1 Input and Output Filters

1.1 Description

This chapter discusses mod_perl's input and output filter handlers.

If all you need is to lookup the filtering API proceed directly to the `Apache::Filter` and `Apache::FilterRec` manpages.

1.2 Your First Filter

You certainly already know how filters work. That's because you encounter filters so often in real life. If you are unfortunate to live in smog-filled cities like Saigon or Bangkok you are probably used to wear a dust filter mask:

If you are smoker, chances are that you smoke cigarettes with filters:

If you are a coffee gourmand, you have certainly tried a filter coffee:

The shower that you use, may have a water filter:

When the sun is too bright, you protect your eyes by wearing sun goggles with UV filter:

If are a photographer you can’t go a step without using filter lenses:

If you love music, you might be unaware of it, but your super-modern audio system is literally loaded with various electronic filters:

There are many more places in our lives where filters are used. The purpose of all filters is to apply some transformation to what's coming into the filter, letting something different out of the filter. Certainly in some cases it's possible to modify the source itself, but that makes things unflexible, and but most of the time we have no control over the source. The advantage of using filters to modify something is that they can be replaced when requirements change Filters also can be stacked, which allows us to make each filter do simple transformations. For example by combining several different filters, we can apply multiple transformations. In certain situations combining several filters of the same kind let's us achieve a better quality output.

The mod_perl filters are not any different, they receive some data, modify it and send it out. In the case of filtering the output of the response handler, we could certainly change the response handler's logic to do something different, since we control the response handler. But this may make the code unnecessary complex. If we can apply transformations to the response handler's output, it certainly gives us more flexibility and simplifies things. For example if a response needs to be compressed before sent out, it'd be very inconvenient and inefficient to code in the response handler itself. Using a filter for that purpose is a perfect solution. Similarly, in certain cases, using an input filter to transform the incoming request data is the most wise solution. Think of the same example of having the incoming data coming compressed.

Just like with real life filters, you can pipe several filters to modify each other's output. You can also customize a selection of different filters at run time.

Without much further ado, let's write a simple but useful obfuscation filter for our HTML documents.

We are going to use a very simple obfuscation -- turn an HTML document into a one liner, which will make it harder to read its source without a special processing. To accomplish that we are going to remove characters \012 (`\n`) and \015 (`\r`), which depending on the platform alone or as a combination represent the end of line and a carriage return.

And here is the filter handler code:

```
#file:MyApache/FilterObfuscate.pm
#--------------------------------
package MyApache::FilterObfuscate;

use strict;
use warnings;
```

```
use Apache::Filter ();
use Apache::RequestRec ();
use APR::Table ();

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

use constant BUFF_LEN => 1024;

sub handler {
    my $f = shift;

    unless ($f->ctx) {
        $f->r->headers_out->unset('Content-Length');
        $f->ctx(1);
    }

    while ($f->read(my $buffer, BUFF_LEN)) {
        $buffer =~ s/[\r\n]//g;
        $f->print($buffer);
    }

    return Apache::OK;
}
1;
```

Next we configure Apache to apply the `MyApache::FilterObfuscate` filter to all requests that get mapped to files with an *".html"* extension:

```
<Files ~ "\.html">
    PerlOutputFilterHandler MyApache::FilterObfuscate
</Files>
```

Filter handlers are similar to HTTP handlers, they are expected to return `Apache::OK` or `Apache::DECLINED`, but instead of receiving `$r` (the request object) as the first argument, they receive `$f` (the filter object).

The filter starts by unsetting of the `Content-Length` response header, because it modifies the length of the response body (shrinks it). If the response handler had set the `Content-Length` header and the filter hasn't unset it, the client may have problems receiving the response since it'd expect more data than it was sent.

The core of this filter is a read-modify-print expression in a while loop. The logic is very simple: read at most `BUFF_LEN` characters of data into `$buffer`, apply the regex to remove any occurences of `\n` and `\r` in it, and print the resulting data out. The input data may come from a response handler, or from an upstream filter. The output data goes to the next filter in the output chain. Even though in this example we haven't configured any more filters, internally Apache by itself uses several core filters to manipulate the data and send it out to the client.

As we are going to explain in great detail in the next sections, the same filter may be called many times during a single request, every time receiving a chunk of data. For example if the POSTed request data is 64k long, an input filter could be invoked 8 times, each time receiving 8k of data. The same may happen during response phase, where an upstream filter may split 64k output in 8 8k chunks. The while loop that

we just saw is going to read each of these 8k in 8 calls, since it requests 1k on every `read()` call.

Since it's enough to unset the `Content-Length` header when the filter is called the first time, we need to have some flag telling us whether we have done the job. The method `ctx()` provides this functionality:

```
unless ($f->ctx) {
    $f->r->headers_out->unset('Content-Length');
    $f->ctx(1);
}
```

the `unset()` call will be made only on the first filter call for each request. Of course you can store any kind of a Perl data structure in `$f->ctx` and retrieve it later in subsequent filter invocations of the same request. We will show plenty of examples using this method in the following sections.

Of course the `MyApache::FilterObfuscate` filter logic should take into account situations where removing new line characters will break the correct rendering, as is the case if there are multi-line `<pre>...</pre>` entries, but since it escalates the complexity of the filter, we will disregard this requirement for now.

A positive side effect of this obfuscation algorithm is in shortening the amount of the data sent to the client. If you want to look at the production ready implementation, which takes into account the HTML markup specifics, the `Apache::Clean` module, available from CPAN, does just that.

mod_perl I/O filtering follows the Perl's principle of making simple things easy and difficult things possible. You have seen that it's trivial to write simple filters. As you read through this tutorial you will see that much more difficult things are possible, even though a more elaborated code will be needed.

1.3 I/O Filtering Concepts

Before introducing the APIs, mod_perl provides for Apache Filtering, there are several important concepts to understand.

1.3.1 Two Methods for Manipulating Data

Apache 2.0 considers all incoming and outgoing data as chunks of information, disregarding their kind and source or storage methods. These data chunks are stored in *buckets*, which form bucket brigades. Input and output filters massage the data in *bucket brigades*. Response and protocol handlers also receive and send data using bucket brigades, though in most cases this is hidden behind wrappers, such as `read()` and `print()`.

mod_perl 2.0 filters can directly manipulate the bucket brigades or use the simplified streaming interface where the filter object acts similar to a filehandle, which can be read from and printed to.

Even though you don't use bucket brigades directly when you use the streaming filter interface (which works on bucket brigades behind the scenes), it's still important to understand bucket brigades. For example you need to know that an output filter will be invoked as many times as the number of bucket brigades sent from an upstream filter or a content handler. Or you need to know that the end of stream

indicator (EOS) is sometimes sent in a separate bucket brigade, so it shouldn't be a surprise that the filter was invoked even though no real data went through. As we delve into the filter details you will see that understanding bucket brigades, will help to understand how filters work.

Moreover you will need to understand bucket brigades if you plan to implement protocol modules.

1.3.2 HTTP Request Versus Connection Filters

HTTP request filters are applied when Apache serves an HTTP request.

HTTP request input filters get invoked on the body of the HTTP request only if the body is consumed by the content handler. HTTP request headers are not passed through the HTTP request input filters.

HTTP response output filters get invoked on the body of the HTTP response if the content handler has generated one. HTTP response headers are not passed through the HTTP response output filters.

Connection level filters are applied at the connection level.

A connection may be configured to serve one or more HTTP requests, or handle other protocols. Connection filters see all the incoming and outgoing data. If an HTTP request is served, connection filters can modify the HTTP headers and the body of request and response. If a different protocol is served over connection (e.g. IMAP), the data could have a completely different pattern, than the HTTP protocol (headers + body).

Apache supports several other filter types, which mod_perl 2.0 may support in the future.

1.3.3 Multiple Invocations of Filter Handlers

Unlike other Apache handlers, filter handlers may get invoked more than once during the same request. Filters get invoked as many times as the number of bucket brigades sent from an upstream filter or a content provider.

For example if a content generation handler sends a string, and then forces a flush, following by more data:

```
# assuming buffered STDOUT ($|==0)
$r->print("foo");
$r->rflush;
$r->print("bar");
```

Apache will generate one bucket brigade with two buckets (there are several types of buckets which contain data, one of them is *transient*):

```
bucket type       data
----------------------
1st    transient   foo
2nd    flush
```

and send it to the filter chain. Then assuming that no more data was sent after `print("bar")`, it will create a last bucket brigade containing data:

```
bucket type       data
----------------------
1st    transient   bar
```

and send it to the filter chain. Finally it'll send yet another bucket brigade with the EOS bucket indicating that there will be no more data sent:

```
bucket type       data
----------------------
1st    eos
```

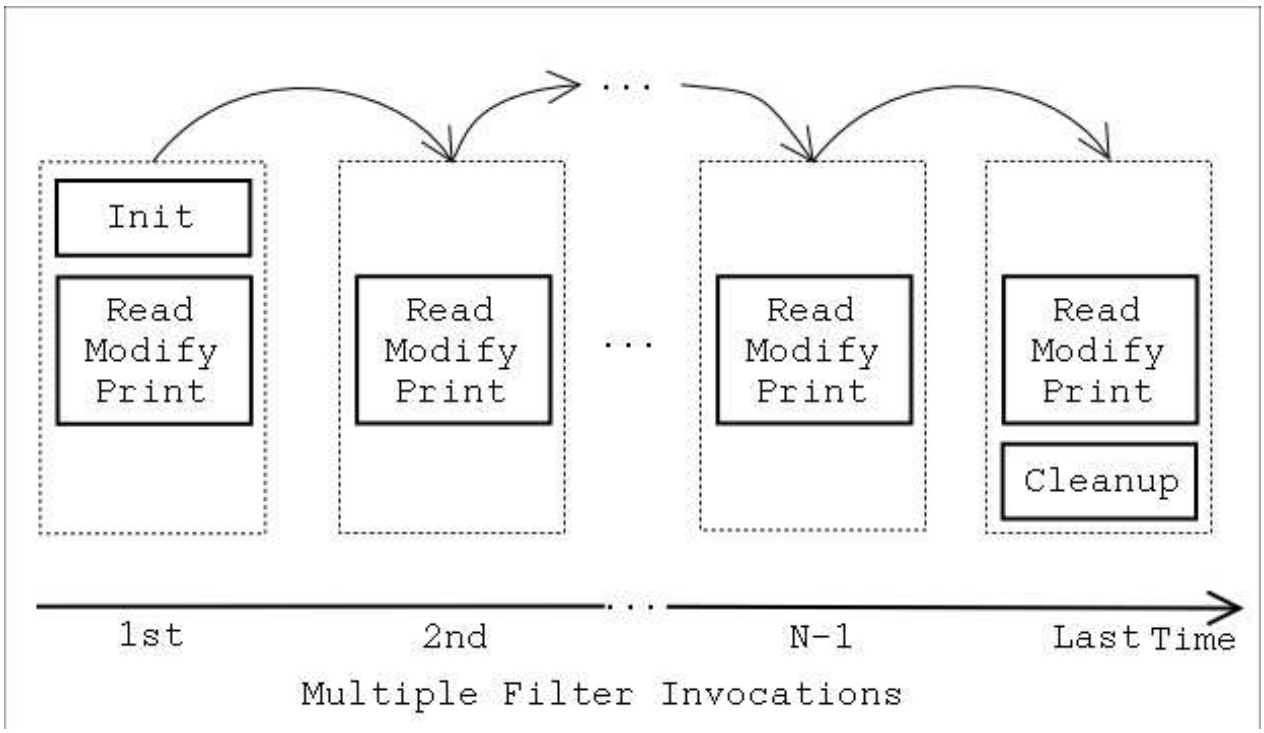
Notice that the EOS bucket may come attached to the last bucket brigade with data, instead of coming in its its own bucket brigade. Filters should never make an assumption that the EOS bucket is arriving alone in a bucket brigade. Therefore the first output filter will be invoked two or three times (three times if EOS is coming in its own brigade), depending on the number of bucket brigades sent by the response handler.

A user may install an upstream filter, and that filter may decide to insert extra bucket brigades or collect all the data in all bucket brigades passing through it and send it all down in one brigade. What's important to remember is when coding a filter, one should never assume that the filter is always going to be invoked once, or a fixed number of times. Neither one can make assumptions on the way the data is going to come in. Therefore a typical filter handler may need to split its logic in three parts.

Jumping ahead we will show some pseudo-code that represents all three parts. This is how a typical stream-oriented filter handler looks like:

```
sub handler {
    my $f = shift;

    # runs on first invocation
    unless ($f->ctx) {
        init($f);
        $f->ctx(1);
    }

    # runs on all invocations
    process($f);

    # runs on the last invocation
    if ($f->seen_eos) {
        finalize($f);
    }

    return Apache::OK;
}
sub init     { ... }
sub process  { ... }
sub finalize { ... }
```

The following diagram depicts all three parts:



Let's explain each part using this pseudo-filter.

1. **Initialization**

   During the initialization, the filter runs all the code that should be performed only once across multiple invocations of the filter (this is during a single request). The filter context is used to accomplish that task. For each new request the filter context is created before the filter is called for the first time and its destroyed at the end of the request.

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

   When the filter is invoked for the first time `$f->ctx` returns `undef` and the custom function init() is called. This function could, for example, retrieve some configuration data, set in *httpd.conf* or initialize some datastructure to its default value.

   To make sure that init() won't be called on the following invocations, we must set the filter context before the first invocation is completed:

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

In practice, the context is not just served as a flag, but used to store real data. For example the following filter handler counts the number of times it was invoked during a single request:

```
sub handler {
    my $f = shift;

    my $ctx = $f->ctx;
    $ctx->{invoked}++;
    $f->ctx($ctx);
    warn "filter was invoked $ctx->{invoked} times\n";

    return Apache::DECLINED;
}
```

Since this filter handler doesn't consume the data from the upstream filter, it's important that this handler returns `Apache::DECLINED`, in which case mod_perl passes the current bucket brigade to the next filter. If this handler returns `Apache::OK`, the data will be simply lost. And if that data included a special EOS token, this may wreck havoc.

Unsetting the `Content-Length` header for filters that modify the response body length is a good example of the code to be used in the initialization phase:

```
unless ($f->ctx) {
    $f->r->headers_out->unset('Content-Length');
    $f->ctx(1);
}
```

We will see more of initialization examples later in this chapter.

2. **Processing**

The next part:

```
process($f);
```

is unconditionally invoked on every filter invocation. That's where the incoming data is read, modified and sent out to the next filter in the filter chain. Here is an example that lowers the case of the characters passing through:

```
use constant READ_SIZE  => 1024;
sub process {
    my $f = shift;
    while ($f->read(my $data, READ_SIZE)) {
        $f->print(lc $data);
    }
}
```

Here the filter operates only on a single bucket brigade. Since it manipulates every character separately the logic is really simple.

In more complicated filters the filters may need to buffer data first before the transformation can be applied. For example if the filter operates on html tokens (e.g., '<img src="me.jpg">'), it's possible that one brigade will include the beginning of the token ('<img ') and the remainder of the token ('src="me.jpg">') will come in the next bucket brigade (on the next filter invocation). In certain cases it may involve more than two bucket brigades to get the whole token. In such a case the filter will have to store the remainder of unprocessed data in the filter context and then reuse it on the next invocation. Another good example is a filter that performs data compression (compression is usually effective only when applied to relatively big chunks of data), so if a single bucket brigade doesn't contain enough data, the filter may need to buffer the data in the filter context till it collects enough of it.

We will see the implementation examples in this chapter.

3. **Finalization**

   Finally, some filters need to know when they are invoked for the last time, in order to perform various cleanups and/or flush any remaining data. As mentioned earlier, Apache indicates this event by a special end of stream "token", represented by a bucket of type `EOS`. If the filter is using the streaming interface, rather than manipulating the bucket brigades directly, and it was calling read() in a while loop, it can check whether this is the last time it's invoked, using the `$f->seen_eos` method:

   ```
   if ($f->seen_eos) {
       finalize($f);
   }
   ```

   This check should be done at the end of the filter handler, because sometimes the EOS "token" comes attached to the tail of data (the last invocation gets both the data and EOS) and sometimes it comes all alone (the last invocation gets only EOS). So if this test is performed at the beginning of the handler and the EOS bucket was sent in together with the data, the EOS event may be missed and filter won't function properly.

   Jumping ahead, filters, directly manipulating bucket brigades, have to look for a bucket whose type is `EOS` to accomplish this. We will see examples later in the chapter.

Some filters may need to deploy all three parts of the described logic, others will need to do only initialization and processing, or processing and finalization, while the simplest filters might perform only the normal processing (as we saw in the example of the filter handler that lowers the case of the characters going through it).

1.3.4 Blocking Calls

All filters (excluding the core filter that reads from the network and the core filter that writes to it) block at least once when invoked. Depending on whether this is an input or an output filter, the blocking happens when the bucket brigade is requested from the upstream filter or when the bucket brigade is passed to the downstream filter.

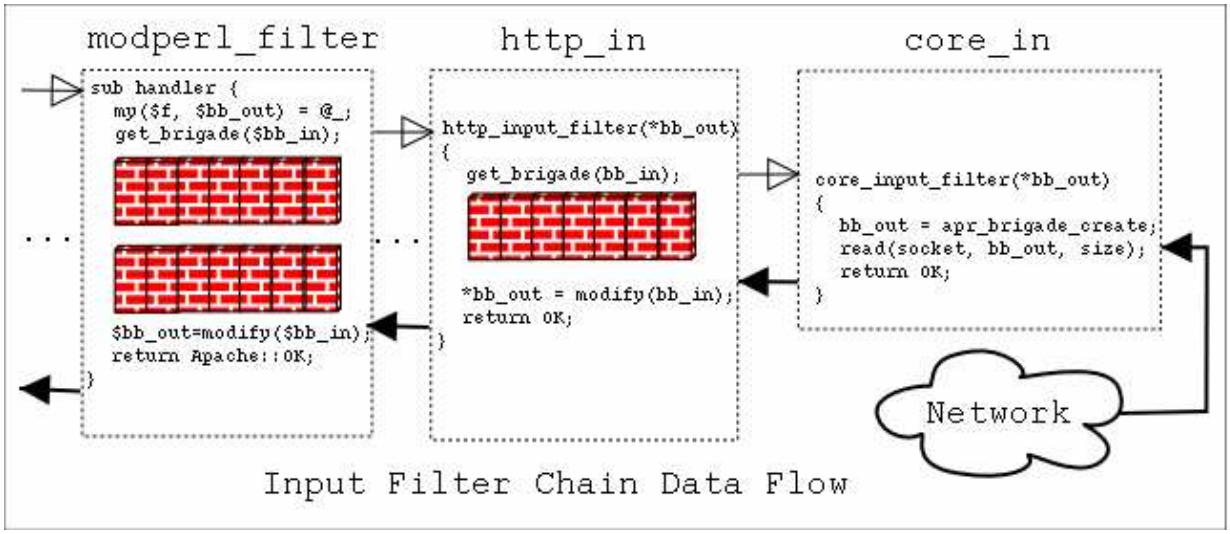
First of all, the input and output filters differ in the ways they acquire the bucket brigades (which includes the data that they filter). Even though when a streaming API is used the difference can't be seen, it's important to understand how things work underneath. Therefore we are going to show examples of transparent filters, which pass data through them unmodified. Instead of reading the data in and printing it out the bucket brigades are now passed as is.

Here is a code for a transparent input filter:

```
#file:MyApache/FilterTransparent.pm (first part)
#-----------------------------------------------
package MyApache::FilterTransparent;

use Apache::Const -compile => qw(OK);
use APR::Const -compile => ':common';

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

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

    return Apache::OK;
}
```

When the input filter *in()* is invoked, it first asks the upstream filter for the next bucket brigade (using the `get_brigade()` call). That upstream filter is in turn going to ask for the bucket brigade from the next upstream filter in chain, etc., till the last filter (called `core_in`), that reads from the network is reached. The `core_in` filter reads, using a socket, a portion of the incoming data from the network, processes it and sends it to its downstream filter, which will process the data and send it to its downstream filter, etc., till it reaches the very first filter who has asked for the data. (In reality some other handler triggers the request for the bucket brigade, e.g., an HTTP response handler, or a protocol module, but for our discussion it's good enough to assume that it's the first filter that issues the `get_brigade()` call.)

The following diagram depicts a typical input filters chain data flow in addition to the program control flow.



The black- and white-headed arrows show when the control is switched from one filter to another. In addition the black-headed arrows show the actual data flow. The diagram includes some pseudo-code, both for in Perl for the mod_perl filters and in C for the internal Apache filters. You don't have to understand C to understand this diagram. What's important to understand is that when input filters are invoked they first call each other via the `get_brigade()` call and then block (notice the brick wall on the diagram), waiting for the call to return. When this call returns all upstream filters have already completed finishing their filtering task.

As mentioned earlier, the streaming interface hides these details, however the first `$f->read()` call will block, as underneath it performs the `get_brigade()` call.

The diagram shows a part of the actual input filter chain for an HTTP request, the `...` shows that there are more filters in between the mod_perl filter and `http_in`.

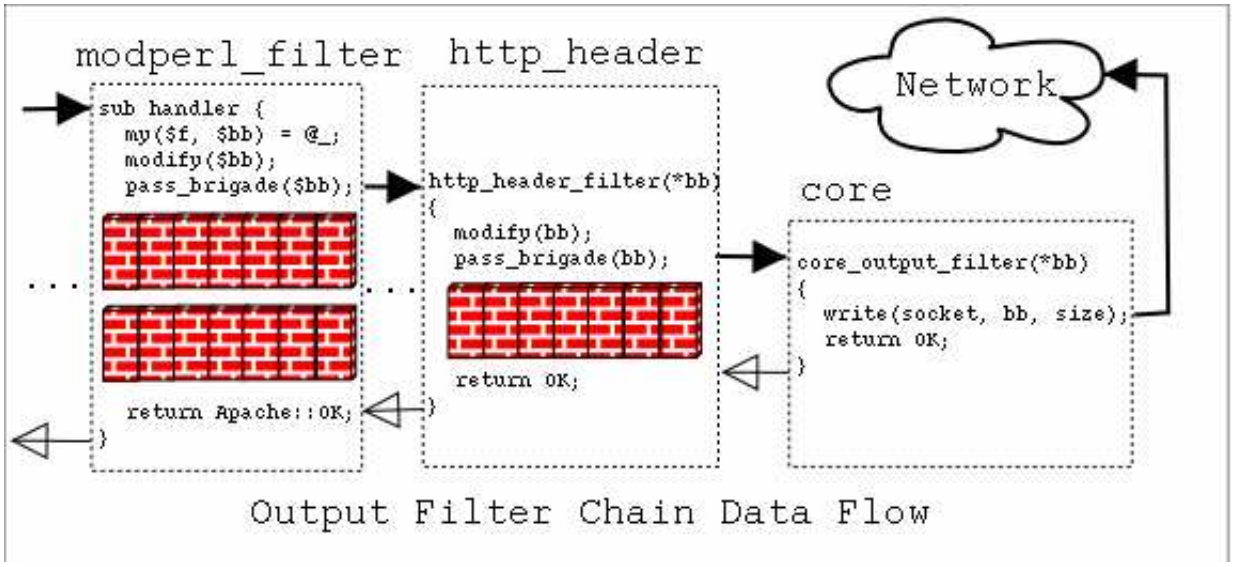
Now let's look at what happens in the output filters chain. Here the first filter acquires the bucket brigades containing the response data, from the content handler (or another protocol handler if we aren't talking HTTP), it then may apply some modification and pass the data to the next filter (using the `pass_brigade()` call), which in turn applies its modifications and sends the bucket brigade to the next filter, etc., all the way down to the last filter (called `core`) which writes the data to the network, via the socket the client is listening to. Even though the output filters don't have to wait to acquire the bucket brigade (since the upstream filter passes it to them as an argument), they still block in a similar fashion to input filters, since they have to wait for the `pass_brigade()` call to return.

Here is an example of a transparent output filter:

```
#file:MyApache/FilterTransparent.pm (continued)
#-----------------------------------------------
sub out {
    my ($f, $bb) = @_;

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

    return Apache::OK;
}
1;
```

The *out()* filter passes `$bb` to the downstream filter unmodified and if you add debug prints before and after the `pass_brigade()` call and configure the same filter twice, the debug print will show the blocking call.

The following diagram depicts a typical output filters chain data flow in addition to the program control flow:



Similar to the input filters chain diagram, the arrows show the program control flow and in addition the black-headed arrows show the data flow. Again, it uses a Perl pseudo-code for the mod_perl filter and C pseudo-code for the Apache filters, similarly the brick walls represent the waiting. And again, the diagram shows a part of the real HTTP response filters chain, where `...` stands for the omitted filters.

1.4 mod_perl Filters Declaration and Configuration

Now let's see how mod_perl filters are declared and configured.

1.4.1 Filter Priority Types

When Apache filters are configured they are inserted into the filters chain according to their priority/type. In most cases when using one or two filters things will just work, however if you find that the order of filter invocation is wrong, the filter priority type should be consulted. Unfortunately this information is available only by consulting the source code, unless it's documented in the module man pages. Numerical definitions of priority types, such as `AP_FTYPE_CONTENT_SET`, `AP_FTYPE_RESOURCE`, can be found in *include/util_filter.h*.

As of this writing Apache comes with two core filters: `DEFLATE` and `INCLUDES`. For example in the following configuration:

```
SetOutputFilter DEFLATE
SetOutputFilter INCLUDES
```

the `DEFLATE` filter will be inserted in the filters chain after the `INCLUDES` filter, even though it was configured before it. This is because the `DEFLATE` filter is of type `AP_FTYPE_CONTENT_SET` (20), whereas the `INCLUDES` filter is of type `AP_FTYPE_RESOURCE` (10).

As of this writing mod_perl provides two kind of filters with fixed priority type:

```
Handler                  Priority           Value
-------------------------------------------------
FilterRequestHandler     AP_FTYPE_RESOURCE    10
FilterConnectionHandler  AP_FTYPE_PROTOCOL    30
```

Therefore `FilterRequestHandler` filters (10) will be always invoked before the `DEFLATE` filter (20), whereas `FilterConnectionHandler` filters (30) after it. The `INCLUDES` filter (10) has the same priority as `FilterRequestHandler` filters (10), and therefore it'll be inserted according to the configuration order, when `PerlSetOutputFilter` or `PerlSetInputFilter` is used.

1.4.2 `PerlInputFilterHandler`

The `PerlInputFilterHandler` directive registers a filter, and inserts it into the relevant input filters chain.

This handler is of type `VOID`.

The handler's configuration scope is `DIR`.

The following sections include several examples that use the `PerlInputFilterHandler` handler.

1.4.3 `PerlOutputFilterHandler`

The `PerlOutputFilterHandler` directive registers a filter, and inserts it into the relevant output filters chain.

This handler is of type VOID.

The handler's configuration scope is DIR.

The following sections include several examples that use the PerlOutputFilterHandler handler.

1.4.4 PerlSetInputFilter

The SetInputFilter directive, documented at *http://httpd.apache.org/docs-2.0/mod/core.html#setinputfilter* sets the filter or filters which will process client requests and POST input when they are received by the server (in addition to any filters configured earlier).

To mix mod_perl and non-mod_perl input filters of the same priority nothing special should be done. For example if we have an imaginary Apache filter FILTER_FOO and mod_perl filter MyApache::FilterInputFoo, this configuration:

```
SetInputFilter FILTER_FOO
PerlInputFilterHandler MyApache::FilterInputFoo
```

will add both filters, however the order of their invocation might be not the one that you've expected. To make the invocation order the same as the insertion order replace SetInputFilter with PerlSet-InputFilter, like so:

```
PerlSetInputFilter FILTER_FOO
PerlInputFilterHandler MyApache::FilterInputFoo
```

now FILTER_FOO filter will be always executed before the MyApache::FilterInputFoo filter, since it was configured before MyApache::FilterInputFoo (i.e., it'll apply its transformations on the incoming data last). Here is a diagram input filters chain and the data flow from the network to the response handler for the presented configuration:

```
      response handler
            /\
            ||
        FILTER_FOO
            /\
            ||
MyApache::FilterInputFoo
            /\
            ||
  core input filters
            /\
            ||
          network
```

As explained in the section Filter Priority Types this directive won't affect filters of different priority. For example assuming that MyApache::FilterInputFoo is a FilterRequestHandler filter, the configurations:

```
PerlInputFilterHandler MyApache::FilterInputFoo
PerlSetInputFilter DEFLATE
```

and

```
PerlSetInputFilter DEFLATE
PerlInputFilterHandler MyApache::FilterInputFoo
```

are equivalent, because mod_deflate's DEFLATE filter has a higher priority than MyApache::Filter-InputFoo, thefore it'll always be inserted into the filter chain after MyApache::FilterInputFoo, (i.e. the DEFLATE filter will apply its transformations on the incoming data first). Here is a diagram input filters chain and the data flow from the network to the response handler for the presented configuration:

```
    response handler
           /\
           ||
 MyApache::FilterInputFoo
           /\
           ||
        DEFLATE
           /\
           ||
   core input filters
           /\
           ||
         network
```

SetInputFilter's ; semantics are supported as well. For example, in the following configuration:

```
PerlInputFilterHandler MyApache::FilterInputFoo
PerlSetInputFilter FILTER_FOO;FILTER_BAR
```

MyApache::FilterOutputFoo will be executed first, followed by FILTER_FOO and finally by FILTER_BAR (again, assuming that all three filters have the same priority).

The PerlSetInputFilter directives's configuration scope is DIR.

1.4.5 PerlSetOutputFilter

The SetOutputFilter directive, documented at *http://httpd.apache.org/docs-2.0/mod/core.html#setoutputfilter* sets the filters which will process responses from the server before they are sent to the client (in addition to any filters configured earlier).

To mix mod_perl and non-mod_perl output filters of the same priority nothing special should be done. This configuration:

```
SetOutputFilter INCLUDES
PerlOutputFilterHandler MyApache::FilterOutputFoo
```

will add all two filters to the filter chain, however the order of their invocation might be not the one that you've expected. To preserve the insertion order replace SetOutputFilter with PerlSetOutput-Filter, like so:

```
PerlSetOutputFilter INCLUDES
PerlOutputFilterHandler MyApache::FilterOutputFoo
```

now mod_include's `INCLUDES` filter will be always executed before the `MyApache::FilterOutputFoo` filter. Here is a diagram input filters chain and the data flow from the response handler to the network for the presented configuration:

```
    response handler
           ||
           \/
        INCLUDES
           ||
           \/
MyApache::FilterOutputFoo
           ||
           \/
  core output filters
           ||
           \/
         network
```

`SetOutputFilter`'s `;` semantics are supported as well. For example, in the following configuration:

```
PerlOutputFilterHandler MyApache::FilterOutputFoo
PerlSetOutputFilter INCLUDES;FILTER_FOO
```

`MyApache::FilterOutputFoo` will be executed first, followed by `INCLUDES` and finally by `FILTER_FOO` (again, assuming that all three filters have the same priority).

Just as explained in the `PerlSetInputFilter` section, if filters have different priorities, the insertion order might be different. For example in the following configuration:

```
PerlSetOutputFilter DEFLATE
PerlSetOutputFilter INCLUDES
PerlOutputFilterHandler MyApache::FilterOutputFoo
```

mod_include's `INCLUDES` filter will be always executed before the `MyApache::FilterOutputFoo` filter. The latter will be followed by mod_deflate's `DEFLATE` filter, even though it was configured before the other two filters. This is because it has a higher priority. And the corresponding diagram looks like so:

```
    response handler
           ||
           \/
        INCLUDES
           ||
           \/
MyApache::FilterOutputFoo
           ||
           \/
         DEFLATE
           ||
           \/
```

```
    core output filters
            ||
            \/
          network
```

The `PerlSetOutputFilter` directives's configuration scope is `DIR`.

1.4.6 HTTP Request vs. Connection Filters

mod_perl 2.0 supports connection and HTTP request filtering. mod_perl filter handlers specify the type of the filter using the method attributes.

HTTP request filter handlers are declared using the `FilterRequestHandler` attribute. Consider the following request input and output filters skeleton:

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

sub input  : FilterRequestHandler {
    my($f, $bb, $mode, $block, $readbytes) = @_;
    #...
}

sub output : FilterRequestHandler {
    my($f, $bb) = @_;
    #...
}

1;
```

If the attribute is not specified, the default `FilterRequestHandler` attribute is assumed. Filters specifying subroutine attributes must subclass `Apache::Filter`, others only need to:

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

The request filters are usually configured in the `<Location>` or equivalent sections:

```
PerlModule MyApache::FilterRequestFoo
PerlModule MyApache::NiceResponse
<Location /filter_foo>
    SetHandler modperl
    PerlResponseHandler     MyApache::NiceResponse
    PerlInputFilterHandler  MyApache::FilterRequestFoo::input
    PerlOutputFilterHandler MyApache::FilterRequestFoo::output
</Location>
```

Now we have the request input and output filters configured.

The connection filter handler uses the `FilterConnectionHandler` attribute. Here is a similar example for the connection input and output filters.

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

sub input  : FilterConnectionHandler {
    my($f, $bb, $mode, $block, $readbytes) = @_;
    #...
}

sub output : FilterConnectionHandler {
    my($f, $bb) = @_;
    #...
}

1;
```

This time the configuration must be done outside the `<Location>` or equivalent sections, usually within the `<VirtualHost>` or the global server configuration:

```
Listen 8005
<VirtualHost _default_:8005>
    PerlModule MyApache::FilterConnectionBar
    PerlModule MyApache::NiceResponse

    PerlInputFilterHandler  MyApache::FilterConnectionBar::input
    PerlOutputFilterHandler MyApache::FilterConnectionBar::output
    <Location />
        SetHandler modperl
        PerlResponseHandler MyApache::NiceResponse
    </Location>

</VirtualHost>
```

This accomplishes the configuration of the connection input and output filters.

Notice that for HTTP requests the only difference between connection filters and request filters is that the former see everything: the headers and the body, whereas the latter see only the body.

mod_perl provides two interfaces to filtering: a direct bucket brigades manipulation interface and a simpler, stream-oriented interface. The examples in the following sections will help you to understand the difference between the two interfaces.

1.4.7 Filter Initialization Phase

Like in any cool application, there is a hidden door, that let’s you do cool things. mod_perl is not an exception.

where you can plug yet another callback. This *init* callback runs immediately after the filter handler is inserted into the filter chain, before it was invoked for the first time. Here is a skeleton of an init handler:

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

The attribute `FilterInitHandler` marks the Perl 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.

For example you may decide to dynamically remove a filter before it had a chance to run, if some condition is true:

```
sub init : FilterInitHandler {
    my $f = shift;
    $f->remove() if should_remove_filter();
    return Apache::OK;
}
```

Not all `Apache::Filter` methods can be used in the init handler, because it's not a filter. Hence you can use methods that operate on the filter itself, such as `remove()` and `ctx()` or retrieve request information, such as `r()` and `c()`. But not methods that operate on data, such as `read()` and `print()`.

In order to hook an init filter handler, the real filter has to assign this callback using the `FilterHasInitHandler` which accepts a reference to the callback function, similar to `push_handlers()`. 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;
}
```

While attributes are parsed during the code compilation (it's really a sort of source filter), the argument to the `FilterHasInitHandler()` attribute is compiled at a later stage once the module is compiled.

The argument to `FilterHasInitHandler()` can be any Perl code which when `eval()`'ed returns a code reference. For example:

```
package MyApache::OtherFilter;
use base qw(Apache::Filter);
sub init  : FilterInitHandler { ... }

package MyApache::FilterBar;
use MyApache::OtherFilter;
use base qw(Apache::Filter);
sub get_pre_handler { \&MyApache::OtherFilter::init }
sub filter : FilterHasInitHandler(get_pre_handler()) { ... }
```

Here the `MyApache::FilterBar::filter` handler is configured to run the `MyApache::Other-Filter::init` init handler.

Notice that the argument to `FilterHasInitHandler()` is always `eval()`'ed in the package of the real filter handler (not the init handler). So the above code leads to the following evaluation:

```
$init_sub = eval "package MyApache::FilterBar; get_pre_handler()";
```

though, this is done in C, using the `eval_pv()` C call.

META: currently only one initialization callback can be registered per filter handler. If the need to register more than one arises it should be very easy to extend the functionality.

1.5 All-in-One Filter

Before we delve into the details of how to write filters that do something with the data, lets first write a simple filter that does nothing but snooping on the data that goes through it. We are going to develop the `MyApache::FilterSnoop` handler which can snoop on request and connection filters, in input and output modes.

But first let's develop a simple response handler that simply dumps the request's *args* and *content* as strings:

```
file:MyApache/Dump.pm
---------------------
package MyApache::Dump;

use strict;
use warnings;

use Apache::RequestRec ();
use Apache::RequestIO ();
use APR::Table ();

use Apache::Const -compile => qw(OK M_POST);

sub handler {
    my $r = shift;
    $r->content_type('text/plain');

    $r->print("args:\n", $r->args, "\n");

    if ($r->method_number == Apache::M_POST) {
        my $data = content($r);
        $r->print("content:\n$data\n");
    }

    return Apache::OK;
}

sub content {
    my $r = shift;
```

```
    $r->setup_client_block;

    return '' unless $r->should_client_block;

    my $len = $r->headers_in->get('content-length');
    my $buf;
    $r->get_client_block($buf, $len);

    return $buf;
}


1;
```

which is configured as:

```
PerlModule MyApache::Dump
<Location /dump>
    SetHandler modperl
    PerlResponseHandler MyApache::Dump
</Location>
```

If we issue the following request:

```
% echo "mod_perl rules" | POST 'http://localhost:8002/dump?foo=1&bar=2'
```

the response will be:

```
args:
foo=1&bar=2
content:
mod_perl rules
```

As you can see it simply dumped the query string and the posted data.

Now let's write the snooping filter:

```
file:MyApache/FilterSnoop.pm
----------------------------
package MyApache::FilterSnoop;

use strict;
use warnings;

use base qw(Apache::Filter);
use Apache::FilterRec ();
use APR::Brigade ();
use APR::Bucket ();

use Apache::Const -compile => qw(OK DECLINED);
use APR::Const -compile => ':common';

sub connection : FilterConnectionHandler { snoop("connection", @_) }
sub request    : FilterRequestHandler    { snoop("request",    @_) }

sub snoop {
```

```
    my $type = shift;
    my($f, $bb, $mode, $block, $readbytes) = @_; # filter args

    # $mode, $block, $readbytes are passed only for input filters
    my $stream = defined $mode ? "input" : "output";

    # read the data and pass-through the bucket brigades unchanged
    if (defined $mode) {
        # input filter
        my $rv = $f->next->get_brigade($bb, $mode, $block, $readbytes);
        return $rv unless $rv == APR::SUCCESS;
        bb_dump($type, $stream, $bb);
    }
    else {
        # output filter
        bb_dump($type, $stream, $bb);
        my $rv = $f->next->pass_brigade($bb);
        return $rv unless $rv == APR::SUCCESS;
    }

    return Apache::OK;

}

sub bb_dump {
    my($type, $stream, $bb) = @_;

    my @data;
    for (my $b = $bb->first; $b; $b = $bb->next($b)) {
        $b->read(my $bdata);
        $bdata = '' unless defined $bdata;
        push @data, $b->type->name, $bdata;
    }

    # send the sniffed info to STDERR so not to interfere with normal
    # output
    my $direction = $stream eq 'output' ? ">>>" : "<<<";
    print STDERR "\n$direction $type $stream filter\n";

    my $c = 1;
    while (my($btype, $data) = splice @data, 0, 2) {
        print STDERR "    o bucket $c: $btype\n";
        print STDERR "[$data]\n";
        $c++;
    }
}
1;
```

This package provides two filter handlers, one for connection and another for request filtering:

```
sub connection : FilterConnectionHandler { snoop("connection", @_) }
sub request    : FilterRequestHandler    { snoop("request",    @_) }
```

Both handlers forward their arguments to the `snoop()` function that does the real job. We needed to add these two subroutines in order to assign the two different attributes. Plus the functions pass the filter type to `snoop()` as the first argument, which gets shifted off `@_` and the rest of the `@_` are the arguments that

were originally passed to the filter handler.

It's easy to know whether a filter handler is running in the input or the output mode. The arguments `$f` and `$bb` are always passed, whereas the arguments `$mode`, `$block`, and `$readbytes` are passed only to input filter handlers.

If we are in the input mode, in the same call we retrieve the bucket brigade from the previous filter on the input filters stack and immediately link it to the `$bb` variable which makes the bucket brigade available to the next input filter when the filter handler returns. If we forget to perform this linking our filter will become a black hole in which data simply disappears. Next we call `bb_dump()` which dumps the type of the filter and the contents of the bucket brigade to `STDERR`, without influencing the normal data flow.

If we are in the output mode, the `$bb` variable already points to the current bucket brigade. Therefore we can read the contents of the brigade right away. After that we pass the brigade to the next filter.

Let's snoop on connection and request filter levels in both directions by applying the following configuration:

```
Listen 8008
<VirtualHost _default_:8008>
    PerlModule MyApache::FilterSnoop
    PerlModule MyApache::Dump

    # Connection filters
    PerlInputFilterHandler  MyApache::FilterSnoop::connection
    PerlOutputFilterHandler MyApache::FilterSnoop::connection

    <Location /dump>
        SetHandler modperl
        PerlResponseHandler MyApache::Dump
        # Request filters
        PerlInputFilterHandler  MyApache::FilterSnoop::request
        PerlOutputFilterHandler MyApache::FilterSnoop::request
    </Location>

</VirtualHost>
```

Notice that we use a virtual host because we want to install connection filters.

If we issue the following request:

```
% echo "mod_perl rules" | POST 'http://localhost:8008/dump?foo=1&bar=2'
```

We get the same response, when using `MyApache::FilterSnoop`, because our snooping filter didn't change anything. Though there was a lot of output printed to *error_log*. We present it all here, since it helps a lot to understand how filters work.

First we can see the connection input filter at work, as it processes the HTTP headers. We can see that for this request each header is put into a separate brigade with a single bucket. The data is conveniently enclosed by `[]` so you can see the new line characters as well.

```
<<< connection input filter
    o bucket 1: HEAP
[POST /dump?foo=1&bar=2 HTTP/1.1
]

<<< connection input filter
    o bucket 1: HEAP
[TE: deflate,gzip;q=0.3
]

<<< connection input filter
    o bucket 1: HEAP
[Connection: TE, close
]

<<< connection input filter
    o bucket 1: HEAP
[Host: localhost:8008
]

<<< connection input filter
    o bucket 1: HEAP
[User-Agent: lwp-request/2.01
]

<<< connection input filter
    o bucket 1: HEAP
[Content-Length: 14
]

<<< connection input filter
    o bucket 1: HEAP
[Content-Type: application/x-www-form-urlencoded
]

<<< connection input filter
    o bucket 1: HEAP
[
]
```

Here the HTTP header has been terminated by a double new line. So far all the buckets were of the *HEAP* type, meaning that they were allocated from the heap memory. Notice that the HTTP request input filters will never see the bucket brigades with HTTP headers, as it has been consumed by the last core connection filter.

The following two entries are generated when `MyApache::Dump::handler` reads the POSTed content:

```
<<< connection input filter
    o bucket 1: HEAP
[mod_perl rules]

<<< request input filter
    o bucket 1: HEAP
[mod_perl rules]
    o bucket 2: EOS
[]
```

as we saw earlier on the diagram, the connection input filter is run before the request input filter. Since our connection input filter was passing the data through unmodified and no other custom connection input filter was configured, the request input filter sees the same data. The last bucket in the brigade received by the request input filter is of type *EOS*, meaning that all the input data from the current request has been received.

Next we can see that `MyApache::Dump::handler` has generated its response. However we can see that only the request output filter gets run at this point:

```
>>> request output filter
    o bucket 1: TRANSIENT
[args:
foo=1&bar=2
content:
mod_perl rules
]
```

This happens because Apache hasn't sent yet the response HTTP headers to the client. The request filter sees a bucket brigade with a single bucket of type *TRANSIENT* which is allocated from the stack memory.

The moment the first bucket brigade of the response body has entered the connection output filters, Apache injects a bucket brigade with the HTTP headers. Therefore we can see that the connection output filter is filtering the brigade with HTTP headers (notice that the request output filters don't see it):

```
>>> connection output filter
    o bucket 1: HEAP
[HTTP/1.1 200 OK
Date: Tue, 19 Nov 2002 15:59:32 GMT
Server: Apache/2.0.44-dev (Unix) mod_perl/1.99_08-dev
Perl/v5.8.0 mod_ssl/2.0.44-dev OpenSSL/0.9.6d DAV/2
Connection: close
Transfer-Encoding: chunked
Content-Type: text/plain; charset=ISO-8859-1

]
```

and followed by the first response body's brigade:

```
>>> connection output filter
    o bucket 1: TRANSIENT
[2b
]
    o bucket 2: TRANSIENT
[args:
```

```
foo=1&bar=2
content:
mod_perl rules

]
    o bucket 3: IMMORTAL
[
]
```

If the response is large, the request and connection filters will filter chunks of the response one by one.

META: what's the size of the chunks? 8k?

Finally, Apache sends a series of the bucket brigades to finish off the response, including the end of stream meta-bucket to tell filters that they shouldn't expect any more data, and flush buckets to flush the data, to make sure that any buffered output is sent to the client:

```
>>> connection output filter
    o bucket 1: IMMORTAL
[0

]
    o bucket 2: EOS
[]

>>> connection output filter
    o bucket 1: FLUSH
[]

>>> connection output filter
    o bucket 1: FLUSH
[]
```

This module helps to understand that each filter handler can be called many time during each request and connection. It's called for each bucket brigade.

Also it's important to mention that HTTP request input filters are invoked only if there is some POSTed data to read and it's consumed by a content handler.

1.6 Input Filters

mod_perl supports Connection and HTTP Request input filters:

1.6.1 Connection Input Filters

Let's say that we want to test how our handlers behave when they are requested as `HEAD` requests, rather than `GET`. We can alter the request headers at the incoming connection level transparently to all handlers.

This example's filter handler looks for data like:

```
GET /perl/test.pl HTTP/1.1
```

and turns it into:

```
HEAD /perl/test.pl HTTP/1.1
```

The following input filter handler does that by directly manipulating the bucket brigades:

```
file:MyApache/InputFilterGET2HEAD.pm
------------------------------------
package MyApache::InputFilterGET2HEAD;

use strict;
use warnings;

use base qw(Apache::Filter);

use APR::Brigade ();
use APR::Bucket ();

use Apache::Const -compile => 'OK';
use APR::Const    -compile => ':common';

sub handler : FilterConnectionHandler {
    my($f, $bb, $mode, $block, $readbytes) = @_;

    return Apache::DECLINED if $f->ctx;

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

    for (my $b = $bb->first; $b; $b = $bb->next($b)) {
        my $data;
        my $status = $b->read($data);
        return $status unless $status == APR::SUCCESS;
        warn("data: $data\n");

        if ($data and $data =~ s|^GET|HEAD|) {
            my $bn = APR::Bucket->new($data);
            $b->insert_after($bn);
            $b->remove; # no longer needed
            $f->ctx(1); # flag that that we have done the job
            last;
        }
    }

    Apache::OK;
}

1;
```

The filter handler is called for each bucket brigade, which in turn includes buckets with data. The gist of any input filter handler is to request the bucket brigade from the upstream filter, and return it downstream filter using the second argument `$bb`. It's important to remember that you can call methods on this argument, but you shouldn't assign to this argument, or the chain will be broken. You have two techniques to

choose from to retrieve-modify-return bucket brigades:

1. Create a new empty bucket brigade `$ctx_bb`, pass it to the upstream filter via `get_brigade()` and wait for this call to return. When it returns, `$ctx_bb` is populated with buckets. Now the filter should move the bucket from `$ctx_bb` to `$bb`, on the way modifying the buckets if needed. Once the buckets are moved, and the filter returns, the downstream filter will receive the populated bucket brigade.

2. Pass `$bb` to `get_brigade()` to the upstream filter, so it will be populated with buckets. Once `get_brigade()` returns, the filter can go through the buckets and modify them in place, or it can do nothing and just return (in which case, the downstream filter will receive the bucket brigade unmodified).

Both techniques allow addition and removal of buckets. Though the second technique is more efficient since it doesn't have the overhead of create the new brigade and moving the bucket from one brigade to another. In this example we have chosen to use the second technique, in the next example we will see the first technique.

Our filter has to perform the substitution of only one HTTP header (which normally resides in one bucket), so we have to make sure that no other data gets mangled (e.g. there could be POSTED data and it may match `/^GET/` in one of the buckets). We use `$f->ctx` as a flag here. When it's undefined the filter knows that it hasn't done the required substitution, though once it completes the job it sets the context to 1.

To optimize the speed, the filter immediately returns `Apache::DECLINED` when it's invoked after the substitution job has been done:

```
return Apache::DECLINED if $f->ctx;
```

In that case mod_perl will call `get_brigade()` internally which will pass the bucket brigade to the downstream filter. Alternatively the filter could do:

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

but this is a bit less efficient.

[META: the most efficient thing to do is to remove the filter itself once the job is done, so it won't be even invoked after the job has been done.

```
if ($f->ctx) {
    $f->remove;
    return Apache::DECLINED;
}
```

However, this can't be used with Apache 2.0.46 and lower, since it has a bug when trying to remove the edge connection filter (it doesn't remove it). Don't know if it's going to be fixed in 2.0.47]

If the job wasn't done yet, the filter calls `get_brigade`, which populates the `$bb` bucket brigade. Next, the filter steps through the buckets looking for the bucket that matches the regex: `/^GET/`. If that happens, a new bucket is created with the modified data (`s/^GET/HEAD/`. Now it has to be inserted in place of the old bucket. In our example we insert the new bucket after the bucket that we have just modified and immediately remove that bucket that we don't need anymore:

```
$b->insert_after($bn);
$b->remove; # no longer needed
```

Finally we set the context to 1, so we know not to apply the substitution on the following data and break from the *for* loop.

The handler returns `Apache::OK` indicating that everything was fine. The downstream filter will receive the bucket brigade with one bucket modified.

Now let's check that the handler works properly. For example, consider the following response handler:

```
file:MyApache/RequestType.pm
----------------------------
package MyApache::RequestType;

use strict;
use warnings;

use Apache::RequestIO ();
use Apache::RequestRec ();
use Apache::Response ();

use Apache::Const -compile => 'OK';

sub handler {
    my $r = shift;

    $r->content_type('text/plain');
    my $response = "the request type was " . $r->method;
    $r->set_content_length(length $response);
    $r->print($response);

    Apache::OK;
}

1;
```

which returns to the client the request type it has issued. In the case of the `HEAD` request Apache will discard the response body, but it'll will still set the correct `Content-Length` header, which will be 24 in case of the `GET` request and 25 for `HEAD`. Therefore if this response handler is configured as:

```
Listen 8005
<VirtualHost _default_:8005>
    <Location />
        SetHandler modperl
        PerlResponseHandler +MyApache::RequestType
    </Location>
</VirtualHost>
```

and a GET request is issued to /:

```
panic% perl -MLWP::UserAgent -le \
'$r = LWP::UserAgent->new()->get("http://localhost:8005/"); \
print $r->headers->content_length . ": ".  $r->content'
24: the request type was GET
```

where the response's body is:

```
the request type was GET
```

And the Content-Length header is set to 24.

However if we enable the MyApache::InputFilterGET2HEAD input connection filter:

```
Listen 8005
<VirtualHost _default_:8005>
    PerlInputFilterHandler +MyApache::InputFilterGET2HEAD

    <Location />
        SetHandler modperl
        PerlResponseHandler +MyApache::RequestType
    </Location>
</VirtualHost>
```

And issue the same GET request, we get only:

```
25:
```

which means that the body was discarded by Apache, because our filter turned the GET request into a HEAD request and if Apache wasn't discarding the body on HEAD, the response would be:

```
the request type was HEAD
```

that's why the content length is reported as 25 and not 24 as in the real GET request.

1.6.2 HTTP Request Input Filters

Request filters are really non-different from connection filters, other than that they are working on request and response bodies and have an access to a request object.

1.6.3 Bucket Brigade-based Input Filters

Let's look at the request input filter that lowers the case of the request's body: MyApache::InputRequestFilterLC:

```
file:MyApache/InputRequestFilterLC.pm
-------------------------------------
package MyApache::InputRequestFilterLC;

use strict;
use warnings;
```

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

use Apache::Connection ();
use APR::Brigade ();
use APR::Bucket ();

use Apache::Const -compile => 'OK';
use APR::Const    -compile => ':common';

sub handler : FilterRequestHandler {
    my($f, $bb, $mode, $block, $readbytes) = @_;

    my $c = $f->c;
    my $bb_ctx = APR::Brigade->new($c->pool, $c->bucket_alloc);
    my $rv = $f->next->get_brigade($bb_ctx, $mode, $block, $readbytes);
    return $rv unless $rv == APR::SUCCESS;

    while (!$bb_ctx->empty) {
        my $b = $bb_ctx->first;

        $b->remove;

        if ($b->is_eos) {
            $bb->insert_tail($b);
            last;
        }

        my $data;
        my $status = $b->read($data);
        return $status unless $status == APR::SUCCESS;

        $b = APR::Bucket->new(lc $data) if $data;

        $bb->insert_tail($b);
    }

    Apache::OK;
}

1;
```

As promised, in this filter handler we have used the first technique of bucket brigade modification. The handler creates a temporary bucket brigade (`ctx_bb`), populates it with data using `get_brigade()`, and then moves buckets from it to the bucket brigade `$bb`, which is then retrieved by the downstream filter when our handler returns.

This filter doesn't need to know whether it was invoked for the first time or whether it has already done something. It's state-less handler, since it has to lower case everything that passes through it. Notice that this filter can't be used as the connection filter for HTTP requests, since it will invalidate the incoming request headers; for example the first header line:

```
GET /perl/TEST.pl HTTP/1.1
```

will become:

```
get /perl/test.pl http/1.1
```

which messes up the request method, the URL and the protocol.

Now if we use the `MyApache::Dump` response handler, we have developed before in this chapter, which dumps the query string and the content body as a response, and configure the server as follows:

```
<Location /lc_input>
    SetHandler modperl
    PerlResponseHandler    +MyApache::Dump
    PerlInputFilterHandler +MyApache::InputRequestFilterLC
</Location>
```

When issuing a POST request:

```
% echo "mOd_pErl RuLeS" | POST 'http://localhost:8002/lc_input?FoO=1&BAR=2'
```

we get a response:

```
args:
FoO=1&BAR=2
content:
mod_perl rules
```

indeed we can see that our filter has lowercased the POSTed body, before the content handler received it. You can see that the query string wasn't changed.

1.6.4 Stream-oriented Input Filters

Let's now look at the same filter implemented using the stream-oriented API.

```
file:MyApache/InputRequestFilterLC2.pm
--------------------------------------
package MyApache::InputRequestFilterLC2;

use strict;
use warnings;

use base qw(Apache::Filter);

use Apache::Const -compile => 'OK';

use constant BUFF_LEN => 1024;

sub handler : FilterRequestHandler {
    my $f = shift;

    while ($f->read(my $buffer, BUFF_LEN)) {
        $f->print(lc $buffer);
    }
```

```
    Apache::OK;
}
1;
```

Now you probably ask yourself why did we have to go through the bucket brigades filters when this all can be done so much simpler. The reason is that we wanted you to understand how the filters work underneath, which will assist a lot when you will need to debug filters or optimize their speed. In certain cases a bucket brigade filter may be more efficient than the stream-oriented. For example if the filter applies transformation to selected buckets, certain buckets may contain open filehandles or pipes, rather than real data. And when you call read() the buckets will be forced to read that data in. But if you didn't want to modify these buckets you could pass them as they are and let Apache do faster techniques for sending data from the file handles or pipes.

The logic is very simple here, the filter reads in loop, and prints the modified data, which at some point will be sent to the next filter. This point happens every time the internal mod_perl buffer is full or when the filter returns.

`read()` populates `$buffer` to a maximum of `BUFF_LEN` characters (1024 in our example). Assuming that the current bucket brigade contains 2050 chars, `read()` will get the first 1024 characters, then 1024 characters more and finally the remaining 2 characters. Notice that even though the response handler may have sent more than 2050 characters, every filter invocation operates on a single bucket brigade so you have to wait for the next invocation to get more input. In one of the earlier examples we have shown that you can force the generation of several bucket brigades in the content handler by using `rflush()`. For example:

```
$r->print("string");
$r->rflush();
$r->print("another string");
```

It's only possible to get more than one bucket brigade from the same filter handler invocation if the filter is not using the streaming interface and by simply calling `get_brigade()` as many times as needed or till EOS is received.

The configuration section is pretty much identical:

```
<Location /lc_input2>
    SetHandler modperl
    PerlResponseHandler    +MyApache::Dump
    PerlInputFilterHandler +MyApache::InputRequestFilterLC2
</Location>
```

When issuing a POST request:

```
% echo "mOd_pErl RuLeS" | POST 'http://localhost:8002/lc_input2?FoO=1&BAR=2'
```

we get a response:

```
args:
FoO=1&BAR=2
content:
mod_perl rules
```

indeed we can see that our filter has lowercased the POSTed body, before the content handler received it. You can see that the query string wasn't changed.

1.7 Output Filters

mod_perl supports Connection and HTTP Request output filters:

1.7.1 Connection Output Filters

Connection filters filter **all** the data that is going through the server. Therefore if the connection is of HTTP request type, connection output filters see the headers and the body of the response, whereas request output filters see only the response body.

META: for now see the request output filter explanations and examples, connection output filter examples will be added soon. Interesting ideas for such filters are welcome (possible ideas: mangling output headers for HTTP requests, pretty much anything for protocol modules).

1.7.2 HTTP Request Output Filters

As mentioned earlier output filters can be written using the bucket brigades manipulation or the simplified stream-oriented interface.

First let's develop a response handler that sends two lines of output: numerals 1234567890 and the English alphabet in a single string:

```
file:MyApache/SendAlphaNum.pm
------------------------------
package MyApache::SendAlphaNum;

use strict;
use warnings;

use Apache::RequestRec ();
use Apache::RequestIO ();

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

sub handler {
    my $r = shift;

    $r->content_type('text/plain');

    $r->print(1..9, "0\n");
    $r->print('a'..'z', "\n");
```

```
    Apache::OK;
}
1;
```

The purpose of our filter handler is to reverse every line of the response body, preserving the new line characters in their places. Since we want to reverse characters only in the response body, without breaking the HTTP headers, we will use the HTTP request output filter.

1.7.2.1 Stream-oriented Output Filters

The first filter implementation is using the stream-oriented filtering API:

```
file:MyApache/FilterReverse1.pm
---------------------------
package MyApache::FilterReverse1;

use strict;
use warnings;

use base qw(Apache::Filter);

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

use constant BUFF_LEN => 1024;

sub handler : FilterRequestHandler {
    my $f = shift;

    while ($f->read(my $buffer, BUFF_LEN)) {
        for (split "\n", $buffer) {
            $f->print(scalar reverse $_);
            $f->print("\n");
        }
    }

    Apache::OK;
}
1;
```

Next, we add the following configuration to *httpd.conf*:

```
PerlModule MyApache::FilterReverse1
PerlModule MyApache::SendAlphaNum
<Location /reverse1>
    SetHandler modperl
    PerlResponseHandler     MyApache::SendAlphaNum
    PerlOutputFilterHandler MyApache::FilterReverse1
</Location>
```

Now when a request to */reverse1* is made, the response handler `MyApache::SendAlphaNum::handler()` sends:

```
1234567890
abcdefghijklmnopqrstuvwxyz
```

as a response and the output filter handler `MyApache::FilterReverse1::handler` reverses the lines, so the client gets:

```
0987654321
zyxwvutsrqponmlkjihgfedcba
```

The `Apache::Filter` module loads the `read()` and `print()` methods which encapsulate the stream-oriented filtering interface.

The reversing filter is quite simple: in the loop it reads the data in the *readline()* mode in chunks up to the buffer length (1024 in our example), and then prints each line reversed while preserving the new line control characters at the end of each line. Behind the scenes `$f->read()` retrieves the incoming brigade and gets the data from it, and `$f->print()` appends to the new brigade which is then sent to the next filter in the stack. `read()` breaks the *while* loop, when the brigade is emptied or the end of stream is received.

In order not to distract the reader from the purpose of the example the used code is oversimplified and won't handle correctly input lines which are longer than 1024 characters and possibly using a different line termination token (could be "\n", "\r" or "\r\n" depending on a platform). Moreover a single line may be split between across two or even more bucket brigades, so we have to store the unprocessed string in the filter context, so it can be used on the following invocations. So here is an example of a more complete handler, which does takes care of these issues:

```
sub handler {
    my $f = shift;

    my $leftover = $f->ctx;
    while ($f->read(my $buffer, BUFF_LEN)) {
        $buffer = $leftover . $buffer if defined $leftover;
        $leftover = undef;
        while ($buffer =~ /([^\r\n]*)([\r\n]*)/g) {
            $leftover = $1, last unless $2;
            $f->print(scalar(reverse $1), $2);
        }
    }

    if ($f->seen_eos) {
        $f->print(scalar reverse $leftover) if defined $leftover;
    }
    else {
        $f->ctx($leftover) if defined $leftover;
    }

    return Apache::OK;
}
```

The handler uses the `$leftover` variable to store unprocessed data as long as it fails to assemble a complete line or there is an incomplete line following the new line token. On the next handler invocation this data is then prepended to the next chunk that is read. When the filter is invoked on the last time, it

unconditionally reverses and flushes any remaining data.

1.7.2.2 Bucket Brigade-based Output Filters

The following filter implementation is using the bucket brigades API to accomplish exactly the same task as the first filter.

```
file:MyApache/FilterReverse2.pm
-------------------------------
package MyApache::FilterReverse2;

use strict;
use warnings;

use base qw(Apache::Filter);

use APR::Brigade ();
use APR::Bucket ();

use Apache::Const -compile => 'OK';
use APR::Const -compile => ':common';

sub handler : FilterRequestHandler {
    my($f, $bb) = @_;

    my $c = $f->c;
    my $bb_ctx = APR::Brigade->new($c->pool, $c->bucket_alloc);

    while (!$bb->empty) {
        my $bucket = $bb->first;

        $bucket->remove;

        if ($bucket->is_eos) {
            $bb_ctx->insert_tail($bucket);
            last;
        }

        my $data;
        my $status = $bucket->read($data);
        return $status unless $status == APR::SUCCESS;

        if ($data) {
            $data = join "",
                map {scalar(reverse $_), "\n"} split "\n", $data;
            $bucket = APR::Bucket->new($data);
        }
```

```
            $bb_ctx->insert_tail($bucket);
        }

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

        Apache::OK;
    }
    1;
```

and the corresponding configuration:

```
    PerlModule MyApache::FilterReverse2
    PerlModule MyApache::SendAlphaNum
    <Location /reverse2>
        SetHandler modperl
        PerlResponseHandler     MyApache::SendAlphaNum
        PerlOutputFilterHandler MyApache::FilterReverse2
    </Location>
```

Now when a request to */reverse2* is made, the client gets:

```
    0987654321
    zyxwvutsrqponmlkjihgfedcba
```

as expected.

The bucket brigades output filter version is just a bit more complicated than the stream-oriented one. The handler receives the incoming bucket brigade `$bb` as its second argument. Since when the handler is completed it must pass a brigade to the next filter in the stack, we create a new bucket brigade into which we are going to put the modified buckets and which eventually we pass to the next filter.

The core of the handler is in removing buckets from the head of the bucket brigade `$bb` while there are some, reading the data from the buckets, reversing and putting it into a newly created bucket which is inserted to the end of the new bucket brigade. If we see a bucket which designates the end of stream, we insert that bucket to the tail of the new bucket brigade and break the loop. Finally we pass the created brigade with modified data to the next filter and return.

Similarly to the original version of `MyApache::FilterReverse1::handler`, this filter is not smart enough to handle incomplete lines. However the exercise of making the filter foolproof should be trivial by porting a better matching rule and using the `$leftover` buffer from the previous section is trivial and left as an exercise to the reader.

1.8 Filter Applications

The following sections provide various filter applications and their implementation.

1.8.1 Handling Data Underruns

Sometimes filters need to read at least N bytes before they can apply their transformation. It's quite possible that reading one bucket brigade is not enough. But two or more are needed. This situation is sometimes referred to as an *underrun*.

Let's take an input filter as an example. When the filter realizes that it doesn't have enough data in the current bucket brigade, it can store the read data in the filter context, and wait for the next invocation of itself, which may or may not satisfy its needs. Meanwhile it must return an empty bb to the upstream input filter. This is not the most efficient technique to resolve underruns.

Instead of returning an empty bb, the input filter can initiate the retrieval of extra bucket brigades, until the underrun condition gets resolved. Notice that this solution is absolutely transparent to any filters before or after the current filter.

Consider this HTTP request:

```
% perl -MLWP::UserAgent -le ' \
  $r = LWP::UserAgent->new()->post("http://localhost:8011/", \
       [content => "x" x (40 * 1024 + 7)]); \
  print $r->is_success ? $r->content : "failed: " . $r->code'
read 40975 chars
```

This client POSTs just a little bit more than 40kb of data to the server. Normally Apache splits incoming POSTed data into 8kb chunks, putting each chunk into a separate bucket brigade. Therefore we expect to get 5 brigades of 8kb, and one brigade with just a few bytes (a total of 6 bucket brigades).

Now let's say that the filter needs to have 1024*16 + 5 bytes to have a complete token and then it can start its processing. The extra 5 bytes are just so we don't perfectly fit into 8bk bucket brigades, making the example closer to real situations. Having 40975 bytes of input and a token size of 16389 bytes, we will have 2 full tokens and 8197 remainder.

Jumping ahead let's look at the filter debug output:

```
filter called
asking for a bb
asking for a bb
asking for a bb
storing the remainder: 7611 bytes

filter called
asking for a bb
asking for a bb
storing the remainder: 7222 bytes

filter called
asking for a bb
seen eos, flushing the remaining: 8197 bytes
```

So we can see that the filter was invoked three times. The first time it has consumed three bucket brigades, collecting one full token of 16389 bytes and has a remainder of 7611 bytes to be processed on the next invocation. The second time it needed only two more bucket brigades and this time after completing the second token, 7222 bytes have remained. Finally on the third invocation it has consumed the last bucket brigade (total of six, just as we have expected), however it didn't have enough for the third token and since EOS has been seen (no more data expected), it has flushed the remaining 8197 bytes as we have calculated earlier.

It is clear from the debugging output that the filter was invoked only three times, instead of six times (there were six bucket brigades). Notice that the upstread input filter (if any) wasn't aware that there were six bucket brigades, since it saw only three. Our example filter didn't do much with those tokens, so it has only repackaged data from 8kb per bucket brigade, to 16389 bytes per bucket brigade. But of course in real world some transformation is applied on these tokens.

Now you understand what did we want from the filter, it's time for the implementation details. First let's look at the `response()` handler (the first part of the module):

```
#file:MyApache/Underrun.pm
#------------------------
package MyApache::Underrun;

use strict;
use warnings;

use constant IOBUFSIZE => 8192;

use Apache::Const -compile => qw(MODE_READBYTES OK M_POST);
use APR::Const    -compile => qw(SUCCESS BLOCK_READ);

sub response {
    my $r = shift;

    $r->content_type('text/plain');

    if ($r->method_number == Apache::M_POST) {
        my $data = read_post($r);
        #warn "HANDLER READ: $data\n";
        my $length = length $data;
        $r->print("read $length chars");
    }

    return Apache::OK;
}

sub read_post {
    my $r = shift;
    my $debug = shift || 0;

    my @data = ();
    my $seen_eos = 0;
    my $filters = $r->input_filters();
    my $ba = $r->connection->bucket_alloc;
    my $bb = APR::Brigade->new($r->pool, $ba);
```

```
    do {
        my $rv = $filters->get_brigade($bb,
            Apache::MODE_READBYTES, APR::BLOCK_READ, IOBUFSIZE);

        if ($rv != APR::SUCCESS) {
            return $rv;
        }

        while (!$bb->empty) {
            my $buf;
            my $b = $bb->first;

            $b->remove;

            if ($b->is_eos) {
                warn "EOS bucket:\n" if $debug;
                $seen_eos++;
                last;
            }

            my $status = $b->read($buf);
            warn "DATA bucket: [$buf]\n" if $debug;
            if ($status != APR::SUCCESS) {
                return $status;
            }
            push @data, $buf;
        }

        $bb->destroy;

    } while (!$seen_eos);

    return join '', @data;
}
```

The `response()` handler is trivial -- it reads the POSTed data and prints how many bytes it has read. `read_post()` sucks all POSTed data without parsing it.

Now comes the filter (which lives in the same package):

```
#file:MyApache/Underrun.pm (continued)
#-------------------------------------
use Apache::Filter ();

use Apache::Const -compile => qw(OK M_POST);

use constant TOKEN_SIZE => 1024*16 + 5; # ~16k

sub filter {
    my($f, $bb, $mode, $block, $readbytes) = @_;
    my $ba = $f->r->connection->bucket_alloc;
    my $ctx = $f->ctx;
    my $buffer = defined $ctx ? $ctx : '';
    $ctx = '';  # reset
    my $seen_eos = 0;
```

```
    my $data;
    warn "\nfilter called\n";

    # fetch and consume bucket brigades untill we have at least TOKEN_SIZE
    # bytes to work with
    do {
        my $tbb = APR::Brigade->new($f->r->pool, $ba);
        my $rv = $f->next->get_brigade($tbb, $mode, $block, $readbytes);
        warn "asking for a bb\n";
        ($data, $seen_eos) = flatten_bb($tbb);
        $tbb->destroy;
        $buffer .= $data;
    } while (!$seen_eos && length($buffer) < TOKEN_SIZE);

    # now create a bucket per chunk of TOKEN_SIZE size and put the remainder
    # in ctx
    for (split_buffer($buffer)) {
        if (length($_) == TOKEN_SIZE) {
            $bb->insert_tail(APR::Bucket->new($_));
        }
        else {
            $ctx .= $_;
        }
    }

    my $len = length($ctx);
    if ($seen_eos) {
        # flush the remainder
        $bb->insert_tail(APR::Bucket->new($ctx));
        $bb->insert_tail(APR::Bucket::eos_create($ba));
        warn "seen eos, flushing the remaining: $len bytes\n";
    }
    else {
        # will re-use the remainder on the next invocation
        $f->ctx($ctx);
        warn "storing the remainder: $len bytes\n";
    }

    return Apache::OK;
}

# split a string into tokens of TOKEN_SIZE bytes and a remainder
sub split_buffer {
    my $buffer = shift;
    if ($] < 5.007) {
        my @tokens = $buffer =~ /(.{@{[TOKEN_SIZE]}}|.+)/g;
        return @tokens;
    }
    else {
        # available only since 5.7.x+
        return unpack "(A" . TOKEN_SIZE . ")*", $buffer;
    }
}

sub flatten_bb {
    my ($bb) = shift;
```

```
    my $seen_eos = 0;

    my @data;
    for (my $b = $bb->first; $b; $b = $bb->next($b)) {
        $seen_eos++, last if $b->is_eos;
        $b->read(my $bdata);
        $bdata = '' unless defined $bdata;
        push @data, $bdata;

    }
    return (join('', @data), $seen_eos);
}

1;
```

The filter calls `get_brigade()` in a do-while loop till it reads enough data or sees EOS. Notice that it may get underruns for several times, and then suddenly receive a lot of data at once, which will be enough for more than one minimal size token, so we have to take care this into an account. Once the underrun condition is satisfied (we have at least one complete token) the tokens are put into a bucket brigade and returned to the upstream filter for processing, keeping any remainders in the filter context, for the next invocations or flushing all the remaining data if EOS has been seen.

Notice that this won't be possible with streaming filters where every invocation gives the filter exactly one bucket brigade to work with and provides not facilities to fetch extra brigades. (META: however this can be fixed, by providing a method which will fetch the next bucket brigade, so the read in a while loop can be repeated)

And here is the configuration for this setup:

```
PerlModule MyApache::Underrun
<Location />
  PerlInputFilterHandler MyApache::Underrun::filter
  SetHandler modperl
  PerlResponseHandler MyApache::Underrun::response
</Location>
```

1.9 Filter Tips and Tricks

Various tips to use in filters.

1.9.1 Altering the Content-Type Response Header

Let's say that you want to modify the `Content-Type` header in the request output filter:

```
sub handler : FilterRequestHandler {
    my $f = shift;
    ...
    $f->r->content_type("text/html; charset=$charset");
    ...
```

Request filters have an access to the request object, so we simply modify it.

1.10 Writing Well-Behaving Filters

Filter writers must follow the following rules:

1.10.1 Adjusting HTTP Headers

The following information is relevant for HTTP filters

- **Unsetting the Content-Length header**

  HTTP response filters modifying the length of the body they process must unset the `Content-Length` header. For example, a compression filter modifies the body length, whereas a lowercasing filter doesn't; therefore the former has to unset the header, and the latter doesn't have to.

  The header must be unset before any output is sent from the filter. If this rule is not followed, an HTTP response header with incorrect `Content-Length` value might be sent.

  Since you want to run this code once during the multiple filter invocations, use the `ctx()` method to set the flag:

  ```
  unless ($f->ctx) {
      $f->r->headers_out->unset('Content-Length');
      $f->ctx(1);
  }
  ```

- META: Same goes for last-modified/etags, which may need to be unset, "vary" might need to be added if you want caching to work properly (depending on what your filter does.

1.10.2 Other issues

META: to be written. Meanwhile collecting important inputs from various sources.

[

This one will be expanded by Geoff at some point:

HTTP output filter developers are ought to handle conditional GETs properly... (mostly for the reason of efficiency?)

]

[

talk about issues like not losing meta-buckets. e.g. if the filter runs a switch statement and propagates buckets types that were known at the time of writing, it may drop buckets of new types which may be added later, so it's important to ensure that there is a default cause where the bucket is passed as is.

of course mention the fact where things like EOS buckets must be passed, or the whole chain will be broken. Or if some filter decides to inject an EOS bucket by itself, it should probably consume and destroy the rest of the incoming bb. need to check on this issue.

]

[

Need to document somewhere (concepts?) that the buckets should never be modified directly, because the filter can't know ho else could be referencing it at the same time. (shared mem/cache/memory mapped files are examples on where you don't want to modify the data). Instead the data should be moved into a new bucket.

Also it looks like we need to $b->destroy (need to add the API) in addition to $b->remove. Which can be done in one stroke using $b->delete (need to add the API).

]

[

Mention mod_bucketeer as filter debugging tool (in addition to FilterSnoop)

]

1.11 Writing Efficient Filters

META: to be written

[

As of this writing the network input filter reads in 8000B chunks (not 8192B!), and making each bucket 8000B in size, so it seems that the most efficient reading technique is:

```
use constant BUFF_LEN => 8000;
while ($f->read(my $buffer, BUFF_LEN)) {
    # manip $buffer
    $f->print($buffer);
}
```

however if there is some filter in between, it may change the size of the buckets. Also this number may change in the future.

Hmm, I've also seen it read in 7819 chunks. I suppose this is not very reliable. But it's probably a good idea to ask at least 8k, so if a bucket brigade has < 8k, nothing will need to be stored in the internal buffer. i.e. read() will return less than asked for.

]

[

Bucket Brigades are used to make the data flow between filters and handlers more efficient. e.g. a file handle can be put in a bucket and the read from the file can be postponed to the very moment when the data is sent to the client, thus saving a lot of memory and CPU cycles. though filters writers should be aware that if they call $bucket->read(), or any other operation that internally forces the bucket to read the information into the memory (like the length() op) and thus making the data handling inefficient. therefore a care should be taken so not to read the data in, unless it's really necessary.

]

1.12 Maintainers

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

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

1.13 Authors

-

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

Table of Contents: