

NOVEDADES EN PERL 5.10

Enrique Nell
Barcelona Perl Mongers

Octubre de 2008

Perl 5.10

- Primera actualización importante de Perl desde 2002
(1994 => Perl 5, 2002 => Perl 5.8)
- Lanzada el 18 de diciembre de 2007, día del vigésimo aniversario de Perl
- Compatible con las versiones anteriores

Nuevo pragma: feature

- Se usa para activar nuevos elementos de sintaxis que pueden entrar en conflicto con elementos o nombres subrutinas de versiones anteriores del lenguaje.
En Perl 5.10.0 hay 3: **switch**, **state**, **say**
- Se pueden activar individualmente:

use feature 'say';
use feature qw(switch say);
- O en grupos: use feature ':5.10'; (prefijo ':')

Nuevo pragma: feature (cont.)

- En la versión Perl 5.10.X, **use feature ‘:5.10’** equivaldrá a **use feature ‘:5.10.X’**
- También se pueden cargar implícitamente:
`use 5.10.0;`
`use 5.010;` (esta variante no muestra advertencias de portabilidad)

Nuevo pragma: feature (cont.)

- **feature** es un pragma léxico (sólo está vigente hasta el final del bloque actual)
- Para desactivar una característica en un bloque de código se usa **no feature**.
P. ej.: no feature 'switch';
- En la línea de comandos (one-liners) se pueden cargar utilizando el modificador -E en lugar de -e.

say

- Nueva función similar a **print** que añade un carácter de nueva línea a continuación del texto que se va a imprimir.
 - **print** “algo interesante\n”; equivale a
say “algo interesante”;
- Aquí nos ahorramos 4 caracteres.

say (cont.)

- **print** una_subrutina(), "\n";

say una_subrutina();

En este caso nos ahorramos 7 caracteres.

- También evita problemas de interpolación cuando se escriben por error comillas simples:

```
print 'otra cadena\n'; # imprime \n por error
```

state

- En Perl 5.10 hay un nuevo tipo de variables léxicas, denominadas **variables de estado**, que favorecen la encapsulación.
- En una subrutina, el valor de estas variables se conserva entre llamadas.
- Ofrecen una alternativa a las variables globales (lo que evita los efectos colaterales) y una sintaxis más simple que la de los cierres (*closures*).

state (cont.)

- Se declaran con la palabra clave **state**.
- Ejemplo:

use feature 'state';

```
sub incrementar { state $x = 42; return ++$x }
```

La declaración y asignación `state $x = 42` se realiza en tiempo de compilación.

Smart match (~~)

- Realiza una “comparación astuta” de sus argumentos.
- Permite comparar escalares, arrays, hashes, escalares con expresiones regulares, etc.
- Permite buscar escalares o expresiones regulares en arrays o referencias de arrays.
- Es conmutativo: $\$x \sim\sim \y devuelve el mismo resultado que $\$y \sim\sim \x .

Smart match (~~) (cont.)

- Ejemplos:

`$a ~~ /\D+?\d/`

`$a ~~ @valores (i@valores contiene $a?)`

`$a ~~ $valores_eref (i@valores contiene $a?)`

`/^\w+\b\s/ ~~ @valores`

`$a ~~ %hash (i$a es una clave de %hash?)`

Smart match (~~) (cont.)

- Más ejemplos:

@a ~~ @b (¿arrays idénticos?)

%a ~~ %b (¿contienen las mismas claves?)

\$sub_ref ~~ \$arg (¿\$sub_ref->(\$arg) = true?)

- Se puede modificar la forma de comparar objetos sobrecargando el operador ~~.

switch

- Ha tardado en llegar, pero es mucho más flexible que la existente en otros lenguajes, como C o Java, y proporciona mayor expresividad.
- Se implementa con las palabras clave: **given**, **when**, **default**, **continue** y **break**.
- Se activa con **use feature 'switch'**;

switch (cont.)

- given (\$var) {
 when (EXPR1) { código1 }
 when (EXPR2) { código2 }

 default { códigoN }
}

switch (cont.)

- Los argumentos de **given** y **when** deben escribirse entre paréntesis.
- **given** asigna el valor de su argumento a $\$_{..}$.
- En la cláusula **when**(EXPR) se realiza un “smart match” implícito de EXPR con $\$_{..}$.
- EXPR también puede ser una mera comparación booleana.

switch (cont.)

- La palabra clave **break** se usa para salir del bloque **given**. El código de una cláusula **when** finaliza implícitamente con una instrucción **break**.
- La palabra clave **continue** se usa para pasar a la siguiente cláusula **when**.

foreach .. when

- use feature 'switch';

```
foreach (@quiniela) {  
    when ("1") { $uno++ }  
    when ("X") { $equis++ }  
    when ("2") { $dos++ }  
}
```

foreach .. when (cont.)

- Cuando se activa una cláusula **when**, se pasa a la siguiente iteración del bucle (a menos que se use **continue** en el código de la cláusula).
- Sólo funciona con la variable predeterminada `$_` para la iteración.

Defined-OR (//)

- El operador OR (||) permite determinar si una variable es verdadera o falsa, pero no distingue si una variable es falsa (0 o "") o no está definida.
- $\$a \ || \ \b equivale a: **$\$a \ ? \ \$a \ : \ \$b$**
- El operador Defined-OR (//) proporciona una manera rápida de comprobar si una variable está definida.
- $\$a \ // \ \b equivale a: **$\text{defined } \$a \ ? \ \$a \ : \ \$b$**

Defined-OR (//) (cont.)

- $\$a \//= \b equivale a: **$\$a = \b unless defined $\$a$**

Si $\$a$ está definida, mantiene el valor. Si no lo está, se le asigna el valor $\$b$.

- Se puede usar para seleccionar el primer valor definido de un grupo de valores, o el último si no hay ningún valor definido:

$\$x = \$a \// \$b \//$ “Desconocido”;

Expresiones regulares

- Se ha reestructurado el motor de expresiones regulares (recursivo => iterativo). Ahora es más rápido.
- Perl 5.10 permite extender el motor de expresiones regulares o utilizar otros motores de expresiones regulares (p. ej., use re::engine::PCRE;)

Rx: Backreferences

- `\1`, ¿es una referencia a una captura previa ('backreference') o un valor octal?
- Nueva sintaxis para evitar ambigüedades. Ahora `\1` se puede escribir como `\g{1}` o `\g1`.
- 'Backreferences' relativas: `\g{-n}` o `\g-n` hace referencia a la n-ésima captura anterior

```
my $duplicadas = qr/(\w+)\s+\g{-1}/;
```

Rx: Captura con nombre

- Variables de captura con nombre: (?<nombre>rx)
- Las capturas se almacenan en el hash especial %+

Valor de captura denominada “nombre”: \$+{nombre}

- También se pueden utilizar los números de captura habituales

Rx: Captura con nombre (cont.)

- Si una expresión regular contiene el mismo nombre de variable de captura varias veces, en %+ sólo se almacenará la última captura.
- En %- se almacenan todas. El valor de cada clave del hash es una referencia a un array.
- Sintaxis de 'backreferences' de capturas con nombre:

`\k<nombre>` o `\k'nombre'`

Rx: Captura con nombre (cont.)

En las sustituciones, las capturas con nombre se usan con la nueva secuencia `\k`:

```
s{
    ^
    (?<inicial> \w+)
    (?<intermedia> .* ) \b
    (?<final> \w+)
    $
}
{\k<final>\k<intermedia>\k<inicial>}x;
```

Rx: Alternativa a `$``, `$&`, `$'`

- En Perl 5.10, el modificador `/p` ofrece el mismo resultado sin penalización de rendimiento:

`/regex/p`

Texto antes de la coincidencia: `${^PREMATCH}`

Texto coincidente: `${^MATCH}`

Texto después de la coincidencia: `${^POSTMATCH}`

Rx: Cuantificadores posesivos

- Para evitar la explosión combinatoria se usa “atomic matching”, (`?>`). Lo malo es que reduce la legibilidad de la expresión regular.
- Ahora se puede usar un cuantificador posesivo (un signo `+` a continuación del cuantificador principal):

`/(?>x+)/` se puede escribir así: `/x++/`

Rx: Cuantificadores posesivos (cont.)

- 'aaaa' = ~ m/ a+ a/x; Comparación positiva
(con backtracking)
- 'aaaa' = ~ m/ a++ a/x; Comparación negativa
(sin backtracking)

Rx: Depuración

- Ahora se puede usar **use re 'debug'**; con ámbito léxico para depurar expresiones regulares (en Perl 5.8 era global; devolvía información de depuración para todas las regex del programa).
- Se puede desactivar con **no re 'debug'**;

Variable `$_` léxica

- Se puede declarar `my $_;`
- Permite utilizar la variable predeterminada `$_` sin modificar el valor de `$_` en código que esté fuera del bloque actual.

Operadores de tests de ficheros acumulables

- Ahora se puede sustituir una expresión como

`-x $file && -w _ && -f _`

por

`-x -w -f $file`

Nuevos módulos

- Se han actualizado algunos módulos y se han incluido otros nuevos, entre ellos:

Module::Build, CPANPLUS, Hash::Util::FieldHash,
Module::CoreList

(lista completa en perl5100delta)

Nueva documentación

- Expresiones regulares: perlreapi, perlreguts
- Unicode: perlunitut, perlunifaq
- perlglossary
- perlcommunity
- perlpragma (crear pragmas léxicos personalizados)

Más novedades

- Se ha refactorizado, ordenado y optimizado el código fuente de Perl, y se ha mejorado la gestión de memoria.
- Mensajes de error más claros
- Función **sort()** optimizada (Salvador Fandiño)
- Mejor soporte para UTF-8

Todavía más...

- Mejoras en subprocessos (threads)
- Pragmas léxicos definidos por el usuario
- UNIVERSAL::DOES
- ... y mucho más.

La lista completa está en `perl5100delta.pod` (ver Referencias a continuación).

Referencias

- <http://search.cpan.org/~rgarcia/perl-5.10.0/pod/perl5100delta.pod>
- Perl Training Australia - Perl Tips
 - <http://perltraining.com.au/tips/2008-03-12.html>
 - <http://perltraining.com.au/tips/2008-04-18.html>
 - <http://perltraining.com.au/tips/2008-02-08.html>
 - <http://perltraining.com.au/tips/2008-03-03.html>
 - <http://perltraining.com.au/tips/2008-03-25.html>

Referencias (cont.)

- Perl is Dead, Long Live Perl! - Renée Baecker
The Perl Review (Spring 2007)
- Perl 5.10 is here - David M. Williams
ITWare.com (2/01/2008)
- Perl 5.10 Advanced Regular Expressions - Yves Orton

[http://www.regex-engineer.org/slides/
perl510_regex.html](http://www.regex-engineer.org/slides/perl510_regex.html)

Referencias (cont.)

- Perl 5.10 for People Who Aren't Totally Insane
Ricardo Signes, <http://www.slideshare.net/rjbs/perl-510-for-people-who-arent-totally-insane>
- Named Captures in Perl 5.9.5 - brian d foy
The Perl Review (Fall 2007)
- What's New in Perl 5.10 - Paul Fenwick
<http://perltraining.com.au/talks/>