Mordiditas de aquí y de allá

Bits and Pieces - Typeglobs

Generalmente las cosas y lo afectos, se valoran cuando se conocen, y esta valoración se acrecienta aún más cuando los perdemos. Los typeglobs no son la excepción.

Yendo a lo nuestro, los typeglobs son la forma de acceder a las tablas de símbolos (symbol tables) de perl. Pero ... qué es esto ??

Cada vez que perl guarda un scalar, array, hash, sub, file handle o nombre de package (módulo) lo hace con la estructura que vimos en el número anterior (CaFe Perl 0.7b) pero la forma de direccionarlo/accederlo es a través de un hash, relacionado con el namespace y nombre del dato. Esta estructura interna es accesible desde un programa de usuario, a través de la siguiente estructura :

```
*package::variable{SCALAR}    ## \$package::variable
*package::array{ARRAY}        ## \@package::array
*package::hash{HASH}          ## \%package::hash
*package::sub{CODE}           ## \&package::sub
*package::glob{GLOB}          ## \*package::glob
*package::filehandle{IO}      ##   file   handle   interno,   no   hay
equivalente
*package::name{NAME}          ## nombre de un símbolo
*package::package{PACKAGE}    ## nombre del package
```

Noten que esta variable se accede con el prefijo (sigil) asterisco (*).

De esta forma si tenemos el siguiente package :

```
package myTest;

$DEBUG = 1;

sub DEBUG() {
      return "Esto es una rutina de debugging";
};

1;
```

el acceso a cada una de las estructuras es como sigue :

```
$myTest::DEBUG;               ## scalar DEBUG, forma tradicional.
${ *myTest::DEBUG{SCALAR} };  ## scalar DEBUG
${ *myTest::DEBUG };          ## scalar DEBUG
&{ *myTest::DEBUG{CODE} };    ## sub DEBUG
&{ *myTest::DEBUG };          ## sub DEBUG
*myTest::DEBUG->();           ## sub DEBUG
```

Hay algo más que interesante, y característico de perl, que es el manejo del contexto. Vemos como un scalar puede ser accedido bien a través de la key SCALAR o bien anteponiendo el sigil $, indicando que se desea la key SCALAR del hash (lo mismo ocurre con CODE y el sigil & o cualquiera de las otras estructuras). Como generalización el programa principal tiene

como nombre de módulo main y los typeglobs correspondientes se acceden como *:: o *main::

Hasta acá no se trata nada mas ni nada menos que lo que se suele llamar como “sintactic sugar” (“azúcar sintáctica” o “quiero que el código se vea lindo como a mi me gusta”), ya que no es más que otra forma de acceder los mismos datos que venimos accediendo desde nuestro primer programa Perl ... o al menos lo usamos así hasta ahora. Analicemos un poco más en detalle.

El que tengamos acceso a la estructura de cómo Perl busca los datos en memoria nos puede ayudar a manipularlo en ciertas formas y ocasiones. Por ejemplo el typeglob *main::var1 contiene referencias a todas las estructuras ubicadas en el programa principal cuyo nombre comienza con var1 ($var1, @var1, etc.) y algo similar ocurre con el typeglob *main::var2, que a su vez ambos son referencias a un hash que contiene las estructuras de datos correspondientes, con lo cual si una de ellas le asignamos la otra lo que estamos haciendo es, por ejemplo, copiar la referencia de var2 a var1 y ambas van a representar el mismo hash ... lo que equivale a generar dos puntos de entrada distintos a las mismas variable pero con distinto nombre !!!.

Tomemos el siguiente código :

```
local *var1;
$var1 = "Esto es un scalar";
$var1{'aaa'} = 'bbbb';
$var1{'ccc'} = 'dddd';
push @var1, 'eeeee';
push @var1, 'fffff';
push @var1, 'ggggg';

print "-" x 40 . "\n";
my $refScalar = *var1{SCALAR};
print "Scalar => ". $$refScalar . "\n";

my $refArray = *var1{ARRAY};
foreach ( @$refArray ) {
    print "Array  => $_\n";
};

my $refHash = *var1{HASH};
foreach ( keys %$refHash ) {
    print "Hash   => $_ : ". $refHash->{$_} ."\n";
};

print "-" x 40 . "\n";
local $var2 = "Nada 2";
*var1 = \$var2;
my $refScalar = *var1{SCALAR};
print "Scalar => ". $$refScalar . "\n";

$refArray = *var1{ARRAY};
foreach ( @$refArray ) {   ## Sigue existiendo el ARRAY
    print "Array  => $_\n";
};


print "-" x 40 . "\n";
```

```
local $var3 = "Nada 3";
*var1 = *var3;
my $refScalar = *var1{SCALAR};
print "Scalar => ". $$refScalar . "\n";

$refArray = *var1{ARRAY};
foreach ( @$refArray ) {   ## NO HAY ARRAY
    print "Array  => $_\n";
};
```

la salida que obtenemos al ejecutarlo es :

```
----------------------------------------
Scalar => Esto es un scalar
Array  => eeeee
Array  => fffff
Array  => ggggg
Hash   => aaa : bbbb
Hash   => ccc : dddd
----------------------------------------
Scalar => Nada 2
Array  => eeeee
Array  => fffff
Array  => ggggg
----------------------------------------
Scalar => Nada 3
```

En el primer bloque simplemente accedimos las variables con el nombre var1 a través de su correspondiente typeglob *var1, que es lo que veníamos viendo hasta ahora, pero en el segundo bloque hicimos algo un poco más interesante : asignar a la referencia de $var2 a $var1 (*var1 = \$var2), de tal forma que todo el namespace main::var1 siga igual excepto $var1 que ahora apunta al contenido de $var2 (y se sigue accediendo como $var1 !!!). En el tercer bloque se hizo lo mismo peor a un nivel global, reemplazando todo el namespace main::var1 por main::var3 (*var1 = *var3).

Copias, copias y más copias sabemos que es la base de la economía de escalas : producir lo mismo infinitamente y reducir los costos fijos a una suma infinitesimal, pero ... y dónde queda el arte, la inventiva, el ocio creativo, eh ??

Hay algo muy particular en esto, y es que al poder manipular los typeglobs podemos hacer lo propio con los namespaces de un package (no nos olvidemos que al programa principal lleva main como nombre de package), y en lugar de copiar una estructura podemos inyectarle un nuevo valor dentro de la misma. Tratándose de un scalar o un hash no es más que una asignación hecha de otra forma, pero tratándose de código ... mmmm ... podemos generar nuevas sub dentro del namespace del package que necesitemos. Ahí va un ejemplo :

```
use strict;

sub installsub($$) {
    my $glob = shift; ## nombre de la sub a instalar
    my $code = shift; ## código de la sub

    no strict 'refs';
    *{ $glob } = $code;
```

```
}


eval( 'test();');
print "Error del eval() : \n\t$@\n";
installsub( 'test', sub { print "TEST!\n" } );
eval( 'test();');
print "Error del eval() : \n\t$@\n";
```

que al ejecutarlo nos muestra la siguiente salida :

```
Error del eval() :
        Undefined subroutine &main::test called at (eval 1) line 1.

TEST!
Error del eval() :
```

La magia está toda contenida dentro de las líneas minimalistas de installsub(), más precisamente en su última línea donde se asigna al código de la sub a un nombre dentro del package main usando typeglobs. Como se puede ver en la ejecución antes de llamar a installsub() la función eval() hace un trap del error de no definición de la sub test(), error que desaparece luego de ejecutar installsub(). Una nota de color, y algo para tener cuidado, es la sentencia previa a la asignación del typeglob, y es que hay que quitar le uso estricto en cuento a referencias ya que en realidad estamos usando una referencia a una estructura a través de su nombre, cosa que no está permitida si estamos haciendo uso estricto (use strict).

Señoras y señores, si esto no es magia y creatividad entonces ni David Copperfield existiría (tampoco Larry Wall).

Si avanzamos un paso más, podemos ver que desde cualquier lugar (main o package) podemos manipular los typeglobs/namespace de cualquier otro lugar, llevándonos un paso más lejos en nuestra intrépida búsqueda, con lo cual podríamos insertar subs de un package en el package main y así poderlas usar sin poner el calificador del package antes del nombre de cada sub. Otro ejemplo más para divertirnos :

```
use strict;

BEGIN {
    package myTest;

    sub hello() {
        print "Hello !!\n";
    };

    my @definedSub = ('hello', 'new' );

    sub importSubs() {
        foreach ( @definedSub ) {
            print "    --> importing sub '$_'\n";
            *{ "main::$_" } = *{ "myTest::$_" };
        };
    };
}
```

```
eval( 'hello();' );
print "Error del eval() --> : \n\t$@\n";
myTest::importSubs();
eval( 'hello();' );
print "Error del eval() --> : \n\t$@\n";
```

Obtenemos como salida :

```
Error del eval() --> :
        Undefined subroutine &main::hello called at (eval 1) line 1.

    --> importing sub 'hello'
    --> importing sub 'new'
Hello !!(plain)
Error del eval() --> :
```

Nuevamente la magia está con nosotros (“The magic is back”), y de una forma muy particular ya que exactamente este mismo proceso es el que usa para importar las subs al namespace main cada vez que hacemos un use() en nuestros scripts. Exactamente lo que hace Exporter.pm en su sub import(). Si la analizamos, salvo algunas consideraciones extras como manejo de cache , las lineas que hacen el import en el namespace que es el invocante son las mismas que hacen nuestra magia en installsub() !!!

Y hablando de manejar namespaces, por qué no manejar nuestro propio namespace ?? Por ejemplo supongamos que tenemos un package que tiene una cierta cantidad de subs, y ahora generamos una nueva versión pero que, por problemas de diseño, no va a mantener la misma interfaz que la anterior : hay nuevas subs, hay subs que desaparecen y las subs que quedan han cambiado la cantidad, el tipo o el orden de sus parámetros. Este es un típico problema de compatibilidad que podemos resolver usando typeglobs, adjuntando el package anterior (con un nombre distinto) y modificando el namespace actual para que aparezcan las subs actuales sean reemplazadas por las anteriores.

En este ejemplo hay tres packages con las siguientes características :

- myTest_v1 : package original que sólo posee la sub hello()
- myTest_v2 : package de versión posterior que agrega la sub bye()
- myTest : package final que sólo posee hello() y new() que permite habilitar alguna de las versiones de package anteriores

```
package myTest_v1;
our @definedSub = ('hello' );
sub hello() {
    print "Hello (v1)\n";
};
```

```
package myTest_v2;

our @definedSub = ('hello', 'bye' );
sub hello() {
    print "Hello (v2)\n";
};

sub bye() {
```

```
    print "Bye (v2)\n";
};


package myTest;

sub hello() {
    print "Hello !!(plain)\n";
};

sub new($) {
    my $version = shift;
    print "About to install package myTest_v$version\n";
    no strict 'refs';

    foreach ( @{ "myTest_v".$version."::definedSub" } ) {
        print "    --> Assigning sub '$_'\n";
        *{ "myTest::$_" } = *{ "myTest_v".$version."::$_" };
    };
};
```

y el programa principal que lo convoca :

```
use strict;
require myTest;

myTest::hello();
myTest::new(1);
myTest::hello();

myTest::new(2);
myTest::hello();
myTest::bye();
```

teniendo como salida de la ejecución :

```
Hello !!(plain)
About to install package myTest_v1
    --> Assigning sub 'hello'
Hello (v1)
About to install package myTest_v2
    --> Assigning sub 'hello'
    --> Assigning sub 'bye'
Hello (v2)
Bye (v2)
```

Como vemos se puede hacer un manejo de muy bajo nivel con los typeglobs que puede contribuir, o no, a la estabilidad y claridad de nuestro código.

Conclusiones

Los typeglobs nos permiten manipular las estructuras de datos dentro de un namespace, pero :

- no hay que abusar, es una característica muy potente y que puede dificultar la lectura del código
- hay que utilizarla en puntos del código específico para manejos internos (un buen ejemplo es Exporter.pm) y con una clara documentación que incluya uso y efectos
- no usarla sólo para “renombrar” ciertas estructuras con un nombre más corto o conveniente (además de ser un uso pobre puede llevar a confusión), es preferible cambiar el nombre original y ajustar el código donde sea necesario

Debido a estas características y al cuidado necesario en su uso los typeglobs no van a estar presentes en Perl 6, aunque seguramente habrá una forma más elegante y cómoda de hacer este tipo de tareas.

Infografía

- Programming Perl, 3rd. Edition
- Learnign Perl, 4th. edition
- Advanced Perl Programming
- Chromatic journal (http://use.perl.org/~chromatic/journal/11712)
- Class::BuildMethods (http://use.perl.org/~Ovid/journal/27712)
- Futility (http://use.perl.org/~Whammo/journal/3066)
- Exporter.pm (http://search.cpan.org/~nwclark/perl-5.8.7/lib/Exporter.pm)
- CPAN (http://www.cpan.org)
- Apocalipsis 2 : Bits and Pieces (http://dev.perl.org/perl6/doc/design/apo/A02.html)
- Sinopsis 2 : Bits and Pieces (http://dev.perl.org/perl6/doc/design/syn/S02.html)
- Exégesis 2 : Bits and Pieces (http://dev.perl.org/perl6/doc/design/exe/E02.html)