NAME

perlreapi - perl regular expression plugin interface

DESCRIPTION

As of Perl 5.9.5 there is a new interface for plugging and using other regular expression engines than the default one.

Each engine is supposed to provide access to a constant structure of the following format:

```
    typedef struct regexp_engine {
        REGEXP* (*comp) (pTHX_ const SV * const pattern, const U32 flags);
        I32     (*exec) (pTHX_ REGEXP * const rx, char* stringarg, char*
strend,
                         char* strbeg, I32 minend, SV* screamer,
                         void* data, U32 flags);
        char*   (*intuit) (pTHX_ REGEXP * const rx, SV *sv, char *strpos,
                           char *strend, U32 flags,
                           struct re_scream_pos_data_s *data);
        SV*     (*checkstr) (pTHX_ REGEXP * const rx);
        void    (*free) (pTHX_ REGEXP * const rx);
        void    (*numbered_buff_FETCH) (pTHX_ REGEXP * const rx, const I32
paren,
                                SV * const sv);
        void    (*numbered_buff_STORE) (pTHX_ REGEXP * const rx, const I32
paren,
                                    SV const * const value);
        I32     (*numbered_buff_LENGTH) (pTHX_ REGEXP * const rx, const SV
* const sv,
                                     const I32 paren);
        SV*     (*named_buff) (pTHX_ REGEXP * const rx, SV * const key,
                               SV * const value, U32 flags);
        SV*     (*named_buff_iter) (pTHX_ REGEXP * const rx, const SV *
const lastkey,
                                    const U32 flags);
        SV*     (*qr_package)(pTHX_ REGEXP * const rx);
    #ifdef USE_ITHREADS
        void*   (*dupe) (pTHX_ REGEXP * const rx, CLONE_PARAMS *param);
    #endif
```

When a regexp is compiled, its `engine` field is then set to point at the appropriate structure, so that when it needs to be used Perl can find the right routines to do so.

In order to install a new regexp handler, `$^H{regcomp}` is set to an integer which (when casted appropriately) resolves to one of these structures. When compiling, the `comp` method is executed, and the resulting regexp structure's engine field is expected to point back at the same structure.

The pTHX_ symbol in the definition is a macro used by perl under threading to provide an extra argument to the routine holding a pointer back to the interpreter that is executing the regexp. So under threading all routines get an extra argument.

Callbacks

comp

```
    REGEXP* comp(pTHX_ const SV * const pattern, const U32 flags);
```

Compile the pattern stored in `pattern` using the given `flags` and return a pointer to a prepared REGEXP structure that can perform the match. See *The REGEXP structure* below for an explanation

of the individual fields in the REGEXP struct.

The `pattern` parameter is the scalar that was used as the pattern. previous versions of perl would pass two `char*` indicating the start and end of the stringified pattern, the following snippet can be used to get the old parameters:

```
STRLEN plen;
char*  exp = SvPV(pattern, plen);
char* xend = exp + plen;
```

Since any scalar can be passed as a pattern it's possible to implement an engine that does something with an array (`"ook" =~ [ qw/ eek hlagh / ]`) or with the non-stringified form of a compiled regular expression (`"ook" =~ qr/eek/`). perl's own engine will always stringify everything using the snippet above but that doesn't mean other engines have to.

The `flags` parameter is a bitfield which indicates which of the `msixp` flags the regex was compiled with. It also contains additional info such as whether `use locale` is in effect.

The `eogc` flags are stripped out before being passed to the comp routine. The regex engine does not need to know whether any of these are set as those flags should only affect what perl does with the pattern and its match variables, not how it gets compiled and executed.

By the time the comp callback is called, some of these flags have already had effect (noted below where applicable). However most of their effect occurs after the comp callback has run in routines that read the `rx->extflags` field which it populates.

In general the flags should be preserved in `rx->extflags` after compilation, although the regex engine might want to add or delete some of them to invoke or disable some special behavior in perl. The flags along with any special behavior they cause are documented below:

The pattern modifiers:

`/m` - RXf_PMf_MULTILINE

> If this is in `rx->extflags` it will be passed to `Perl_fbm_instr` by `pp_split` which will treat the subject string as a multi-line string.

`/s` - RXf_PMf_SINGLELINE

`/i` - RXf_PMf_FOLD

`/x` - RXf_PMf_EXTENDED

> If present on a regex # comments will be handled differently by the tokenizer in some cases.
>
> TODO: Document those cases.

`/p` - RXf_PMf_KEEPCOPY

Additional flags:

RXf_PMf_LOCALE

> Set if `use locale` is in effect. If present in `rx->extflags` `split` will use the locale dependent definition of whitespace under when RXf_SKIPWHITE or RXf_WHITE are in effect. Under ASCII whitespace is defined as per *isSPACE*, and by the internal macros `is_utf8_space` under UTF-8 and `isSPACE_LC` under `use locale`.

RXf_UTF8

> Set if the pattern is *SvUTF8()*, set by Perl_pmruntime.
>
> A regex engine may want to set or disable this flag during compilation. The perl engine for instance may upgrade non-UTF-8 strings to UTF-8 if the pattern includes constructs such as `\x{...}` that can only match Unicode values.

RXf_SPLIT

If `split` is invoked as `split ' '` or with no arguments (which really means `split(' ', $_)`, see *split*), perl will set this flag. The regex engine can then check for it and set the SKIPWHITE and WHITE extflags. To do this the perl engine does:

```
if (flags & RXf_SPLIT && r->prelen == 1 && r->precomp[0] == ' ')
    r->extflags |= (RXf_SKIPWHITE|RXf_WHITE);
```

These flags can be set during compilation to enable optimizations in the `split` operator.

RXf_SKIPWHITE

If the flag is present in `rx->extflags` `split` will delete whitespace from the start of the subject string before it's operated on. What is considered whitespace depends on whether the subject is a UTF-8 string and whether the `RXf_PMf_LOCALE` flag is set.

If RXf_WHITE is set in addition to this flag `split` will behave like `split " "` under the perl engine.

RXf_START_ONLY

Tells the split operator to split the target string on newlines (`\n`) without invoking the regex engine.

Perl's engine sets this if the pattern is `/^/` (`plen == 1 && *exp == '^'`), even under `/^/s`, see *split*. Of course a different regex engine might want to use the same optimizations with a different syntax.

RXf_WHITE

Tells the split operator to split the target string on whitespace without invoking the regex engine. The definition of whitespace varies depending on whether the target string is a UTF-8 string and on whether RXf_PMf_LOCALE is set.

Perl's engine sets this flag if the pattern is `\s+`.

RXf_NULL

Tells the split operator to split the target string on characters. The definition of character varies depending on whether the target string is a UTF-8 string.

Perl's engine sets this flag on empty patterns, this optimization makes `split //` much faster than it would otherwise be. It's even faster than `unpack`.

exec

```
I32 exec(pTHX_ REGEXP * const rx,
         char *stringarg, char* strend, char* strbeg,
         I32 minend, SV* screamer,
         void* data, U32 flags);
```

Execute a regexp.

intuit

```
char* intuit(pTHX_ REGEXP * const rx,
             SV *sv, char *strpos, char *strend,
             const U32 flags, struct re_scream_pos_data_s *data);
```

Find the start position where a regex match should be attempted, or possibly whether the regex engine should not be run because the pattern can't match. This is called as appropriate by the core depending on the values of the extflags member of the regexp structure.

checkstr

```
SV* checkstr(pTHX_ REGEXP * const rx);
```

Return a SV containing a string that must appear in the pattern. Used by `split` for optimising matches.

free

```
void free(pTHX_ REGEXP * const rx);
```

Called by perl when it is freeing a regexp pattern so that the engine can release any resources pointed to by the `pprivate` member of the regexp structure. This is only responsible for freeing private data; perl will handle releasing anything else contained in the regexp structure.

Numbered capture callbacks

Called to get/set the value of `$``, `$'`, $& and their named equivalents, ${^PREMATCH}, ${^POSTMATCH} and $^{MATCH}, as well as the numbered capture buffers ($1, $2, ...).

The `paren` parameter will be `-2` for `$``, `-1` for `$'`, `0` for $&, `1` for `$1` and so forth.

The names have been chosen by analogy with *Tie::Scalar* methods names with an additional **LENGTH** callback for efficiency. However named capture variables are currently not tied internally but implemented via magic.

numbered_buff_FETCH

```
void numbered_buff_FETCH(pTHX_ REGEXP * const rx, const I32 paren,
                         SV * const sv);
```

Fetch a specified numbered capture. `sv` should be set to the scalar to return, the scalar is passed as an argument rather than being returned from the function because when it's called perl already has a scalar to store the value, creating another one would be redundant. The scalar can be set with `sv_setsv`, `sv_setpvn` and friends, see *perlapi*.

This callback is where perl untaints its own capture variables under taint mode (see *perlsec*). See the `Perl_reg_numbered_buff_fetch` function in *regcomp.c* for how to untaint capture variables if that's something you'd like your engine to do as well.

numbered_buff_STORE

```
    void    (*numbered_buff_STORE) (pTHX_ REGEXP * const rx, const I32
paren,
                                    SV const * const value);
```

Set the value of a numbered capture variable. `value` is the scalar that is to be used as the new value. It's up to the engine to make sure this is used as the new value (or reject it).

Example:

```
if ("ook" =~ /(o*)/) {
    # `paren' will be `1' and `value' will be `ee'
    $1 =~ tr/o/e/;
}
```

Perl's own engine will croak on any attempt to modify the capture variables, to do this in another engine use the following callback (copied from `Perl_reg_numbered_buff_store`):

```
void
Example_reg_numbered_buff_store(pTHX_ REGEXP * const rx, const I32
```

```
paren,              SV const * const value)
    {
        PERL_UNUSED_ARG(rx);
        PERL_UNUSED_ARG(paren);
        PERL_UNUSED_ARG(value);

        if (!PL_localizing)
            Perl_croak(aTHX_ PL_no_modify);
    }
```

Actually perl will not *always* croak in a statement that looks like it would modify a numbered capture variable. This is because the STORE callback will not be called if perl can determine that it doesn't have to modify the value. This is exactly how tied variables behave in the same situation:

```
package CaptureVar;
use base 'Tie::Scalar';

sub TIESCALAR { bless [] }
sub FETCH { undef }
sub STORE { die "This doesn't get called" }

package main;

tie my $sv => "CatptureVar";
$sv =~ y/a/b/;
```

Because `$sv` is `undef` when the `y///` operator is applied to it the transliteration won't actually execute and the program won't `die`. This is different to how 5.8 and earlier versions behaved since the capture variables were READONLY variables then, now they'll just die when assigned to in the default engine.

numbered_buff_LENGTH

```
I32 numbered_buff_LENGTH (pTHX_ REGEXP * const rx, const SV * const sv,
                          const I32 paren);
```

Get the `length` of a capture variable. There's a special callback for this so that perl doesn't have to do a FETCH and run `length` on the result, since the length is (in perl's case) known from an offset stored in `rx->offs` this is much more efficient:

```
I32 s1  = rx->offs[paren].start;
I32 s2  = rx->offs[paren].end;
I32 len = t1 - s1;
```

This is a little bit more complex in the case of UTF-8, see what `Perl_reg_numbered_buff_length` does with *is_utf8_string_loclen*.

Named capture callbacks

Called to get/set the value of `%+` and `%-` as well as by some utility functions in *re*.

There are two callbacks, `named_buff` is called in all the cases the FETCH, STORE, DELETE, CLEAR, EXISTS and SCALAR *Tie::Hash* callbacks would be on changes to `%+` and `%-` and `named_buff_iter` in the same cases as FIRSTKEY and NEXTKEY.

The `flags` parameter can be used to determine which of these operations the callbacks should respond to, the following flags are currently defined:

Which *Tie::Hash* operation is being performed from the Perl level on %+ or %+, if any:

```
RXapif_FETCH
RXapif_STORE
RXapif_DELETE
RXapif_CLEAR
RXapif_EXISTS
RXapif_SCALAR
RXapif_FIRSTKEY
RXapif_NEXTKEY
```

Whether %+ or %- is being operated on, if any.

```
RXapif_ONE /* %+ */
RXapif_ALL /* %- */
```

Whether this is being called as `re::regname`, `re::regnames` or `re::regnames_count`, if any. The first two will be combined with `RXapif_ONE` or `RXapif_ALL`.

```
RXapif_REGNAME
RXapif_REGNAMES
RXapif_REGNAMES_COUNT
```

Internally %+ and %- are implemented with a real tied interface via *Tie::Hash::NamedCapture*. The methods in that package will call back into these functions. However the usage of *Tie::Hash::NamedCapture* for this purpose might change in future releases. For instance this might be implemented by magic instead (would need an extension to mgvtbl).

named_buff

```
SV*     (*named_buff) (pTHX_ REGEXP * const rx, SV * const key,
                       SV * const value, U32 flags);
```

named_buff_iter

```
SV*     (*named_buff_iter) (pTHX_ REGEXP * const rx, const SV * const
lastkey,
                            const U32 flags);
```

qr_package

```
SV* qr_package(pTHX_ REGEXP * const rx);
```

The package the qr// magic object is blessed into (as seen by `ref qr//`). It is recommended that engines change this to their package name for identification regardless of whether they implement methods on the object.

