

Pistas y trucos para programar fácilmente Perl en Vim

PICANDO LO JUSTO

El editor Vim tiene algunos trucos para ayudarnos a reducir las pulsaciones de teclas. En el artículo de este mes veremos cómo los programadores de Perl pueden ahorrar esfuerzos usando las técnicas de Vim.

POR MICHAEL SCHILLI

Puede que no haya otra decisión más importante en la vida de un programador que la elección de un editor. Una vez que hemos optado por Vi o Emacs, nos dedicaremos a amoldarnos a sus armas e intentaremos exprimir hasta el último gramo de rendimiento de nuestra herramienta favorita. Una vez que hemos elegido el editor, lo mejor

será aprender todo lo que podamos sobre él. Un uso efectivo del editor no sólo reduce el daño del síndrome del túnel carpiano, sino que también nos ayudará a obtener código más rápidamente y con pocos errores tipográficos.

El editor Vim (VI Improved, mejorado) tiene un cierto número de ventajas sobre su venerable predecesor Vi. A través de

los años, Vim se ha extendido drásticamente respaldado por un grupo incondicional de programadores. Es extremadamente configurable y extensible a través de plugins. De hecho Vim puede amoldarse a conveniencia para casi cualquier gusto o estilo de trabajo.

Vim guarda un archivo de configuración llamado `.vimrc` bajo el directorio de



inicio del usuario y éste es un buen sitio para almacenar los trucos que veremos a continuación.

Las distribuciones de Linux no siempre vienen con la última versión de Vim, así que no está de más que ejecutemos `vim --version` para verificar qué versión tenemos. Se necesita la versión 6.1 o posterior. Si tenemos una versión más antigua solamente tendremos que obtenerla de [2].

Resaltado de Sintaxis

El resaltado de las palabras claves y las estructuras en el código de los programas es de gran ayuda para los irritados ojos de alguien que ha pasado montones de horas leyéndolas. Vim tiene un resaltado de sintaxis excelente para diversos lenguajes de programación y asombrosamente se obtienen unos resultados muy precisos con él. Incluso para lenguajes tan complicados de analizar como Perl. En las Figuras 1a y 1b se puede apreciar cómo es

más fácil reconocer las estructuras de código con el resaltado por colores.

Por supuesto que, para esta funcionalidad, será necesario que nuestra Xterm permita el uso de colores. Si el resaltado de sintaxis no está habilitado por omisión en Vim, la orden `:syntax on` la activará. Vim evalúa la extensión del archivo (`.pl` o `.pm`) e incluso la secuencia `#!/usr/bin/perl` de la línea Shebang, para detectar el código Perl y realizar el resaltado de acuerdo a la sintaxis de Perl. Cuando comenzamos a editar un nuevo archivo que no tiene una extensión específica de Perl o una línea Shebang, se le puede indicar explícitamente a Vim el tipo de archivo tecleando: `:set filetype=perl`.

Atajos

Si siempre programamos en el mismo lenguaje, nos encontraremos tecleando las mismas secuencias una y otra vez. Como defensor de `Log::Log4perl`, he per-

dido la cuenta del número de veces que he tecleado `use Log::Log4perl qw(:easy);`. Afortunadamente, Vim me ha ayudado a poner fin a esta práctica.

La orden `:abbreviate ul4p use Log::Log4perl qw (:easy); <RETURN>` define a `ul4p` como un atajo. Cada vez que tecleemos el atajo en el modo de entrada de texto y pulsemos algo como la barra espaciadora o la tecla del retorno de carro, Vim expandirá la cadena automáticamente y nos dará nuestra línea `Log4perl`. El literal `<RETURN>`, que se coloca al final de la definición del atajo, simula la pulsación de la tecla del retorno de carro y por eso añade un corte de línea. Si queremos parar el modo de entrada y pasar al modo de órdenes, después de expandir un atajo, solamente hay que añadir `<ESC>`.

Otra forma de insertar secuencias de texto largas en los atajos es leyéndolas de un archivo: `:abb ul4p < BACKSPACE > < ESC > :r ~/.tmpl_4p < RETURN >`. Esta orden le dice a Vim que sustituya el atajo `ul4p` con el contenido del archivo que se ha indicado.

Macros de teclado

Las Macros se pueden usar para repetir pasos de edición recurrentes que modifi-

Listado 1: órdenes de Vim

```
01 # Buscar 'sub'
02 /sub
03 # Comenzar a
04 # grabar la macro
05 qa
06 # Insertar una línea encima,
07 # Volver a modo órdenes
08 O<ESC>
09 # Insertar 20 '#'
10 20i#<ESC>
11 # Copiar línea
12 yy
13 # Bajar una línea
14 # Insertar la línea
15 # copiada abajo.
16 jp
17 # Parar la grabación de la
   macro
18 q
19 # Buscar el siguiente 'sub'
20 n
21 # Reproducir la macro 'a'
22 @a
23 # ... repetir.
```

can múltiples áreas no continuas. Las Figuras 2a y 2b muestran tres cabeceras de función que queremos enmarcar con almohadillas.

Se necesitan las siguientes órdenes: Primero utilizamos `/sub` para buscar `sub`; después comenzamos la grabación de la macro `a`, dibujamos las almohadillas alrededor de la primera cabecera y paramos la grabación de la macro. Luego pulsamos `n` para buscar el siguiente `sub` y reproducimos la macro pulsando `@a`. En el Listado 1 se muestra una lista de las órdenes.

Si lo preferimos, podemos añadir las marcas cada vez que añadamos una nueva función, lo hacemos definiendo

un atajo de teclado para la tecla `F`, de la siguiente manera: `:map F o<ESC>43i#<ESC>yyosub {<ENTER><ESC>Pk$!`. A partir de ahora, cada vez que pulsemos la tecla `F`, en el modo de órdenes, Vim insertará una cabecera de función, cambiará al modo de entrada y colocará el cursor a la derecha de él para dejarnos escribir el nombre de la nueva función.

El revoltijo de letras en la definición del `mapa` incluye otra vez el típico estilo `vi` de órdenes con una sola tecla en el modo de órdenes, que seguro que reconocerán los incondicionales de `vi`. El número de marcas es una cuestión de gusto. Hemos usado 43 en la definición

anterior. Asignar las órdenes puede ahorrar mucho tiempo y molestias cuando se trata de secuencias que se repiten, como por ejemplo las cabeceras de las funciones. Si les gusta, pueden utilizar la misma aproximación para otras secuencias de texto que también se repitan, como por ejemplo, el código para recopilar parámetros de funciones como en `my(...) = @_;`

Otra tarea común es guardar el guión en el que estamos trabajando pulsando `:w` e invocar `perl -c script.pl`, para comprobar la sintaxis del guión. La siguiente orden asigna las acciones de guardar y verificar la sintaxis a la tecla `X` en modo de órdenes: `:nnoremap X :w`

Listado 2: ppitags

```

001 #!/usr/bin/perl -w
002 #####
003 # ppitags-PPI-basado en ctags
004 # Mike Schilli, 2005
005 # (m@perlmeister.com)
006 #####
007 use strict;
008
009 use PPI::Document;
010 use File::Find;
011 use Sysadm::Install qw(:all);
012 use Log::Log4perl qw(:easy);
013
014 my $outfile =
015 "$ENV{HOME}/.ptags.txt";
016 my %dirs = ();
017 my @found = ();
018
019 find \&file_wanted,
020 grep { $_ ne "." } @INC;
021
022 blurt
023 join("\n", sort @found),
024 $outfile;
025
026 #####
027 sub file_wanted {
028 #####
029 my $abs =
030 $File::Find::name;
031
032 # Evita dirs erroneos
033 $File::Find::prune = 1
034 if -d and $dirs{$abs}++;
035
036 # Solo módulos Perl
037 return unless /\.pm$/;
038
039 my $d =
040 PPI::Document->load(
041 $abs);
042
043 unless ($d) {
044 WARN "Cannot load $abs" .
045 " ($! @$)";
046 return;
047 }
048
049 # Encuentra paquetes y
050 # todas las subrutinas nom-
051 $d->find(
052 \&document_wanted);
053 }
054
055 #####
056 sub document_wanted {
057 #####
058 our $package;
059 my $tag;
060
061 if(ref( $_[1] ) eq
062 'PPI::Statement::Package'
063 ) {
064 $tag =
065 $_[1]->child(2)
066 ->content();
067 $package = $tag;
068
069 } elsif(ref($_[1]) eq
070 'PPI::Statement::Sub'
071 and $_[1]->name() {
072 $tag =
073 "$package\::";
074 . $_[1]->name();
075 }
076
077 return 1
078 unless defined $tag;
079
080 push @found,
081 $tag . "\t"
082 . $File::Find::name
083 . "\t"
084 . regex_from_node(
085 $_[1]);
086
087 return 1;
088 }
089
090 #####
091 sub regex_from_node {
092 #####
093 my ($node) = @_;
094
095 my $regex =
096 $node->content();
097
098 $regex =~ s/\n.*//gs;
099
100 while (
101 my $prev =
102 $node->previous_sibling()
103 ) {
104 last if $prev =~ /\n/;
105 $regex =
106 $prev->content()
107 . $regex;
108 $node = $prev;
109 }
110
111 $regex =~
112 s#[/.*[\]^$]#\&#g;
113
114 return "/^$regex/";
115 }

```

<Enter>:!perl -c % <Enter> .

