1 Overview of mod_perl 2.0

1.1 Description

This chapter should give you a general idea about what mod_perl 2.0 is and how it differs from mod_perl 1.0. This chapter presents the new features of Apache 2.0, Perl 5.6.0 -- 5.8.0 and their influence on mod_perl 2.0. The new MPM models from Apache 2.0 are discussed.

1.2 Version Naming Conventions

In order to keep things simple, here and in the rest of the documentation we refer to mod_perl 1.x series as mod_perl 1.0 and to 2.0.x series as mod_perl 2.0. Similarly we call Apache 1.3.x series as Apache 1.3 and 2.0.x as Apache 2.0. There is also Apache 2.1, which is a development track towards Apache 2.2.

1.3 Why mod_perl, The Next Generation

mod_perl was introduced in early 1996, both Perl and Apache have changed a great deal since that time. mod_perl has adjusted to both along the way over the past 4 and a half years or so using the same code base. Over this course of time, the mod_perl sources have become more and more difficult to maintain, in large part to provide compatibility between the many different flavors of Apache and Perl. And, compatibility across these versions and flavors is a more difficult goal for mod_perl to reach that a typical Apache or Perl module, since mod_perl reaches a bit deeper into the corners of Apache and Perl internals than most. Discussions of the idea to rewrite mod_perl as version 2.0 started in 1998, but never made it much further than an idea. When Apache 2.0 development was underway it became clear that a rewrite of mod_perl would be required to adjust to the new Apache architecture and API.

Of the many changes happening in Apache 2.0, the one which has the most significant impact on mod_perl is the introduction of threads to the overall design. Threads have been a part of Apache on the win32 side since the Apache port was introduced. The mod_perl port to win32 happened in version 1.00b1, released in June of 1997. This port enabled mod_perl to compile and run in a threaded windows environment, with one major caveat: only one concurrent mod_perl request could be handled at any given time. This was due to the fact that Perl did not introduce thread-safe interpreters until version 5.6.0, released in March of 2000. Contrary to popular belief, the "threads support" implemented in Perl 5.005 (released July 1998), did not make Perl thread-safe internally. Well before that version, Perl had the notion of "Multiplicity", which allowed multiple interpreter instances in the same process. However, these instances were not thread safe, that is, concurrent callbacks into multiple interpreters were not supported.

It just so happens that the release of Perl 5.6.0 was nearly at the same time as the first alpha version of Apache 2.0. The development of mod_perl 2.0 was underway before those releases, but as both Perl 5.6.0 and Apache 2.0 were reaching stability, mod_perl 2.0 was becoming more of a reality. In addition to the adjustments for threads and Apache 2.0 API changes, this rewrite of mod_perl is an opportunity to clean up the source tree. This includes both removing the old backward compatibility bandaids and building a smarter, stronger and faster implementation based on lessons learned over the 4.5 years since mod_perl was introduced.

The new version includes a mechanism for an automatic building of the Perl interface to Apache API, which allowed us to easily adjust mod_perl 2.0 to ever changing Apache 2.0 API, during its development period. Another important feature is the `Apache::Test` framework, which was originally developed for mod_perl 2.0, but then was adopted by Apache 2.0 developers to test the core server features and third party modules. Moreover the tests written using the `Apache::Test` framework could be run with Apache 1.0 and 2.0, assuming that both supported the same features.

There are multiple other interesting changes that have already happened to mod_perl in version 2.0 and more will be developed in the future. Some of these are discussed in this chapter, others can be found in the rest of the mod_perl 2.0 documentation.

1.4 What's new in Apache 2.0

Apache 2.0 has introduced numerous new features and enhancements. Here are the most important new features:

- ***Apache Portable Runtime*** **(APR)**

  Apache 1.3 has been ported to a very large number of platforms including various flavors of unix, win32, os/2, the list goes on. However, in 1.3 there was no clear-cut, pre-designed portability layer for third-party modules to take advantage of. APR provides this API layer in a very clean way. APR assists a great deal with mod_perl portability. Combined with the portablity of Perl, mod_perl 2.0 needs only to implement a portable build system, the rest comes "for free". A Perl interface is provided for certain areas of APR, such as the shared memory abstraction, but the majority of APR is used by mod_perl "under the covers".

  The APR uses the concept of memory pools, which significantly simplifies the memory management code and reduces the possibility of having memory leaks, which always haunt C programmers.

- **I/O Filtering**

  Filtering of Perl modules output has been possible for years since tied filehandle support was added to Perl. There are several modules, such as `Apache::Filter` and `Apache::OutputChain` which have been written to provide mechanisms for filtering the `STDOUT` stream. There are several of these modules because no one's approach has quite been able to offer the ease of use one would expect, which is due simply to limitations of the Perl tied filehandle design. Another problem is that these filters can only filter the output of other Perl modules. C modules in Apache 1.3 send data directly to the client and there is no clean way to capture this stream. Apache 2.0 has solved this problem by introducing a filtering API. With the baseline I/O stream tied to this filter mechansim, any module can filter the output of any other module, with any number of filters in between. Using this new feature things like SSL, data (de-)compression and other data manipulations are done very easily.

- ***Multi Processing Model modules*** **(MPMs).**

  In Apache 1.3 concurrent requests were handled by multiple processes, and the logic to manage these processes lived in one place, *http_main.c*, 7700 some odd lines of code. If Apache 1.3 is compiled on

a Win32 system large parts of this source file are redefined to handle requests using threads. Now suppose you want to change the way Apache 1.3 processes requests, say, into a DCE RPC listener. This is possible only by slicing and dicing *http_main.c* into more pieces or by redefining the *standalone_main* function, with a `-DSTANDALONE_MAIN=your_function` compile time flag. Neither of which is a clean, modular mechanism.

Apache-2.0 solves this problem by introducing *Multi Processing Model modules*, better known as *MPMs*. The task of managing incoming requests is left to the MPMs, shrinking *http_main.c* to less than 500 lines of code. Now it's possible to write different processing modules specific to various platforms. For example the Apache 2.0 on Windows is much more efficient now, since it uses *mpm_winnt* which deploys the native Windows features.

Here is a partial list of major MPMs available as of this writing.

- **prefork**

  The *prefork* MPM emulates Apache 1.3's preforking model, where each request is handled by a different forked child process.

- **worker**

  The *worker* MPM implements a hybrid multi-process multi-threaded approach based on the *pthreads* standard. It uses one acceptor thread, multiple worker threads.

- **mpmt_os2, netware, winnt and beos**

  These MPMs also implement the hybrid multi-process/multi-threaded model, with each based on native OS thread implementations.

- **perchild**

  The *perchild* MPM is similar to the *worker* MPM, but is extended with a mechanism which allows mapping of requests to virtual hosts to a process running under the user id and group configured for that host. This provides a robust replacement for the *suexec* mechanism.

  META: as of this writing this mpm is not working

On platforms that support more than one MPM, it's possible to switch the used MPMs as the need change. For example on Unix it's possible to start with a preforked module. Then when the demand is growing and the code matures, it's possible to migrate to a more efficient threaded MPM, assuming that the code base is capable of running in the threaded environment.

- **New Hook Scheme**

  In Apache 1.3, modules were registered using the *module* structure, normally static to *mod_foo.c*. This structure contains pointers to the command table, configuration creation and merging functions, response handler table and function pointers for all of the other hooks, such as *child_init* and *check_user_id*. In Apache 2.0, this structure has been pruned down to the first three items mentioned and a new function pointer added called *register_hooks*. It is the job of *register_hooks* to register

functions for all other hooks (such as *child_init* and *check_user_id*). Not only is hook registration now dynamic, it is also possible for modules to register more than one function per hook, unlike 1.3. The new hook mechanism also makes it possible to sort registered functions, unlike 1.3 with function pointers hardwired into the module structure, and each module structure into a linked list. Order in 1.3 depended on this list, which was possible to order using compile-time and startup-time configuration, but that was left to the user. Whereas in 2.0, the *add_hook* functions accept an order preference parameter, those commonly used are:

- **FIRST**
- **MIDDLE**
- **LAST**

For mod_perl, dynamic registration provides a cleaner way to bypass the `Perl*Handler` configuration directives. By simply adding this configuration:

```
PerlModule Apache::Foo
```

`Apache::Foo` can register hooks itself at server startup:

```
Apache::Hook->add(PerlAuthenHandler => \&authenticate,
                  Apache::Hook::MIDDLE);
Apache::Hook->add(PerlLogHandler    => \&logger,
                  Apache::Hook::LAST);
```

META: Not implemented yet (API will change?)

However, this means that Perl subroutines registered via this mechanism will be called for **every** request. It will be left to that subroutine to decide if it was to handle or decline the given phase. As there is overhead in entering the Perl runtime, it will most likely be to your advantage to continue using `Perl*Handler` configuration directives to reduce this overhead. If it is the case that your `Perl*Handler` should be invoked for every request, the hook registration mechanism will save some configuration keystrokes.

- **Protocol Modules**

  Apache 1.3 is hardwired to speak only one protocol, HTTP. Apache 2.0 has moved to more of a "server framework" architecture making it possible to plugin handlers for protocols other than HTTP. The protocol module design also abstracts the transport layer so protocols such as SSL can be hooked into the server without requiring modifications to the Apache source code. This allows Apache to be extended much further than in the past, making it possible to add support for protocols such as FTP, SMTP, RPC flavors and the like. The main advantage being that protocol plugins can take advantage of Apache's portability, process/thread management, configuration mechanism and plugin API.

- **Parsed Configuration Tree**

  When configuration files are read by Apache 1.3, it hands off the parsed text to module configuration directive handlers and discards that text afterwards. With Apache 2.0, the configuration files are first parsed into a tree structure, which is then walked to pass data down to the modules. This tree is then left in memory with an API for accessing it at request time. The tree can be quite useful for other

modules. For example, in 1.3, mod_info has its own configuration parser and parses the configuration files each time you access it. With 2.0 there is already a parse tree in memory, which mod_info can then walk to output its information.

If a mod_perl 1.0 module wants access to configuration information, there are two approaches. A module can "subclass" directive handlers, saving a copy of the data for itself, then returning **DECLINE_CMD** so the other modules are also handed the info. Or, the `$Apache::Server::SaveConfig` variable can be set to save <Perl> configuration in the `%Apache::ReadConfig::` namespace. Both methods are rather kludgy, version 2.0 provides a Perl interface to the Apache configuration tree.

All these new features boost the Apache performance, scalability and flexibility. The APR helps the overall performance by doing lots of platform specific optimizations in the APR internals, and giving the developer the API which was already greatly optimized.

Apache 2.0 now includes special modules that can boost performance. For example the mod_mmap_static module loads webpages into the virtual memory and serves them directly avoiding the overhead of *open()* and *read()* system calls to pull them in from the filesystem.

The I/O layering is helping performance too, since now modules don't need to waste memory and CPU cycles to manually store the data in shared memory or *pnotes* in order to pass the data to another module, e.g., in order to provide response's gzip compression.

And of course a not least important impact of these features is the simplification and added flexibility for the core and third party Apache module developers.

1.5 What's new in Perl 5.6.0 - 5.8.0

As we have mentioned earlier Perl 5.6.0 is the minimum requirement for mod_perl 2.0. Though as we will see later certain new features work only with Perl 5.8.0 and higher.

These are the important changes in the recent Perl versions that had an impact on mod_perl. For a complete list of changes see the corresponding to the used version *perldelta* manpages (*http://perldoc.com/perl5.8.0/pod/perl56delta.html*, *http://perldoc.com/perl5.8.0/pod/perl561delta.html* and *http://perldoc.com/perl5.8.0/pod/perldelta.html*).

The 5.6 Perl generation has introduced the following features:

- The beginnings of support for running multiple interpreters concurrently in different threads. In conjunction with the perl_clone() API call, which can be used to selectively duplicate the state of any given interpreter, it is possible to compile a piece of code once in an interpreter, clone that interpreter one or more times, and run all the resulting interpreters in distinct threads. See the *perlembed* (*http://perldoc.com/perl5.6.1/pod/perlembed.html*) and *perl561delta* (*http://perldoc.com/perl5.6.1/pod/perl561delta.html*) manpages.

- The core support for declaring subroutine attributes, which is used by mod_perl 2.0's *method handlers*. See the *attributes* manpage.

- The *warnings* pragma, which allows to force the code to be super clean, via the setting:

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

  which will abort any code that generates warnings. This pragma also allows a fine control over what warnings should be reported. See the *perllexwarn* (*http://perldoc.com/perl5.6.1/pod/perllexwarn.html*) manpage.

- Certain `CORE::` functions now can be overridden via `CORE::GLOBAL::` namespace. For example mod_perl now can override `CORE::exit()` via `CORE::GLOBAL::exit`. See the *perlsub* (*http://perldoc.com/perl5.6.1/pod/perlsub.html*) manpage.

- The `XSLoader` extension as a simpler alternative to `DynaLoader`. See the *XSLoader* manpage.

- The large file support. If you have filesystems that support "large files" (files larger than 2 gigabytes), you may now also be able to create and access them from Perl. See the *perl561delta* (*http://perldoc.com/perl5.6.1/pod/perl561delta.html*) manpage.

- Multiple performance enhancements were made. See the *perl561delta* (*http://perldoc.com/perl5.6.1/pod/perl561delta.html*) manpage.

- Numerous memory leaks were fixed. See the *perl561delta* (*http://perldoc.com/perl5.6.1/pod/perl561delta.html*) manpage.

- Improved security features: more potentially unsafe operations taint their results for improved security. See the *perlsec* (*http://perldoc.com/perl5.6.1/pod/perlsec.html*) and *perl561delta* (*http://perldoc.com/perl5.6.1/pod/perl561delta.html*) manpages.

- Available on new platforms: GNU/Hurd, Rhapsody/Darwin, EPOC.

Overall multiple bugs and problems very fixed in the Perl 5.6.1, so if you plan on running the 5.6 generation, you should run at least 5.6.1. It is possible that when this tutorial is printed 5.6.2 will be out.

The Perl 5.8.0 has introduced the following features:

- The introduced in 5.6.0 experimental PerlIO layer has been stabilized and become the default IO layer in 5.8.0. Now the IO stream can be filtered through multiple layers. See the *perlapio* (*http://perldoc.com/perl5.8.0/pod/perlapio.html*) and *perliol* (*http://perldoc.com/perl5.8.0/pod/perliol.html*) manpages.

  For example this allows mod_perl to inter-operate with the APR IO layer and even use the APR IO layer in Perl code. See the `APR::PerlIO` manpage.

  Another example of using the new feature is the extension of the `open()` functionality to create anonymous temporary files via:

```
open my $fh, "+>", undef or die $!;
```

That is a literal `undef()`, not an undefined value. See the `open()` entry in the *perlfunc* manpage (*http://perldoc.com/perl5.8.0/pod/func/open.html*).

- More overridable via `CORE::GLOBAL::` keywords. See the *perlsub* (*http://perldoc.com/perl5.8.0/pod/perlsub.html*) manpage.

- The signal handling in Perl has been notoriously unsafe because signals have been able to arrive at inopportune moments leaving Perl in inconsistent state. Now Perl delays signal handling until it is safe.

- `File::Temp` was added to allow a creation of temporary files and directories in an easy, portable, and secure way. See the *File::Temp* manpage.

- A new command-line option, `-t` is available. It is the little brother of `-T`: instead of dying on taint violations, lexical warnings are given. This is only meant as a temporary debugging aid while securing the code of old legacy applications. **This is not a substitute for `-T`.** See the *perlrun* (*http://perldoc.com/perl5.8.0/pod/perlrun.html*) manpage.

  A new special variable `${^TAINT}` was introduced. It indicates whether taint mode is enabled. See the *perlvar* (*http://perldoc.com/perl5.8.0/pod/perlvar.html*) manpage.

- Threads implementation is much improved since 5.6.

- A much better support for Unicode.

- Numerous bugs and memory leaks fixed. For example now you can localize the tied `Apache::DBI` filehandles without leaking memory.

- Available on new platforms: AtheOS, Mac OS Classic, Mac OS X, MinGW, NCR MP-RAS, NonStop-UX, NetWare and UTS. The following platforms are again supported: BeOS, DYNIX/ptx, POSIX-BC, VM/ESA, z/OS (OS/390).

1.6 What's new in mod_perl 2.0

The new features introduced by Apache 2.0 and Perl 5.6 and 5.8 generations provide the base of the new mod_perl 2.0 features. In addition mod_perl 2.0 re-implements itself from scratch providing such new features as new build and testing framework. Let's look at the major changes since mod_perl 1.0.

1.6.1 Threads Support

In order to adapt to the Apache 2.0 threads architecture (for threaded MPMs), mod_perl 2.0 needs to use thread-safe Perl interpreters, also known as "ithreads" (Interpreter Threads). This mechanism can be enabled at compile time and ensures that each Perl interpreter uses its private `PerlInterpreter` structure for storing its symbol tables, stacks and other Perl runtime mechanisms. When this separation is engaged any number of threads in the same process can safely perform concurrent callbacks into Perl. This of course requires each thread to have its own `PerlInterpreter` object, or at least that each instance

is only accessed by one thread at any given time.

The first mod_perl generation has only a single `PerlInterpreter`, which is constructed by the parent process, then inherited across the forks to child processes. mod_perl 2.0 has a configurable number of `PerlInterpreters` and two classes of interpreters, *parent* and *clone*. A *parent* is like that in mod_perl 1.0, where the main interpreter created at startup time compiles any pre-loaded Perl code. A *clone* is created from the parent using the Perl API *perl_clone()* (*http://www.perldoc.com/perl5.8.0/pod/perlapi.html#Cloning-an-interpreter*) function. At request time, *parent* interpreters are only used for making more *clones*, as the *clones* are the interpreters which actually handle requests. Care is taken by Perl to copy only mutable data, which means that no runtime locking is required and read-only data such as the syntax tree is shared from the *parent*, which should reduce the overall mod_perl memory footprint.

Rather than create a `PerlInterperter` per-thread by default, mod_perl creates a pool of interpreters. The pool mechanism helps cut down memory usage a great deal. As already mentioned, the syntax tree is shared between all cloned interpreters. If your server is serving more than mod_perl requests, having a smaller number of PerlInterpreters than the number of threads will clearly cut down on memory usage. Finally and perhaps the biggest win is memory re-use: as calls are made into Perl subroutines, memory allocations are made for variables when they are used for the first time. Subsequent use of variables may allocate more memory, e.g. if a scalar variable needs to hold a longer string than it did before, or an array has new elements added. As an optimization, Perl hangs onto these allocations, even though their values "go out of scope". mod_perl 2.0 has a much better control over which PerlInterpreters are used for incoming requests. The interpreters are stored in two linked lists, one for available interpreters and another for busy ones. When needed to handle a request, one interpreter is taken from the head of the available list and put back into the head of the same list when done. This means if for example you have 10 interpreters configured to be cloned at startup time, but no more than 5 are ever used concurrently, those 5 continue to reuse Perl’s allocations, while the other 5 remain much smaller, but ready to go if the need arises.

Various attributes of the pools are configurable using threads mode specific directives.

The interpreters pool mechanism has been abstracted into an API known as "tipool", *Thread Item Pool*. This pool can be used to manage any data structure, in which you wish to have a smaller number than the number of configured threads. For example a replacement for `Apache::DBI` based on the *tipool* will allow to reuse database connections between multiple threads of the same process.

1.6.2 Thread-environment Issues

While mod_perl itself is thread-safe, you may have issues with the thread-safety of your code. For more information refer to Threads Coding Issues Under mod_perl.

Another issue is that "global" variables are only global to the interpreter in which they are created. It’s possible to share variables between several threads running in the same process. For more information see: Shared Variables.

1.6.3 Perl Interface to the APR and Apache APIs

As we have mentioned earlier, Apache 2.0 uses two APIs:

- the Apache Portable APR (APR) API, which implements a portable and efficient API to handle generically work with files, sockets, threads, processes, shared memory, etc.
- the Apache API, which handles issues specific to the web server.

In mod_perl 1.0, the Perl interface back into the Apache API and data structures was done piecemeal. As functions and structure members were found to be useful or new features were added to the Apache API, the XS code was written for them here and there.

mod_perl 2.0 generates the majority of XS code and provides thin wrappers were needed to make the API more Perlish. As part of this goal, nearly the entire APR and Apache API, along with their public data structures are covered from the get-go. Certain functions and structures which are considered "private" to Apache or otherwise un-useful to Perl aren't glued. Most of the API behaves just as it did in mod_perl 1.0, so users of the API will not notice the difference, other than the addition of many new methods. Where API has changed a special back compatibility module can be used.

In mod_perl 2.0 the APR API resides in the `APR::` namespace, and obviously the `Apache::` namespace is mapped to the Apache API.

And in the case of `APR`, it is possible to use `APR` modules outside of Apache, for example:

```
% perl  -MApache2 -MAPR -MAPR::UUID -le 'print APR::UUID->new->format'
b059a4b2-d11d-b211-bc23-d644b8ce0981
```

The mod_perl 2.0 generator is a custom suite of modules specifically tuned for gluing Apache and allows for complete control over *everything*, providing many possibilities none of *xsubpp*, *SWIG* or *Inline.pm* are designed to do. Advantages to generating the glue code include:

- Not tied tightly to xsubpp
- Easy adjustment to Apache 2.0 API/structure changes
- Easy adjustment to Perl changes (e.g., Perl 6)
- Ability to "discover" hookable third-party C modules.
- Cleanly take advantage of features in newer Perls
- Optimizations can happen across-the-board with one-shot
- Possible to AUTOLOAD XSUBs
- Documentation can be generated from code

- Code can be generated from documentation

1.7 Integration with 2.0 Filtering

The mod_perl 2.0 interface to the Apache filter API comes in two flavors. First, similar to the C API, where bucket brigades need to be manipulated. Second, streaming filtering, is much simpler than the C API, since it hides most of the details underneath. For a full discussion on filters and implementation examples refer to the Input and Output Filters chapter.

1.7.1 Other New Features

In addition to the already mentioned new features, the following are of a major importance:

- Apache 2.0 protocol modules are supported. Later we will see an example of a protocol module running on top of mod_perl 2.0.

- mod_perl 2.0 provides a very simply to use interface to the Apache filtering API. We will present a filter module example later on.

- A feature-full and flexible `Apache::Test` framework was developed especially for mod_perl testing. While used to test the core mod_perl features, it is used by third-party module writers to easily test their modules. Moreover `Apache::Test` was adopted by Apache and currently used to test both Apache 1.3, 2.0 and other ASF projects. Anything that runs top of Apache can be tested with `Apache::Test`, be the target written in Perl, C, PHP, etc.

- The support of the new MPMs model makes mod_perl 2.0 can scale better on wider range of platforms. For example if you've happened to try mod_perl 1.0 on Win32 you probably know that the requests had to be serialized, i.e. only a single request could be processed at a time, rendering the Win32 platform unusable with mod_perl as a heavy production service. Thanks to the new Apache MPM design, now mod_perl 2.0 can be used efficiently on Win32 platforms using its native *win32* MPM.

1.7.2 Optimizations

The rewrite of mod_perl gives us the chances to build a smarter, stronger and faster implementation based on lessons learned over the 4.5 years since mod_perl was introduced. There are optimizations which can be made in the mod_perl source code, some which can be made in the Perl space by optimizing its syntax tree and some a combination of both. In this section we'll take a brief look at some of the optimizations that are being considered.

The details of these optimizations from the most part are hidden from mod_perl users, the exception being that some will only be turned on with configuration directives. A few of which include:

- "Compiled" `Perl*Handlers`

- Inlined `Apache::*.xs` calls
- Use of Apache pools for memory allocations

1.8 Maintainers

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

- Stas Bekman <stas (at) stason.org>

1.9 Authors

- Doug MacEachern <dougm (at) covalent.net>
- Stas Bekman <stas (at) stason.org>

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

Table of Contents: