reads a byte stream and recreates the compiled patterns in new memory blocks, setting pointers to them in a vector. The first two arguments are a pointer to a suitable vector and its length, and the third argument points to a byte stream. The final argument is a pointer to a general context, which can be used to specify custom memory mangagement functions for the decoded patterns. If this argument is NULL, **malloc()** and **free()** are used. After deserialization, the byte stream is no longer needed and can be discarded.

```
int32_t number_of_codes;
pcre2_code *list_of_codes[2];
uint8_t *bytes = <serialized data>;
int32_t number_of_codes =
  pcre2_serialize_decode(list_of_codes, 2, bytes, NULL);
```

If the vector is not large enough for all the patterns in the byte stream, it is filled with those that fit, and the remainder are ignored. The yield of the function is the number of decoded patterns, or one of the following negative error codes:

PCRE2_ERROR_BADDATA	second argument is zero or less
PCRE2_ERROR_BADMAGIC	mismatch of id bytes in the data
PCRE2_ERROR_BADMODE	mismatch of code unit size or PCRE2 version
PCRE2_ERROR_BADSERIALIZEDDATA	other sanity check failure
PCRE2_ERROR_MEMORY	memory allocation failed
PCRE2_ERROR_NULL	first or third argument is NULL

PCRE2_ERROR_BADMAGIC may mean that the data is corrupt, or that it was compiled on a system with different endianness.

Decoded patterns can be used for matching in the usual way, and must be freed by calling **pcre2_code_free()**. However, be aware that there is a potential race issue if you are using multiple patterns that were decoded from a single byte stream in a multithreaded application. A single copy of the character tables is used by all the decoded patterns and a reference count is used to arrange for its memory to be automatically freed when the last pattern is freed, but there is no locking on this reference count. Therefore, if you want to call **pcre2_code_free()** for these patterns in different threads, you must arrange your own locking, and ensure that **pcre2_code_free()** cannot be called by two threads at the same time.

If a pattern was processed by **pcre2_jit_compile()** before being serialized, the JIT data is discarded and so is no longer available after a save/restore cycle. You can, however, process a restored pattern with **pcre2_jit_compile()** if you wish.

AUTHOR

Philip Hazel
University Computing Service
Cambridge, England.

REVISION

Last updated: 27 June 2018