Introduction to mod_perl (part IV): Perl Basics

By Stas Bekman

If you like this article, please make a donation to the Perl development grant fund.

Table of Contents

Introduction

Before I delve into the details of mod_perl programming in the future articles it's probably a very good idea to review some very important Perl basics. You will discover these invaluable when you start coding for mod_perl. I will start with pure Perl notes and gradually move to explaining the peculiarities of coding for mod_perl, presenting the traps one might fall into and explaining things obvious for some of us but may be not for the others.

Using Global Variables and Sharing Them Between Modules/Packages

It helps when you code your application in a structured way, using the perl packages, but as you probably know once you start using packages it's much harder to share the variables between the various packagings. A configuration package comes to mind as a good example of the package that will want its variables to be accessible from the other modules.

Of course using the Object Oriented (OO) programming is the best way to provide an access to variables through the access methods. But if you are not yet ready for OO techniques you can still benefit from using the techniques I'm going to talk about.

Making Variables Global

When you first wrote `$x` in your code you created a (package) global variable. It is visible everywhere in your program, although if used in a package other than the package in which it was declared (`main::` by default), it must be referred to with its fully qualified name, unless you have imported this variable with import(). This will work only if you do not use `strict` pragma; but it's very important to use this pragma if you want to run your scripts under mod_perl.

Making Variables Global With strict Pragma On

First you use :

```
  use strict;
```

Then you use:

```
 use vars qw($scalar %hash @array);
```

This declares the named variables as package globals in the current package. They may be referred to within the same file and package with their unqualified names; and in different files/packages with their fully qualified names.

With perl5.6 you can use the `our` operator instead:

```
  our qw($scalar %hash @array);
```

If you want to share package global variables between packages, here is what you can do.

Using Exporter.pm to Share Global Variables

Assume that you want to share the `CGI.pm` object (I will use `$q`) between your modules. For example, you create it in `script.pl`, but you want it to be visible in `My::HTML`. First, you make `$q` global.

```
script.pl:
----------------
use vars qw($q);
use CGI;
use lib qw(.);
use My::HTML qw($q); # My/HTML.pm is in the same dir as script.pl
$q = CGI->new;


My::HTML::printmyheader();
```

Note that I have imported `$q` from `My::HTML`. And `My::HTML` does the export of `$q`:

```
My/HTML.pm
----------------
package My::HTML;
use strict;


BEGIN {
  use Exporter ();


  @My::HTML::ISA         = qw(Exporter);
  @My::HTML::EXPORT      = qw();
  @My::HTML::EXPORT_OK   = qw($q);


}


use vars qw($q);


sub printmyheader{
  # Whatever you want to do with $q... e.g.
  print $q->header();
}
1;
```

So the `$q` is shared between the `My::HTML` package and `script.pl`. It will work vice versa as well, if you create the object in `My::HTML` but use it in `script.pl`. You have true sharing, since if you change `$q` in `script.pl`, it will be changed in `My::HTML` as well.

What if you need to share `$q` between more than two packages? For example you want My::Doc to share `$q` as well.

You leave `My::HTML` untouched, and modify *script.pl* to include:

```
use My::Doc qw($q);
```

Then you add the same `Exporter` code that I used in `My::HTML`, into `My::Doc`, so that it also exports `$q`.

One possible pitfall is when you want to use `My::Doc` in both `My::HTML` and *script.pl*. Only if you add

```
use My::Doc qw($q);
```

into `My::HTML` will `$q` be shared. Otherwise `My::Doc` will not share `$q` any more. To make things clear here is the code:

```
script.pl:
----------------
use vars qw($q);
use CGI;
use lib qw(.);
use My::HTML qw($q); # My/HTML.pm is in the same dir as script.pl
use My::Doc  qw($q); # Ditto
$q = new CGI;


My::HTML::printmyheader();

My/HTML.pm
----------------
package My::HTML;
use strict;


BEGIN {
  use Exporter ();


  @My::HTML::ISA         = qw(Exporter);
  @My::HTML::EXPORT      = qw();
  @My::HTML::EXPORT_OK   = qw($q);


}


use vars     qw($q);
use My::Doc  qw($q);


sub printmyheader{
  # Whatever you want to do with $q... e.g.
  print $q->header();


  My::Doc::printtitle('Guide');
}
1;

My/Doc.pm
----------------
```

```
package My::Doc;
use strict;


BEGIN {
  use Exporter ();


  @My::Doc::ISA         = qw(Exporter);
  @My::Doc::EXPORT      = qw();
  @My::Doc::EXPORT_OK   = qw($q);


}


use vars qw($q);


sub printtitle{
  my $title = shift || 'None';


  print $q->h1($title);
}
1;
```

Using the Perl Aliasing Feature to Share Global Variables

As the title says you can import a variable into a script or module without using `Exporter.pm`. I have found it useful to keep all the configuration variables in one module `My::Config`. But then I have to export all the variables in order to use them in other modules, which is bad for two reasons: polluting other packages' name spaces with extra tags which increases the memory requirements; and adding the overhead of keeping track of what variables should be exported from the configuration module and what imported, for some particular package. I solve this problem by keeping all the variables in one hash `%c` and exporting that. Here is an example of `My::Config`:

```
package My::Config;
use strict;
use vars qw(%c);
%c = (
  # All the configs go here
  scalar_var => 5,


  array_var  => [qw(foo bar)],


  hash_var   => {
                 foo => 'Foo',
                 bar => 'BARRR',
                },
);
1;
```

Now in packages that want to use the configuration variables I have either to use the fully qualified names like `$My::Config::test`, which I dislike or import them as described in the previous section. But hey, since I have only one variable to handle, I can make things even simpler and save the loading of the `Exporter.pm` package. I will use the Perl aliasing feature for exporting and saving the keystrokes:

```
package My::HTML;
use strict;
use lib qw(.);
  # Global Configuration now aliased to global %c
use My::Config (); # My/Config.pm in the same dir as script.pl
use vars qw(%c);
*c = \%My::Config::c;


  # Now you can access the variables from the My::Config
print $c{scalar_var};
print $c{array_var}[0];
print $c{hash_var}{foo};
```

Of course $c is global everywhere you use it as described above, and if you change it somewhere it will affect any other packages you have aliased `$My::Config::c` to.

Note that aliases work either with global or `local()` vars − you cannot write:

```
my *c = \%My::Config::c; # ERROR!
```

Which is an error. But you can write:

```
local *c = \%My::Config::c;
```

For more information about aliasing, refer to the Camel book, second edition, pages 51−52.

Using Non−Hardcoded Configuration Module Names

You have just seen how to use a configuration module for configuration centralization and an easy access to the information stored in this module. However, there is somewhat of a chicken−and−egg problem−−how to let your other modules know the name of this file? Hardcoding the name is brittle−−if you have only a single project it should be fine, but if you have more projects which use different configurations and you will want to reuse their code you will have to find all instances of the hardcoded name and replace it.

Another solution could be to have the same name for a configuration module, like `My::Config` but putting a different copy of it into different locations. But this won't work under mod_perl because of the namespace collision. You cannot load different modules which uses the same name, only the first one will be loaded.

Luckily, there is another solution which allows us to stay flexible. `PerlSetVar` comes to rescue. Just like with environment variables, you can set server's global Perl variables which can be retrieved from any module and script. Those statements are placed into the *httpd.conf* file. For example

```
PerlSetVar FooBaseDir       /home/httpd/foo
PerlSetVar FooConfigModule  Foo::Config
```

Now I `require()` the file where the above configuration will be used.

```
PerlRequire /home/httpd/perl/startup.pl
```

In the *startup.pl* I might have the following code:

```
  # retrieve the configuration module path
use Apache:
my $s             = Apache->server;
my $base_dir      = $s->dir_config('FooBaseDir')      || '';
my $config_module = $s->dir_config('FooConfigModule') || '';
die "FooBaseDir and FooConfigModule aren't set in httpd.conf"
  unless $base_dir and $config_module;


  # build the real path to the config module
my $path = "$base_dir/$config_module";
$path =~ s|::|/|;
$path .= ".pm";
  # I have something like "/home/httpd/foo/Foo/Config.pm"


  # now I can pull in the configuration module
require $path;
```

Now I know the module name and it's loaded, so for example if I need to use some variables stored in this module to open a database connection, I will do:

```
Apache::DBI->connect_on_init
("DBI:mysql:${$config_module.'::DB_NAME'}::${$config_module.'::SERVER'}",
 ${$config_module.'::USER'},
 ${$config_module.'::USER_PASSWD'},
```

```
 {
  PrintError => 1, # warn() on errors
  RaiseError => 0, # don't die on error
  AutoCommit => 1, # commit executes immediately
 }
);
```

Where variable like:

```
${$config_module.'::USER'}
```

In my example are really:

```
$Foo::Config::USER
```

If you want to access these variable from within your code at the run time, instead accessing to the server object `$c`, use the request object `$r`:

```
my $r = shift;
my $base_dir      = $r->dir_config('FooBaseDir')      || '';
my $config_module = $r->dir_config('FooConfigModule') || '';
```

The Scope of the Special Perl Variables

Now let's talk about Special Perl Variables.

Special Perl variables like `$|` (buffering), `$^T` (script's start time), `$^W` (warnings mode), `$/` (input record separator), `$\` (output record separator) and many more are all true global variables; they do not belong to any particular package (not even `main::`) and are universally available. This means that if you change them, you change them anywhere across the entire program; furthermore you cannot scope them with my(). However you can local()ise them which means that any changes you apply will only last until the end of the enclosing scope. In the mod_perl situation where the child server doesn't usually exit, if in one of your scripts you modify a global variable it will be changed for the rest of the process' life and will affect all the scripts executed by the same process. Therefore localising these variables is highly recommended, I'd say mandatory.

I will demonstrate the case on the input record separator variable. If you undefine this variable, the diamond operator (readline) will suck in the whole file at once if you have enough memory. Remembering this you should never write code like the example below.

```
$/ = undef; # BAD!
open IN, "file" ....
  # slurp it all into a variable
$all_the_file = <IN>;
```

The proper way is to have a `local()` keyword before the special variable is changed, like this:

```
local $/ = undef;
open IN, "file" ....
  # slurp it all inside a variable
$all_the_file = <IN>;
```

But there is a catch. `local()` will propagate the changed value to the code below it. The modified value will be in effect until the script terminates, unless it is changed again somewhere else in the script.

A cleaner approach is to enclose the whole of the code that is affected by the modified variable in a block, like this:

```
{
  local $/ = undef;
  open IN, "file" ....
    # slurp it all inside a variable
  $all_the_file = <IN>;
}
```

That way when Perl leaves the block it restores the original value of the `$/` variable, and you don't need to worry elsewhere in your program about its value being changed here.

Note that if you call a subroutine after you've set a global variable but within the enclosing block, the global variable will be visible with its new value inside the subroutine.

Compiled Regular Expressions

And finally I want to cover the pitfall many people has fallen into. Let's talk about regular expressions use under mod_perl.

When using a regular expression that contains an interpolated Perl variable, if it is known that the variable (or variables) will not change during the execution of the program, a standard optimization technique is to add the `/o` modifier to the regex pattern. This directs the compiler to build the internal table once, for the entire lifetime of the script, rather than every time the pattern is executed. Consider:

```
my $pat = '^foo$'; # likely to be input from an HTML form field
foreach( @list ) {
  print if /$pat/o;
}
```

This is usually a big win in loops over lists, or when using the `grep()` or `map()` operators.

In long−lived mod_perl scripts, however, the variable may change with each invocation and this can pose a problem. The first invocation of a fresh httpd child will compile the regex and perform the search correctly. However, all subsequent uses by that child will continue to match the original pattern, regardless of the current contents of the Perl variables the pattern is supposed to depend on. Your script will appear to be broken.

There are two solutions to this problem:

The first is to use `eval q//`, to force the code to be evaluated each time. Just make sure that the eval block covers the entire loop of processing, and not just the pattern match itself.

The above code fragment would be rewritten as:

```
my $pat = '^foo$';
eval q{
  foreach( @list ) {
    print if /$pat/o;
  }
}
```

Just saying:

```
foreach( @list ) {
  eval q{ print if /$pat/o; };
}
```

means that I recompile the regex for every element in the list even though the regex doesn't change.

You can use this approach if you require more than one pattern match operator in a given section of code. If the section contains only one operator (be it an `m//` or `s///`), you can rely on the property of the null pattern, that reuses the last pattern seen. This leads to the second solution, which also eliminates the use of eval.

The above code fragment becomes:

```
my $pat = '^foo$';
"something" =~ /$pat/; # dummy match (MUST NOT FAIL!)
foreach( @list ) {
  print if //;
}
```

The only gotcha is that the dummy match that boots the regular expression engine must absolutely, positively succeed, otherwise the pattern will not be cached, and the `//` will match everything. If you can't count on fixed text to ensure the match succeeds, you have two possibilities.

If you can guarantee that the pattern variable contains no meta−characters (things like *, +, ^, $...), you can use the dummy match:

```
$pat =~ /\Q$pat\E/; # guaranteed if no meta−characters present
```

If there is a possibility that the pattern can contain meta−characters, you should search for the pattern or the non−searchable \377 character as follows:

```
"\377" =~ /$pat|^\377$/; # guaranteed if meta−characters present
```

Another approach:

It depends on the complexity of the regex to which you apply this technique. One common usage where a compiled regex is usually more efficient is to ``*match any one of a group of patterns*" over and over again.

Maybe with a helper routine, it's easier to remember. Here is one slightly modified from Jeffery Friedl's example in his book ``*Mastering Regular Expressions*".

```
########################################################
# Build_MatchMany_Function
# -- Input:  list of patterns
# -- Output: A code ref which matches its $_[0]
#            against ANY of the patterns given in the
#            "Input", efficiently.
#
sub Build_MatchMany_Function {
  my @R = @_;
  my $expr = join '||', map { "\$_[0] =~ m/\$R[$_]/o" } ( 0..$#R );
  my $matchsub = eval "sub { $expr }";
  die "Failed in building regex @R: $@" if $@;
  $matchsub;
}
```

Example usage:

```
@some_browsers = qw(Mozilla Lynx MSIE AmigaVoyager lwp libwww);
$Known_Browser=Build_MatchMany_Function(@some_browsers);


while (<ACCESS_LOG>) {
  # ...
  $browser = get_browser_field($_);
  if ( ! &$Known_Browser($browser) ) {
    print STDERR "Unknown Browser: $browser\n";
  }
```

```
  # ...
}
```

In the next article I'll present a few other Perl basics directly related to the mod_perl programming.

References

- The book ``*Mastering Regular Expressions*'' by Jeffrey E. Friedl.
- The book ``*Learning Perl*'' by Randal L. Schwartz (also known as the ``*Llama*'' book, named after the llama picture on the cover of the book).
- The book ``*Programming Perl*'' by L.Wall, T. Christiansen and J.Orwant (also known as the ``*Camel*'' book, named after the camel picture on the cover of the book).
- The *Exporter*, *perlre*, *perlvar*, *perlmod* and *perlmodlib* man pages.