

1 Writing mod_perl Handlers and Scripts

1.1 Description

This chapter covers the `mod_perl` coding specifics, different from normal Perl coding. Most other perl coding issues are covered in the perl manpages and rich literature.

1.2 Prerequisites

1.3 Where the Methods Live

`mod_perl` 2.0 has all its methods spread across many modules. In order to use these methods the modules containing them have to be loaded first. If you don't do that `mod_perl` will complain that it can't find the methods in question. The module `ModPerl::MethodLookup` can be used to find out which modules need to be used.

1.4 Goodies Toolkit

1.4.1 Environment Variables

`mod_perl` sets the following environment variables:

- `$ENV{MOD_PERL}` - is set to the `mod_perl` version the server is running under. e.g.:

```
mod_perl/1.99_03-dev
```

If `$ENV{MOD_PERL}` doesn't exist, most likely you are not running under `mod_perl`.

```
die "I refuse to work without mod_perl!" unless exists $ENV{MOD_PERL};
```

However to check which version is used it's better to use the following technique:

```
use mod_perl;
use constant MP2 => ($mod_perl::VERSION >= 1.99);
# die "I want mod_perl 2.0!" unless MP2;
```

- `$ENV{GATEWAY_INTERFACE}` - is set to `CGI-Perl/1.1` for compatibility with `mod_perl` 1.0. This variable is deprecated in `mod_perl` 2.0. Use `$ENV{MOD_PERL}` instead.

`mod_perl` passes (exports) the following shell environment variables (if they are set) :

- `PATH` - Executables search path.
- `TZ` - Time Zone.

Any of these environment variables can be accessed via `%ENV`.

1.4.2 Threaded MPM or not?

If the code needs to behave differently depending on whether it's running under one of the threaded MPMs, or not, the class method `Apache::MPM->is_threaded` can be used. For example:

```
use Apache::MPM ();
if (Apache::MPM->is_threaded) {
    require APR::OS;
    my $tid = APR::OS::thread_current();
    print "current thread id: $tid (pid: $$)";
}
else {
    print "current process id: $$";
}
```

This code prints the current thread id if running under a threaded MPM, otherwise it prints the process id.

1.4.3 Writing MPM-specific Code

If you write a CPAN module it's a bad idea to write code that won't run under all MPMs, and developers should strive to write a code that works with all mpms. However it's perfectly fine to perform different things under different mpms.

If you don't develop CPAN modules, it's perfectly fine to develop your project to be run under a specific MPM.

```
use Apache::MPM ();
my $mpm = lc Apache::MPM->show;
if ($mpm eq 'prefork') {
    # prefork-specific code
}
elsif ($mpm eq 'worker') {
    # worker-specific code
}
elsif ($mpm eq 'winnt') {
    # winnt-specific code
}
else {
    # others...
}
```

1.5 Code Developing Nuances

1.5.1 Auto-Reloading Modified Modules with `Apache::Reload`

META: need to port `Apache::Reload` notes from the guide here. but the gist is:

```
PerlModule Apache::Reload
PerlInitHandler Apache::Reload
#PerlPreConnectionHandler Apache::Reload
PerlSetVar ReloadAll Off
PerlSetVar ReloadModules "ModPerl:* Apache:*
```

Use:

```
PerlInitHandler Apache::Reload
```

if you need to debug HTTP protocol handlers. Use:

```
PerlPreConnectionHandler Apache::Reload
```

for any handlers.

Though notice that we have started to practice the following style in our modules:

```
package Apache::Whatever;

use strict;
use warnings FATAL => 'all';
```

FATAL => 'all' escalates all warnings into fatal errors. So when `Apache::Whatever` is modified and reloaded by `Apache::Reload` the request is aborted. Therefore if you follow this very healthy style and want to use `Apache::Reload`, flex the strictness by changing it to:

```
use warnings FATAL => 'all';
no warnings 'redefine';
```

but you probably still want to get the *redefine* warnings, but downgrade them to be non-fatal. The following will do the trick:

```
use warnings FATAL => 'all';
no warnings 'redefine';
use warnings 'redefine';
```

Perl 5.8.0 allows to do all this in one line:

```
use warnings FATAL => 'all', NONFATAL => 'redefine';
```

but if your code may be used with older perl versions, you probably don't want to use this new functionality.

Refer to the *perllexwarn* manpage for more information.

1.6 Integration with Apache Issues

In the following sections we discuss the specifics of Apache behavior relevant to `mod_perl` developers.

1.6.1 Sending HTTP Response Headers

Apache 2.0 doesn't provide a method to force HTTP response headers sending (what used to be done by `send_http_header()` in Apache 1.3). HTTP response headers are sent as soon as the first bits of the response body are seen by the special core output filter that generates these headers. When the response handler send the first chunks of body it may be cached by the mod_perl internal buffer or even by some of the output filters. The response handler needs to flush in order to tell all the components participating in the sending of the response to pass the data out.

For example if the handler needs to perform a relatively long-running operation (e.g. a slow db lookup) and the client may timeout if it receives nothing right away, you may want to start the handler by setting the *Content-Type* header, following by an immediate flush:

```
sub handler {
    my $r = shift;
    $r->content_type('text/html');
    $r->rflush; # send the headers out

    $r->print(long_operation());
    return Apache::OK;
}
```

If this doesn't work, check whether you have configured any third-party output filters for the resource in question. Improperly written filter may ignore the orders to flush the data.

META: add a link to the notes on how to write well-behaved filters at handlers/filters

1.6.2 Sending HTTP Response Body

In mod_perl 2.0 a response body can be sent only during the response phase. Any attempts to do that in the earlier phases will fail with an appropriate explanation logged into the *error_log* file.

This happens due to the Apache 2.0 HTTP architecture specifics. One of the issues is that the HTTP response filters are not setup before the response phase.

1.7 Perl Specifics in the mod_perl Environment

In the following sections we discuss the specifics of Perl behavior under mod_perl.

1.7.1 Request-localized Globals

mod_perl 2.0 provides two types of `SetHandler` handlers: `modperl` and `perl-script`. Remember that the `SetHandler` directive is only relevant for the response phase handlers, it neither needed nor affects non-response phases.

Under the handler:

```
SetHandler perl-script
```

several special global Perl variables are saved before the handler is called and restored afterwards. This includes: %ENV, @INC, \$/, STDOUT's \$| and END blocks array (PL_endav).

Under:

```
SetHandler modperl
```

nothing is restored, so you should be especially careful to remember localize all special Perl variables so the local changes won't affect other handlers.

1.7.2 *exit()*

In the normal Perl code `exit()` is used to stop the program flow and exit the Perl interpreter. However under `mod_perl` we only want to stop the program flow without killing the Perl interpreter.

You should take no action if your code includes `exit()` calls and it's OK to continue using them. `mod_perl` worries to override the `exit()` function with its own version which stops the program flow, and performs all the necessary cleanups, but doesn't kill the server. This is done by overriding:

```
*CORE::GLOBAL::exit = \&ModPerl::Util::exit;
```

so if you mess up with `*CORE::GLOBAL::exit` yourself you better know what you are doing.

You can still call `CORE::exit` to kill the interpreter, again if you know what you are doing.

1.8 Threads Coding Issues Under mod_perl

The following sections discuss threading issues when running `mod_perl` under a threaded MPM.

1.8.1 *Thread-environment Issues*

The "only" thing you have to worry about your code is that it's thread-safe and that you don't use functions that affect all threads in the same process.

Perl 5.8.0 itself is thread-safe. That means that operations like `push()`, `map()`, `chomp()`, `=`, `/`, `+=`, etc. are thread-safe. Operations that involve system calls, may or may not be thread-safe. It all depends on whether the underlying C libraries used by the perl functions are thread-safe.

For example the function `localtime()` is not thread-safe when the implementation of `asctime(3)` is not thread-safe. Other usually problematic functions include `readdir()`, `srand()`, etc.

Another important issue that shouldn't be missed is what some people refer to as *thread-locality*. Certain functions executed in a single thread affect the whole process and therefore all other threads running inside that process. For example if you `chdir()` in one thread, all other thread now see the current

working directory of that thread that `chdir()`'ed to that directory. Other functions with similar effects include `umask()`, `chroot()`, etc. Currently there is no cure for this problem. You have to find these functions in your code and replace them with alternative solutions which don't incur this problem.

For more information refer to the *perlthrtut* (<http://perldoc.com/perl5.8.0/pod/perlthrtut.html>) manpage.

1.8.2 Deploying Threads

This is actually quite unrelated to mod_perl 2.0. You don't have to know much about Perl threads, other than Thread-environment Issues, to have your code properly work under threaded MPM mod_perl.

If you want to spawn your own threads, first of all study how the new *ithreads* Perl model works, by reading the *perlthrtut*, *threads* (<http://search.cpan.org/search?query=threads>) and *threads::shared* (<http://search.cpan.org/search?query=threads%3A%3Ashared>) manpages.

Artur Bergman wrote an article which explains how to port pure Perl modules to work properly with Perl *ithreads*. Issues with `chdir()` and other functions that rely on shared process' datastructures are discussed. <http://www.perl.com/lpt/a/2002/06/11/threads.html>.

1.8.3 Shared Variables

Global variables are only global to the interpreter in which they are created. Other interpreters from other threads can't access that variable. Though it's possible to make existing variables shared between several threads running in the same process by using the function `threads::shared::share()`. New variables can be shared by using the *shared* attribute when creating them. This feature is documented in the *threads::shared* (<http://search.cpan.org/search?query=threads%3A%3Ashared>) manpage.

1.9 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

-

1.10 Authors

-

Only the major authors are listed above. For contributors see the Changes file.

Table of Contents:

1	Writing mod_perl Handlers and Scripts	1
1.1	Description	2
1.2	Prerequisites	2
1.3	Where the Methods Live	2
1.4	Goodies Toolkit	2
1.4.1	Environment Variables	2
1.4.2	Threaded MPM or not?	3
1.4.3	Writing MPM-specific Code	3
1.5	Code Developing Nuances	3
1.5.1	Auto-Reloading Modified Modules with Apache::Reload	3
1.6	Integration with Apache Issues	4
1.6.1	Sending HTTP Response Headers	5
1.6.2	Sending HTTP Response Body	5
1.7	Perl Specifics in the mod_perl Environment	5
1.7.1	Request-localized Globals	5
1.7.2	exit()	6
1.8	Threads Coding Issues Under mod_perl	6
1.8.1	Thread-environment Issues	6
1.8.2	Deploying Threads	7
1.8.3	Shared Variables	7
1.9	Maintainers	7
1.10	Authors	7