The package this method returns should also have the internal `Regexp` package in its `@ISA`. `qr//-isa("Regexp")>` should always be true regardless of what engine is being used.

Example implementation might be:

```
SV*
Example_qr_package(pTHX_ REGEXP * const rx)
{
 PERL_UNUSED_ARG(rx);
 return newSVpvs("re::engine::Example");
}
```

Any method calls on an object created with `qr//` will be dispatched to the package as a normal object.

```
use re::engine::Example;
my $re = qr//;
$re->meth; # dispatched to re::engine::Example::meth()
```

To retrieve the REGEXP object from the scalar in an XS function use the SvRX macro, see *"REGEXP Functions" in perlapi*.

```
void meth(SV * rv)
PPCODE:
    REGEXP * re = SvRX(sv);
```

dupe

```
void* dupe(pTHX_ REGEXP * const rx, CLONE_PARAMS *param);
```

On threaded builds a regexp may need to be duplicated so that the pattern can be used by multiple threads. This routine is expected to handle the duplication of any private data pointed to by the `pprivate` member of the regexp structure. It will be called with the preconstructed new regexp structure as an argument, the `pprivate` member will point at the **old** private structure, and it is this routine's responsibility to construct a copy and return a pointer to it (which perl will then use to overwrite the field as passed to this routine.)

This allows the engine to dupe its private data but also if necessary modify the final structure if it really must.

On unthreaded builds this field doesn't exist.

The REGEXP structure

The REGEXP struct is defined in *regexp.h*. All regex engines must be able to correctly build such a structure in their *comp* routine.

The REGEXP structure contains all the data that perl needs to be aware of to properly work with the regular expression. It includes data about optimisations that perl can use to determine if the regex engine should really be used, and various other control info that is needed to properly execute patterns in various contexts such as is the pattern anchored in some way, or what flags were used during the compile, or whether the program contains special constructs that perl needs to be aware of.

In addition it contains two fields that are intended for the private use of the regex engine that compiled the pattern. These are the `intflags` and `pprivate` members. `pprivate` is a void pointer to an arbitrary structure whose use and management is the responsibility of the compiling engine. perl will never modify either of these values.

```
typedef struct regexp {
    /* what engine created this regexp? */
    const struct regexp_engine* engine;

    /* what re is this a lightweight copy of? */
    struct regexp* mother_re;

    /* Information about the match that the perl core uses to manage
things */
    U32 extflags;   /* Flags used both externally and internally */
    I32 minlen;     /* mininum possible length of string to match */
```

```
            I32 minlenret;  /* mininum possible length of $& */
            U32 gofs;       /* chars left of pos that we search from */

            /* substring data about strings that must appear
               in the final match, used for optimisations */
            struct reg_substr_data *substrs;

            U32 nparens;  /* number of capture buffers */

            /* private engine specific data */
            U32 intflags;   /* Engine Specific Internal flags */
            void *pprivate; /* Data private to the regex engine which
                               created this object. */

            /* Data about the last/current match. These are modified during
        matching*/
            U32 lastparen;            /* last open paren matched */
            U32 lastcloseparen;       /* last close paren matched */
            regexp_paren_pair *swap;  /* Swap copy of *offs */
            regexp_paren_pair *offs;  /* Array of offsets for (@-) and (@+) */

            char *subbeg;  /* saved or original string so \digit works forever.
             */
            SV_SAVED_COPY  /* If non-NULL, SV which is COW from original */
            I32 sublen;    /* Length of string pointed by subbeg */

            /* Information about the match that isn't often used */
            I32 prelen;           /* length of precomp */
            const char *precomp;  /* pre-compilation regular expression */

            char *wrapped;  /* wrapped version of the pattern */
            I32 wraplen;    /* length of wrapped */

            I32 seen_evals;   /* number of eval groups in the pattern - for
        security checks */
            HV *paren_names;  /* Optional hash of paren names */

            /* Refcount of this regexp */
            I32 refcnt;             /* Refcount of this regexp */
        } regexp;
```

The fields are discussed in more detail below:

engine

This field points at a regexp_engine structure which contains pointers to the subroutines that are to be used for performing a match. It is the compiling routine's responsibility to populate this field before returning the regexp object.

Internally this is set to `NULL` unless a custom engine is specified in `$^H{regcomp}`, perl's own set of callbacks can be accessed in the struct pointed to by `RE_ENGINE_PTR`.

mother_re

TODO, see *http://www.mail-archive.com/perl5-changes@perl.org/msg17328.html*

extflags

This will be used by perl to see what flags the regexp was compiled with, this will normally be set to the value of the flags parameter by the *comp* callback. See the *comp* documentation for valid flags.

minlen minlenret

The minimum string length required for the pattern to match. This is used to prune the search space by not bothering to match any closer to the end of a string than would allow a match. For instance there is no point in even starting the regex engine if the minlen is 10 but the string is only 5 characters long. There is no way that the pattern can match.

`minlenret` is the minimum length of the string that would be found in $& after a match.

The difference between `minlen` and `minlenret` can be seen in the following pattern:

```
/ns(?=\d)/
```

where the `minlen` would be 3 but `minlenret` would only be 2 as the \d is required to match but is not actually included in the matched content. This distinction is particularly important as the substitution logic uses the `minlenret` to tell whether it can do in-place substitution which can result in considerable speedup.

gofs

Left offset from pos() to start match at.

substrs

Substring data about strings that must appear in the final match. This is currently only used internally by perl's engine for but might be used in the future for all engines for optimisations.

nparens, lasparen, and lastcloseparen

These fields are used to keep track of how many paren groups could be matched in the pattern, which was the last open paren to be entered, and which was the last close paren to be entered.

intflags

The engine's private copy of the flags the pattern was compiled with. Usually this is the same as `extflags` unless the engine chose to modify one of them.

pprivate

A void* pointing to an engine-defined data structure. The perl engine uses the `regexp_internal` structure (see *"Base Structures" in perlreguts*) but a custom engine should use something else.

swap

TODO: document

offs

A `regexp_paren_pair` structure which defines offsets into the string being matched which correspond to the $& and $1, $2 etc. captures, the `regexp_paren_pair` struct is defined as follows:

```
typedef struct regexp_paren_pair {
    I32 start;
    I32 end;
} regexp_paren_pair;
```

If `->offs[num].start` or `->offs[num].end` is `-1` then that capture buffer did not match.

`->offs[0].start/end` represents `$&` (or `${^MATCH` under `//p`) and `->offs[paren].end` matches `$$paren` where `$paren` = 1>.

precomp prelen

Used for optimisations. `precomp` holds a copy of the pattern that was compiled and `prelen` its length. When a new pattern is to be compiled (such as inside a loop) the internal `regcomp` operator checks whether the last compiled `REGEXP`'s `precomp` and `prelen` are equivalent to the new one, and if so uses the old pattern instead of compiling a new one.

The relevant snippet from `Perl_pp_regcomp`:

```
 if (!re || !re->precomp || re->prelen != (I32)len ||
     memNE(re->precomp, t, len))
        /* Compile a new pattern */
```

paren_names

This is a hash used internally to track named capture buffers and their offsets. The keys are the names of the buffers the values are dualvars, with the IV slot holding the number of buffers with the given name and the pv being an embedded array of I32. The values may also be contained independently in the data array in cases where named backreferences are used.

substrs

Holds information on the longest string that must occur at a fixed offset from the start of the pattern, and the longest string that must occur at a floating offset from the start of the pattern. Used to do Fast-Boyer-Moore searches on the string to find out if its worth using the regex engine at all, and if so where in the string to search.

subbeg sublen saved_copy

Used during execution phase for managing search and replace patterns.

wrapped wraplen

Stores the string `qr//` stringifies to. The perl engine for example stores `(?-xism:eek)` in the case of `qr/eek/`.

When using a custom engine that doesn't support the `(?:)` construct for inline modifiers, it's probably best to have `qr//` stringify to the supplied pattern, note that this will create undesired patterns in cases such as:

```
my $x = qr/a|b/;  # "a|b"
my $y = qr/c/i;   # "c"
my $z = qr/$x$y/; # "a|bc"
```

There's no solution for this problem other than making the custom engine understand a construct like `(?:)`.

seen_evals

This stores the number of eval groups in the pattern. This is used for security purposes when embedding compiled regexes into larger patterns with `qr//`.

refcnt

The number of times the structure is referenced. When this falls to 0 the regexp is automatically freed by a call to pregfree. This should be set to 1 in each engine's *comp* routine.

HISTORY

Originally part of *perlreguts*.

AUTHORS

Originally written by Yves Orton, expanded by Ævar Arnfjörð Bjarmason.

LICENSE