1 Apache::Filter - Perl API for Apache 2.0 Filtering

1.1 Synopsis

```
use Apache::Filter ();
```

META: to be completed

1.2 Description

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

Make sure to read `the Filtering tutorial|docs::2.0::user::handlers::filters`.

1.3 Common Filter API

The following methods can be called from any filter handler:

1.3.1 `c`

The current connection object can be retrieved from a connection or a request filter with:

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

- **arg1: `$f` (`Apache::Filter`)**
- **ret: `$c` (`Apache::Connection`)**

1.3.2 `ctx`

Get and set the filter context data.

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

- **arg1: `$f` (`Apache::Filter`)**
- **opt arg2: `$ctx` (scalar)**

  Could be any perl SCALAR.

- **ret: `$ctx` (scalar)**

  Could be any perl SCALAR.

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.

Besides its usage to store data between filter invocations, this method is also useful when as a flag. For example here is how to ensure that something happens only once during the filter's life:

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

1.3.3 frec

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

```
my $frec = $f->frec();
$f->frec($frec);
```

- **arg1: `$f` (`Apache::Filter`)**
- **opt arg2: `$frec` (`Apache::FilterRec`)**
- **opt ret: `$frec` (`Apache::FilterRec`)**

1.3.4 next

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

```
$next_f = $f->next;
```

- **arg1: `$f` (`Apache::Filter`)**
- **ret: `$next_f` (`Apache::Filter`)**

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*).

1.3.5 `r`

Inside an HTTP request filter retrieve the current request object:

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

- **arg1: `$f` (`Apache::Filter`)**
- **ret: `$r` (`Apache::RequestRec`)**

If a sub-request adds filters, then the sub-request is the request associated with the filter.

1.3.6 `remove`

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

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

- **arg1: `$f` (`Apache::Filter`)**
- **ret: no return value**

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.

1.4 Bucket Brigade Filter API

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

1.4.1 `fflush`

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

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

- **arg1: `$f` (`Apache::Filter`)**

  The current filter

- **arg2: `$bb` (`Apache::Filter`)**

  The brigade to flush

- **ret: XXX**

1.4.2 `get_brigade`

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

```
$ret = $next_f->get_brigade($bb, $mode, $block, $readbytes);
```

- **arg1: `$next_f` (`Apache::Filter`)**

  The next filter in the chain

- **arg2: `$bb` (`APR::Brigade`)**

  The original brigade passed to get_brigade() must be empty. On return it gets populated with the next bucket brigade, or nothing if there is no more data to read.

- **arg3: `$mode` (integer)**

  The way in which the data should be read

- **arg4: `$block` (integer)**

  How the operations should be performed APR::BLOCK_READ, APR::NONBLOCK_READ

- **arg5: `$readbytes` (integer)**

  How many bytes to read from the next filter.

- **ret: `$ret` (integer)**

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

If the bottom-most filter doesn't read from the network, then Apache::NOBODY_READ is returned (META: need to add this constant).

For example:

```
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;
}
```

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.

1.4.3 pass_brigade

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

```
$ret = $next_f->pass_brigade($bb);
```

- **arg1: `$next_f` (`Apache::Filter`)**

  The next filter in the chain

- **arg2: `$bb` (`APR::Brigade`)**

  The current bucket brigade

- **ret: `$ret` (integer)**

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

  If the bottom-most filter doesn't write to the network, then Apache::NOBODY_WROTE is returned (META: need to add this constant).

The caller relinquishes ownership of the brigade (i.e. it may get destroyed/overwritten/etc by the callee).

For example:

```
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;
}
```

1.5 Streaming Filter API

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

1.5.1 `seen_eos`

This methods returns a true value when the EOS bucket is seen by the `read` method.

```
$ret = $f->seen_eos;
```

- **arg1: `$f` (`Apache::Filter`)**

  The filter to remove

- **ret: `$ret` (integer)**

  a true value if seen, otherwise a false value

This method 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.

1.5.2 `read`

Read data from the filter

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

- **arg1: `$f` (`Apache::Filter`)**
- **arg2: `$buffer` (scalar)**
- **arg3: `$read_len` (integer)**
- **ret: `$ret` (number)**

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

This is a streaming filter method, which acquires a single bucket brigade behind the scenes and reads data from all its buckets. Therefore it can only read from one bucket brigade per filter invocation.

If the EOS bucket is read, the `seen_eos` method will return a true value.

1.5.3 `fputs`

META: Autogenerated - needs to be reviewed/completed

```
$ret = $f->fputs($bb, $str);
```

- **arg1: `$f` (`Apache::Filter`)**
- **arg2: `$bb` (`APR::Brigade`)**
- **arg3: `$str` (string)**
- **ret: `$ret` (integer)**

1.5.4 `print`

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

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

- **arg1: `$f` (`Apache::Filter`)**
- **arg2: `$buffer` (scalar)**
- **ret: XXX**

This method should be used only in streaming filters.

1.6 Other Filter-related API

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

1.6.1 `add_input_filter`

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

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

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

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

- **arg1: `$c` (`Apache::Connection`) or `$r` (`Apache::RequestRec`)**
- **arg2: `&callback` (CODE ref)**
- **ret: XXX**

1.6.2 `add_output_filter`

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

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

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

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

- **arg1: `$c` (`Apache::Connection`) or `$r` (`Apache::RequestRec`)**
- **arg2: `&callback` (CODE ref)**
- **ret: XXX**

1.7 TIE Interface

`Apache::Filter` also implements a tied interface, so you can work with the `$f` object as a hash reference.

META: complete

1.7.1 TIEHANDLE

META: Autogenerated - needs to be reviewed/completed

```
$ret = TIEHANDLE($stashsv, $sv);
```

- **arg1: `$stashsv` (scalar)**
- **arg2: `$sv` (scalar)**
- **ret: `$ret` (scalar)**

1.7.2 PRINT

META: Autogenerated - needs to be reviewed/completed

```
$ret = PRINT(...);
```

- **arg1: `...` (XXX)**
- **ret: `$ret` (integer)**

1.8 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.

1.8.1 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.

1.8.2 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.

1.8.3 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.

1.8.4 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.

1.9 Configuration

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

1.9.1 `PerlInputFilterHandler`

See `PerlInputFilterHandler`.

1.9.2 `PerlOutputFilterHandler`

See `PerlOutputFilterHandler`.

1.9.3 `PerlSetInputFilter`

See `PerlSetInputFilter`.

1.9.4 `PerlSetOutputFilter`

See `PerlSetInputFilter`.

1.10 See Also

mod_perl 2.0 documentation.

1.11 Copyright

1.12 Authors

The mod_perl development team and numerous contributors.

Table of Contents: