Apache handlers with mod_perl

Author : *rbowen*

Date : 2002/10/0718 : 31 : 58

Contents

1 mod_perl as CGI accellerator **2**

2 What mod_perl is really for **2**

3 Example mod_perl handler **3**

4 So what does this gain me **4**

5 Writing a Perl module **4**

6 Where the modules live **5**

7 Installing a mod_perl handler **6**

7.1 Installing a Perl module manually 6

7.2 Installing a Perl module the right way 6

7.3 Auto-generating this stuff 7

8 Apache configuration **8**

9 Actual useful handlers **9**

10 What next? **10**

11 Apache::DBI **10**

12 Things not to do **11**

12.1 exit 11

12.2 Global variables 11

12.3 Other random tips 12

1 mod_perl as CGI accellerator

`mod_perl` is a very popular, widely installed module, which is used primarily as a CGI accelerator. Simply by running a CGI program under the `mod_perl` handlers `Apache::Registry` or `Apache::PerlRun`, you can get pretty significant speed improvements in your CGI performance, without changing a line of code.

At least, this seems to be what most people use it for.

And, it's really good at it. Here's some sample benchmarks using the `ab` utility that comes with Apache:

```
./ab -n 1000 -c 10 http://localhost/cgi-bin/example1.cgi

Requests per second:    58.59 [#/sec] (mean)

...

./ab -n 1000 -c 10 http://localhost/cgi-perl/example1.cgi

Requests per second:    108.70 [#/sec] (mean)

...

./ab -n 1000 -c 10 http://localhost/perl/example1.cgi

Requests per second:    213.40 [#/sec] (mean)
```

So those are some pretty significant results.

2 What mod_perl is really for

However, the real power of `mod_perl` is that you have direct access to the Apache API, so that you can write handlers for any phase of the Apache transations, directly in Perl.

The Apache API provides hooks into every part of the HTTP transaction, so that modules can affect authentication, URL mapping, content negotiation, or any other aspect of the transaction.

`mod_perl` lets you do this in Perl.

Now, this is only going to be an advantage if you happen to know Perl, but, aside from that little piece of trivia, this is a great tool for rapid development of web based applications, without all of the pain and slowness associated with CGI.

`mod_perl` handlers are most frequently used for content generation - that is, as a replacement for CGI programs that do roughly the same thing. However, they are also frequently used for AAA (Authentication, Authorization and Access Control) handlers.

A `mod_perl` handler is a Perl module with a method called `handler`. Actually, you can call your method whatever you want, but the convention is to call it `handler`, and this is the method that `mod_perl` will call by default if you don't specify something else.

This method will receive a single argument, which is an object in the `Apache` class. It is traditional to call this object `$r`, which presumably stands for "Request". This object can then be used to make calls into the Apache API.

This method, by means of calling methods via `$r`, returns content in much the same way that a CGI program would do.

3 Example mod_perl handler

The following is an example `mod_perl` handler - the standard beginner "hello world" example program which does nothing more interesting than to display a message in the browser.

```
package Apache::HandlerTest;
# File is called Apache/HandlerTest.pm
# Path: /usr/lib/perl5/site_perl/5.6.0/Apache/HandlerTest.pm

sub handler
    my $r = shift; # Apache session object
    $r->content_type('text/html');
    $r->send_http_header;
    $r->print( "Hello, world." );


1;
```

To configure Apache to use this handler, you would need to add a configuration section like the following to your `httpd.conf` file:

```
<Location /handlertest>
   SetHandler perl-script
   PerlModule Apache::HandlerTest
   PerlHandler Apache::HandlerTest
</Location>
```

This configuration will cause requests to the URL `/handlertest` to be handled by `Apache::HandlerTest`. If you want to call your method something other than `handler`, or if you have several methods in the same module, you can specify a method name in the `PerlHandler` directive:

```
PerlHandler Apache::HandlerTest::othermethod
```

As this handler does not do anything particularly interesting, there are only a few things worth commenting on here.

First, rather than printing output, as we are used to doing in CGI programs, output is sent directly to the

Apache API using the various methods illustrated. A CGI program is a forked process, and whatever it prints to `STDOUT` gets piped back to Apache for redelivery to the client. A `mod_perl` handler, on the other hand, is running in process in the Apache child, and sends things directly through the API. This, among several other things, leads to a substantial performance boost over CGI.

Second, it is useful to note that we set just the `content_type` header, but we can set any of the other standard headers in a similar manner. Alternately, if we are only interested in sending the one header, we can actually do this with a single function call as shown below:

```
$r->send_http_header( 'text/html' );
```

This would reduce the above handler by one line. Some people find that important. `;-)`

4 So what does this gain me

There are a number of things that this gains you over just writing CGI programs. The first of these things is performance. Our fastest CGI running under `Apache::Registry` ran at 213 requests per second on my crusty old laptop. The above `mod_perl` handler, which does essentially the same thing, ran at 296 requests per second. - almost another 50% improvement.

The reasons for this are numerous, and are even more numerous when your handler is doing something slightly more useful, such as database access, as we'll discuss later.

First, Whereas CGI forks an external process, the `mod_perl` handler runs in process, meaning that no inter-process communication is needed. The `mod_perl` module is loaded in with the Apache process, as is any other module

Second, your Perl code does not need to be loaded and compiled each time, as is the case with CGI programs. The `PerlModule` directive causes your Perl module to be loaded into the Apache child process. The code contained in it is evaluated, and cached in memory as the compiled bytecode. Each time your handler is invoked, this compiled code is merely executed. No interpretation and compilation phases are required.

Of course, this is a bit of an over-simplification, and you need to see the `mod_perl` documentation for the gory details. But that's the basic idea.

5 Writing a Perl module

In order to write a `mod_perl` handler, you need to know how to write a Perl module. Specifically, you need to know how to write an OO (Object Oriented) Perl module. This is much easier than one might expect. OO is really simple in Perl.

Here are the basics.

A class, which is one of the fundamental building blocks of OO terminology, is just a module. A module is simply a file beginning with a `package` declaration, and containing one or more methods.

A method is a function which receives, as its first argument, an object. An object method is called by an object using the special arrow syntax:

```
$return = $object->method( @arguments );
```

And an object is a reference (usually a hash reference) which has been acted upon by the `bless()` operator to associate it with a particular class.

Strictly speaking, you don't really need to know all of this stuff to write `mod_per` handlers. But it helps.

In a particular module, methods will typically receive, as their argument, an object in that class.

For example, if you were to create an object in the `Object::Namespace` class:

```
use Object::Namespace;
my $object = Object::Namespace->new;
$return = $object->methodname;
```

The the method, in the `Object::Namespace` class, would look something like this:

```
sub methodname
    my $self = shift;
    # $self is an object in the Object::Namspace class
```

The class `Object::Namespace` will (usually) be located in a file called `Namespace.pm`, which is located in a `Object/` subdirectory somewhere.

But where ...

6 Where the modules live

When a Perl module is loaded, using the `use` keyword, Perl loads that particular module file from the file system, by looking in several different directories until it finds the file.

The list of directories in which it will look - the module path - is stored in the global variable `INC`. To see the contents of the variable `INC`, run the following command:

```
rbowen@rhiannon:~% perl -le 'print join "\ n",@INC;'
/usr/lib/perl5/5.6.0/i686-linux
/usr/lib/perl5/5.6.0
/usr/lib/perl5/site_perl/5.6.0/i686-linux
/usr/lib/perl5/site_perl/5.6.0
/usr/lib/perl5/site_perl
.
```

The sample output there shows the module path for an installation of Perl 5.6.0 on Linux. This may be different on your system.

In each of these directories, the modules name is used to find the file path. For example a module called `Apache::HandlerTest` will be located in the file `Apache/HandlerTest.pm`, somewhere in the Perl library path. In this particular case, that file is located at
`/usr/lib/perl5/site_perl/5.6.0/Apache/HandlerTest.pm`

7 Installing a mod_perl handler

Installing a `mod_perl` handler consists of two steps. First, you have to install the Perl module itself, and then you have to tell Apache that you want to use it.

Installing a Perl module is a fairly simple process. It can be done one of two ways. You can either install the module file manually, or, for a module built in a standard way, you can run the installation script (sort of) that comes with it. Of course, if you are writing your own handler module, you will need to know how to generate this installation program yourself.

7.1 Installing a Perl module manually

As described above, a Perl module is located somewhere in your Perl module path, in a location corresponding to its name. Typically, the `site_perl` directory is reserved for modules that are installed on your system after the initial installation of Perl - as opposed to modules that came with Perl when it was initially installed.

So, for a module `Apache::Example`, we'll put it at `/usr/lib/perl5/site_perl/5.6.0/Apache/Example.pm`, for example. If you want to use the example module shown above, then, type in that file, and put it in this location so that Perl can find it when we `use` the module.

7.2 Installing a Perl module the right way

Most Perl modules (ie, those you get off of CPAN) install with:

```
perl Makefile.PL
make
make test
make install
```

If you are writing a real module, for distribution beyond a single machine on which you are writing it, yours should also install with this procedure.

Fortunately, you don't need to know anything about how to write makefiles in order to make this happen. Perl comes with a wonderful module called `ExtUtils::MakeMaker` which handles all of this stuff for you.

`Makefile.PL` is a simple script that describes some of the basic information about your module, and, when it is run, it generates a full-blown makefile which can be used to build and install your module.

The contents of `Makefile.PL` are really simple (or at least they can be). Here's an example `Makefile.PL`:

```
use ExtUtils::MakeMaker;

WriteMakefile(
  'NAME'      => 'Apache::Example',
  'VERSION_FROM' => 'lib/Apache/Example.pm',    # finds $VERSION
);
```

Yes, that's all there is to it.

so, now, what does it mean? Well, since there's not much to it, it's simple to explain.

The first line loads the `Extutils::MakeMaker` module, which knows how to build makefiles from the simple hash provided in your `Makefile.PL`

Next, we have a call to the `WriteMakeFile` function, to which we can pass a number of arguments describing your module. We've passed a very minimal list of arguments. You should see the documentation for `Extutils::Makemaker` for a more complete list of arguments that you can pass.

The `NAME` argument specifies the name of the package. This is used when a distribution is built with the `make dist` command.

The `VERSION_FROM` argument specifies which file contains the variable `$VERSION`, which will determine the version number put on a distribution.

That's really all there is to it. You may also want to have a file called `MANIFEST` which contains a listing of all files that are to be distributed with your module. This will generally consist of the module itself, the `Makiefile.PL`, perhaps a `README` file, and perhaps some tests.

7.3 Auto-generating this stuff

While it is nice to know how to write these things from scratch, it is often very convenient to be able to automatically generate a skeleton from which you can work. This is helpful so that you don't forget anything, and also makes sure that you get the syntax correct.

There are a variety of ways to generate these things. The most widely advocated one is a utility called `h2xs` which gets installed when you install Perl. You can find out more about `h2sx` by looking at its documentation, and I won't say much more about it here, because I don't recommend that you use it.

By far the best way to generate Perl modules is `ExtUtils::ModuleMaker`, by Geoff Avery. In its simplest incarnation, you can generate a module distribution by running the following command:

```
perl -MExtUtils::ModuleMaker -e 'Quick_Module("Apache::Example");'
```

This command generates all of the necessary files for a Perl module, including the module itself, the `Makefile.PL`, stub documentation, and some basic tests. The directory structure created is much more conducive to growth than is that generated by `h2xs`, and there are a large number of additional arguments that you can pass to it to generate a more complex module tree.

For example, the following script is what I use to generate modules for internal use at my company:

```
#!/usr/bin/perl
use ExtUtils::ModuleMaker;

Generate_Module_Files (
    NAME     => 'Acme::Time::Asparagus',
    ABSTRACT => 'Time on the vegetable clock',
    AUTHOR   =>
        NAME    => 'Rich Bowen',
        EMAIL   => 'rbowen@CooperMcGregor.com',
        CPANID  => 'RBOW',
        WEBSITE => 'http://www.CooperMcGregor.com/',
    ,
    VERSION  => 1.10,
    LICENSE  => 'perl',
);
```

And, once you have generated the module distribution, your module can be installed using the standard incantation:

```
perl Makefile.PL && make && make install
```

Note that this module (`ExtUtils::ModuleMaker` is not (yet) a standard part of Perl, and needs to be installed from CPAN before you can use it. So go to your favorite CPAN mirror, or to CPAN.org, and download the latest version.

8 Apache configuration

Once you have the module installed, you then need to tell Apache to use it as a handler. This is usually a simple matter of a `<Location>` section and a few directives.

```
<Location /url_goes_here>
  SetHandler perl-script

  PerlModule Your::Module
  PerlHander Your::Module
</Location>
```

The `SetHandler` directive tells Apache that all requests for this url are to be handled by a `mod_perl` handler.

The `PerlModule` directive loads the module. This directive may not actually be required, but it is a good practice to have it in there. This directive can be used to load any other modules that may be needed also.

And the `PerlHandler` directive specifies that the url is to be handled using the methods in this module. If your handler method is not called `handler`, you may specify a particular method name in this directive:

```
PerlHandler Your::Module::othermethod
```

There are a plethora of other directives that may be used, if you wish to specify that something else is to be done using a Perl module. For example, if you want to do the authentication using a Perl module, this would be specified using the `PerlAuthenHandler` directive.

9 Actual useful handlers

Handlers that print "Hello World" are not particularly useful for anything other than demonstrating basic concepts. If we want a web handler that is even marginally useful, we need to be able to read in form content, or input from the URL `QUERY_STRING`, and do something useful with that.

Fortunately, mod_perl offers a simple way to do this.

```
sub handler
   my $r = shift;

   if ($r->method eq 'POST')
       %form = $r->content;
    else
       %form = $r->args;


   $r->send_http_header( 'text/html' );
   $r->print( "Name = " . $formname );
```

The method above would, of course, be contained in a module, which would in turn be loaded using the `PerlHandler` directive, as demonstrated in the section above.

The `method` method determines the HTTP method used (`GET` or `POST` in this case).

If the request was a `POST` request, then the form content will be available via `STDIN`, which is retrieved, and split into its component parts, using the `content` method. Otherwise, it is Epresumed that the request was a `GET` request, and the form contents are read out of the `QUERY_STRING` variable, using the `args` method.

In either case, we get a return value of a hash. This hash is composed of name and value pairs, corresponding to the fields that were in the HTML form that is being sent to us. The last lines of the handler return the content-type header, and display the value of the form fiend `name`, if there was such a field.

10 What next?

Well, once you've gotten this far, there's not much more to say. Particularly if you are used to writing CGI programs. After the first dozen or so CGI programs, they are all pretty much the same. You get form content, you do something with it, and you return a document to the client. The variations are minor.

The same is mostly true of `mod_perl` content generation handlers. There are, however, a few additional things that you might want to know.

11 Apache::DBI

As you may already be aware, the Perl `DBI` module facilitates connections to your favorite database. Most databases in common use have a DBD (database driver) that can be used with the DBI so that the database is accessible from Perl.

One of the big performance problems with CGI is that you have to reestabilsh your connection to the database with every request for the CGI resource. And something that `mod_perl` gives you is persistent database connections, to remove this limitation, and give a substantial performance boost.

This persistence is provided through the module `Apache::DBI`, which hijacks all calls to the `DBI` module, and subverts them slightly. In particular, it maintains a pool of database connections, and gives you one of these when you attempt to connect to the database. Also, when you call the `disconnect` method, `Apache::DBI` politely ignores you, maintaining the open database handle to be used the next time.

Additionally, you can tell Apache to connect to the database as soon as a child process is launched, and save even the overhead of the initial connection.

This, and other initialization steps, can be performed using the `PerlRequire` directive, and an initialization script. As this is not a `mod_perl` introduction, I'll assume that you already know basically how that works.

In `httpd.conf`, put:

```
PerlRequire /path/to/apache_preload.pl
```

Then, in `apache_preload.pl`:

```
use Apache::DBI ();
use DBI ();

Apache::DBI->connect_on_init (
       $odatabase, $DBI_username, $DBI_password,
        AutoCommit => 1,
);
```

Note that `Apache::DBI` should get loaded before `DBI`. `Apache::DBI` will create the database connection each time a new child process is started. Then, in your code, you just use DBI as you would ordinarily.

12 Things not to do

When you are writing your handler modules, there are several things that you want to avoid doing.

12.1 exit

In Perl, the `exit` command is used to terminate a running program. It is often used in scripts to cause a program to end abruptly.

Don't use this function when writing `mod_perl` handlers. It does Bad Things. In particular, it causes the Perl interpeter to exit. Since one of the primary functions of `mod_perl` is to keep a Perl interpreter running in the Apache process, this causes a problem. You end up with an Apache process that thinks it has a Perl interpreter, but which doesn't.

This can cause a number of unpleasant symptoms. In one incarnation, a request will be sent to `mod_perl`, which is unable to respond, and then Apache will return to the client whatever happened to be in the send buffer from the last time. In other words, you can get in a situation where every request gets served the same content, regardless of what the request was for. Clearly this is a bad thing, and may even result in a security problem whereby sensitive documents are distributed to unauthorized users.

So, don't use the `exit` function, ever.

12.2 Global variables

Don't use global variables. Of course, you should not use global variables anyway. It's a really bad practice, and leads to unexpected results. However, this is particularly true with `mod_perl` applications. Remember that the Perl interpreter does not exit between accesses, and your Perl code also does not get reloaded or purged between accesses. Thus, and global variable set within one access may persist until the next user comes to the web site, with unpleasant side effects.

In one anecdote, variabes `$username` and `$password` were set as global variables, causing anyone coming to the site and not providing a valid username and password to be admitted with the authentication of the previous visitor to the site. You can imagine that this would cause problems.

12.3 Other random tips

`use strict;` and `use warnings;` at the start of every piece of Perl you write. It will make your life better. Use `my` on every one of your variables.

Of course, you should already be doing these things anyway, right?