Usando la orden `:noremap` en vez de `:map` nos aseguramos que la "X" no sea evaluada si es que aparece en el lado derecho de otra expresión asignada. Además, `:noremap` solamente expande la definición en el modo de órdenes. El símbolo % representa el nombre del archivo actual.

Autoformato

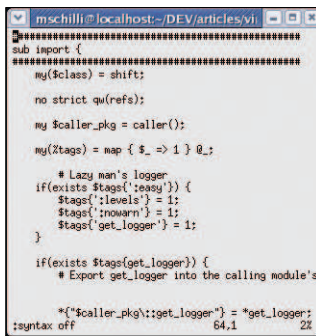
Si estamos componiendo un texto largo, por ejemplo documentación POD, probablemente nos llevará un tiempo, como seis o siete intentos, antes de conseguir un buen aspecto.

Si continuamente estamos añadiendo y borrando pasajes, terminaremos con unos párrafos de aspecto andrajoso y difíciles de revisar. Los procesadores de texto del estilo de Word, trabajan muy duro en segundo plano, reformateando continuamente el texto, pero los auténticos maestros pulen el filo ellos mismos.

Solamente necesitamos cuatro teclas en el modo de órdenes de Vim para hacer esto: `{gq}`. Primero, al pulsar `{` nos lleva al comienzo del párrafo actual, con la orden `gq` obtenemos el texto justificado a la izquierda y por último, `}` define dónde se aplica la orden: en este caso al final del párrafo.

El módulo Perl `Text::Autoformat`, del gran maestro Damian Conway, proporciona una aproximación muy elegante. Junto a la justificación a la izquierda, el módulo comprende todo tipo de estilos inteligentes: por ejemplo, puede manejar listas marcadas con `>` (las líneas seguidas de `>` se sangran de la misma forma) y maneja las comillas de los sangrados del email usando `>` o `>>`, o incluso corchetes, igual que lo haría un ser humano.

Para asignar la orden de formato, en el modo de órdenes, a la tecla `f`, usamos



```
mschilli@localhost:~/DEV/articles/vi
sub import {
my($class) = shift;
no strict qw(refs);
my $caller_pkg = caller();
my($tags) = map { $_ => 1 } @_;
# Lazy man's logger
if(exists $tags{'easy'}) {
$tags{'levels'} = 1;
$tags{'nowarn'} = 1;
$tags{'get_logger'} = 1;
}
if(exists $tags{'get_logger'}) {
# Export get_logger into the calling module's
*{"$caller_pkg\::get_logger"} = *get_logger;
:syntax: off 64,1 2K
```

Figura 1a: Un trozo de código Perl en VIM, sin ...



```
mschilli@localhost:~/DEV/articles/vi
sub import {
my($class) = shift;
no strict qw(refs);
my $caller_pkg = caller();
my($tags) = map { $_ => 1 } @_;
# Lazy man's logger
if(exists $tags{'easy'}) {
$tags{'levels'} = 1;
$tags{'nowarn'} = 1;
$tags{'get_logger'} = 1;
}
if(exists $tags{'get_logger'}) {
# Export get_logger into the calling module's
*{"$caller_pkg\::get_logger"} = *get_logger;
:syntax: on 64,1 2K
```

Figura 1b: ... y con el resaltado de sintaxis activado.

el cursor al siguiente carácter que haya, con `fe` nos colocará en la siguiente `e` del texto, por ejemplo. Si realmente queremos usar esta función, deberemos elegir otra tecla, o incluso realizar un atajo de teclado de dos caracteres: por ejemplo, `:map !f` espera que primero pulsemos `!` en modo de órdenes, antes de pulsar `f`.

Sangrado

La discusión sobre qué está bien o qué está mal al sangrar el código de los programas, no ha terminado. ¿Dónde se ponen las llaves? ¿Hasta qué punto necesitamos sangrar el código anidado? ¿Se deben utilizar espacios o tabuladores?

Como programadores todos tenemos nuestras propias preferencias, Vim permite escoger una opción.

El sangrado basado en tabuladores es una cuestión de gustos; muchos lo rechazan por principio. Si activamos la opción `:set expandtab`, Vim convertirá los tabuladores en espacios. Para establecer el número de espacios por tabulador, usamos `:set shiftwidth = 4`.

Pero no cometamos el error de utilizar `expandtab` sin reflexionar

`:map f !Gperl -Mtext::Autoformat -e'autoformat' <RETURN>`. Más tarde, durante la edición de un párrafo, simplemente cambiamos al modo de órdenes y pulsamos la tecla `f` con el cursor colocado en algún lugar del pasaje de texto y el texto se formateará automáticamente y correctamente. Las figuras 3a y 3b muestran el texto crudo y formateado.

Si el tamaño predeterminado de 72 líneas es demasiado ancho (o demasiado estrecho) para nuestro gusto, tenemos la opción de cambiarlo: `:map f !Gperl -Mtext::Autoformat -e'autoformat {right = >65}' <RETURN>` restringe el tamaño máximo de línea a 65 caracteres.

Los usuarios experimentados de Vim alegrarán que `f` está asignada por omisión en el modo de órdenes; llevando

acerca de lo que estamos haciendo, de lo contrario nos encontraremos con una desagradable sorpresa cuando editemos un Makefile. Los objetivos de Make están seguidos por órdenes sangradas con tabuladores, y reemplazarlos por espacios provocará un error de sintaxis. Para evitar esto, dejaremos que Vim detecte el tipo de archivo, utilizando `autocmd` y configurando la opción `expandtab` solamente para los programas en Perl:

```
:filetype on
:autocmd FileType perl :set expandtab
```

Para localizar problemas como este, la orden `:set list` nos permitirá ver los caracteres no imprimibles en Vim. Los tabuladores se ven como `^I` y el carácter de fin de línea se muestra como un '\$' azul. Con `:set nolist` volvemos al modo de visualización normal.

La opción `shiftwidth` que se mencionó anteriormente tiene otra función: En combinación con la opción `cindent` se puede usar `shiftwidth` para ahorrarnos un montón de pulsaciones. Siempre que tecleemos un condicional como `if($really) {` y pulsemos retorno de carro, Vim sangrará la siguiente línea con los valores definidos en `shiftwidth` y `expandtab`. Sin embargo, si pulsamos `}` y retorno de carro, Vim pondrá automáticamente las llaves de cierre hacia fuera, al comienzo de la línea. Como este funcionamiento no es el adecuado para algunos tipos de archivos, habría que añadir una orden `autocmd` que identifique primero el tipo de archivo antes de establecer la opción: `autocmd FileType perl :set cindent`.

Hay veces que no nos damos cuenta que un segmento de código se ha de sangrar, hasta que



hemos terminado de ‘picarlo’. En estos casos, colocamos el fragmento entre llaves, como se ve en la Figura 4a, pasamos al modo de orden y pulsamos `>i{`, para sangrar bloque ‘interno’ con el valor definido en `shiftwidth` (Véase la Figura 4b).

La opción `:set smarttab` añade otra característica cuando usamos la opción `expandtab`: Pulsando la tecla de retroceso con el cursor colocado sobre el primer carácter de una línea sangrada, envía la línea de vuelta al margen izquierdo y pulsando la tecla de tabulador vuelve a sangrar la línea, pero sin utilizar realmente los tabuladores.

Otra sugerencia: Para navegar de una llave a su pareja, solamente tenemos que colocar el cursor, en modo de órdenes, en la primera llave y pulsar el símbolo del porcentaje (%). Esto hace más fácil encontrar las llaves perdidas si Perl indica un error de sintaxis.

Si estamos usando un teclado americano y necesitamos teclear con `vi` un carácter no estándar tal como un umlaut (Ä por ejemplo), lo podemos hacer con sólo pulsar `Ctrl-K A`: en el modo de entrada. Para saber más acerca de cómo introducir caracteres no estándar, introducimos la orden `:digraphs` para obtener una lista de lo que está disponible.

Un buen comienzo

La herramienta `tmpl` desde [5] ofrece un buen comienzo cuando queremos zambullirnos dentro de un nuevo guión en Perl: por ejemplo, `$ tmpl -p cooltool` creará un nuevo archivo llamado `cooltool`. Como podemos ver en la Figura 5, el esqueleto del guión contiene unas cuantas líneas de cabecera, unos cuantos módulos típicos, algún código para las opciones de análisis del guión y visualización de `manpage`. `tmpl` lee unos cuantos parámetros configurables, como el nombre del autor, del archivo `.tmpl` en el directorio de inicio del usuario.

El esqueleto `cooltool` ya tiene dos posibilidades: `$ cooltool -v` muestra la versión actual del guión, que se guarda en la variable `$CVSVERSION` y que es actuali-



Figura 2a: El editor de macro en modo de grabación.

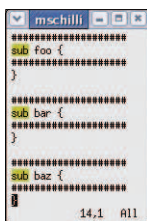


Figura 2b: Después el usuario solamente tiene que reproducir la macro dos veces.

zada automáticamente por CVS. Además, `Pod::Usage` muestra un pequeño archivo de ayuda si se activa la bandera `-h`.

Por supuesto que habrá que rellenar los huecos para que el guión crezca y se convierta en algo práctico, pero el esqueleto será de gran ayuda, ofreciendo una plantilla y un esquema de documentación, que son indispensables en cualquier guión.

Completando el texto

En el modo texto, Vim recuerda automáticamente las palabras que hemos escrito y las completa pulsando `CTRL-n`. Si definimos una variable como `nuestra $GLOBAL_SUPER_VARIABLE`; y la usamos más tarde en el guión, no necesitamos reescribir el nombre; en vez de eso, solamente tenemos que escribir las primeras letras, pulsar `CTRL-n` y dejar que Vim lea nuestros pensamientos.

Si hay más de una posibilidad de expandir las letras que hemos tecleado, pulsamos `CTRL-n` varias veces hasta encontrar la que buscamos, con `CTRL-p` volvemos a la palabra anterior. Esta característica nos permite ahorrar un montón de tiempo y de pulsaciones.

Etiquetas

Los programadores de C estarán familiarizados con el programa `ctags`, que crea un archivo de etiquetas para Vim. Después de que `ctags` ha leído el archivo, el desarrollador solamente necesita colocar el cursor en algún lugar de la llamada

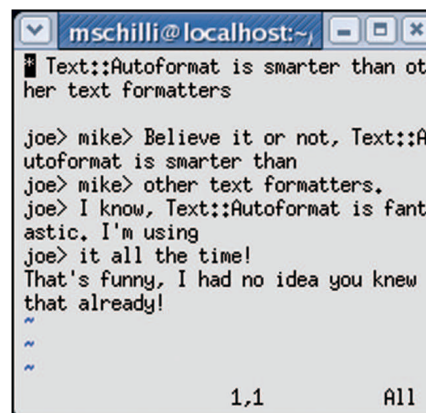


Figura 3a: Un listado con > y parte de un mensaje de email, antes ...

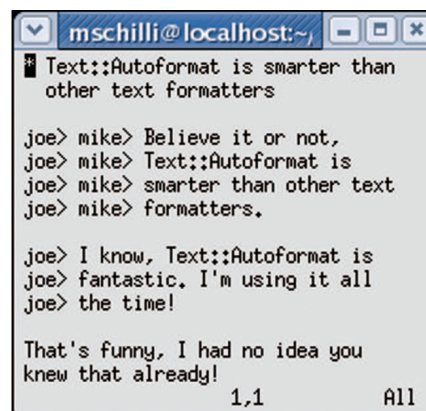


Figura 3b: ... y después de formatearlo con Text::Autoformat.

a la función y pulsar `CTRL-]`, en el modo de órdenes, para decirle a Vim que salte a la definición de función, sin importar en qué archivo está.

Por ejemplo, para decirle a Vim, en Perl, que vaya al archivo fuente para `LWP::UserAgent`, si el cursor está en algún lugar del texto `LWP::UserAgent`, en un programa en Perl, el usuario necesita

Listado 3: .vimrc

```
01 version 6.0
02 :map !L iuse Log::Log4perl
qw(:easy);<RETURN>Log::
Log4perl->easy_init($DEBUG);<RETU
RN><ESC>
03 :map F o<ESC>43i#<ESC>yosub
{<ENTER><ESC>Pk$i
04 map f !Gperl -MText::
Autoformat -e'autoformat{right
=>70}'^V^M
05 set backspace=2
06 set fileencodings=utf-8,latin1
07 set formatoptions=tcql
08 set helplang=en
09 set history=50
10 set hlsearch
11 set ruler
12 set shiftwidth=4
13 :autocmd FileType perl :set
cindent
14 :autocmd FileType perl :set
expandtab
15 set smarttab
16 :nnoemap X :w<Enter>:!perl -c
%<Enter>
17 :set tags=/home/mschilli/.
ptags.txt
18 :set iskeyword+=
```

hacer dos cosas. Antes de nada, Vim debe comprender que las palabras clave de Perl pueden incluir los dos puntos; para hacer esto podemos teclear `:set iskeyword += :`. En segundo lugar, Vim necesita analizar el archivo de etiquetas, el cual proporciona un índice de todos los paquetes instalados, tal como se ve en esta orden

```
:set tags = /home/mschilli/.ptags.txt.
```

Pulsar `CTRL-]`, con el cursor colocado en el nombre del módulo, nos llevará al código fuente del módulo. Alternativamente, podemos facilitar el nombre del módulo como parámetro en el modo de órdenes, como en `:tag LWP::UserAgent`. Si pulsamos `CTRL-T` mientras estamos viendo el archivo del módulo, volveremos a donde estábamos. La magia que hay detrás de esta funcionalidad está almacenada en el archivo de etiquetas `.ptags.txt` que puede crear el código del Listado 2.

A pesar de tener un gestor de ventanas, puede ser que necesitemos ver dos archivos simultáneamente en una sola ventana. Encararemos este problema de otra manera, ya que los usuarios de Vim no son del tipo de gente que utiliza un ratón cuando sus manos están ‘picando’ código.

Si pulsamos `CTRL-W-]`, en vez de `CTRL-]`, con el cursor dentro de una palabra clave, la ventana se dividirá en dos mitades, con el código de nuestro archivo en la mitad inferior y el código del módulo que hemos referenciado en la mitad superior. Pulsando `CTRL-WW` saltamos entre los dos paneles. Tecleando la orden `:quit` en la ventana superior, cerramos esa ventana, quedando solamente el texto

```
mschilli@localhost:~/DEV/articles/vi
1 # $resp = $u->request(GET "http://foo.com");
2 last if $resp->is_error() and
3 $resp->code() ne RC_REQUEST_TIMEOUT;
4 redo if $retires++ < 3;
5
6 1,1 All
```

Figura 4a: Un bloque entre llaves y el cursor al comienzo del bloque.

```
mschilli@localhost:~/DEV/articles/vi
1 # $resp = $u->request(GET "http://foo.com");
2 last if $resp->is_error() and
3 $resp->code() ne RC_REQUEST_TIMEOUT;
4 redo if $retires++ < 3;
5
6 4 lines >ed 1 time 2,5 All
```

Figura 4b: La orden `>{` sangra el bloque interno.

go fuente del módulo, en la superior.

Si no estamos seguros de cómo se escribe el nombre de un método podemos buscarlo, solamente tenemos que introducir una expresión regular. La orden `tselect` buscará cualquier etiqueta que coincida y nos ofrecerá una lista para que la elijamos: `:tselect /^LWP`. Esto permite al usuario seleccionar el número que corresponda con el módulo que necesitamos desde el menú, como se ve en la Figura 7.

Dame etiquetas

¿Cómo creamos un archivo `~/.ptags.txt`? Para hacer esto, necesitamos leer los módulos que utiliza nuestra instalación local de Perl a intervalos regulares. El guión mostrado en el Listado 2 husmea en todas nuestras rutas `@INC`, escribe en un array `@` los directorios para ayudarle a recordar dónde ha estado para evitar volver a mirar en los que ya ha visitado.

Otra opción para crear `ppi-tags` sería utilizar el programa `ctags`, que, en sus versiones más recientes, maneja el código Perl realmente bien. Pero llamando `ctags -R -f`

que estamos editando en la ventana principal. Alternativamente, podemos utilizar la orden `:only` en la ventana inferior, para cerrar la ventana superior.

Se puede ver una sesión de Vim en la Figura 6, con un guión de prueba que utiliza el módulo

`LWP::UserAgent` en la ventana inferior, y el método `new()` del código

```
mschilli@mybox:~
1 #!/usr/bin/perl
2 # =====
3 # cooltool
4 # 2005, Mike Schilli <cpa@perlmeister.com>
5 # =====
6 use warnings;
7 use Getopt::Std;
8 use Pod::Usage;
9
10 use vars qw($CVSVERSION);
11 $CVSVERSION = 'Revision: 1.3 $';
12
13 getopts("hu", \%opt);
14 pod2usage() if $opt{h};
15
16 if ($opt{v}) {
17     my ($version) = $CVSVERSION =~ /(\d\S+)/;
18     die "$0 $version\n";
19 }
20
21 __END__
22
23 =head1 NAME
24
25 cooltool - blah blah blah
26
27 =head1 SYNOPSIS
28
29 cooltool -xyz
30
31 =head1 OPTIONS
32
33 =over 8
34
35 =item B<-x>
36
37 Prints this manual page in text format.
38
39 =back
40
41 =head1 DESCRIPTION
42
43 cooltool blah blah blah.
44
45 =head1 EXAMPLES
46
47 $ cooltool -x foo bar
48
49 =head1 LEGALESE
50
51 Copyright 2005 by Mike Schilli.
52 All rights reserved. This program is free
53 software, you can redistribute it and/or
54 modify it under the same terms as Perl itself.
55
56 =head1 AUTHOR
57
58 2005, Mike Schilli <cpa@perlmeister.com>
59
60 1,1 All
```

Figura 5: Vim en el modo de ventana dividida: Un guión de prueba en la parte inferior y el código del módulo `LWP::UserAgent`, con los guiones utilizados, en la superior.

`~/.ptags.txt/usr/lib/perl5` no encontrará las etiquetas que requieren un análisis más elaborado del código Perl más allá del modelo de simple coincidencia.

Para analizar fuentes Perl, lo que de verdad necesitamos es Perl, pues Perl es extremadamente difícil de analizar. Sin embargo, Adán Kennedy intentó lo imposible y escribió recientemente un programa “bastante bueno” de análisis para Perl; de hecho, el programa de análisis es increíblemente bueno. El módulo `PPI` de CPAN incluye `PPI::Document`; su método `load()` lee un módulo Perl, lo reparte en fichas y las almacena como nodos en una estructura en árbol.

El guión `ppitags` utiliza `File::Find` para analizar los directorios en el array global `@INC` de Perl. Para cada entrada que encuentra, `File::Find` salta a la función `file_wanted`. Si la entrada es un directorio en vez de un archivo, la línea 34 actualiza el hush `%dirs`, para averiguar si la trayectoria se ha recorrido ya. Si es así la línea 33 fija la variable `$File::Find::prune` a 1 para indicarle a `File::Find` que puede saltarse el resto del directorio y todo lo que haya bajo él. La línea 37 ignora cualquier cosa que no sea un módulo Perl terminado en `pm`.

La línea 40 analiza el módulo Perl actual. Cualquier error que ocurra en este punto será manejado por la línea 43 (en el momento de escribir esto, PPI no es per-

fecto) emite una advertencia y descarta el módulo que no podrá manejar.

Después de analizar un módulo, la línea 51 llama al método `find()` para el objeto `PPI::Document`, avanzando a través de las fichas en las fuentes de Perl y llama a la función `document_wanted` definida en la línea 56 para cada ficha encontrada.

La función comprueba si la ficha es del tipo `PPI::Statement::Package` o un objeto



del tipo `PPI::Statement::Sub`, esto es, un paquete o una subdefinición en el código de Perl.

Una definición `package` significa una línea como `package LWP::UserAgent;`, que a su vez significa cuatro fichas en el mundo de PPI: `package`, espacio, el nombre del módulo y el punto y coma de cierre. Solamente el nombre del módulo es de interés para `ppitags`, que es el tercer hijo en el nodo, que pasó `$_[1]` a `document_wanted()`. El método `child()` comienza con el índice `ihijo` a 0, extrae la cadena `"LWP::UserAgent":` `$_[1]` -> `child(2)`.

La línea 69 encuentra definiciones del tipo `sub func {` y extrae los nombres de la función y los métodos para permitir al mecanismo de etiquetado identificar las estructuras tales como `LWP::Debug::trace` y salta a la localización donde se define realmente la función en el módulo `LWP::Debug`.

Cuando se analiza una definición del paquete, `ppitags` almacena el nombre del paquete como el paquete actual, que se utiliza entonces como un prefijo para todas las funciones analizadas. Aunque esto podría fallar con definiciones de paquetes en bloques, no hay ninguna diferencia en el 99,9% de los casos.

La orden `push` en la línea 80 empuja una nueva secuencia al final del array `@found`, que se compone de la etiqueta requerida (Paquete- o el nombre completo de la función cualificada), del nombre absoluto de la fuente, del nombre del archivo y de una expresión regular, que localiza la definición del paquete o de la función dentro del archivo de las fuentes. Para hacer esto, la función define en la línea 91 `FF`, `regex_from_node`, componiendo una expresión regular compuesta de todos los caracteres del comienzo de la línea que coincida con la ficha requerida. En el caso de

subprogramas, `$node->content()` devuelve tanto la cabecera como el cuerpo de la función. Esta es la razón por la cual la línea 98 quita todas las líneas, aparte de la primera, y las líneas de la 100 a la 109 retroceden de ficha en ficha hasta que se alcanza el principio de la línea. Al final del bucle `while`, `$regex` contendrá la línea fuente desde el comienzo de la línea hasta la ficha. La línea 114 utiliza estos datos para generar una expresión regular del tipo de `/^.../` con un ancla para el carácter de comienzo de línea. La operación buscar y substituir en la línea 112 asegura que no entre en conflicto ningún carácter no estándar en el código Perl con los meta-caracteres de la expresión regular escapándolos con barras inversas.

El guión `ppitags` crea un archivo `~/ptags.txt` que contiene una lista en el formato de entradas a tres columnas: `Paquete/Subrutina [tab] NombreFichero [tab] ExpresiónRegular`, que Vim analizará con `:set tags=` tal como se ha descrito anteriormente, permitiendo así que salte elegantemente de las palabras claves al correspondiente código fuente.

Es conveniente ejecutar `ppitags` una vez al día como tarea del `cron` para mantener al día `~/ptags.txt`. Si se prefiere se puede extender el guión para permitir a Vim identificar nuestras variables `our` totalmente cualificadas (del tipo `$Text::Wrap::columns`, por ejemplo) y saltar a sus definiciones dentro del código fuente del módulo.

Almacenamiento permanente

Cuando lanzamos Vim lee el archivo de configuración `.vimrc` de nuestro directo-

rio de inicio y en realidad esto permite que los usuarios ejecutemos una serie de órdenes antes de que Vim se ponga a trabajar. Después de retocar los ajustes predeterminados interactivamente, probablemente queramos guardar los cambios. En vez de volver a teclear las órdenes para añadirlas a nuestro archivo `.vimrc`, solamente tenemos que teclear `:mkvimrc` para indicarle a Vim que guarde la configuración actual en `~/vimrc`.

Hay un ejemplo de configuración en [1] que contiene todas las configuraciones que hemos repasado en el artículo de este mes. Recuerden, ¡ahorren tiempo 'picando' para dedicarlo a pensar!

RECURSOS

- [1] Listados de este artículo: <http://www.linux-magazine.es/Magazine/Downloads/10>
- [2] Página principal del proyecto Vim: <http://www.vim.org/>
- [3] Steve Oualline, "vi Improved - Vim", New Riders, 2001
- [4] Archivo de guiones de Mike: <http://perlmeister.com/scripts/>
- [5] Guión `tmpl`: <http://perlmeister.com/scripts/tmpl>

EL AUTOR Michael Schilli trabaja como desarrollador de software en Yahoo!, Sunnyvale, California. Es el autor de "Perl Power" de la editorial Addison-Wesley y se le puede contactar en mshilli@perlmeister.com. Su página está en <http://perlmeister.com/>.



```
mschilli@localhost:~/DEV/articles/
sub new
{
    my($class, %opt) = @_;
    LWP::Debug::trace(0);
    my $agent = delete $opt{agent};
    $agent = $class->_agent unless defined $agent;
    my $from = delete $opt{from};
    perl/5.8.5/LWP/UserAgent.pm [R0] 41,1
}
use LWP::UserAgent;
my $ua = LWP::UserAgent->new();
!atext.pl 9:10 712
```

Figura 6: Una plantilla para un nuevo guión llamado `cooltool`, creada por el guión `tmpl`.

```
mschilli@mybox:/mnt/big2/mschilli
1 #! perl kind tag file
2 F LWP /home/mschilli/PER
L/1/lib/perl5/site_perl/5.8.3/LWP.pm
package LWP;
2 F LWP::Authen::Basic /home/mschilli/PER
L/1/lib/perl5/site_perl/5.8.3/LWP/Authen/B
asic.pm
package LWP::Authen::Basic;
3 F LWP::Authen::Basic::authenticate /h
ome/mschilli/PER/L/1/lib/perl5/site_perl/5.8.3/LWP/Au
then/B
asic.pm
sub authenticate
4 F LWP::Authen::Digest /home/mschilli/P
ER/L/1/lib/perl5/site_perl/5.8.3/LWP/Authen/Digest.p
m
package LWP::Authen::Digest;
5 F LWP::Authen::Digest::authenticate /h
ome/mschilli/PER/L/1/lib/perl5/site_perl/5.8.3/LWP/Au
then/D
igest.pm
sub authenticate
6 F LWP::Authen::Ntlm /home/mschilli/PER
L/1/lib/perl5/site_perl/5.8.3/LWP/Authen/Ntlm.p
m
package LWP::Authen::Ntlm;
7 F LWP::Authen::Ntlm::authenticate /h
ome/mschilli/PER/L/1/lib/perl5/site_perl/5.8.3/LWP/Au
then/N
tlm.pm
sub authenticate
8 F LWP::Authen::MIME /home/mschilli/PER
L/1/lib/perl5/site_perl/5.8.5/LWP/Authen/MIME.p
m
package LWP::Authen::MIME;
9 F LWP::Authen::MIME::authenticate /h
ome/
!More
```

Figura 7: Utilizando expresiones regulares para buscar etiquetas; si el usuario teclea `/^LWP` obtendrá un menú numerado por entradas.