mod_perl APIs

The Apache::, APR:: and ModPerl:: namespaces APIs for mod_perl 2.0

Last modified Sat Dec 6 12:13:28 2003 GMT

Table of Contents:

Part I: Apache:: Core API

Part II: APR:: Core API

- 14. APR - Perl Interface for libapr and libaprutil Libraries
  Normally you don't need to use this module. However if you are using an `APR::` package outside of mod_perl, you need to load APR first. For example:

- 15. APR::Const - Perl Interface for APR Constants

- 16. APR:PerlIO -- An APR Perl IO layer
  `APR::PerlIO` implements a Perl IO layer using APR's file manipulation as its internals.

- 17. APR::Table -- A Perl API for manipulating opaque string-content table
  `APR::Table` allows its users to manipulate opaque string-content tables.

Part III: ModPerl::

- 18. ModPerl::MethodLookup -- Map mod_perl 2.0 modules, objects and methods
  mod_perl 2.0 provides many methods, which reside in various modules. One has to load each of the modules before using the desired methods. `ModPerl::MethodLookup` provides the Perl API for finding module names which contain methods in question and other helper functions, like figuring out what methods defined by some module, or what methods can be called on a given object.

- 19. ModPerl::MM -- A "subclass" of ExtUtils::MakeMaker for mod_perl 2.0
  `ModPerl::MM` is a "subclass" of `ExtUtils::MakeMaker` for mod_perl 2.0, to a degree of sub-classability of `ExtUtils::MakeMaker`.

- 20. ModPerl::PerlRun - Run unaltered CGI scripts under mod_perl

- 21. ModPerl::Registry - Run unaltered CGI scripts persistently under mod_perl
  URIs in the form of `http://example.com/perl/test.pl` will be compiled as the body of a Perl subroutine and executed. Each child process will compile the subroutine once and store it in memory. It will recompile it whenever the file (e.g. *test.pl* in our example) is updated on disk. Think of it as an object oriented server with each script implementing a class loaded at runtime.

- 22. ModPerl::RegistryBB - Run unaltered CGI scripts persistently under mod_perl
  `ModPerl::RegistryBB` is similar to *`ModPerl::Registry`*, but does the bare minimum (mnemonic: BB = Bare Bones) to compile a script file once and run it many times, in order to get the maximum performance. Whereas *`ModPerl::Registry`* does various checks, which add a slight overhead to response times.

- 23. ModPerl::RegistryCooker - Cook mod_perl 2.0 Registry Modules
  `ModPerl::RegistryCooker` is used to create flexible and overridable registry modules which emulate mod_cgi for Perl scripts. The concepts are discussed in the manpage of the following modules: *`ModPerl::Registry`*, *`ModPerl::Registry`* and *`ModPerl::RegistryBB`*.

- 24. ModPerl::RegistryLoader - Compile ModPerl::RegistryCooker scripts at server startup
  This modules allows compilation of scripts, running under packages derived from `ModPerl::RegistryCooker`, at server startup. The script's handler routine is compiled by the parent server, of which children get a copy and thus saves some memory by initially sharing the

compiled copy with the parent and saving the overhead of script's compilation on the first request in every httpd instance.

- 25. ModPerl::Util -- Helper mod_perl 2.0 Functions
  `ModPerl::Util` provides mod_perl 2.0 util functions.

Part IV: Helper Modules / Applications

- 26. Apache::Reload - Reload Perl Modules when Changed on Disk
  `Apache::Reload` reloads modules that change on the disk.

- 27. Apache::Status - Embedded interpreter status information
  The **Apache::Status** module provides some information about the status of the Perl interpreter embedded in the server.

Part V: Internal Modules

- 28. ModPerl::BuildMM -- A "subclass" of ModPerl::MM used for building mod_perl 2.0
  `ModPerl::BuildMM` is a "subclass" of `ModPerl::MM` used for building mod_perl 2.0. Refer to *`ModPerl::MM`* manpage.

See www.perldoc.com for documentation of the rest of the `Apache::` modules.

1 Apache::Access -- A Perl API for Apache request object

1.1 SYNOPSIS

```
use Apache::Access;
sub handler{
    my $r = shift;
    ...
    my $auth_type = $r->auth_type;
    ...
}
```

1.2 DESCRIPTION

`Apache::Access` provides the Perl API for Apache request object.

1.3 API

Function arguments (if any) and return values are shown in the function's synopsis.

- **auth_type()**

  `$r->auth_type` gets or sets the value of the *AuthType* configuration directive for the current request.

  ```
  my $auth_type = $r->auth_type;
  ```

  or

  ```
  $r->auth_type('Basic');
  ```

-

2 Apache::compat -- 1.0 backward compatibility functions deprecated in 2.0

2 Apache::compat -- 1.0 backward compatibility functions deprecated in 2.0

2.1 SYNOPSIS

```
# either add at the very beginning of startup.pl
use Apache2
use Apache::compat;
# or httpd.conf
PerlModule Apache2
PerlModule Apache::compat
```

2.2 DESCRIPTION

`Apache::compat` provides mod_perl 1.0 compatibility layer and can be used to smooth the transition process to mod_perl 2.0.

It includes functions that have changed their API or were removed in mod_perl 2.0. If your code uses any of those functions, you should load this module at the server startup, and everything should work as it did in 1.0. If it doesn't please report the bug, but before you do that please make sure that your code does work properly under mod_perl 1.0.

However, remember, that it's implemented in pure Perl and not C, therefore its functionality is not optimized and it's the best to try to port your code not to use deprecated functions and stop using the compatibility layer.

2.3 Use in CPAN Modules

The short answer: **Do not use** `Apache::compat` in CPAN modules.

The long answer:

`Apache::compat` is useful during the mod_perl 1.0 code porting. Though remember that it's implemented in pure Perl. In certain cases it overrides mod_perl 2.0 methods, because their API is very different and doesn't map 1:1 to mod_perl 1.0. So if anything, not under user's control, loads `Apache::compat` user's code is forced to use the potentially slower method. Which is quite bad.

Some users may choose to keep using `Apache::compat` in production and it may perform just fine. Other users will choose not to use that module, by porting their code to use mod_perl 2.0 API. However it should be users' choice whether to load this module or not and not to be enforced by CPAN modules.

If you port your CPAN modules to work with mod_perl 2.0, you should follow the porting Perl and XS module guidelines.

Users that are stuck with CPAN modules preloading `Apache::compat`, can prevent this from happening by adding

```
$INC{'Apache/compat.pm'} = __FILE__;
```

at the very beginning of their *startup.pl*. But this will most certainly break the module that needed this module.

2.4 API

You should be reading the mod_perl 1.0 API docs for usage of the methods and functions in this package, since what this module is doing is providing a backwards compatibility and it makes no sense to duplicate documentation.

Another important document to read is: Migrating from mod_perl 1.0 to mod_perl 2.0 which covers all mod_perl 1.0 constants, functions and methods that have changed in mod_perl 2.0.

3 Apache::Const - Perl Interface for Apache Constants

3.1 SYNOPSIS

3.2 CONSTANTS

3.2.1 :cmd_how

```
use Apache::Const -compile => qw(:cmd_how);
```

The :cmd_how group is for XXX constants.

3.2.1.1 Apache::FLAG

3.2.1.2 Apache::ITERATE

3.2.1.3 Apache::ITERATE2

3.2.1.4 Apache::NO_ARGS

3.2.1.5 Apache::RAW_ARGS

3.2.1.6 Apache::TAKE1

3.2.1.7 Apache::TAKE12

3.2.1.8 Apache::TAKE123

3.2.1.9 Apache::TAKE13

3.2.1.10 Apache::TAKE2

3.2.1.11 Apache::TAKE23

3.2.1.12 Apache::TAKE3

3.2.2 :common

```
use Apache::Const -compile => qw(:common);
```

The :common group is for XXX constants.

3.2.2.1 Apache::AUTH_REQUIRED

3.2.2.2 Apache::DECLINED

3.2.2.3 Apache::DONE

3.2.2.4 Apache::FORBIDDEN

3.2.2.5 Apache::NOT_FOUND

3.2.2.6 Apache::OK

3.2.2.7 Apache::REDIRECT

3.2.2.8 Apache::SERVER_ERROR

3.2.3 :config

```
use Apache::Const -compile => qw(:config);
```

The `:config` group is for XXX constants.

3.2.3.1 Apache::DECLINE_CMD

3.2.4 :filter_type

```
use Apache::Const -compile => qw(:filter_type);
```

The `:filter_type` group is for XXX constants.

3.2.4.1 Apache::FTYPE_CONNECTION

3.2.4.2 Apache::FTYPE_CONTENT_SET

3.2.4.3 Apache::FTYPE_NETWORK

3.2.4.4 Apache::FTYPE_PROTOCOL

3.2.4.5 Apache::FTYPE_RESOURCE

3.2.4.6 Apache::FTYPE_TRANSCODE

3.2.5 :http

```
use Apache::Const -compile => qw(:http);
```

The `:http` group is for XXX constants.

3.2.5.1 Apache::HTTP_ACCEPTED

3.2.5.2 Apache::HTTP_BAD_GATEWAY

3.2.5.3 Apache::HTTP_BAD_REQUEST

3.2.5.4 Apache::HTTP_CONFLICT

3.2.5.5 Apache::HTTP_CONTINUE

3.2.5.6 Apache::HTTP_CREATED

3.2.5.7 Apache::HTTP_EXPECTATION_FAILED

3.2.5.8 Apache::HTTP_FAILED_DEPENDENCY

3.2.5.9 Apache::HTTP_FORBIDDEN

3.2.5.10 Apache::HTTP_GATEWAY_TIME_OUT

3.2.5.11 Apache::HTTP_GONE

3.2.5.12 Apache::HTTP_INSUFFICIENT_STORAGE

3.2.5.13 Apache::HTTP_INTERNAL_SERVER_ERROR

3.2.5.14 Apache::HTTP_LENGTH_REQUIRED

3.2.5.15 Apache::HTTP_LOCKED

3.2.5.16 Apache::HTTP_METHOD_NOT_ALLOWED

3.2.5.17 Apache::HTTP_MOVED_PERMANENTLY

3.2.5.18 Apache::HTTP_MOVED_TEMPORARILY

3.2.5.19 Apache::HTTP_MULTIPLE_CHOICES

3.2.5.20 Apache::HTTP_MULTI_STATUS

3.2.5.21 Apache::HTTP_NON_AUTHORITATIVE

3.2.5.22 Apache::HTTP_NOT_ACCEPTABLE

3.2.5.23 Apache::HTTP_NOT_EXTENDED

3.2.5.24 Apache::HTTP_NOT_FOUND

3.2.5.25 Apache::HTTP_NOT_IMPLEMENTED

3.2.5.26 Apache::HTTP_NOT_MODIFIED

3.2.5.27 Apache::HTTP_NO_CONTENT

3.2.5.28 Apache::HTTP_OK

3.2.5.29 Apache::HTTP_PARTIAL_CONTENT

3.2.5.30 Apache::HTTP_PAYMENT_REQUIRED

3.2.5.31 Apache::HTTP_PRECONDITION_FAILED

3.2.5.32 Apache::HTTP_PROCESSING

3.2.5.33 Apache::HTTP_PROXY_AUTHENTICATION_REQUIRED

3.2.5.34 Apache::HTTP_RANGE_NOT_SATISFIABLE

3.2.5.35 Apache::HTTP_REQUEST_ENTITY_TOO_LARGE

3.2.5.36 Apache::HTTP_REQUEST_TIME_OUT

3.2.5.37 Apache::HTTP_REQUEST_URI_TOO_LARGE

3.2.5.38 Apache::HTTP_RESET_CONTENT

3.2.5.39 Apache::HTTP_SEE_OTHER

3.2.5.40 Apache::HTTP_SERVICE_UNAVAILABLE

3.2.5.41 Apache::HTTP_SWITCHING_PROTOCOLS

3.2.5.42 Apache::HTTP_TEMPORARY_REDIRECT

3.2.5.43 Apache::HTTP_UNAUTHORIZED

3.2.5.44 Apache::HTTP_UNPROCESSABLE_ENTITY

3.2.5.45 Apache::HTTP_UNSUPPORTED_MEDIA_TYPE

3.2.5.46 Apache::HTTP_USE_PROXY

3.2.5.47 Apache::HTTP_VARIANT_ALSO_VARIES

3.2.6 :input_mode

```
use Apache::Const -compile => qw(:input_mode);
```

The `:input_mode` group is for XXX constants.

3.2.6.1 Apache::MODE_EATCRLF

3.2.6.2 Apache::MODE_EXHAUSTIVE

3.2.6.3 Apache::MODE_GETLINE

3.2.6.4 Apache::MODE_INIT

3.2.6.5 Apache::MODE_READBYTES

3.2.6.6 Apache::MODE_SPECULATIVE

3.2.7 :log

```
use Apache::Const -compile => qw(:log);
```

The `:log` group is for XXX constants.

3.2.7.1 Apache::LOG_ALERT

3.2.7.2 Apache::LOG_CRIT

3.2.7.3 Apache::LOG_DEBUG

3.2.7.4 Apache::LOG_EMERG

3.2.7.5 Apache::LOG_ERR

3.2.7.6 Apache::LOG_INFO

3.2.7.7 Apache::LOG_LEVELMASK

3.2.7.8 Apache::LOG_NOTICE

3.2.7.9 Apache::LOG_STARTUP

3.2.7.10 Apache::LOG_TOCLIENT

3.2.7.11 Apache::LOG_WARNING

3.2.8 :methods

```
use Apache::Const -compile => qw(:methods);
```

The `:methods` group is for XXX constants.

3.2.8.1 Apache::METHODS

3.2.8.2 Apache::M_BASELINE_CONTROL

3.2.8.3 Apache::M_CHECKIN

3.2.8.4 Apache::M_CHECKOUT

3.2.8.5 Apache::M_CONNECT

3.2.8.6 Apache::M_COPY

3.2.8.7 Apache::M_DELETE

3.2.8.8 Apache::M_GET

3.2.8.9 Apache::M_INVALID

3.2.8.10 Apache::M_LABEL

3.2.8.11 Apache::M_LOCK

3.2.8.12 Apache::M_MERGE

3.2.8.13 Apache::M_MKACTIVITY

3.2.8.14 Apache::M_MKCOL

3.2.8.15 Apache::M_MKWORKSPACE

3.2.8.16 Apache::M_MOVE

3.2.8.17 Apache::M_OPTIONS

3.2.8.18 Apache::M_PATCH

3.2.8.19 Apache::M_POST

3.2.8.20 Apache::M_PROPFIND

3.2.8.21 Apache::M_PROPPATCH

3.2.8.22 Apache::M_PUT

3.2.8.23 Apache::M_REPORT

3.2.8.24 Apache::M_TRACE

3.2.8.25 Apache::M_UNCHECKOUT

3.2.8.26 Apache::M_UNLOCK

3.2.8.27 Apache::M_UPDATE

3.2.8.28 Apache::M_VERSION_CONTROL

3.2.9 :mpmq

```
use Apache::Const -compile => qw(:mpmq);
```

The `:mpmq` group is for querying MPM properties.

3.2.9.1 Apache::MPMQ_NOT_SUPPORTED

3.2.9.2 Apache::MPMQ_STATIC

3.2.9.3 Apache::MPMQ_DYNAMIC

3.2.9.4 Apache::MPMQ_MAX_DAEMON_USED

3.2.9.5 Apache::MPMQ_IS_THREADED

3.2.9.6 Apache::MPMQ_IS_FORKED

3.2.9.7 Apache::MPMQ_HARD_LIMIT_DAEMONS

3.2.9.8 Apache::MPMQ_HARD_LIMIT_THREADS

3.2.9.9 Apache::MPMQ_MAX_THREADS

3.2.9.10 Apache::MPMQ_MIN_SPARE_DAEMONS

3.2.9.11 Apache::MPMQ_MIN_SPARE_THREADS

3.2.9.12 Apache::MPMQ_MAX_SPARE_DAEMONS

3.2.9.13 Apache::MPMQ_MAX_SPARE_THREADS

3.2.9.14 Apache::MPMQ_MAX_REQUESTS_DAEMON

3.2.9.15 Apache::MPMQ_MAX_DAEMONS

3.2.10 :options

```
use Apache::Const -compile => qw(:options);
```

The `:options` group is for XXX constants.

3.2.10.1 Apache::OPT_ALL

3.2.10.2 Apache::OPT_EXECCGI

3.2.10.3 Apache::OPT_INCLUDES

3.2.10.4 Apache::OPT_INCNOEXEC

3.2.10.5 Apache::OPT_INDEXES

3.2.10.6 Apache::OPT_MULTI

3.2.10.7 Apache::OPT_NONE

3.2.10.8 Apache::OPT_SYM_LINKS

3.2.10.9 Apache::OPT_SYM_OWNER

3.2.10.10 Apache::OPT_UNSET

3.2.11 :override

```
use Apache::Const -compile => qw(:override);
```

The `:override` group is for XXX constants.

3.2.11.1 Apache::ACCESS_CONF

3.2.11.2 Apache::OR_ALL

3.2.11.3 Apache::OR_AUTHCFG

3.2.11.4 Apache::OR_FILEINFO

3.2.11.5 Apache::OR_INDEXES

3.2.11.6 Apache::OR_LIMIT

3.2.11.7 Apache::OR_NONE

3.2.11.8 Apache::OR_OPTIONS

3.2.11.9 Apache::OR_UNSET

3.2.11.10 Apache::RSRC_CONF

3.2.12 :platform

```
use Apache::Const -compile => qw(:platform);
```

The `:platform` group is for constants that may differ from OS to OS.

3.2.12.1 Apache::CRLF

3.2.12.2 Apache::CR

3.2.12.3 Apache::LF

3.2.13 :remotehost

```
use Apache::Const -compile => qw(:remotehost);
```

The `:remotehost` group is for XXX constants.

3.2.13.1 Apache::REMOTE_DOUBLE_REV

3.2.13.2 Apache::REMOTE_HOST

3.2.13.3 Apache::REMOTE_NAME

3.2.13.4 Apache::REMOTE_NOLOOKUP

3.2.14 :satisfy

```
use Apache::Const -compile => qw(:satisfy);
```

The `:satisfy` group is for XXX constants.

3.2.14.1 Apache::SATISFY_ALL

3.2.14.2 Apache::SATISFY_ANY

3.2.14.3 Apache::SATISFY_NOSPEC

3.2.15 :types

```
use Apache::Const -compile => qw(:types);
```

The `:types` group is for XXX constants.

3.2.15.1 Apache::DIR_MAGIC_TYPE

4 Apache::Directive -- A Perl API for manipulating Apache configuration tree

4 Apache::Directive -- A Perl API for manipulating Apache configuration tree

4.1 Synopsis

```
use Apache::Directive;

my $tree = Apache::Directive->conftree;

my $documentroot = $tree->lookup('DocumentRoot');

my $vhost = $tree->lookup('VirtualHost', 'localhost:8000');
my $servername = $vhost->{'ServerName'};

print $tree->as_string;

use Data::Dumper;
print Dumper($tree->as_hash);

my $node = $tree;
while ($node) {

    #do something with $node

    if (my $kid = $node->first_child) {
        $node = $kid;
    }
    elsif (my $next = $node->next) {
        $node = $next;
    }
    else {
        if (my $parent = $node->parent) {
            $node = $parent->next;
        }
        else {
            $node = undef;
        }
    }
}
```

4.2 Description

`Apache::Directive` allows its users to search and navigate the internal Apache configuration.

Internally, this information is stored in a tree structure. Each node in the tree has a reference to its parent (if it's not the root), its first child (if any), and to its next sibling.

4.3 Class Methods

Function arguments (if any) and return values are shown in the function's synopsis.

4.3.1 conftree()

```
$tree = Apache::Directive->conftree();
```

Returns the root of the configuration tree.

4.4 Object Methods

Function arguments (if any) and return values are shown in the function's synopsis.

4.4.1 next()

```
$node = $node->next;
```

Returns the next sibling of `$node`, `undef` otherwise

4.4.2 first_child()

```
$subtree = $node->first_child;
```

Returns the first child node of `$node`, undef otherwise

4.4.3 parent()

```
$parent = $node->parent;
```

Returns the parent of `$node`, undef if this node is the root node

4.4.4 directive()

```
$name = $node->directive;
```

Returns the name of the directive in `$node`.

4.4.5 args()

```
$args = $node->args;
```

Returns the arguments to this `$node`

4.4.6 filename()

```
$fname = $node->filename;
```

Returns the filename this `$node` was created from

4.4.7 line_number()

```
$lineno = $node->line_number;
```

Returns the line number in `filename` this `$node` was created from

4.4.8 as_string()

```
print $tree->as_string();
```

Returns a string representation of the configuration tree, in httpd.conf format.

4.4.9 as_hash()

```
$config = $tree->as_hash();
```

Returns a hash representation of the configuration tree, in a format suitable for inclusion in the <Perl> sections.

4.4.10 lookup()

```
lookup($directive, [$args])
```

Returns node(s) matching a certain value. In list context, it will return all matching nodes. In scalar context, it will return only the first matching node.

If called with only one `$directive` value, this will return all nodes from that directive:

```
@Alias = $tree->lookup('Alias');
```

Would return all nodes for Alias directives.

If called with an extra `$args` argument, this will return only nodes where both the directive and the args matched:

```
$VHost = $tree->lookup('VirtualHosts', '_default_:8000');
```

4.5 Authors

4.6 Copyright

5 Apache::Filter -- A Perl API for Apache 2.0 Filtering

5.1 Synopsis

```
use Apache::Filter;
```

5.2 Description

`Apache::Filter` provides the Perl API for Apache 2.0 filtering framework

5.3 Common Filter API

The following methods can be called from any filter handler:

5.3.1 `c`

Inside a connection or a request filter the current connection object can be retrieved with:

```
my $c = $f->c;
```

5.3.2 `ctx`

A filter context is created before the filter is called for the first time and it's destroyed at the end of the request. The context is preserved between filter invocations of the same request. So if a filter needs to store some data between invocations it should use the filter context for that. The filter context is initialized with the `undef` value.

The `ctx` method accepts a single SCALAR argument. Therefore if you want to store any other perl datastructure you should use a reference to it.

For example you can store a hash reference:

```
$f->ctx({ foo => 'bar' });
```

and then access it:

```
$foo = $f->ctx->{foo};
```

if you access the context more than once it's more efficient to copy it's value before using it:

```
my $ctx = $f->ctx;
$foo = $ctx->{foo};
```

to avoid redundant method calls. As of this writing `$ctx` is not a tied variable, so if you modify it need to store it at the end:

```
$f->ctx($ctx);
```

META: later we might make it a TIEd-variable interface, so it'll be stored automatically.

This method is useful when it's acting as a flag which ensures that something happens only once. For example:

```
unless ($f->ctx) {
    do_something_once();
    $f->ctx(1);
}
```

5.3.3 frec

```
my $fr = $f->frec([$frec]);
```

Get/set the `Apache::FilterRec` (filter record) object.

5.3.4 next

```
$f->next;
```

Returns the `Apache::Filter` object of the next filter in chain.

Since Apache inserts several core filters at the end of each chain, normally this method always returns an object. However if it's not a mod_perl filter handler, you can call only the following methods on it: `get_brigade`, `pass_brigade`, `c`, `r`, `frec` and `next`. If you call other methods the behavior is undefined.

META: I doubt anybody will ever need to mess with other filters, from within a mod_perl filter. but if the need arises it's easy to tell a mod_perl filter from non-mod_perl one by calling `$f->frec->name` (it'll return one of the following four names: *modperl_request_output*, *modperl_request_input*, *modperl_connection_output* or *modperl_connection_input*).

5.3.5 r

Inside an HTTP request filter the current request object can be retrieved with:

```
my $r = $f->r;
```

5.3.6 remove

```
$f->remove;
```

Remove the current filter from the filter chain (for the current request).

Notice that you should either complete the current filter invocation normally (by calling `get_brigade` or `pass_brigade` depending on the filter kind) or if nothing was done, return `Apache::DECLINED` and mod_perl will take care of passing the current bucket brigade through unmodified to the next filter in chain.

note: calling remove() on the very top connection filter doesn't affect the filter chain due to a bug in Apache 2.0.46 and lower (may be fixed in 2.0.47). So don't use it with connection filters, till it gets fixed in Apache and then make sure to require the minimum Apache version if you rely on it.

5.4 Bucket Brigade Filter API

The following methods can be called from any filter, directly manipulating bucket brigades:

5.4.1 `fflush`

```
$f->fflush($bb);
```

Flush the `$bb` brigade down the filter stack.

5.4.2 `get_brigade`

```
sub filter {
    my($f, $bb, $mode, $block, $readbytes) = @_;

    my $rv = $f->next->get_brigade($bb, $mode, $block, $readbytes);
    return $rv unless $rv == APR::SUCCESS;

    # ... process $bb

    return Apache::OK;
}
```

This is a method to use in bucket brigade input filters. It acquires a bucket brigade from the upstream input filter.

Normally arguments `$mode`, `$block`, `$readbytes` are the same as passed to the filter itself.

It returns `APR::SUCCESS` on success, otherwise a failure code, in which case it should be returned to the caller.

5.4.3 `pass_brigade`

```
sub filter {
    my($f, $bb) = @_;

    # ... process $bb

    my $rv = $f->next->pass_brigade($bb);
    return $rv unless $rv == APR::SUCCESS;

    # process $bb
    return Apache::OK;
}
```

This is a method to use in bucket brigade output filters. It passes the current bucket brigade to the downstream output filter.

It returns `APR::SUCCESS` on success, otherwise a failure code, in which case it should be returned to the caller.

5.5 Streaming Filter API

The following methods can be called from any filter, which uses the simplified streaming functionality:

5.5.1 seen_eos

```
$f->seen_eos;
```

This methods returns a true value when the EOS bucket is seen by the `read` method. This only works in streaming filters which exhaustively `$f->read` all the incoming data in a while loop, like so:

```
while ($f->read(my $buffer, $read_len)) {
    # do something with $buffer
}
if ($f->seen_eos) {
    # do something
}
```

This method is useful when a streaming filter wants to append something to the very end of data, or do something at the end of the last filter invocation. After the EOS bucket is read, the filter should expect not to be invoked again.

If an input streaming filter doesn't consume all data in the bucket brigade (or even in several bucket brigades), it has to generate the EOS event by itself. So when the filter is done it has to set the EOS flag:

```
$f->seen_eos(1);
```

when the filter handler returns, internally mod_perl will take care of creating and sending the EOS bucket to the upstream input filter.

A similar logic may apply for output filters.

In most other cases you shouldn't set this flag. When this flag is prematurely set (before the real EOS bucket has arrived) in the current filter invocation, instead of invoking the filter again, mod_perl will create and send the EOS bucket to the next filter, ignoring any other bucket brigades that may have left to consume. As mentioned earlier this special behavior is useful in writing special tests that test abnormal situations.

5.5.2 read

```
$f->read(my $buffer, $read_len);
```

Reads at most `$read_len` characters into `$buffer`. It returns a true value as long as it had something to read, and a false value otherwise.

This is a streaming filter method, which acquires the bucket brigade behind the scenes and reads data from all buckets. If the EOS bucket is read, the `seen_eos` method will return a true value.

5.5.3 print

```
$f->print($buffer);
```

Sends the contents of `$buffer` to the next filter in chain (via internal buffer).

This method should be used only in streaming filters.

5.6 Other Filter-related API

Other methods which affect filters, but called on non-`Apache::Filter` objects:

5.6.1 add_input_filter

```
$r->add_input_filter(\&callback);
```

Adds `&callback` filter handler to input request filter chain.

```
$c->add_input_filter(\&callback);
```

Adds `&callback` filter handler to input connection filter chain.

5.6.2 add_output_filter

```
$r->add_output_filter(\&callback);
```

Adds `&callback` filter handler to output request filter chain.

```
$c->add_output_filter(\&callback);
```

Adds `&callback` filter handler to output connection filter chain.

5.7 Filter Handler Attributes

Packages using filter attributes have to subclass `Apache::Filter`:

```
package MyApache::FilterCool;
use base qw(Apache::Filter);
```

Attributes are parsed during the code compilation, by the function `MODIFY_CODE_ATTRIBUTES`, inherited from the `Apache::Filter` package.

5.7.1 `FilterRequestHandler`

The `FilterRequestHandler` attribute tells mod_perl to insert the filter into an HTTP request filter chain.

For example, to configure an output request filter handler, use the `FilterRequestHandler` attribute in the handler subroutine's declaration:

```
package MyApache::FilterOutputReq;
sub handler : FilterRequestHandler { ... }
```

and add the configuration entry:

```
PerlOutputFilterHandler MyApache::FilterOutputReq
```

This is the default mode. So if you are writing an HTTP request filter, you don't have to specify this attribute.

The section HTTP Request vs. Connection Filters delves into more details.

5.7.2 `FilterConnectionHandler`

The `FilterConnectionHandler` attribute tells mod_perl to insert this filter into a connection filter chain.

For example, to configure an output connection filter handler, use the `FilterConnectionHandler` attribute in the handler subroutine's declaration:

```
package MyApache::FilterOutputCon;
sub handler : FilterConnectionHandler { ... }
```

and add the configuration entry:

```
PerlOutputFilterHandler MyApache::FilterOutputCon
```

The section HTTP Request vs. Connection Filters delves into more details.

5.7.3 `FilterInitHandler`

The attribute `FilterInitHandler` marks the function suitable to be used as a filter initialization callback, which is called immediately after a filter is inserted to the filter chain and before it's actually called.

```
sub init : FilterInitHandler {
    my $f = shift;
    #...
    return Apache::OK;
}
```

In order to hook this filter callback, the real filter has to assign this callback using the `Filter-HasInitHandler` which accepts a reference to the callback function.

For further discussion and examples refer to the Filter Initialization Phase tutorial section.

5.7.4 `FilterHasInitHandler`

If a filter wants to run an initialization callback it can register such using the `FilterHasInitHandler` attribute. Similar to `push_handlers` the callback reference is expected, rather than a callback name. The used callback function has to have the `FilterInitHandler` attribute. For example:

```
package MyApache::FilterBar;
use base qw(Apache::Filter);
sub init   : FilterInitHandler { ... }
sub filter : FilterRequestHandler FilterHasInitHandler(\&init) {
    my ($f, $bb) = @_;
    # ...
    return Apache::OK;
}
```

For further discussion and examples refer to the Filter Initialization Phase tutorial section.

5.8 Configuration

mod_perl 2.0 filters configuration is explained in the filter handlers tutorial.

5.8.1 `PerlInputFilterHandler`

See `PerlInputFilterHandler`.

5.8.2 `PerlOutputFilterHandler`

See `PerlOutputFilterHandler`.

5.8.3 `PerlSetInputFilter`

See `PerlSetInputFilter`.

5.8.4 `PerlSetOutputFilter`

See `PerlSetInputFilter`.

5.9 See Also

The filter handlers tutorial and the `Apache::FilterRec` manpage.

6 Apache::FilterRec -- A Perl API for Apache 2.0 Filter Records

6.1 Synopsis

```
use Apache::FilterRec;
# ...
sub filter {
    my $f      = shift;
    my $frec   = $f->frec;
    my $next_f = $frec->next;
    my $name   = $frec->name;
    # ...
}
```

6.2 Description

`Apache::FilterRec` provides the Perl API for Apache 2.0 filter records.

6.3 API

Function arguments (if any) and return values are shown in the function's synopsis.

6.3.1 `name`

META: to be written

6.3.2 `next`

META: to be written

6.4 See Also

The filter handlers tutorial and the `Apache::Filter` manpage.

7 Apache::Log -- Perl API for Apache Logging Methods

7.1 Synopsis

```
#in startup.pl
#-------------
use Apache::Log;

use Apache::Const -compile => qw(OK :log);
use APR::Const    -compile => qw(:error SUCCESS);

my $s = Apache->server;

$s->log_error("server: log_error");
$s->log_serror(__FILE__, __LINE__, Apache::LOG_ERR,
               APR::SUCCESS, "log_serror logging at err level");
$s->log_serror(Apache::LOG_MARK, Apache::LOG_DEBUG,
               APR::ENOTIME, "debug print");
Apache::Server->log_error("routine warning");

Apache->warn("routine warning");
Apache::warn("routine warning");
Apache::Server->warn("routine warning");

#in a handler
#------------
use Apache::Log;

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

use Apache::Const -compile => qw(OK :log);
use APR::Const    -compile => qw(:error SUCCESS);

sub handler{
    my $r = shift;
    $r->log_error("request: log_error");
    $r->warn("whoah!");

    my $rlog = $r->log;
    for my $level qw(emerg alert crit error warn notice info debug) {
        no strict 'refs';
        $rlog->$level($package, "request: $level log level");
    }

    # can use server methods as well
    my $s = $r->server;
    $s->log_error("server: log_error");

    $r->log_rerror(Apache::LOG_MARK, Apache::LOG_DEBUG,
                   APR::ENOTIME, "in debug");

    $s->log_serror(Apache::LOG_MARK, Apache::LOG_INFO,
                   APR::SUCESS, "server info");

    $s->log_serror(Apache::LOG_MARK, Apache::LOG_ERR,
                   APR::ENOTIME, "fatal error");
```

```
    $s->warn('routine server warning');

    return Apache::OK;
}
```

7.2 Description

`Apache::Log` provides the Perl API for Apache logging methods.

Depending on the the current `LogLevel` setting, only logging with the same log level or higher will be loaded. For example if the current `LogLevel` is set to *warning*, only messages with log level of the level *warning* or higher (*err*, *crit*, *elert* and *emerg*) will be logged. Therefore this:

```
$r->log_rerror(Apache::LOG_MARK, Apache::LOG_WARNING,
               APR::ENOTIME, "warning!");
```

will log the message, but this one won't:

```
$r->log_rerror(Apache::LOG_MARK, Apache::LOG_INFO,
               APR::ENOTIME, "just an info");
```

It will be logged only if the server log level is set to *info* or *debug*. `LogLevel` is set in the configuration file, but can be changed using the *`$s->loglevel()`* method.

The filename and the line number of the caller are logged only if `Apache::LOG_DEBUG` is used (because that's how Apache 2.0 logging mechanism works).

7.3 Constants

Log level constants can be compiled all at once:

```
use Apache::Const -compile => qw(:log);
```

or individually:

```
use Apache::Const -compile => qw(LOG_DEBUG LOG_INFO);
```

7.3.1 LogLevel Constants

The following constants (sorted from the most severe level to the least severe) are used in logging methods to specify the log level at which the message should be logged:

7.3.1.1 `Apache::LOG_EMERG`

7.3.1.2 Apache::LOG_ALERT

7.3.1.3 Apache::LOG_CRIT

7.3.1.4 Apache::LOG_ERR

7.3.1.5 Apache::LOG_WARNING

7.3.1.6 Apache::LOG_NOTICE

7.3.1.7 Apache::LOG_INFO

7.3.1.8 Apache::LOG_DEBUG

Make sure to compile the APR status constants before using them. For example to compile APR::SUCESS and all the APR error status constants do:

```
use APR::Const    -compile => qw(:error SUCCESS);
```

7.3.2 Other Constants

7.3.2.1 Apache::LOG_LEVELMASK

used to mask off the level value, to make sure that the log level's value is within the proper bits range. e.g.:

```
$loglevel &= LOG_LEVELMASK;
```

7.3.2.2 Apache::LOG_TOCLIENT

used to give content handlers the option of including the error text in the ErrorDocument sent back to the client. When Apache::LOG_TOCLIENT is passed to log_rerror() the error message will be saved in the $r's notes table, keyed to the string *"error-notes"*, if and only if the severity level of the message is Apache::LOG_WARNING or greater and there are no other *"error-notes"* entry already set in the request record's notes table. Once the *"error-notes"* entry is set, it is up to the error handler to determine whether this text should be sent back to the client. For example:

```
$r->log_rerror(Apache::LOG_MARK, Apache::LOG_ERR|Apache::LOG_TOCLIENT,
               APR::ENOTIME, "request log_rerror");
```

now the log message can be retrieved via:

```
$r->notes->get("error-notes");
```

Remember that client-generated text streams sent back to the client MUST be escaped to prevent CSS attacks.

7.3.2.3 Apache::LOG_STARTUP

is useful for startup message where no timestamps, logging level is wanted. For example:

```
$s->log_serror(Apache::LOG_MARK, Apache::LOG_INFO,
               APR::SUCCESS, "This log message comes with a header");
```

Will print:

```
[Wed May 14 16:47:09 2003] [info] This log message comes with a header
```

whereas, when Apache::LOG_STARTUP is binary ORed as in:

```
$s->log_serror(Apache::LOG_MARK, Apache::LOG_INFO|Apache::LOG_STARTUP,
               APR::SUCCESS, "This log message comes with no header");
```

then the logging will be:

```
This log message comes with no header
```

7.4 Server Logging Methods

7.4.1 $s->log_error()

```
$s->log_error(@message);
```

just logs the supplied message. For example:

```
$s->log_error("running low on memory");
```

7.4.2 $s->log_serror()

```
log_serror($file, $line, $level, $status, @message);
```

where:

```
 * $file    The file in which this function is called
 * $line    The line number on which this function is called
 * $level   The level of this error message
 * $status  The status code from the previous command
 * @message The log message
```

This function provides a fine control of when the message is logged, gives an access to built-in status codes.

For example:

```
$s->log_serror(Apache::LOG_MARK, Apache::LOG_ERR,
               APR::SUCCESS, "log_serror logging at err level");

$s->log_serror(Apache::LOG_MARK, Apache::LOG_DEBUG,
               APR::ENOTIME, "debug print");
```

7.4.3 $s->log()

```
my $slog = $s->log;
```

returns a handle which can be used to log messages of different level. See the next entry.

7.4.4 emerg(), alert(), crit(), error(), warn(), notice(), info(), debug()

```
$s->log->emerg(@message);
```

after getting the log handle with `$s->log`, use these methods to control when messages should be logged.

For example:

```
my $slog = $s->log;
$slog->debug("just ", "some debug info");
$slog->warn(@warnings);
$slog->crit("dying");
```

7.5 Request Logging Methods

7.5.1 $r->log_error()

```
$r->log_error(@message);
```

logs the supplied message (similar to `$s->log_error`). For example:

```
$r->log_error("the request is about to end");
```

the same as `$s->log_error`.

7.5.2 $r->log_rerror()

```
log_rerror($file, $line, $level, $status, @message);
```

same as `$s->log_rerror`. For example:

```
$s->log_rerror(Apache::LOG_MARK, Apache::LOG_ERR,
               APR::SUCCESS, "log_rerror logging at err level");

$s->log_rerror(Apache::LOG_MARK, Apache::LOG_DEBUG,
               APR::ENOTIME, "debug print");
```

7.5.3 $r->log()

```
my $rlog = $r->log;
```

Similar to `$s->log()`

7.5.4 the emerg(), alert(), crit(), error(), warn(), notice(), info(), debug() methods

Similar to the server's log functions with the same names.

For example:

```
$rlog->debug("just ", "some debug info");
$rlog->warn(@req_warnings);
$rlog->crit("dying");
```

7.6 General Functions

7.6.1 Apache::LOG_MARK()

```
my($file, $line) = Apache::LOG_MARK();
```

Though looking like a constant, this is a function, which returns a list of two items: `(__FILE__, __LINE__)`, i.e. the file and the line where the function was called from. It's mostly useful to be passed as the first argument to those logging methods, expecting the filename and the line number as the first arguments.

7.7 Aliases

7.7.1 $s->warn()

```
$s->warn(@warnings);
```

is the same as:

```
$s->log_error(Apache::LOG_MARK, Apache::LOG_WARNING,
              APR::SUCCESS, @warnings)
```

For example:

```
$s->warn('routine server warning');
```

7.7.2 Apache->warn()

7.7.3 Apache::warn()

```
Apache->warn(@warnings);
```

8 Apache::PerlSections - Default Handler for Perl sections

8.1 Synopsis

```
<Perl >
@PerlModule = qw(Mail::Send Devel::Peek);

#run the server as whoever starts it
$User  = getpwuid(>) || >;
$Group = getgrgid()) || );

$ServerAdmin = $User;

</Perl>
```

8.2 Description

With `<Perl >...</Perl>` sections, it is possible to configure your server entirely in Perl.

`<Perl >` sections can contain *any* and as much Perl code as you wish. These sections are compiled into a special package whose symbol table mod_perl can then walk and grind the names and values of Perl variables/structures through the Apache core configuration gears.

Block sections such as `<Location>..</Location>` are represented in a `%Location` hash, e.g.:

```
<Perl>

$Location{"/~dougm/"} = {
  AuthUserFile   => '/tmp/htpasswd',
  AuthType       => 'Basic',
  AuthName       => 'test',
  DirectoryIndex => [qw(index.html index.htm)],
  Limit          => {
      METHODS => 'GET POST',
      require => 'user dougm',
  },
};

</Perl>
```

If an Apache directive can take two or three arguments you may push strings (the lowest number of arguments will be shifted off the `@list`) or use an array reference to handle any number greater than the minimum for that directive:

```
push @Redirect, "/foo", "http://www.foo.com/";

push @Redirect, "/imdb", "http://www.imdb.com/";

push @Redirect, [qw(temp "/here" "http://www.there.com")];
```

Other section counterparts include `%VirtualHost`, `%Directory` and `%Files`.

To pass all environment variables to the children with a single configuration directive, rather than listing each one via `PassEnv` or `PerlPassEnv`, a `<Perl >` section could read in a file and:

```
push @PerlPassEnv, [$key => $val];
```

or

```
Apache->httpd_conf("PerlPassEnv $key $val");
```

These are somewhat simple examples, but they should give you the basic idea. You can mix in any Perl code you desire. See *eg/httpd.conf.pl* and *eg/perl_sections.txt* in the mod_perl distribution for more examples.

Assume that you have a cluster of machines with similar configurations and only small distinctions between them: ideally you would want to maintain a single configuration file, but because the configurations aren't *exactly* the same (e.g. the `ServerName` directive) it's not quite that simple.

`<Perl >` sections come to rescue. Now you have a single configuration file and the full power of Perl to tweak the local configuration. For example to solve the problem of the `ServerName` directive you might have this `<Perl >` section:

```
<Perl >
$ServerName = 'hostname';
</Perl>
```

For example if you want to allow personal directories on all machines except the ones whose names start with *secure*:

```
<Perl >
$ServerName = 'hostname';
if ($ServerName !~ /^secure/) {
    $UserDir = "public.html";
}
else {
    $UserDir = "DISABLED";
}
</Perl>
```

8.3 Configuration Variables

There are a few variables that can be set to change the default behaviour of `<Perl >` sections.

8.3.1 $Apache::Server::SaveConfig

By default, the namespace in which `<Perl >` sections are evaluated is cleared after each block closes. By setting it to a true value, the content of those namespaces will be preserved and will be available for inspection by modules like `Apache::Status`.

8.3.2 $Apache::Server::StrictPerlSections

By default, compilation and run-time errors within <Perl > sections will cause a warning to be printed in the error_log. By setting this variable to a true value, code in the sections will be evaluated as if "use strict" was in usage, and all warning and errors will cause the server to abort startup and report the first error.

8.4 Advanced API

mod_perl 2.0 now introduces the same general concept of handlers to <Perl > sections. Apache::Perl-Sections simply being the default handler for them.

To specify a different handler for a given perl section, an extra handler argument must be given to the section:

```
<Perl handler="My::PerlSection::Handler" somearg="test1">
  $foo = 1;
  $bar = 2;
</Perl>
```

And in My/PerlSection/Handler.pm:

```
sub My::Handler::handler : handler {
    my($self, $parms, $args) = @_;
    #do your thing!
}
```

So, when that given <Perl > block in encountered, the code within will first be evaluated, then the handler routine will be invoked with 3 arguments

$self is self-explanatory

$parms is the *Apache::CmdParms* for this Container, for example, you might want to call $parms->server() to get the current server.

$args is an APR::Table object of the section arguments, the 2 guaranteed ones will be:

```
$args->{'handler'} = 'My::PerlSection::Handler';

$args->{'package'} = 'Apache::ReadConfig';
```

Other name="value" pairs given on the <Perl > line will also be included.

At this point, it's up to the handler routing to inspect the namespace of the $args->{'package'} and chooses what to do.

The most likely thing to do is to feed configuration data back into apache. To do that, use Apache::Server->add_config("directive"), for example:

```
$parms->server->add_config("Alias /foo /bar");
```

Would create a new alias. The source code of `Apache::PerlSections` is a good place to look for a practical example.

8.5 Bugs

8.5.1 <Perl> directive missing closing '>'

httpd-2.0.47 and earlier had a bug in the configuration parser which caused the startup failure with the following error:

```
Starting httpd:
Syntax error on line ... of /etc/httpd/conf/httpd.conf:
<Perl> directive missing closing '>'     [FAILED]
```

This has been fixed in httpd-2.0.48. If you can't upgrade to this or a higher version, please add a space before the closing '>' of the opening tag as a workaround. So if you had:

```
<Perl>
# some code
</Perl>
```

change it to be:

```
<Perl >
# some code
</Perl>
```

9 Apache::porting -- a helper module for mod_perl 1.0 to mod_perl 2.0 porting

9 Apache::porting -- a helper module for mod_perl 1.0 to mod_perl 2.0 porting

9.1 Synopsis

```
# either add at the very beginning of startup.pl
use Apache2;
use Apache::porting;

# or httpd.conf
PerlModule Apache2
PerlModule Apache::porting

# now issue requests and look at the error_log file for hints
```

9.2 Description

`Apache::porting` helps to port mod_perl 1.0 code to run under mod_perl 2.0. It doesn't provide any back-compatibility functionality, however it knows trap calls to methods that are no longer in the mod_perl 2.0 API and tell what should be used instead if at all. If you attempts to use mod_perl 2.0 methods without first loading the modules that contain them, it will tell you which modules you need to load. Finally if your code tries to load modules that no longer exist in mod_perl 2.0 it'll also tell you what are the modules that should be used instead.

`Apache::porting` communicates with users via the *error_log* file. Everytime it traps a problem, it logs the solution (if it finds one) to the error log file. If you use this module coupled with `Apache::Reload` you will be able to port your applications quickly without needing to restart the server on every modification.

It starts to work only when child process start and doesn't work for the code that gets loaded at the server startup. This limitation is explained in the Culprits section.

It relies heavily on `ModPerl::MethodLookup`. which can also be used manually to lookup things.

9.3 Culprits

`Apache::porting` uses the `UNIVERSAL::AUTOLOAD` function to provide its functionality. However it seems to be impossible to create `UNIVERSAL::AUTOLOAD` at the server startup, Apache segfaults on restart. Therefore it performs the setting of `UNIVERSAL::AUTOLOAD` only during the *child_init* phase, when child processes start. As a result it can't help you with things that get preloaded at the server startup.

If you know how to resolve this problem, please let us know. To reproduce the problem try to use an earlier phase, e.g. `PerlPostConfigHandler`:

```
Apache->server->push_handlers(PerlPostConfigHandler => \&porting_autoload);
```

10 Apache::RequestRec -- A Perl API for Apache request object

10.1 SYNOPSIS

```
use Apache::RequestRec;
sub handler{
    my $r = shift;

    my $s = $r->server;
    my $dir_config = $r->dir_config;
    ...
}
```

10.2 DESCRIPTION

`Apache::RequestRec` provides the Perl API for Apache request object.

10.3 API

Function arguments (if any) and return values are shown in the function's synopsis.

10.3.1 server()

```
$s = $r->server;
```

Gets the `Apache::Server` object for the server the request `$r` is running under.

10.3.2 dir_config()

dir_config() provides an interface for the per-directory variable specified by the `PerlSetVar` and `PerlAddVar` directives, and also can be manipulated via the `APR::Table` methods.

The keys are case-insensitive.

```
$apr_table = $r->dir_config();
```

dir_config() called in a scalar context without the `$key` argument returns a *HASH* reference blessed into the `APR::Table` class. This object can be manipulated via the `APR::Table` methods. For available methods see the `APR::Table` manpage.

```
@values = $r->dir_config($key);
```

If the `$key` argument is passed in the list context a list of all matching values will be returned. This method is ineffective for big tables, as it does a linear search of the table. Thefore avoid using this way of calling dir_config() unless you know that there could be more than one value for the wanted key and all the values are wanted.

```
$value = $r->dir_config($key);
```

If the `$key` argument is passed in the scalar context only a single value will be returned. Since the table preserves the insertion order, if there is more than one value for the same key, the oldest value assosiated with the desired key is returned. Calling in the scalar context is also much faster, as it'll stop searching the table as soon as the first match happens.

```
$r->dir_config($key => $val);
```

If the `$key` and the `$val` arguments are used, the set() operation will happen: all existing values associated with the key `$key` (and the key itself) will be deleted and `$value` will be placed instead.

```
$r->dir_config($key => undef);
```

If `$val` is *undef* the unset() operation will happen: all existing values associated with the key `$key` (and the key itself) will be deleted.

10.3.3 ap_auth_type()

`$r->ap_auth_type` gets or sets the *ap_auth_type* slot in the request record.

```
$r->ap_auth_type('Basic');
```

or

```
my $auth_type = $r->ap_auth_type;
```

ap_auth_type holds the authentication type that has been negotiated between the client and server during the actual request. Generally, *ap_auth_type* is populated automatically when you call `$r->get_basic_auth_pw` so you don't really need to worry too much about it, but if you want to roll your own authentication mechanism then you will have to populate *ap_auth_type* yourself.

Note that `$r->ap_auth_type` was `$r->connection->auth_type` in the mod_perl 1.0 API.

10.3.4 main()

```
$main_r = $r->main;
```

If the current request is a sub-request, this method returns a blessed reference to the main request structure. If the current request is the main request, then this method returns undef.

To figure out whether you are inside a main request or a sub-request/internal redirect, use `$r->is_initial_req`.

11 Apache::RequestUtil -- Methods for work with Apache::Request object

11.1 SYNOPSIS

```
use Apache::RequestUtil;
```

11.2 DESCRIPTION

`Apache::RequestUtil` provides the Perl API for Apache request object.

META: complete

11.3 API

Function arguments (if any) and return values are shown in the function's synopsis.

11.4 FUNCTIONS

*11.4.1 * Apache->request()*

11.5 METHODS

11.5.1 new()

11.5.2 get_server_name()

11.5.3 get_server_port()

11.5.4 dir_config()

11.5.5 get_status_line()

11.5.6 is_initial_req()

11.5.7 method_register()

11.5.8 add_config()

11.5.9 location()

11.5.10 location_merge()

11.5.11 no_cache()

11.5.12 pnotes()

11.5.13 set_basic_credentials()

11.5.14 as_string()

11.5.15 push_handlers()

```
$r->push_handlers(PerlResponseHandler => \&handler);
$r->push_handlers(PerlResponseHandler => [\&handler, \&handler2]);

# XXX: not implemented yet
$r->push_handlers(PerlResponseHandler => sub {...});
```

11.5.16 add_handlers()

11.5.17 get_handlers()

11.5.18 slurp_filename()

```
my $ref_content = $r->slurp_filename([$tainted]);
```

Return a reference to contents of `$r->filename`.

By default the returned data is tainted (if run under `-T`). If an optional `$tainted` flag is set to zero, the data will be marked as non-tainted. Do not set this flag to zero unless you know what you are doing, you may create a security hole in your program if you do. For more information see the *perlsec* manpage. If you wonder why this option is available, it is used internally by the `ModPerl::Registry` handler and friends, because the CGI scripts that it reads are considered safe (you could just as well `require()` them).

12 Apache::ServerUtil -- Methods for work with Apache::Server object

12 Apache::ServerUtil -- Methods for work with Apache::Server object

12.1 SYNOPSIS

```
use Apache::ServerUtil;

$s = Apache->server;
my $srv_cfg = $s->dir_config;

# get 'conf/' dir path using $r
my $conf_dir = Apache::server_root_relative('conf', $r->pool);

# get 'log/' dir path using default server startup pool
my $log_dir = Apache::server_root_relative('log');
```

12.2 DESCRIPTION

`Apache::ServerUtil` provides the Perl API for Apache server object.

META: complete

12.3 API

Function arguments (if any) and return values are shown in the function's synopsis.

12.3.1 CONSTANTS

- **server_root**

  returns the value set by the `ServerRoot` directive.

12.3.2 FUNCTIONS

- **server_root_relative()**

  Returns the canonical form of the filename made absolute to `ServerRoot`:

  ```
  Apache::server_root_relative($pool, $fname);
  ```

  `$fname` is appended to the value of `ServerRoot` and return it. e.g.:

  ```
  my $log_dir = Apache::server_root_relative($r->pool, 'logs');
  ```

  If `$fname` is not specified, the value of `ServerRoot` is returned with a trailing `/`. (it's the same as using `''` as `$fname`'s value).

  Also see the `server_root` constant.

12.3.3 METHODS

- **server()**

  The main server's object can be retrieved with:

  ```
  $s = Apache->server;
  ```

  Gets the `Apache::Server` object for the main server.

- **dir_config()**

  dir_config() provides an interface for the per-server variables specified by the `PerlSetVar` and `PerlAddVar` directives, and also can be manipulated via the `APR::Table` methods.

  The keys are case-insensitive.

  ```
  $t = $s->dir_config();
  ```

  dir_config() called in a scalar context without the `$key` argument returns a *HASH* reference blessed into the *APR::Table* class. This object can be manipulated via the *APR::Table* methods. For available methods see *APR::Table*.

  ```
  @values = $s->dir_config($key);
  ```

  If the `$key` argument is passed in the list context a list of all matching values will be returned. This method is ineffective for big tables, as it does a linear search of the table. Thefore avoid using this way of calling dir_config() unless you know that there could be more than one value for the wanted key and all the values are wanted.

  ```
  $value = $s->dir_config($key);
  ```

  If the `$key` argument is passed in the scalar context only a single value will be returned. Since the table preserves the insertion order, if there is more than one value for the same key, the oldest value assosiated with the desired key is returned. Calling in the scalar context is also much faster, as it'll stop searching the table as soon as the first match happens.

  ```
  $s->dir_config($key => $val);
  ```

  If the `$key` and the `$val` arguments are used, the set() operation will happen: all existing values associated with the key `$key` (and the key itself) will be deleted and `$value` will be placed instead.

  ```
  $s->dir_config($key => undef);
  ```

  If `$val` is *undef* the unset() operation will happen: all existing values associated with the key `$key` (and the key itself) will be deleted.

- **push_handlers()**

```
$s->push_handlers(PerlResponseHandler => \&handler);
$s->push_handlers(PerlResponseHandler => [\&handler, \&handler2]);

# XXX: not implemented yet
$s->push_handlers(PerlResponseHandler => sub {...});
```

- **add_handlers()**
- **get_handlers()**

13 Apache::SubProcess -- Executing SubProcesses from mod_perl

13.1 SYNOPSIS

```
use Apache::SubProcess ();

use Config;
use constant PERLIO_IS_ENABLED => $Config{useperlio};

# pass @ARGV / read from the process
$command = "/tmp/argv.pl";
@argv = qw(foo bar);
$out_fh = Apache::SubProcess::spawn_proc_prog($r, $command, \@argv);
$output = read_data($out_fh);

# pass environment / read from the process
$command = "/tmp/env.pl";
$r->subprocess_env->set(foo => "bar");
$out_fh = Apache::SubProcess::spawn_proc_prog($r, $command);
$output = read_data($out_fh);

# write to/read from the process
$command = "/tmp/in_out_err.pl";
($in_fh, $out_fh, $err_fh) =
    Apache::SubProcess::spawn_proc_prog($r, $command);
print $in_fh "hello\n";
$output = read_data($out_fh);
$error  = read_data($err_fh);

# helper function to work w/ and w/o perlio-enabled Perl
sub read_data {
    my($fh) = @_;
    my $data;
    if (PERLIO_IS_ENABLED || IO::Select->new($fh)->can_read(10)) {
        $data = <$fh>;
    }
    return defined $data ? $data : '';
}
```

13.2 DESCRIPTION

`Apache::SubProcess` provides the Perl API for running and communicating with processes spawned from mod_perl handlers.

13.3 API

13.3.1 spawn_proc_prog()

```
$out_fh =
    Apache::SubProcess::spawn_proc_prog($r, $command, [\@argv]);
($in_fh, $out_fh, $err_fh) =
    Apache::SubProcess::spawn_proc_prog($r, $command, [\@argv]);
```

spawn_proc_prog() spawns a sub-process which exec()'s `$command` and returns the output pipe filehandle in the scalar context, or input, output and error pipe filehandles in the list context. Using these three pipes it's possible to communicate with the spawned process.

The third optional argument is a reference to an array which if passed becomes ARGV to the spawned program.

It's possible to pass environment variables as well, by calling:

```
$r->subprocess_env->set($key => $value);
```

before spawning the subprocess.

There is an issue with reading from the read filehandle (`$in_fh`)):

A pipe filehandle returned under perlio-disabled Perl needs to call select() if the other end is not fast enough to send the data, since the read is non-blocking.

A pipe filehandle returned under perlio-enabled Perl on the other hand does the select() internally, because it's really a filehandle opened via `:APR` layer, which internally uses APR to communicate with the pipe. The way APR is implemented Perl's select() cannot be used with it (mainly because select() wants fileno() and APR is a crossplatform implementation which hides the internal datastructure).

Therefore to write a portable code, you want to use select for perlio-disabled Perl and do nothing for perlio-enabled Perl, hence you can use something similar to the read_data() wrapper shown in the SYNOPSIS section.

14 APR - Perl Interface for libapr and libaprutil Libraries

14.1 Synopsis

```
use APR;
```

14.2 Description

Normally you don’t need to use this module. However if you are using an APR:: package outside of mod_perl, you need to load APR first. For example:

```
% perl -MApache2 -MAPR -MAPR::UUID -le 'print APR::UUID->new->format'
```

15 APR::Const - Perl Interface for APR Constants

15.1 SYNOPSIS

15.2 CONSTANTS

15.2.1 :common

```
use APR::Const -compile => qw(:common);
```

The `:common` group is for XXX constants.

15.2.1.1 APR::SUCCESS

15.2.2 :error

```
use APR::Const -compile => qw(:error);
```

The `:error` group is for XXX constants.

15.2.2.1 APR::EABOVEROOT

15.2.2.2 APR::EABSOLUTE

15.2.2.3 APR::EACCES

15.2.2.4 APR::EAGAIN

15.2.2.5 APR::EBADDATE

15.2.2.6 APR::EBADF

15.2.2.7 APR::EBADIP

15.2.2.8 APR::EBADMASK

15.2.2.9 APR::EBADPATH

15.2.2.10 APR::EBUSY

15.2.2.11 APR::ECONNABORTED

15.2.2.12 APR::ECONNREFUSED

15.2.2.13 APR::ECONNRESET

15.2.2.14 APR::EDSOOPEN

15.2.2.15 APR::EEXIST

15.2.2.16 APR::EFTYPE

15.2.2.17 APR::EGENERAL

15.2.2.18 APR::EHOSTUNREACH

15.2.2.19 APR::EINCOMPLETE

15.2.2.20 APR::EINIT

15.2.2.21 APR::EINPROGRESS

15.2.2.22 APR::EINTR

15.2.2.23 APR::EINVAL

15.2.2.24 APR::EINVALSOCK

15.2.2.25 APR::EMFILE

15.2.2.26 APR::EMISMATCH

15.2.2.27 APR::ENAMETOOLONG

15.2.2.28 APR::END

15.2.2.29 APR::ENETUNREACH

15.2.2.30 APR::ENFILE

15.2.2.31 APR::ENODIR

15.2.2.32 APR::ENOENT

15.2.2.33 APR::ENOLOCK

15.2.2.34 APR::ENOMEM

15.2.2.35 APR::ENOPOLL

15.2.2.36 APR::ENOPOOL

15.2.2.37 APR::ENOPROC

15.2.2.38 APR::ENOSHMAVAIL

15.2.2.39 APR::ENOSOCKET

15.2.2.40 APR::ENOSPC

15.2.2.41 APR::ENOSTAT

15.2.2.42 APR::ENOTDIR

15.2.2.43 APR::ENOTEMPTY

15.2.2.44 APR::ENOTHDKEY

15.2.2.45 APR::ENOTHREAD

15.2.2.46 APR::ENOTIME

15.2.2.47 APR::ENOTIMPL

15.2.2.48 APR::ENOTSOCK

15.2.2.49 APR::EOF

15.2.2.50 APR::EPIPE

15.2.2.51 APR::ERELATIVE

15.2.2.52 APR::ESPIPE

15.2.2.53 APR::ETIMEDOUT

15.2.2.54 APR::EXDEV

15.2.3 :filemode

```
use APR::Const -compile => qw(:filemode);
```

The `:filemode` group is for XXX constants.

15.2.3.1 APR::BINARY

15.2.3.2 APR::BUFFERED

15.2.3.3 APR::CREATE

15.2.3.4 APR::DELONCLOSE

15.2.3.5 APR::EXCL

15.2.3.6 APR::PEND

15.2.3.7 APR::READ

15.2.3.8 APR::TRUNCATE

15.2.3.9 APR::WRITE

15.2.4 :filepath

```
use APR::Const -compile => qw(:filepath);
```

The `:filepath` group is for XXX constants.

15.2.4.1 APR::FILEPATH_NATIVE

15.2.4.2 APR::FILEPATH_NOTABOVEROOT

15.2.4.3 APR::FILEPATH_NOTABSOLUTE

15.2.4.4 APR::FILEPATH_NOTRELATIVE

15.2.4.5 APR::FILEPATH_SECUREROOT

15.2.4.6 APR::FILEPATH_SECUREROOTTEST

15.2.4.7 APR::FILEPATH_TRUENAME

15.2.5 :fileperms

```
use APR::Const -compile => qw(:fileperms);
```

The `:fileperms` group is for XXX constants.

15.2.5.1 APR::GEXECUTE

15.2.5.2 APR::GREAD

15.2.5.3 APR::GWRITE

15.2.5.4 APR::UEXECUTE

15.2.5.5 APR::UREAD

15.2.5.6 APR::UWRITE

15.2.5.7 APR::WEXECUTE

15.2.5.8 APR::WREAD

15.2.5.9 APR::WWRITE

15.2.6 :filetype

```
use APR::Const -compile => qw(:filetype);
```

The `:filetype` group is for XXX constants.

15.2.6.1 APR::NOFILE

15.2.6.2 APR::REG

15.2.6.3 APR::DIR

15.2.6.4 APR::CHR

15.2.6.5 APR::BLK

15.2.6.6 APR::PIPE

15.2.6.7 APR::LNK

15.2.6.8 APR::SOCK

15.2.6.9 APR::UNKFILE

15.2.7 :finfo

```
use APR::Const -compile => qw(:finfo);
```

The `:finfo` group is for XXX constants.

15.2.7.1 APR::FINFO_ATIME

15.2.7.2 APR::FINFO_CSIZE

15.2.7.3 APR::FINFO_CTIME

15.2.7.4 APR::FINFO_DEV

15.2.7.5 APR::FINFO_DIRENT

15.2.7.6 APR::FINFO_GPROT

15.2.7.7 APR::FINFO_GROUP

15.2.7.8 APR::FINFO_ICASE

15.2.7.9 APR::FINFO_IDENT

15.2.7.10 APR::FINFO_INODE

15.2.7.11 APR::FINFO_LINK

15.2.7.12 APR::FINFO_MIN

15.2.7.13 APR::FINFO_MTIME

15.2.7.14 APR::FINFO_NAME

15.2.7.15 APR::FINFO_NLINK

15.2.7.16 APR::FINFO_NORM

15.2.7.17 APR::FINFO_OWNER

15.2.7.18 APR::FINFO_PROT

15.2.7.19 APR::FINFO_SIZE

15.2.7.20 APR::FINFO_TYPE

15.2.7.21 APR::FINFO_UPROT

15.2.7.22 APR::FINFO_USER

15.2.7.23 APR::FINFO_WPROT

15.2.8 :flock

```
use APR::Const -compile => qw(:flock);
```

The `:flock` group is for XXX constants.

15.2.8.1 APR::FLOCK_EXCLUSIVE

15.2.8.2 APR::FLOCK_NONBLOCK

15.2.8.3 APR::FLOCK_SHARED

15.2.8.4 APR::FLOCK_TYPEMASK

15.2.9 :hook

```
use APR::Const -compile => qw(:hook);
```

The `:hook` group is for XXX constants.

15.2.9.1 APR::HOOK_FIRST

15.2.9.2 APR::HOOK_LAST

15.2.9.3 APR::HOOK_MIDDLE

15.2.9.4 APR::HOOK_REALLY_FIRST

15.2.9.5 APR::HOOK_REALLY_LAST

15.2.10 :limit

```
use APR::Const -compile => qw(:limit);
```

The `:limit` group is for XXX constants.

15.2.10.1 APR::LIMIT_CPU

15.2.10.2 APR::LIMIT_MEM

15.2.10.3 APR::LIMIT_NOFILE

15.2.10.4 APR::LIMIT_NPROC

15.2.11 :lockmech

```
use APR::Const -compile => qw(:lockmech);
```

The `:lockmech` group is for XXX constants.

15.2.11.1 APR::LOCK_DEFAULT

15.2.11.2 APR::LOCK_FCNTL

15.2.11.3 APR::LOCK_FLOCK

15.2.11.4 APR::LOCK_POSIXSEM

15.2.11.5 APR::LOCK_PROC_PTHREAD

15.2.11.6 APR::LOCK_SYSVSEM

15.2.12 :poll

```
use APR::Const -compile => qw(:poll);
```

The `:poll` group is for XXX constants.

15.2.12.1 APR::POLLERR

15.2.12.2 APR::POLLHUP

15.2.12.3 APR::POLLIN

15.2.12.4 APR::POLLNVAL

15.2.12.5 APR::POLLOUT

15.2.12.6 APR::POLLPRI

15.2.13 :read_type

```
use APR::Const -compile => qw(:read_type);
```

The `:read_type` group is for XXX constants.

15.2.13.1 APR::BLOCK_READ

15.2.13.2 APR::NONBLOCK_READ

15.2.14 :shutdown_how

```
use APR::Const -compile => qw(:shutdown_how);
```

The `:shutdown_how` group is for XXX constants.

15.2.14.1 APR::SHUTDOWN_READ

15.2.14.2 APR::SHUTDOWN_READWRITE

15.2.14.3 APR::SHUTDOWN_WRITE

15.2.15 :socket

```
use APR::Const -compile => qw(:socket);
```

The `:socket` group is for XXX constants.

15.2.15.1 APR::SO_DEBUG

15.2.15.2 APR::SO_DISCONNECTED

15.2.15.3 APR::SO_KEEPALIVE

15.2.15.4 APR::SO_LINGER

15.2.15.5 APR::SO_NONBLOCK

15.2.15.6 APR::SO_RCVBUF

15.2.15.7 APR::SO_REUSEADDR

15.2.15.8 APR::SO_SNDBUF

15.2.16 :table

```
use APR::Const -compile => qw(:table);
```

The `:table` group is for `overlap()` and `compress()` constants. See `APR::Table` for details.

15.2.16.1 APR::OVERLAP_TABLES_MERGE

15.2.16.2 APR::OVERLAP_TABLES_SET

15.2.17 :uri

```
use APR::Const -compile => qw(:uri);
```

The `:uri` group is for XXX constants.

15.2.17.1 APR::URI_ACAP_DEFAULT_PORT

15.2.17.2 APR::URI_FTP_DEFAULT_PORT

15.2.17.3 APR::URI_GOPHER_DEFAULT_PORT

15.2.17.4 APR::URI_HTTPS_DEFAULT_PORT

15.2.17.5 APR::URI_HTTP_DEFAULT_PORT

15.2.17.6 APR::URI_IMAP_DEFAULT_PORT

15.2.17.7 APR::URI_LDAP_DEFAULT_PORT

15.2.17.8 APR::URI_NFS_DEFAULT_PORT

15.2.17.9 APR::URI_NNTP_DEFAULT_PORT

15.2.17.10 APR::URI_POP_DEFAULT_PORT

15.2.17.11 APR::URI_PROSPERO_DEFAULT_PORT

15.2.17.12 APR::URI_RTSP_DEFAULT_PORT

15.2.17.13 APR::URI_SIP_DEFAULT_PORT

15.2.17.14 APR::URI_SNEWS_DEFAULT_PORT

15.2.17.15 APR::URI_SSH_DEFAULT_PORT

15.2.17.16 APR::URI_TELNET_DEFAULT_PORT

15.2.17.17 APR::URI_TIP_DEFAULT_PORT

15.2.17.18 APR::URI_UNP_OMITPASSWORD

15.2.17.19 APR::URI_UNP_OMITPATHINFO

15.2.17.20 APR::URI_UNP_OMITQUERY

15.2.17.21 APR::URI_UNP_OMITSITEPART

15.2.17.22 APR::URI_UNP_OMITUSER

15.2.17.23 APR::URI_UNP_OMITUSERINFO

15.2.17.24 APR::URI_UNP_REVEALPASSWORD

15.2.17.25 APR::URI_WAIS_DEFAULT_PORT

16 APR:PerlIO -- An APR Perl IO layer

16.1 SYNOPSIS

```
# under mod_perl
use APR::PerlIO ();

sub handler {
    my $r = shift;

    die "This Perl build doesn't support PerlIO layers"
        unless APR::PerlIO::PERLIO_LAYERS_ARE_ENABLED;

    open my $fh, ">:APR", $filename, $r->pool or die $!;
    # work with $fh as normal $fh
    close $fh;

    return Apache::OK;
}

# outside mod_perl
% perl -MApache2 -MAPR -MAPR::PerlIO -MAPR::Pool -le \
'open my $fh, ">:APR", "/tmp/apr", APR::Pool->new or die "$!"; \
 print $fh "whoah!"; \
 close $fh;'
```

16.2 DESCRIPTION

`APR::PerlIO` implements a Perl IO layer using APR's file manipulation as its internals.

Why do you want to use this? Normally you shouldn't, probably it won't be faster than Perl's default layer. It's only useful when you need to manipulate a filehandle opened at the APR side, while using Perl.

Normally you won't call open() with APR layer attribute, but some mod_perl functions will return a filehandle which is internally hooked to APR. But you can use APR Perl IO directly if you want.

16.3 Constants

16.3.1 PERLIO_LAYERS_ARE_ENABLED

Before using the Perl IO APR layer one has to check whether it's supported by the used perl build.

```
die "This Perl build doesn't support PerlIO layers"
    unless APR::PerlIO::PERLIO_LAYERS_ARE_ENABLED;
```

Notice that loading `APR::PerlIO` won't fail when Perl IO layers aren't available since `APR::PerlIO` provides functionality for Perl builds not supporting Perl IO layers.

16.4 API

Perl Interface:

16.4.1 open()

To use APR Perl IO to open a file the four arguments open() should be used. For example:

```
open my $fh, ">:APR", $filename, $r->pool or die $!;
```

where:

the second argument is the mode to open the file, constructed from two sections separated by the `:` character: the first section is the mode to open the file under (>, <, etc) and the second section must be a string *APR*.

the fourth argument must be an `APR::Pool` object.

the rest of the arguments are the same as described by the *open()* manpage.

16.4.2 seek()

```
seek($fh, $offset, $whence);
```

If `$offset` is zero, `seek()` works normally.

However if `$offset` is non-zero and Perl has been compiled with with large files support (`-Duselargefiles`), whereas APR wasn't, this function will croak. This is because largefile size `Off_t` simply cannot fit into a non-largefile size `apr_off_t`.

To solve the problem, rebuild Perl with `-Uuselargefiles`. Currently there is no way to force APR to build with large files support.

16.5 C API

The C API provides functions to convert between Perl IO and APR Perl IO filehandles.

META: document these

16.6 SEE ALSO

The *perliol(1)*, *perlapio(1)* and *perl(1)* manpages.

17 APR::Table -- A Perl API for manipulating opaque string-content table

17.1 SYNOPSIS

```
use APR::Table;

$table = make($pool, $nelts);
$table_copy = $table->copy($pool);

$table->clear();

$table->set($key => $val);
$table->unset($key);
$table->add($key, $val);

$val = $table->get($key);
@val = $table->get($key);

$table->merge($key => $val);
overlap($table_a, $table_b, $flags);
$new_table = overlay($table_base, $table_overlay, $pool);

$table->do(sub {print "key $_[0], value $_[1]\n"}, @valid_keys);

#Tied Interface
$value = $table->{$key};
$table->{$key} = $value;
$table->{$key} = $value;
exists $table->{$key};

foreach my $key (keys %{$table}) {
    print "$key = $table->{$key}\n";
}
```

17.2 DESCRIPTION

`APR::Table` allows its users to manipulate opaque string-content tables.

The table's structure is somewhat similar to the Perl's hash structure, but allows multiple values for the same key. An access to the records stored in the table always requires a key.

The key-value pairs are stored in the order they are added.

The keys are case-insensitive.

However as of the current implementation if more than value for the same key is requested, the whole table is lineary searched, which is very inefficient unless the table is very small.

`APR::Table` provides a TIE Interface.

See *apr/include/apr_tables.h* in ASF's *apr* project for low level details.

17.3 API

The variables used in the API definition have the following *"types"*:

- **APR::Table**

  `$table_*`

- **APR::Pool**

  `$pool`

- **scalars: unsigned integers only (SVIV) (as C expects them)**

  `$nelts`, `$flags`

- **scalars: (numerical (SVIV/SVNV) and strings (SVPV))**

  `$key`, `$val`

Function arguments (if any) and return values are shown in the function's synopsis.

- **make()**

  ```
  $table = make($pool, $nelts);
  ```

  Make a new table.

  param `$pool`: The pool to allocate the pool out of.

  param `$nelts`: The number of elements in the initial table.

  return: a new table.

  warning: This table can only store text data

- **copy()**

  ```
  $table_copy = $table->copy($pool);
  ```

  Create a new table and copy another table into it

  param `$pool`: The pool to allocate the new table out of

  param `$table`: The table to copy

  return: A copy of the table passed in

- **clear()**

```
$table->clear();
```

Delete all of the elements from a table.

param `$table`: A copy of the table passed in

- **set();**

```
$table->set($key => $val);
```

Add a key/value pair to a table, if another element already exists with the same key, this will over-write the old data.

param `$table`: The table to add the data to.

param `$key`: The key fo use.

param `$val`: The value to add.

- **add()**

```
$table->add($key, $val);
```

Add data to a table, regardless of whether there is another element with the same key.

param `$table`: The table to add to

param `$key`: The key to use

param `$val`: The value to add.

- **do()**

```
$table->do(sub {[...]}, [@filter]);
```

Iterate over all the elements of the table, invoking provided subroutine for each element. The subroutine gets passed as argument, a key-value pair.

The subroutine can abort the iteration by returning 0 and should always return 1 otherwise.

param `sub`: A subroutine reference or name to be called on each item in the table

param `@filter`: Only keys matching one of the entries in the filter will be processed

- **get()**

```
$val = $table->get($key);
@val = $table->get($key);
```

Get the value(s) associated with a given key.

After this call, the data is still in the table.

param `$table`: The table to search for the key

param `$key`: The key to search for

return: In the scalar context the first matching value returned. (The oldest in the table, if there is more than one value.) In the list context the whole table is traversed and all matching values are returned. If nothing matches *undef* is returned.

- **unset();**

```
$table->unset($key);
```

Remove data from the table

param `$table`: The table to remove data from

param `$key`: The key of the data being removed

- **merge()**

```
$table->merge($key => $val);
```

Add data to a table by merging the value with data that has already been stored

param `$table`: The table to search for the data

param `$key`: The key to merge data for

param `$val`: The data to add

remark: If the key is not found, then this function acts like add()

- **overlap()**

```
overlap($table_a, $table_b, $flags);
```

For each key/value pair in `$table_b`, add the data to `$table_a`. The definition of `$flags` explains how `$flags` define the overlapping method.

param `$table_a`: The table to add the data to.

param `$table_b`: The table to iterate over, adding its data to `%table_a`.

param `$flags`: How to add the `$table_b` to `$table_a`.

When `$flags == APR::OVERLAP_TABLES_SET`, if another element already exists with the same key, this will over-write the old data.

When `$flags` == `APR::OVERLAP_TABLES_MERGE`, the key/value pair from `$table_b` is added, regardless of whether there is another element with the same key in `$table_a`.

remark: This function is highly optimized, and uses less memory and CPU cycles than a function that just loops through table b calling other functions.

- **overlay()**

```
$new_table = overlay($table_base, $table_overlay, $pool);
```

Merge two tables into one new table. The resulting table may have more than one value for the same key.

param `$pool`: The pool to use for the new table

param `$table_overlay`: The first table to put in the new table

param `$table_base`: The table to add at the end of the new table

return: A new table containing all of the data from the two passed in

- **compress()**

```
compress($table, $flag);
```

Converts multi-valued keys in `$table` to single-valued keys. This function takes duplicate table entries and flattens them into a single entry. The flattening behavior is controlled by the (mandatory) flag.

param `$table`: The table to add the data to.

param `$flag`: How to compress `$table`.

When `$flag` == `APR::OVERLAP_TABLES_SET`, each key will be set to the last value seen for that key. For example, given key/value pairs 'foo => bar' and 'foo => baz', 'foo' would have a final value of 'baz' after compression - the 'bar' value would be lost.

When `$flag` == `APR::OVERLAP_TABLES_MERGE`, multiple values for the same key are flattened into a comma-separated list. Given key/value pairs 'foo => bar' and 'foo => baz', 'foo' would have a final value of 'bar, baz' after compression.

17.3.1 TIE Interface

`APR::Table` also implements a tied interface, so you can work with the `$table` object as a hash reference.

The following tied-hash function are supported: FETCH, STORE, DELETE, CLEAR, EXISTS, FIRSTKEY, NEXTKEY and DESTROY.

remark: `APR::Table` can hold more than one key-value pair sharing the same key, so when using a table through the tied interface, the first entry found with the right key will be used, completely disregarding possible other entries with the same key. The only exception to this is if you iterate over the list with *each*, then you can access all key-value pairs that share the same key.

18 ModPerl::MethodLookup -- Map mod_perl 2.0 modules, objects and methods

18.1 Synopsis

```
use ModPerl::MethodLookup;

# return all module names containing XS method 'print'
my($hint, @modules) =
    ModPerl::MethodLookup::lookup_method('print');

# return only module names containing method 'print' which
# expects the first argument to be of type 'Apache::Filter'
# (here $filter is an Apache::Filter object)
my($hint, @modules) =
    ModPerl::MethodLookup::lookup_method('print', $filter);
# or
my($hint, @modules) =
    ModPerl::MethodLookup::lookup_method('print', 'Apache::Filter');

# what XS methods defined by module 'Apache::Filter'
my($hint, @methods) =
    ModPerl::MethodLookup::lookup_module('Apache::Filter');

# what XS methods can be invoked on the object $r (or a ref)
my($hint, @methods) =
    ModPerl::MethodLookup::lookup_object($r);
# or
my($hint, @methods) =
    ModPerl::MethodLookup::lookup_object('Apache::RequestRec');

# preload all mp2 modules in startup.pl
ModPerl::MethodLookup::preload_all_modules();

# command line shortcuts
% perl -MApache2 -MModPerl::MethodLookup -e print_module \
  Apache::RequestRec Apache::Filter
% perl -MApache2 -MModPerl::MethodLookup -e print_object Apache
% perl -MApache2 -MModPerl::MethodLookup -e print_method \
  get_server_built request
% perl -MApache2 -MModPerl::MethodLookup -e print_method read
% perl -MApache2 -MModPerl::MethodLookup -e print_method read APR::Bucket
```

18.2 Description

mod_perl 2.0 provides many methods, which reside in various modules. One has to load each of the modules before using the desired methods. `ModPerl::MethodLookup` provides the Perl API for finding module names which contain methods in question and other helper functions, like figuring out what methods defined by some module, or what methods can be called on a given object.

18.3 API

18.3.1 lookup_method()

```
my($hint, @modules) =
    ModPerl::MethodLookup::lookup_method($method_name);
```

The `lookup_method()` function accepts the method name as the first argument.

The first returned value is a string returning a human readable lookup result. Normally suggesting which modules should be loaded, ready for copy-n-paste or explaining the failure if the lookup didn't succeed.

The second returned value is an array of modules which have matched the query, i.e. the names of the modules which contain the requested method.

```
my($hint, @modules) =
    ModPerl::MethodLookup::lookup_method($method_name, $object);
```

or:

```
my($hint, @modules) =
    ModPerl::MethodLookup::lookup_method($method_name, ref($object));
```

The `lookup_method()` function accepts a second optional argument, which is a blessed object or the class it's blessed into. If there is more than one matches this extra information is used to return only modules of those methods which operate on the objects of the same kind. This usage is useful when the `AUTOLOAD` is used to find yet-unloaded modules which include called methods.

Examples:

Return all module names containing XS method *print*:

```
my($hint, @modules) =
    ModPerl::MethodLookup::lookup_method('print');
```

Return only module names containing method *print* which expects the first argument to be of type `Apache::Filter`:

```
my $filter = bless {}, 'Apache::Filter';
my($hint, @modules) =
    ModPerl::MethodLookup::lookup_method('print', $filter);
```

or:

```
my($hint, @modules) =
    ModPerl::MethodLookup::lookup_method('print', 'Apache::Filter');
```

18.3.2 lookup_module()

```
my($hint, @methods) =
    ModPerl::MethodLookup::lookup_module($module_name);
```

The `lookup_module()` function accepts the module name as an argument.

The first returned value is a string returning a human readable lookup result. Normally suggesting, which methods the given module implements, or explaining the failure if the lookup didn't succeed.

The second returned value is an array of methods which have matched the query, i.e. the names of the methods defined in the requested module.

Example:

What XS methods defined by module `Apache::Filter`:

```
my($hint, @methods) =
    ModPerl::MethodLookup::lookup_module('Apache::Filter');
```

18.3.3 lookup_object()

```
my($hint, @methods) =
    ModPerl::MethodLookup::lookup_object($object);
```

or:

```
my($hint, @methods) =
    ModPerl::MethodLookup::lookup_object(
        $the_class_object_is_blessed_into);
```

The `lookup_object()` function accepts the object or the class name the object is blessed into as an argument.

The first returned value is a string returning a human readable lookup result. Normally suggesting, which methods the given object can invoke (including module names that need to be loaded to use those methods), or explaining the failure if the lookup didn't succeed.

The second returned value is an array of methods which have matched the query, i.e. the names of the methods that can be invoked on the given object (or its class name).

META: As of this writing this method may miss some of the functions/methods that can be invoked on the given object. Currently we can't programmatically deduct the objects they are invoked on, because these methods are written in pure XS and manipulate the arguments stack themselves. Currently these are mainly XS functions, not methods, which of course aren't invoked on objects. There are also logging function wrappers (`Apache::Log`).

Examples:

What XS methods can be invoked on the object `$r`:

```
my($hint, @methods) =
    ModPerl::MethodLookup::lookup_object($r);
```

or `$r`'s class -- `Apache::RequestRec`:

```
my($hint, @methods) =
    ModPerl::MethodLookup::lookup_object('Apache::RequestRec');
```

18.3.4 print_method()

`print_method()` is a convenience wrapper for `lookup_method()`, mainly designed to be used from the command line. For example to print all the modules which define method *read* execute:

```
% perl -MApache2 -MModPerl::MethodLookup -e print_method read
```

Since this will return more than one module, we can narrow the query to only those methods which expect the first argument to be blessed into class `APR::Bucket`:

```
% perl -MApache2 -MModPerl::MethodLookup -e print_method read APR::Bucket
```

You can pass more than one method and it'll perform a lookup on each of the methods. For example to lookup methods `get_server_built` and `request` you can do:

```
% perl -MApache2 -MModPerl::MethodLookup -e print_method \
  get_server_built request
```

The function `print_method()` is exported by default.

18.3.5 print_module()

`print_module()` is a convenience wrapper for `lookup_module()`, mainly designed to be used from the command line. For example to print all the methods defined in the module `Apache::RequestRec`, followed by methods defined in the module `Apache::Filter` you can run:

```
% perl -MApache2 -MModPerl::MethodLookup -e print_module \
  Apache::RequestRec Apache::Filter
```

The function `print_module()` is exported by default.

18.3.6 print_object()

`print_object()` is a convenience wrapper for `lookup_object()`, mainly designed to be used from the command line. For example to print all the methods that can be invoked on object blessed into a class `Apache::RequestRec` run:

```
% perl -MApache2 -MModPerl::MethodLookup -e print_object \
  Apache::RequestRec
```

Similar to `print_object()`, more than one class can be passed to this function.

The function `print_object()` is exported by default.

18.3.7 `preload_all_modules()`

The function `preload_all_modules()` preloads all mod_perl 2.0 modules, which implement their API in XS. This is similar to the mod_perl 1.0 behavior which has most of its methods loaded at the startup.

CPAN modules developers should make sure their distribution loads each of the used mod_perl 2.0 modules explicitly, and not use this function, as it takes the fine control away from the users. One should avoid doing this the production server (unless all modules are used indeed) in order to save memory.

18.4 Applications

18.4.1 AUTOLOAD

When Perl fails to locate a method it checks whether the package the object belongs to has an `AUTOLOAD` function defined and if so, calls it with the same arguments as the missing method while setting a global variable `$AUTOLOAD` (in that package) to the name of the originally called method. We can use this facility to lookup the modules to be loaded when such a failure occurs. Though since we have many packages to take care of we will use a special `UNIVERSAL::AUTOLOAD` function which Perl calls if can't find the `AUTOLOAD` function in the given package.

In that function you can query `ModPerl::MethodLookup`, require() the module that includes the called method and call that method again using the goto() trick:

```
use ModPerl::MethodLookup;
sub UNIVERSAL::AUTOLOAD {
    my($hint, @modules) =
        ModPerl::MethodLookup::lookup_method($UNIVERSAL::AUTOLOAD, @_);
    if (@modules) {
        eval "require $_" for @modules;
        goto &$UNIVERSAL::AUTOLOAD;
    }
    else {
        die $hint;
    }
}
```

However we don't endorse this approach. It's a better approach to always abort the execution which printing the `$hint`and use fix the code to load the missing module. Moreover installing `UNIVERSAL::AUTOLOAD` may cause a lot of problems, since once it's installed Perl will call it every time some method is missing (e.g. undefined `DESTROY` methods). The following approach seems to somewhat work for me. It installs `UNIVERSAL::AUTOLOAD` only when the the child process starts.

```
httpd.conf:
-----------
PerlChildInitHandler ModPerl::MethodLookupAuto

startup.pl:
-----------
{
  package ModPerl::MethodLookupAuto;
  use ModPerl::MethodLookup;

  use Carp;
  sub handler {

      # exclude DESTROY resolving
      my $skip = '^(?!DESTROY$';
      *UNIVERSAL::AUTOLOAD = sub {
          my $method = $AUTOLOAD;
          return if $method =~ /DESTROY/;
          my ($hint, @modules) =
              ModPerl::MethodLookup::lookup_method($method, @_);
          $hint ||= "Can't find method $AUTOLOAD";
          croak $hint;
      };
      return 0;
  }
}
```

This example doesn't load the modules for you. It'll print to STDERR what module should be loaded, when a method from the not-yet-loaded module is called.

A similar technique is used by `Apache::porting`.

18.4.2 Command Line Lookups

When a method is used and mod_perl has reported a failure to find it, it's often useful to use the command line query to figure out which module needs to be loaded. For example if when executing:

```
$r->construct_url();
```

mod_perl complains:

```
Can't locate object method "construct_url" via package
"Apache::RequestRec" at ...
```

you can ask `ModPerl::MethodLookup` for help:

```
% perl -MApache2 -MModPerl::MethodLookup -e print_method construct_url
To use method 'construct_url' add:
        use Apache::URI ();
```

and after copy-n-pasting the use statement in our code, the problem goes away.

One can create a handy alias for this technique. For example, C-style shell users can do:

```
% alias lookup "perl -MApache2 -MModPerl::MethodLookup -e print_method"
```

For Bash-style shell users:

```
% alias lookup="perl -MApache2 -MModPerl::MethodLookup -e print_method"
```

Now the lookup is even easier:

```
% lookup construct_url
to use method 'construct_url' add:
        use Apache::URI;
```

Similar aliases can be provided for `print_object()` and `print_module()`.

18.5 Todo

These methods aren't yet picked by this module (the extract from the map file):

```
modperl_filter_attributes     | MODIFY_CODE_ATTRIBUTES
modperl_spawn_proc_prog       | spawn_proc_prog
apr_sockaddr_ip_get           | sockaddr
apr_sockaddr_port_get         | sockaddr
apr_ipsubnet_create           | new
```

Please report if you find any other missing methods. But remember that as of this moment the module reports only XS function. In the future we may add support for pure perl functions/methods as well.

18.6 See Also

- the mod_perl 1.0 backward compatibility document
- porting Perl modules
- porting XS modules
- `Apache::porting`

18.7 Author

Stas Bekman

19 ModPerl::MM -- A "subclass" of ExtUtils::MakeMaker for mod_perl 2.0

19.1 Synopsis

```
use ModPerl::MM;

# ModPerl::MM takes care of doing all the dirty job of overriding
ModPerl::MM::WriteMakefile(...);

# if there is a need to extend the default methods
sub MY::constants {
    my $self = shift;
    $self->ModPerl::MM::MY::constants;
    # do something else;
}

# or prevent overriding completely
sub MY::constants { shift->MM::constants(@_); }";

# override the default value of WriteMakefile's attribute
my $extra_inc = "/foo/include";
ModPerl::MM::WriteMakefile(
    ...
    INC => $extra_inc,
    ...
);

# extend the default value of WriteMakefile's attribute
my $extra_inc = "/foo/include";
ModPerl::MM::WriteMakefile(
    ...
    INC => join " ", $extra_inc, ModPerl::MM::get_def_opt('INC'),
    ...
);
```

19.2 Description

`ModPerl::MM` is a "subclass" of `ExtUtils::MakeMaker` for mod_perl 2.0, to a degree of sub-classability of `ExtUtils::MakeMaker`.

When `ModPerl::MM::WriteMakefile()` is used instead of `ExtUtils::MakeMaker::WriteMakefile()`, `ModPerl::MM` overrides several `ExtUtils::MakeMaker` methods behind the scenes and supplies default `WriteMakefile()` arguments adjusted for mod_perl 2.0 build. It's written in such a way so that normally 3rd party module developers for mod_perl 2.0, don't need to mess with *Makefile.PL* at all.

19.3 `MY::` Default Methods

`ModPerl::MM` overrides method *foo* as long as *Makefile.PL* hasn't already specified a method *MY::foo*. If the latter happens, `ModPerl::MM` will DWIM and do nothing.

In case the functionality of `ModPerl::MM` methods needs to be extended, rather than completely overriden, the `ModPerl::MM` methods can be called internally. For example if you need to modify constants in addition to the modifications applied by `ModPerl::MM::MY::constants`, call the `ModPerl::MM::MY::constants` method (notice that it resides in the package `ModPerl::MM::MY` and not `ModPerl::MM`), then do your extra manipulations on constants:

```
# if there is a need to extend the methods
sub MY::constants {
    my $self = shift;
    $self->ModPerl::MM::MY::constants;
    # do something else;
}
```

In certain cases a developers may want to prevent from `ModPerl::MM` to override certain methods. In that case an explicit override in *Makefile.PL* will do the job. For example if you don't want the `constants()` method to be overriden by `ModPerl::MM`, add to your *Makefile.PL*:

```
sub MY::constants { shift->MM::constants(@_); }";
```

`ModPerl::MM` overrides the following methods:

19.3.1 `ModPerl::MM::MY::constants`

This method makes sure that everything gets installed relative to the `Apache2/` subdir if `MP_INST_APACHE2=1` was used to build mod_perl 2.0.

19.3.2 `ModPerl::MM::MY::post_initialize`

This method makes sure that everything gets installed relative to the `Apache2/` subdir if `MP_INST_APACHE2=1` was used to build mod_perl 2.0.

19.4 `WriteMakefile()` Default Arguments

`ModPerl::MM::WriteMakefile` supplies default arguments such as `INC` and `TYPEMAPS` unless they weren't passed to `ModPerl::MM::WriteMakefile` from *Makefile.PL*.

If the default values aren't satisfying these should be overriden in *Makefile.PL*. For example to supply an empty INC, explicitly set the argument in *Makefile.PL*.

```
ModPerl::MM::WriteMakefile(
    ...
    INC => '',
    ...
);
```

If instead of fully overriding the default arguments, you want to extend or modify them, they can be retrieved using the `ModPerl::MM::get_def_opt()` function. The following example appends an extra value to the default `INC` attribute:

```
my $extra_inc = "/foo/include";
ModPerl::MM::WriteMakefile(
    ...
    INC => join " ", $extra_inc, ModPerl::MM::get_def_opt('INC'),
    ...
);
```

`ModPerl::MM` supplies default values for the following `ModPerl::MM::WriteMakefile` attributes:

19.4.1 CCFLAGS

19.4.2 LIBS

19.4.3 INC

19.4.4 OPTIMIZE

19.4.5 LDDLFLAGS

19.4.6 TYPEMAPS

19.4.7 dynamic_lib

19.4.7.1 OTHERLDFLAGS

```
dynamic_lib => { OTHERLDFLAGS => ... }
```

19.4.8 macro

19.4.8.1 MOD_INSTALL

```
macro => { MOD_INSTALL => ... }
```

arranges for modules to be installed under the subdir *Apache2/* if mod_perl was built with `MP_INST_APACHE2=1`.

19.5 Public API

The following functions are a part of the public API. They are described elsewhere in this document.

19.5.1 WriteMakefile()

```
ModPerl::MM::WriteMakefile(...);
```

19.5.2 get_def_opt()

```
my $def_val = ModPerl::MM::get_def_opt($key);
```

20 ModPerl::PerlRun - Run unaltered CGI scripts under mod_perl

20.1 SYNOPSIS

20.2 DESCRIPTION

20.3 AUTHORS

Doug MacEachern

Stas Bekman

20.4 SEE ALSO

ModPerl::RegistryCooker(3), Apache(3), mod_perl(3)

21 ModPerl::Registry - Run unaltered CGI scripts persistently under mod_perl

21 ModPerl::Registry - Run unaltered CGI scripts persistently under mod_perl

21.1 Synopsis

```
# httpd.conf
PerlModule ModPerl::Registry
Alias /perl/ /home/httpd/perl/
<Location /perl>
    SetHandler perl-script
    PerlResponseHandler ModPerl::Registry
    #PerlOptions +ParseHeaders
    #PerlOptions -GlobalRequest
    Options +ExecCGI
</Location>
```

21.2 Description

URIs in the form of `http://example.com/perl/test.pl` will be compiled as the body of a Perl subroutine and executed. Each child process will compile the subroutine once and store it in memory. It will recompile it whenever the file (e.g. *test.pl* in our example) is updated on disk. Think of it as an object oriented server with each script implementing a class loaded at runtime.

The file looks much like a "normal" script, but it is compiled into a subroutine.

For example:

```
my $r = Apache->request;
$r->content_type("text/html");
$r->send_http_header;
$r->print("mod_perl rules!");
```

XXX: STOPPED here. Below is the old Apache::Registry document which I haven't worked through yet.

META: document that for now we don't chdir() into the script's dir, because it affects the whole process under threads.

This module emulates the CGI environment, allowing programmers to write scripts that run under CGI or mod_perl without change. Existing CGI scripts may require some changes, simply because a CGI script has a very short lifetime of one HTTP request, allowing you to get away with "quick and dirty" scripting. Using mod_perl and ModPerl::Registry requires you to be more careful, but it also gives new meaning to the word "quick"!

Be sure to read all mod_perl related documentation for more details, including instructions for setting up an environment that looks exactly like CGI:

```
print "Content-type: text/html\n\n";
print "Hi There!";
```

Note that each httpd process or "child" must compile each script once, so the first request to one server may seem slow, but each request there after will be faster. If your scripts are large and/or make use of many Perl modules, this difference should be noticeable to the human eye.

21.3 Security

ModPerl::Registry::handler will preform the same checks as mod_cgi before running the script.

21.4 Environment

The Apache function ‘exit’ overrides the Perl core built-in function.

The environment variable **GATEWAY_INTERFACE** is set to `CGI-Perl/1.1`.

21.5 Commandline Switches In First Line

Normally when a Perl script is run from the command line or under CGI, arguments on the ‘#!’ line are passed to the perl interpreter for processing.

`ModPerl::Registry` currently only honors the **-w** switch and will enable the `warnings` pragma in such case.

Another common switch used with CGI scripts is **-T** to turn on taint checking. This can only be enabled when the server starts with the configuration directive:

```
PerlSwitches -T
```

However, if taint checking is not enabled, but the **-T** switch is seen, `ModPerl::Registry` will write a warning to the *error_log* file.

21.6 Debugging

You may set the debug level with the $ModPerl::Registry::Debug bitmask

```
1 => log recompile in errorlog
2 => ModPerl::Debug::dump in case of $@
4 => trace pedantically
```

21.7 Caveats

ModPerl::Registry makes things look just the CGI environment, however, you must understand that this *is not CGI*. Each httpd child will compile your script into memory and keep it there, whereas CGI will run it once, cleaning out the entire process space. Many times you have heard "always use `-w`, always use `-w` and ’use strict’". This is more important here than anywhere else!

Your scripts cannot contain the __END__ or __DATA__ token to terminate compilation. (META: works in 2.0).

21.8 Authors

Andreas J. Koenig, Doug MacEachern and Stas Bekman.

21.9 See Also

`ModPerl::RegistryCooker`, `ModPerl::RegistryBB`, `ModPerl::PerlRun`, Apache(3), mod_perl(3)

22 ModPerl::RegistryBB - Run unaltered CGI scripts persistently under mod_perl

22 ModPerl::RegistryBB - Run unaltered CGI scripts persistently under mod_perl

22.1 Synopsis

```
# httpd.conf
PerlModule ModPerl::RegistryBB
Alias /perl/ /home/httpd/perl/
<Location /perl>
    SetHandler perl-script
    PerlResponseHandler ModPerl::RegistryBB
    #PerlOptions +ParseHeaders
    #PerlOptions -GlobalRequest
    Options +ExecCGI
</Location>
```

22.2 Description

`ModPerl::RegistryBB` is similar to `ModPerl::Registry`, but does the bare minimum (mnemonic: BB = Bare Bones) to compile a script file once and run it many times, in order to get the maximum performance. Whereas `ModPerl::Registry` does various checks, which add a slight overhead to response times.

22.3 Authors

Doug MacEachern

Stas Bekman

22.4 See Also

`ModPerl::RegistryCooker`, `ModPerl::Registry`, Apache(3), mod_perl(3)

23 ModPerl::RegistryCooker - Cook mod_perl 2.0 Registry Modules

23.1 Synopsis

```
# shouldn't be used as-is but sub-classed first
# see ModPerl::Registry for an example
```

23.2 Description

`ModPerl::RegistryCooker` is used to create flexible and overridable registry modules which emulate mod_cgi for Perl scripts. The concepts are discussed in the manpage of the following modules: `ModPerl::Registry`, `ModPerl::Registry` and `ModPerl::RegistryBB`.

`ModPerl::RegistryCooker` has two purposes:

- Provide ingredients that can be used by registry sub-classes
- Provide a default behavior, which can be overriden in sub-classed

  META: in the future this functionality may move into a separate class.

Here are the current overridable methods:

META: these are all documented in RegistryCooker.pm, though not using pod. please help to port these to pod and move the descriptions here.

- **new()**

  create the class's object, bless it and return it

  ```
  my $obj = $class->new($r);
  ```

  `$class` -- the registry class, usually `__PACKAGE__` can be used.

  `$r` -- *`Apache::Request`* object.

  default: new()

- **init()**

  initializes the data object's fields: `REQ`, `FILENAME`, `URI`. Called from the new().

  default: init()

- **default_handler()**

  default: default_handler()

- **run()**

default: run()

- **can_compile()**

  default: can_compile()

- **make_namespace()**

  default: make_namespace()

- **namespace_root()**

  default: namespace_root()

- **namespace_from()**

  If `namespace_from_uri` is used and the script is called from the virtual host, by default the virtual host name is prepended to the uri when package name for the compiled script is created. Sometimes this behavior is undesirable, e.g., when the same (physical) script is accessed using the same path_info but different virtual hosts. In that case you can make the script compiled only once for all vhosts, by specifying:

  ```
  $ModPerl::RegistryCooker::NameWithVirtualHost = 0;
  ```

  The drawback is that it affects the global environment and all other scripts will be compiled ignoring virtual hosts.

  default: namespace_from()

- **is_cached()**

  default: is_cached()

- **should_compile()**

  default: should_compile()

- **flush_namespace()**

  default: flush_namespace()

- **cache_table()**

  default: cache_table()

- **cache_it()**

  default: cache_it()

- **read_script()**

  default: read_script()

- **rewrite_shebang()**

  default: rewrite_shebang()

- **set_script_name()**

  default: set_script_name()

- **chdir_file()**

  default: chdir_file()

- **get_mark_line()**

  default: get_mark_line()

- **compile()**

  default: compile()

- **error_check()**

  default: error_check()

- **strip_end_data_segment()**

  default: strip_end_data_segment()

- **convert_script_to_compiled_handler()**

  default: convert_script_to_compiled_handler()

23.2.1 Special Predefined Functions

The following functions are implemented as constants.

- **NOP()**

  Use when the function shouldn’t do anything.

- **TRUE()**

  Use when a function should always return a true value.

- **FALSE()**

  Use when a function should always return a false value.

23.3 Sub-classing Techniques

To override the default `ModPerl::RegistryCooker` methods, first, sub-class `ModPerl::RegistryCooker` or one of its existing sub-classes, using `use base`. Second, override the methods.

Those methods that weren't overridden will be resolved at run time when used for the first time and cached for the future requests. One way to to shortcut this first run resolution is to use the symbol aliasing feature. For example to alias `ModPerl::MyRegistry::flush_namespace` as `ModPerl::RegistryCooker::flush_namespace`, you can do:

```
package ModPerl::MyRegistry;
use base qw(ModPerl::RegistryCooker);
*ModPerl::MyRegistry::flush_namespace =
    \&ModPerl::RegistryCooker::flush_namespace;
1;
```

In fact, it's a good idea to explicitly alias all the methods so you know exactly what functions are used, rather then relying on the defaults. For that purpose `ModPerl::RegistryCooker` class method install_aliases() can be used. Simply prepare a hash with method names in the current package as keys and corresponding fully qualified methods to be aliased for as values and pass it to install_aliases(). Continuing our example we could do:

```
package ModPerl::MyRegistry;
use base qw(ModPerl::RegistryCooker);
my %aliases = (
    flush_namespace => 'ModPerl::RegistryCooker::flush_namespace',
);
__PACKAGE__->install_aliases(\%aliases);
1;
```

The values use fully qualified packages so you can mix methods from different classes.

23.4 Examples

The best examples are existing core registry modules: `ModPerl::Registry`, `ModPerl::Registry` and `ModPerl::RegistryBB`. Look at the source code and their manpages to see how they subclass `ModPerl::RegistryCooker`.

For example by default `ModPerl::Registry` uses the script's path when creating a package's namespace. If for example you want to use a uri instead you can override it with:

```
*ModPerl::MyRegistry::namespace_from =
    \&ModPerl::RegistryCooker::namespace_from_uri;
1;
```

Since the `namespace_from_uri` component already exists in `ModPerl::RegistryCooker`. If you want to write your own method, e.g., that creates a namespace based on the inode, you can do:

```
sub namespace_from_inode {
    my $self = shift;
    return (stat $self->[FILENAME])[1];
}
```

META: when $r->finfo will be ported it'll be more effecient. (stat $r->finfo)[1]

23.5 Authors

Doug MacEachern

Stas Bekman

23.6 See Also

`ModPerl::Registry`, `ModPerl::RegistryBB`, `ModPerl::PerlRun`, Apache(3), mod_perl(3)

24 ModPerl::RegistryLoader - Compile ModPerl::RegistryCooker scripts at server startup

24 ModPerl::RegistryLoader - Compile ModPerl::RegistryCooker scripts at server startup

24.1 Synopsis

```
# in startup.pl
use ModPerl::RegistryLoader ();

# explicit uri => filename mapping
my $rlbb = ModPerl::RegistryLoader->new(
    package => 'ModPerl::RegistryBB',
    debug   => 1, # default 0
);

$rlbb->handler($uri, $filename);

###
# uri => filename mapping using a helper function
sub trans {
    my $uri = shift;
    $uri =~ s|^/registry/|cgi-bin/|;
    return Apache::server_root_relative($uri);
}
my $rl = ModPerl::RegistryLoader->new(
    package => 'ModPerl::Registry',
    trans   => \&trans,
);
$rl->handler($uri);

###
$rlbb->handler($uri, $filename, $virtual_hostname);
```

24.2 Description

This modules allows compilation of scripts, running under packages derived from `ModPerl::RegistryCooker`, at server startup. The script's handler routine is compiled by the parent server, of which children get a copy and thus saves some memory by initially sharing the compiled copy with the parent and saving the overhead of script's compilation on the first request in every httpd instance.

This module is of course useless for those running the `ModPerl::PerlRun` handler, because the scripts get recompiled on each request under this handler.

24.3 Methods

- **new()**

  When creating a new `ModPerl::RegistryLoader` object, one has to specify which of the `ModPerl::RegistryCooker` derived modules to use. For example if a script is going to run under `ModPerl::RegistryBB` the object is initialized as:

  ```
  my $rlbb = ModPerl::RegistryLoader->new(
      package => 'ModPerl::RegistryBB',
  );
  ```

If the package is not specified `ModPerl::Registry` is assumed:

```
my $rlbb = ModPerl::RegistryLoader->new();
```

To turn the debugging on, set the *debug* attribute to a true value:

```
my $rlbb = ModPerl::RegistryLoader->new(
    package => 'ModPerl::RegistryBB',
    debug   => 1,
);
```

Instead of specifying explicitly a filename for each uri passed to handler(), a special attribute *trans* can be set to a subroutine to perform automatic remapping.

```
my $rlbb = ModPerl::RegistryLoader->new(
    package => 'ModPerl::RegistryBB',
    trans   => \&trans,
);
```

See the handler() item for an example of using the *trans* attribute.

- **handler()**

```
$rl->handler($uri, [$filename, [$virtual_hostname]]);
```

The handler() method takes argument of `uri` and optionally of `filename` and of `virtual_hostname`.

URI to filename translation normally doesn't happen until HTTP request time, so we're forced to roll our own translation. If the filename is supplied it's used in translation.

If the filename is omitted and a `trans` subroutine was not set in new(), the loader will try using the `uri` relative to the `ServerRoot` configuration directive. For example:

```
httpd.conf:
-----------
ServerRoot /usr/local/apache
Alias /registry/ /usr/local/apache/cgi-bin/

startup.pl:
-----------
use ModPerl::RegistryLoader ();
my $rl = ModPerl::RegistryLoader->new(
    package => 'ModPerl::Registry',
);
# preload /usr/local/apache/cgi-bin/test.pl
$rl->handler(/registry/test.pl);
```

To make the loader smarter about the URI->filename translation, you may provide the `new()` method with a `trans()` function to translate the uri to filename.

The following example will pre-load all files ending with *.pl* in the *cgi-bin* directory relative to `ServerRoot`.

```
httpd.conf:
-----------
ServerRoot /usr/local/apache
Alias /registry/ /usr/local/apache/cgi-bin/

startup.pl:
-----------
{
    # test the scripts pre-loading by using trans sub
    use ModPerl::RegistryLoader ();
    use DirHandle ();
    use strict;

    my $dir = Apache::server_root_relative("cgi-bin");

    sub trans {
        my $uri = shift;
        $uri =~ s|^/registry/|cgi-bin/|;
        return Apache::server_root_relative($uri);
    }

    my $rl = ModPerl::RegistryLoader->new(
        package => "ModPerl::Registry",
        trans   => \&trans,
    );
    my $dh = DirHandle->new($dir) or die $!;

    for my $file ($dh->read) {
        next unless $file =~ /\.pl$/;
        $rl->handler("/registry/$file");
    }
}
```

If `$virtual_hostname` argument is passed it'll be used in the creation of the package name the script will be compiled into for those registry handlers that use *namespace_from_uri()* method. See also the notes on `$ModPerl::RegistryCooker::NameWithVirtualHost` in the `ModPerl::RegistryCooker` documentation.

Also explained in the `ModPerl::RegistryLoader` documentation, this only has an effect at run time if `$ModPerl::RegistryCooker::NameWithVirtualHost` is set to true, otherwise the `$virtual_hostname` argument is ignored.

24.4 Implementation Notes

`ModPerl::RegistryLoader` performs a very simple job, at run time it loads and sub-classes the module passed via the *package* attribute and overrides some of its functions, to emulate the run-time environment. This allows to preload the same script into different registry environments.

24.5 Authors

The original `Apache::RegistryLoader` implemented by Doug MacEachern.

Stas Bekman did the porting to the new registry framework based on `ModPerl::RegistryLoader`.

24.6 SEE ALSO

`ModPerl::RegistryCooker`, `ModPerl::Registry`, `ModPerl::RegistryBB`, `ModPerl::PerlRun`, Apache(3), mod_perl(3)

25 ModPerl::Util -- Helper mod_perl 2.0 Functions

25.1 SYNOPSIS

```
use ModPerl::Util;

$callback = Apache::current_callback;

ModPerl::Util::exit();

ModPerl::Util::untaint($) # secret API?
```

25.2 DESCRIPTION

`ModPerl::Util` provides mod_perl 2.0 util functions.

META: complete

26 Apache::Reload - Reload Perl Modules when Changed on Disk

26.1 Synopsis

```
# Monitor and reload all modules in %INC:
# httpd.conf:
PerlModule Apache::Reload
PerlInitHandler Apache::Reload

# when working with protocols and connection filters
# PerlPreConnectionHandler Apache::Reload

# Reload groups of modules:
# httpd.conf:
PerlModule Apache::Reload
PerlInitHandler Apache::Reload
PerlSetVar ReloadAll Off
PerlSetVar ReloadModules "ModPerl::* Apache::*"
#PerlSetVar ReloadDebug On
#PerlSetVar ReloadConstantRedefineWarnings Off

# Reload a single module from within itself:
package My::Apache::Module;
use Apache::Reload;
sub handler { ... }
1;
```

26.2 Description

`Apache::Reload` reloads modules that change on the disk.

When Perl pulls a file via `require`, it stores the filename in the global hash `%INC`. The next time Perl tries to `require` the same file, it sees the file in `%INC` and does not reload from disk. This module's handler can be configured to iterate over the modules in `%INC` and reload those that have changed on disk or only specific modules that have registered themselves with `Apache::Reload`. It can also do the check for modified modules, when a special touch-file has been modified.

Note that `Apache::Reload` operates on the current context of `@INC`. Which means, when called as a `Perl*Handler` it will not see `@INC` paths added or removed by `Apache::Registry` scripts, as the value of `@INC` is saved on server startup and restored to that value after each request. In other words, if you want `Apache::Reload` to work with modules that live in custom `@INC` paths, you should modify `@INC` when the server is started. Besides, `'use lib'` in the startup script, you can also set the `PERL5LIB` variable in the httpd's environment to include any non-standard 'lib' directories that you choose. For example, to accomplish that you can include a line:

```
PERL5LIB=/home/httpd/perl/extra; export PERL5LIB
```

in the script that starts Apache. Alternatively, you can set this environment variable in *httpd.conf*:

```
PerlSetEnv PERL5LIB /home/httpd/perl/extra
```

26.2.1 Monitor All Modules in %INC

To monitor and reload all modules in `%INC` at the beginning of request's processing, simply add the following configuration to your *httpd.conf*:

```
PerlModule Apache::Reload
PerlInitHandler Apache::Reload
```

When working with connection filters and protocol modules `Apache::Reload` should be invoked in the pre_connection stage:

```
PerlPreConnectionHandler Apache::Reload
```

See also the discussion on `PerlPreConnectionHandler`.

26.2.2 Register Modules Implicitly

To only reload modules that have registered with `Apache::Reload`, add the following to the *httpd.conf*:

```
PerlModule Apache::Reload
PerlInitHandler Apache::Reload
PerlSetVar ReloadAll Off
# ReloadAll defaults to On
```

Then any modules with the line:

```
use Apache::Reload;
```

Will be reloaded when they change.

26.2.3 Register Modules Explicitly

You can also register modules explicitly in your *httpd.conf* file that you want to be reloaded on change:

```
PerlModule Apache::Reload
PerlInitHandler Apache::Reload
PerlSetVar ReloadAll Off
PerlSetVar ReloadModules "My::Foo My::Bar Foo::Bar::Test"
```

Note that these are split on whitespace, but the module list **must** be in quotes, otherwise Apache tries to parse the parameter list.

The `*` wild character can be used to register groups of files under the same namespace. For example the setting:

```
PerlSetVar ReloadModules "ModPerl::* Apache::*"
```

will monitor all modules under the namespaces `ModPerl::` and `Apache::`.

26.2.4 Monitor Only Certain Sub Directories

To reload modules only in certain directories (and their subdirectories) add the following to the *httpd.conf*:

```
PerlModule Apache::Reload
PerlInitHandler Apache::Reload
PerlSetVar ReloadDirectories "/tmp/project1 /tmp/project2"
```

You can further narrow the list of modules to be reloaded from the chosen directories with `ReloadModules` as in:

```
PerlModule Apache::Reload
PerlInitHandler Apache::Reload
PerlSetVar ReloadDirectories "/tmp/project1 /tmp/project2"
PerlSetVar ReloadAll Off
PerlSetVar ReloadModules "MyApache::*"
```

In this configuration example only modules from the namespace `MyApache::` found in the directories */tmp/project1/* and */tmp/project2/* (and their subdirectories) will be reloaded.

26.2.5 Special "Touch" File

You can also declare a file, which when gets `touch(1)`ed, causes the reloads to be performed. For example if you set:

```
PerlSetVar ReloadTouchFile /tmp/reload_modules
```

and don't `touch(1)` the file */tmp/reload_modules*, the reloads won't happen until you go to the command line and type:

```
% touch /tmp/reload_modules
```

When you do that, the modules that have been changed, will be magically reloaded on the next request. This option works with any mode described before.

26.3 Performance Issues

This modules is perfectly suited for a development environment. Though it's possible that you would like to use it in a production environment, since with `Apache::Reload` you don't have to restart the server in order to reload changed modules during software updates. Though this convenience comes at a price:

- If the "touch" file feature is used, `Apache::Reload` has to stat(2) the touch file on each request, which adds a slight but most likely insignificant overhead to response times. Otherwise `Apache::Reload` will stat(2) each registered module or even worse--all modules in `%INC`, which will significantly slow everything down.

- Once the child process reloads the modules, the memory used by these modules is not shared with the parent process anymore. Therefore the memory consumption may grow significantly.

Therefore doing a full server stop and restart is probably a better solution.

26.4 Debug

If you aren't sure whether the modules that are supposed to be reloaded, are actually getting reloaded, turn the debug mode on:

```
PerlSetVar ReloadDebug On
```

26.5 Silencing 'Constant subroutine ... redefined at' Warnings

If a module defines constants, e.g.:

```
use constant PI => 3.14;
```

and gets re-loaded, Perl issues a mandatory warnings which can't be silenced by conventional means (since Perl 5.8.0). This is because constants are inlined at compile time, so if there are other modules that are using constants from this module, but weren't reloaded they will see different values. Hence the warning is mandatory. However chances are that most of the time you won't modify the constant subroutine and you don't want *error_log* to be cluttered with (hopefully) irrelevant warnings. In such cases, if you haven't modified the constant subroutine, or you know what you are doing, you can tell `Apache::Reload` to shut those for you (it overrides `$SIG{__WARN__}` to accomplish that):

```
PerlSetVar ReloadConstantRedefineWarnings Off
```

For the reasons explained above this option is turned on by default.

since: mod_perl 1.99_10

26.6 Caveats

26.6.1 Problems With Reloading Modules Which Do Not Declare Their Package Name

If you modify modules, which don't declare their `package`, and rely on `Apache::Reload` to reload them, you may encounter problems: i.e., it'll appear as if the module wasn't reloaded when in fact it was. This happens because when `Apache::Reload` `require()`s such a module all the global symbols end up in the `Apache::Reload` namespace! So the module does get reloaded and you see the compile time errors if there are any, but the symbols don't get imported to the right namespace. Therefore the old version of the code is running.

26.6.2 Problems with Scripts Running with Registry Handlers that Cache the Code

The following problem is relevant only to registry handlers that cache the compiled script. For example it concerns `ModPerl::Registry` but not `ModPerl::PerlRun`.

26.6.2.1 The Problem

Let's say that there is a module `My::Utils`:

```
#file:My/Utils.pm
#----------------
package My::Utils;
BEGIN { warn __PACKAGE__ , " was reloaded\n" }
use base qw(Exporter);
@EXPORT = qw(colour);
sub colour { "white" }
1;
```

And a registry script *test.pl*:

```
#file:test.pl
#------------
use My::Utils;
print "Content-type: text/plain\n\n";
print "the color is " . colour();
```

Assuming that the server is running in a single mode, we request the script for the first time and we get the response:

```
the color is white
```

Now we change *My/Utils.pm*:

```
-  sub colour { "white" }
+  sub colour { "red" }
```

And issue the request again. `Apache::Reload` does its job and we can see that `My::Utils` was reloaded (look in the *error_log* file). However the script still returns:

```
the color is white
```

26.6.2.2 The Explanation

Even though *My/Utils.pm* was reloaded, `ModPerl::Registry`'s cached code won't run '`use My::Utils;`' again (since it happens only once, i.e. during the compile time). Therefore the script doesn't know that the subroutine reference has been changed.

This is easy to verify. Let's change the script to be:

```
#file:test.pl
#------------
use My::Utils;
print "Content-type: text/plain\n\n";
my $sub_int = \&colour;
my $sub_ext = \&My::Utils::colour;
print "int $sub_int\n";
print "ext $sub_ext\n";
```

Issue a request, you will see something similar to:

```
int CODE(0x8510af8)
ext CODE(0x8510af8)
```

As you can see both point to the same CODE reference (meaning that it's the same symbol). After modifying *My/Utils.pm* again:

```
-  sub colour { "red" }
+  sub colour { "blue" }
```

and calling the script on the secondnd time, we get:

```
int CODE(0x8510af8)
ext CODE(0x851112c)
```

You can see that the internal CODE reference is not the same as the external one.

26.6.2.3 The Solution

There are two solutions to this problem:

Solution 1: replace `use()` with an explicit `require()` + `import()`.

```
- use My::Utils;
+ require My::Utils; My::Utils->import();
```

now the changed functions will be reimported on every request.

Solution 2: remember to touch the script itself every time you change the module that it requires.

26.7 Threaded MPM and Multiple Perl Interpreters

If you use `Apache::Reload` with a threaded MPM and multiple Perl interpreters, the modules will be reloaded by each interpreter as they are used, not every interpreters at once. Similar to mod_perl 1.0 where each child has its own Perl interpreter, the modules are reloaded as each child is hit with a request.

If a module is loaded at startup, the syntax tree of each subroutine is shared between interpreters (big win), but each subroutine has its own padlist (where lexical my variables are stored). Once `Apache::Reload` reloads a module, this sharing goes away and each Perl interpreter will have its own copy of the syntax tree for the reloaded subroutines.

26.8 Pseudo-hashes

The short summary of this is: Don't use pseudo-hashes. They are deprecated since Perl 5.8 and are removed in 5.9.

Use an array with constant indexes. Its faster in the general case, its more guaranteed, and generally, it works.

The long summary is that some work has been done to get this module working with modules that use pseudo-hashes, but it's still broken in the case of a single module that contains multiple packages that all use pseudo-hashes.

So don't do that.

26.9 Authors

Matt Sergeant, matt@sergeant.org

Stas Bekman (porting to mod_perl 2.0)

A few concepts borrowed from `Stonehenge::Reload` by Randal Schwartz and `Apache::StatINC` (mod_perl 1.x) by Doug MacEachern and Ask Bjoern Hansen.

26.10 See Also

`Stonehenge::Reload`

27 Apache::Status - Embedded interpreter status information

27.1 SYNOPSIS

```
<Location /perl-status>
    SetHandler modperl
    PerlResponseHandler Apache::Status
</Location>
```

27.2 DESCRIPTION

The **Apache::Status** module provides some information about the status of the Perl interpreter embedded in the server.

Configure like so:

```
<Location /perl-status>
     SetHandler modperl
     PerlResponseHandler Apache::Status
</Location>
```

Notice that under the "modperl" core handler the *Environment* menu option will show only the environment under that handler. To see the environment seen by handlers running under the "perl-script" core handler, configure `Apache::Status` as:

```
<Location /perl-status>
     SetHandler perl-script
     PerlResponseHandler Apache::Status
</Location>
```

Other modules can "plugin" a menu item like so:

```
Apache::Status->menu_item(
   'DBI' => "DBI connections", #item for Apache::DBI module
   sub {
       my($r,$q) = @_; #request and CGI objects
       my(@strings);
       push @strings,  "blobs of html";
       return \@strings;     #return an array ref
   }
) if Apache->module("Apache::Status"); #only if Apache::Status is loaded
```

WARNING: Apache::Status must be loaded before these modules via the PerlModule or PerlRequire directives.

27.3 OPTIONS

- **StatusOptionsAll**

  This single directive will enable all of the options described below.

```
PerlSetVar StatusOptionsAll On
```

- **StatusDumper**

  When browsing symbol tables, the values of arrays, hashes and scalars can be viewed via **Data::Dumper** if this configuration variable is set to On:

  ```
  PerlSetVar StatusDumper On
  ```

- **StatusPeek**

  With this option On and the **Apache::Peek** module installed, functions and variables can be viewed ala **Devel::Peek** style:

  ```
  PerlSetVar StatusPeek On
  ```

- **StatusLexInfo**

  With this option On and the **B::LexInfo** module installed, subroutine lexical variable information can be viewed.

  ```
  PerlSetVar StatusLexInfo On
  ```

- **StatusDeparse**

  With this option On and **B::Deparse** version 0.59 or higher (included in Perl 5.005_59+), subroutines can be "deparsed".

  ```
  PerlSetVar StatusDeparse On
  ```

  Options can be passed to B::Deparse::new like so:

  ```
  PerlSetVar StatusDeparseOptions "-p -sC"
  ```

  See the **B::Deparse** manpage for details.

- **StatusTerse**

  With this option On, text-based op tree graphs of subroutines can be displayed, thanks to **B::Terse**.

  ```
  PerlSetVar StatusTerse On
  ```

- **StatusTerseSize**

  With this option On and the **B::TerseSize** module installed, text-based op tree graphs of subroutines and their size can be displayed. See the **B::TerseSize** docs for more info.

  ```
  PerlSetVar StatusTerseSize On
  ```

- **StatusTerseSizeMainSummary**

With this option On and the **B::TerseSize** module installed, a "Memory Usage" will be added to the Apache::Status main menu. This option is disabled by default, as it can be rather cpu intensive to summarize memory usage for the entire server. It is strongly suggested that this option only be used with a development server running in **-X** mode, as the results will be cached.

```
PerlSetVar StatusTerseSizeMainSummary On
```

- **StatusGraph**

  When **StatusDumper** is enabled, another link "OP Tree Graph" will be present with the dump if this configuration variable is set to On:

  ```
  PerlSetVar StatusGraph
  ```

  This requires the B module (part of the Perl compiler kit) and B::Graph (version 0.03 or higher) module to be installed along with the **dot** program.

  Dot is part of the graph visualization toolkit from AT&T:
  `http://www.research.att.com/sw/tools/graphviz/`).

  WARNING: Some graphs may produce very large images, some graphs may produce no image if B::Graph's output is incorrect.

- **Dot**

  Location of the dot program for StatusGraph, if other than /usr/bin or /usr/local/bin

- **GraphDir**

  Directory where StatusGraph should write it's temporary image files. Default is $Server-Root/logs/b_graphs

27.4 PREREQUISITES

The *Devel::Symdump* module, version **2.00** or higher.

27.5 SEE ALSO

perl(1), Apache(3), Devel::Symdump(3), Data::Dumper(3), B(3), B::Graph(3)

27.6 AUTHORS

Doug MacEachern with contributions from Stas Bekman

28 ModPerl::BuildMM -- A "subclass" of ModPerl::MM used for building mod_perl 2.0

28.1 SYNOPSIS

```
use ModPerl::BuildMM;

# ModPerl::BuildMM takes care of doing all the dirty job of overriding
ModPerl::BuildMM::WriteMakefile(...);

# if there is a need to extend the methods
sub MY::postamble {
    my $self = shift;

    my $string = $self->ModPerl::BuildMM::MY::postamble;

    $string .= "\nmydist : manifest tardist\n";

    return $string;
}
```

28.2 DESCRIPTION

`ModPerl::BuildMM` is a "subclass" of `ModPerl::MM` used for building mod_perl 2.0. Refer to `ModPerl::MM` manpage.

28.3 OVERRIDEN METHODS

`ModPerl::BuildMM` overrides the following methods:

28.3.1 ModPerl::BuildMM::MY::constants

28.3.2 ModPerl::BuildMM::MY::top_targets

28.3.3 ModPerl::BuildMM::MY::postamble

28.3.4 ModPerl::BuildMM::MY::post_initialize

28.3.5 ModPerl::BuildMM::MY::libscan

Table of Contents: