CHAPTER 25

Programming for mod_perl 2.0

In this chapter, we discuss how to migrate services from mod_perl 1.0 to 2.0, and how to make the new services based on mod_perl 2.0 backward compatible with mod_perl 1.0 (if possible). We also cover all the new `Perl*Handlers` in mod_perl 2.0.

Migrating to and Programming with mod_perl 2.0

In mod_perl 2.0, several configuration directives were renamed or removed. Several APIs also were changed, renamed, removed, or moved to new packages. Certain functions, while staying exactly the same as in mod_perl 1.0, now reside in different packages. Before using them, you need to find and load the new packages.

Since mod_perl 2.0 hasn't yet been released as of this writing, it's possible that certain things will change after the book is published. If something doesn't work as explained here, please refer to the documentation in the mod_perl distribution or the online version at *http://perl.apache.org/docs/2.0/* for the updated documentation.

The Shortest Migration Path

mod_perl 2.0 provides two backward-compatibility layers: one for the configuration files and the other for the code. If you are concerned about preserving backward compatibility with mod_perl 1.0, or are just experimenting with mod_perl 2.0 while continuing to run mod_perl 1.0 on your production server, simply enable the code-compatibility layer by adding:

```
use Apache2;
use Apache::compat;
```

at the top of your startup file. Backward compatibility of the configuration is enabled by default.

Migrating Configuration Files

To migrate the configuration files to mod_perl 2.0 syntax, you may need to make certain adjustments. Several configuration directives are deprecated in 2.0 but are still available for backward compatibility with mod_perl 1.0. If you don't need backward compatibility, consider using the directives that have replaced them.

PerlHandler

`PerlHandler` has been replaced with `PerlResponseHandler`.

PerlSendHeader

`PerlSendHeader` has been replaced with the `PerlOptions +/-ParseHeaders` directive:

```
PerlSendHeader On  => PerlOptions +ParseHeaders
PerlSendHeader Off => PerlOptions -ParseHeaders
```

PerlSetupEnv

`PerlSetupEnv` has been replaced with the `PerlOptions +/-SetupEnv` directive:

```
PerlSetupEnv On  => PerlOptions +SetupEnv
PerlSetupEnv Off => PerlOptions -SetupEnv
```

PerlTaintCheck

Taint mode can now be turned on with:

```
PerlSwitches -T
```

As with standard Perl, taint mode is disabled by default. Once enabled, taint mode cannot be turned off.

PerlWarn

Warnings now can be enabled globally with:

```
PerlSwitches -w
```

PerlFreshRestart

`PerlFreshRestart` is a mod_perl 1.0 legacy option and doesn't exist in mod_perl 2.0. A full tear-down and startup of interpreters is done on restart.

If you need to use the same *httpd.conf* file for 1.0 and 2.0, use:

```
<IfDefine !MODPERL2>
    PerlFreshRestart On
</IfDefine>
```

Code Porting

mod_perl 2.0 is trying hard to be backward compatible with mod_perl 1.0. However, some things (mostly APIs) have changed. To gain complete compatibility with 1.0 while running under 2.0, you should load the compatibility module as early as possible:

```
use Apache::compat;
```

at server startup. Unless there are forgotten things or bugs, your code should work without any changes under the 2.0 series.

However, if you don't have a good reason to keep 1.0 compatibility, you should try to remove the compatibility layer and adjust your code to work under 2.0 without it. This will improve performance. The online mod_perl documentation includes a document (*http://perl.apache.org/docs/2.0/user/porting/compat.html*) that explains what APIs have changed and what new APIs should be used instead.

If you have mod_perl 1.0 and 2.0 installed on the same system and the two use the same Perl libraries directory (e.g., */usr/lib/perl5*), to use mod_perl 2.0 make sure to first load the `Apache2` module, which will perform the necessary adjustments to `@INC`:

```
use Apache2; # if you have 1.0 and 2.0 installed
use Apache::compat;
```

So if before loading `Apache2.pm` the `@INC` array consisted of:

```
/usr/lib/perl5/5.8.0/i686-linux-thread-multi
/usr/lib/perl5/5.8.0
/usr/lib/perl5/site_perl/5.8.0/i686-linux-thread-multi
/usr/lib/perl5/site_perl/5.8.0
/usr/lib/perl5/site_perl
.
```

it will now look like this:

```
/usr/lib/perl5/site_perl/5.8.0/i686-linux-thread-multi/Apache2
/usr/lib/perl5/5.8.0/i686-linux-thread-multi
/usr/lib/perl5/5.8.0
/usr/lib/perl5/site_perl/5.8.0/i686-linux-thread-multi
/usr/lib/perl5/site_perl/5.8.0
/usr/lib/perl5/site_perl
.
```

Notice that a new directory was appended to the search path. If, for example, the code attempts to load `Apache::Server` and there are two versions of this module under */usr/lib/perl5/site_perl/*:

```
5.8.0/i686-linux-thread-multi/Apache/Server.pm
  5.8.0/i686-linux-thread-multi/Apache2/Apache/Server.pm
```

the mod_perl 2.0 version will be loaded first, because the directory *5.8.0/i686-linux-thread-multi/Apache2* comes before the directory *5.8.0/i686-linux-thread-multi* in `@INC`.

Finally, mod_perl 2.0 has all its methods spread across many modules. To use these methods, you first have to load the modules containing them. The `ModPerl::MethodLookup` module can be used to figure out what modules need to be loaded. For example, if you try to use:

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

and mod_perl complains that it can't find the `construct_url()` method, you can ask `ModPerl::MethodLookup`:

```
panic% perl -MApache2 -MModPerl::MethodLookup -e print_method construct_url
```

This will print:

```
to use method 'construct_url' add:
       use Apache::URI ();
```

Another useful feature provided by `ModPerl::MethodLookup` is the `preload_all_modules()` function, which preloads all mod_perl 2.0 modules. This is useful when you start to port your mod_perl 1.0 code (though preferrably avoided in the production environment to save memory). You can simply add the following snippet to your *startup.pl* file:

```
use ModPerl::MethodLookup;
ModPerl::MethodLookup::preload_all_modules();
```

ModPerl::Registry Family

In mod_perl 2.0, `Apache::Registry` and friends (`Apache::PerlRun`, `Apache::RegistryNG`, etc.) have migrated into the `ModPerl::` namespace. The new family is based on the idea of `Apache::RegistryNG` from mod_perl 1.0, where you can customize pretty much all the functionality by providing your own hooks. The functionality of the `Apache::Registry`, `Apache::RegistryBB`, and `Apache::PerlRun` modules hasn't changed from the user's perspective, except for the namespace. All these modules are now derived from the `ModPerl::RegistryCooker` class. So if you want to change the functionality of any of the existing subclasses, or you want to "cook" your own registry module, it can be done easily. Refer to the `ModPerl::RegistryCooker` manpage for more information.

Here is a typical registry section configuration in mod_perl 2.0:

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

As we explained earlier, the `ParseHeaders` option is needed if the headers are being sent via `print()` (i.e., without using the mod_perl API) and comes as a replacement for the `PerlSendHeader` option in mod_perl 1.0.

Example 25-1 shows a simple registry script that prints the environment variables.

Example 25-1. print_env.pl

```
print "Content-type: text/plain\n\n";
for (sort keys %ENV){
    print "$_ => $ENV{$_}\n";
}
```

Save the file in */home/httpd/perl/print_env.pl* and make it executable:

```
panic% chmod 0700 /home/stas/modperl/mod_perl_rules1.pl
```

Now issue a request to *http://localhost/perl/print_env.pl*, and you should see all the environment variables printed out.

One currently outstanding issue with the registry family is the issue with `chdir()`. mod_perl 1.0 registry modules always performed `cdhir()`s to the directory of the script, so scripts could require modules relative to the directory of the script. Since mod_perl 2.0 may run in a threaded environment, the registry scripts can no longer call `chdir()`, because when one thread performs a `chdir()` it affects the whole process—all other threads will see that new directory when calling `Cwd::cwd()`, which will wreak havoc. As of this writing, the registry modules can't handle this problem (they simply don't `chdir()` to the script's directory); however, a satisfying solution will be provided by the time mod_perl 2.0 is released.

Method Handlers

In mod_perl 1.0, method handlers had to be specified by using the `($$)` prototype:

```
package Eagle;
@ISA = qw(Bird);

sub handler ($$) {
    my($class, $r) = @_;
    ...;
}
```

Starting with Perl Version 5.6, you can use subroutine attributes, and that's what mod_perl 2.0 does instead of conventional prototypes:

```
package Eagle;
@ISA = qw(Bird);

sub handler : method {
    my($class, $r) = @_;
    ...;
}
```

See the *attributes* manpage.

mod_perl 2.0 doesn't support the `($$)` prototypes, mainly because several callbacks in 2.0 have more arguments than `$r`, so the `($$)` prototype doesn't make sense any

more. Therefore, if you want your code to work with both mod_perl generations, you should use the subroutine attributes.

Apache::StatINC Replacement

`Apache::StatINC` has been replaced by `Apache::Reload`, which works for both mod_perl generations. To migrate to `Apache::Reload`, simply replace:

```
PerlInitHandler Apache::StatINC
```

with:

```
PerlInitHandler Apache::Reload
```

`Apache::Reload` also provides some extra functionality, covered in the module's manpage.

New Apache Phases and Corresponding Perl*Handlers

Because the majority of the Apache phases supported by mod_perl haven't changed since mod_perl 1.0, in this chapter we will discuss only those phases and corresponding handlers that were added or changed in mod_perl 2.0.

Figure 25-1 depicts the Apache 2.0 server cycle. You can see the mod_perl phases `PerlOpenLogsHandler`, `PerlPostConfigHandler`, and `PerlChildInitHandler`, which we will discuss shortly. Later, we will zoom into the connection cycle depicted in Figure 25-2, which will expose other mod_perl handlers.

Apache 2.0 starts by parsing the configuration file. After the configuration file is parsed, any `PerlOpenLogsHandler` handlers are executed. After that, any `PerlPostConfigHandler` handlers are run. When the *post_config* phase is finished the server immediately restarts, to make sure that it can survive graceful restarts after starting to serve the clients.

When the restart is completed, Apache 2.0 spawns the workers that will do the actual work. Depending on the MPM used, these can be threads, processes, or a mixture of both. For example, the *worker* MPM spawns a number of processes, each running a number of threads. When each child process is started `PerlChildInitHandler`s are executed. Notice that they are run for each starting process, not thread.

From that moment on each working process (or thread) processes connections until it's killed by the server or the server is shut down. When the server is shut down, any registered `PerlChildExitHandler`s are executed.

Example 25-2 demonstrates all the startup phases.

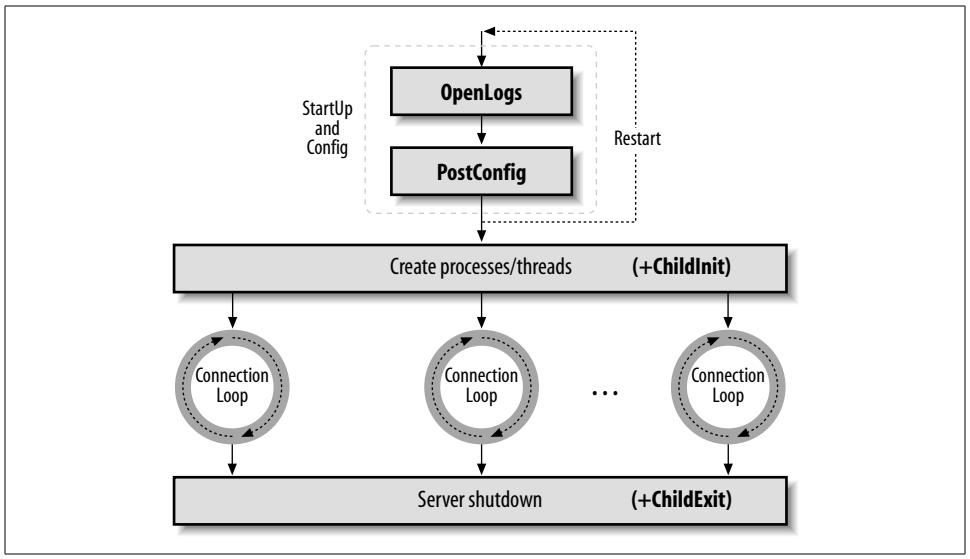

Figure 25-1. Apache 2.0 server lifecycle

Example 25-2. Book/StartupLog.pm

```
package Book::StartupLog;

use strict;
use warnings;

use Apache::Log ();
use Apache::ServerUtil ();

use File::Spec::Functions;

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

my $log_file = catfile "logs", "startup_log";
my $log_fh;

sub open_logs {
    my($conf_pool, $log_pool, $temp_pool, $s) = @_;
    my $log_path = Apache::server_root_relative($conf_pool, $log_file);

    $s->warn("opening the log file: $log_path");
    open $log_fh, ">>$log_path" or die "can't open $log_path: $!";
    my $oldfh = select($log_fh); $| = 1; select($oldfh);

    say("process $$ is born to reproduce");
    return Apache::OK;
}

sub post_config {
    my($conf_pool, $log_pool, $temp_pool, $s) = @_;
```

Example 25-2. Book/StartupLog.pm (continued)

```
    say("configuration is completed");
    return Apache::OK;
}

sub child_exit {
    my($child_pool, $s) = @_;
    say("process $$ now exits");
    return Apache::OK;
}

sub child_init {
    my($child_pool, $s) = @_;
    say("process $$ is born to serve");
    return Apache::OK;
}

sub say {
    my($caller) = (caller(1))[3] =~ /([^:]+)$/;
    if (defined $log_fh) {
        printf $log_fh "[%s] - %-11s: %s\n",
            scalar(localtime), $caller, $_[0];
    }
    else {
        # when the log file is not open
        warn __PACKAGE__ . " says: $_[0]\n";
    }
}

END {
    say("process $$ is shutdown\n");
}

1;
```

Here's the *httpd.conf* configuration section:

```
PerlModule            Book::StartupLog
PerlOpenLogsHandler   Book::StartupLog::open_logs
PerlPostConfigHandler Book::StartupLog::post_config
PerlChildInitHandler  Book::StartupLog::child_init
PerlChildExitHandler  Book::StartupLog::child_exit
```

When we perform a server startup followed by a shutdown, the *logs/startup_log* is created, if it didn't exist already (it shares the same directory with *error_log* and other standard log files), and each stage appends to it its log information. So when we perform:

```
panic% bin/apachectl start && bin/apachectl stop
```

the following is logged to *logs/startup_log*:

```
[Thu Mar  6 15:57:08 2003] - open_logs  : process 21823 is born to reproduce
[Thu Mar  6 15:57:08 2003] - post_config: configuration is completed
[Thu Mar  6 15:57:09 2003] - END        : process 21823 is shutdown
```

```
[Thu Mar  6 15:57:10 2003] - open_logs  : process 21825 is born to reproduce
[Thu Mar  6 15:57:10 2003] - post_config: configuration is completed
[Thu Mar  6 15:57:11 2003] - child_init : process 21830 is born to serve
[Thu Mar  6 15:57:11 2003] - child_init : process 21831 is born to serve
[Thu Mar  6 15:57:11 2003] - child_init : process 21832 is born to serve
[Thu Mar  6 15:57:11 2003] - child_init : process 21833 is born to serve
[Thu Mar  6 15:57:12 2003] - child_exit : process 21833 now exits
[Thu Mar  6 15:57:12 2003] - child_exit : process 21832 now exits
[Thu Mar  6 15:57:12 2003] - child_exit : process 21831 now exits
[Thu Mar  6 15:57:12 2003] - child_exit : process 21830 now exits
[Thu Mar  6 15:57:12 2003] - END        : process 21825 is shutdown
```

First, we can clearly see that Apache always restarts itself after the first *post_config* phase is over. The logs show that the *post_config* phase is preceded by the *open_logs* phase. Only after Apache has restarted itself and has completed the *open_logs* and *post_config* phases again is the *child_init* phase run for each child process. In our example we had the setting `StartServers=4`; therefore, you can see that four child processes were started.

Finally, you can see that on server shutdown, the *child_exit* phase is run for each child process and the `END { }` block is executed by the parent process only.

Apache also specifies the *pre_config* phase, which is executed before the configuration files are parsed, but this is of no use to mod_perl, because mod_perl is loaded only during the configuration phase.

Now let's discuss each of the mentioned startup handlers and their implementation in the `Book::StartupLog` module in detail.

Server Configuration and Startup Phases

open_logs, configured with `PerlOpenLogsHandler`, and *post_config*, configured with `PerlPostConfigHandler`, are the two new phases available during server startup.

PerlOpenLogsHandler

The *open_logs* phase happens just before the *post_config* phase.

Handlers registered by `PerlOpenLogsHandler` are usually used for opening module-specific log files (e.g., *httpd* core and mod_ssl open their log files during this phase).

At this stage the `STDERR` stream is not yet redirected to *error_log*, and therefore any messages to that stream will be printed to the console from which the server is starting (if one exists).

The `PerlOpenLogsHandler` directive may appear in the main configuration files and within `<VirtualHost>` sections.

Apache will continue executing all handlers registered for this phase until the first handler returns something other than `Apache::OK` or `Apache::DECLINED`.

As we saw in the Book::StartupLog::open_logs handler, the *open_logs* phase handlers accept four arguments: the configuration pool,* the logging streams pool, the temporary pool, and the server object:

```
sub open_logs {
    my($conf_pool, $log_pool, $temp_pool, $s) = @_;
    my $log_path = Apache::server_root_relative($conf_pool, $log_file);

    $s->warn("opening the log file: $log_path");
    open $log_fh, ">>$log_path" or die "can't open $log_path: $!";
    my $oldfh = select($log_fh); $| = 1; select($oldfh);

    say("process $$ is born to reproduce");
    return Apache::OK;
}
```

In our example the handler uses the Apache::server_root_relative( ) function to set the full path to the log file, which is then opened for appending and set to unbuffered mode. Finally, it logs the fact that it's running in the parent process.

As you've seen in this example, this handler is configured by adding the following to *httpd.conf*:

```
PerlOpenLogsHandler Book::StartupLog::open_logs
```

PerlPostConfigHandler

The *post_config* phase happens right after Apache has processed the configuration files, before any child processes are spawned (which happens at the *child_init* phase).

This phase can be used for initializing things to be shared between all child processes. You can do the same in the startup file, but in the *post_config* phase you have access to a complete configuration tree.

The *post_config* phase is very similar to the *open_logs* phase. The PerlPostConfigHandler directive may appear in the main configuration files and within <VirtualHost> sections. Apache will run all registered handlers for this phase until a handler returns something other than Apache::OK or Apache::DECLINED. This phase's handlers receive the same four arguments as the *open_logs* phase's handlers. From our example:

```
sub post_config {
    my($conf_pool, $log_pool, $temp_pool, $s) = @_;
    say("configuration is completed");
    return Apache::OK;
}
```

This example handler just logs that the configuration was completed and returns right away.

* Pools are used by Apache for memory-handling functions. You can make use of them from the Perl space, too.

This handler is configured by adding the following to *httpd.conf*:

```
PerlOpenLogsHandler Book::StartupLog::post_config
```

PerlChildInitHandler

The *child_init* phase happens immediately after a child process is spawned. Each child process (not a thread!) will run the hooks of this phase only once in its lifetime.

In the *prefork* MPM this phase is useful for initializing any data structures that should be private to each process. For example, `Apache::DBI` preopens database connections during this phase, and `Apache::Resource` sets the process's resource limits.

The `PerlChildInitHandler` directive should appear in the top-level server configuration file. All `PerlChildInitHandler`s will be executed, disregarding their return values (although mod_perl expects a return value, so returning `Apache::OK` is a good idea).

In the `Book::StartupLog` example we used the `child_init( )` handler:

```
sub child_init {
    my($child_pool, $s) = @_;
    say("process $$ is born to serve");
    return Apache::OK;
}
```

The `child_init( )` handler accepts two arguments: the child process pool and the server object. The example handler logs the PID of the child process in which it's run and returns.

This handler is configured by adding the following to *httpd.conf*:

```
PerlOpenLogsHandler Book::StartupLog::child_init
```

PerlChildExitHandler

The *child_exit* phase is executed before the child process exits. Notice that it happens only when the process exits, not when the thread exits (assuming that you are using a threaded MPM).

The `PerlChildExitHandler` directive should appear in the top-level server configuration file. mod_perl will run all registered `PerlChildExitHandler` handlers for this phase until a handler returns something other than `Apache::OK` or `Apache::DECLINED`.

In the `Book::StartupLog` example we used the `child_exit( )` handler:

```
sub child_exit {
    my($child_pool, $s) = @_;
    say("process $$ now exits");
    return Apache::OK;
}
```

The `child_exit( )` handler accepts two arguments: the child process pool and the server object. The example handler logs the PID of the child process in which it's run and returns.

As you saw in the example, this handler is configured by adding the following to *httpd.conf*:

```
PerlOpenLogsHandler Book::StartupLog::child_exit
```

Connection Phases

Since Apache 2.0 makes it possible to implement protocols other than HTTP, the connection phases *pre_connection*, configured with `PerlPreConnectionHandler`, and *process_connection*, configured with `PerlProcessConnectionHandler`, were added. The *pre_connection* phase is used for runtime adjustments of things for each connection—for example, mod_ssl uses the *pre_connection* phase to add the SSL filters if `SSLEngine On` is configured, regardless of whether the protocol is HTTP, FTP, NNTP, etc. The *process_connection* phase is used to implement various protocols, usually those similar to HTTP. The HTTP protocol itself is handled like any other protocol; internally it runs the request handlers similar to Apache 1.3.

When a connection is issued by a client, it's first run through the `PerlPreConnection-Handler` and then passed to the `PerlProcessConnectionHandler`, which generates the response. When `PerlProcessConnectionHandler` is reading data from the client, it can be filtered by connection input filters. The generated response can also be filtered though connection output filters. Filters are usually used for modifying the data flowing though them, but they can be used for other purposes as well (e.g., logging interesting information). Figure 25-2 depicts the connection cycle and the data flow and highlights which handlers are available to mod_perl 2.0.

Now let's discuss the `PerlPreConnectionHandler` and `PerlProcessConnectionHandler` handlers in detail.

PerlPreConnectionHandler

The *pre_connection* phase happens just after the server accepts the connection, but before it is handed off to a protocol module to be served. It gives modules an opportunity to modify the connection as soon as possible and insert filters if needed. The core server uses this phase to set up the connection record based on the type of connection that is being used. mod_perl itself uses this phase to register the connection input and output filters.

In mod_perl 1.0, during code development `Apache::Reload` was used to automatically reload Perl modules modified since the last request. It was invoked during *post_read_request*, the first HTTP request's phase. In mod_perl 2.0, *pre_connection* is the earliest phase, so if we want to make sure that all modified Perl modules are reloaded for any protocols and their phases, it's best to set the scope of the Perl interpreter to the lifetime of the connection via:

```
PerlInterpScope connection
```

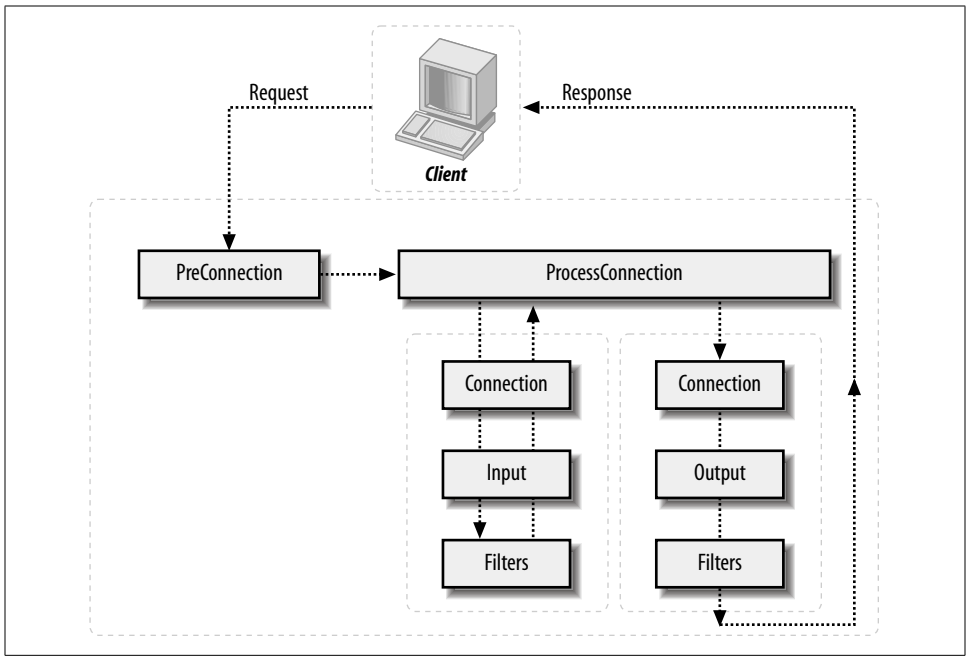

Figure 25-2. Apache 2.0 connection cycle

and invoke the `Apache::Reload` handler during the *pre_connection* phase. However, this development-time advantage can become a disadvantage in production—for example, if a connection handled by the HTTP protocol is configured as `KeepAlive` and there are several requests coming on the same connection (one handled by mod_perl and the others by the default image handler), the Perl interpreter won't be available to other threads while the images are being served.

Apache will continue executing all handlers registered for this phase until the first handler returns something other than `Apache::OK` or `Apache::DECLINED`.

The `PerlPreConnectionHandler` directive may appear in the main configuration files and within `<VirtualHost>` sections.

A *pre_connection* handler accepts a connection record and a socket object as its arguments:

```
sub handler {
    my ($c, $socket) = @_;
    # ...
    return Apache::OK;
}
```

PerlProcessConnectionHandler

The *process_connection* phase is used to process incoming connections. Only protocol modules should assign handlers for this phase, as it gives them an opportunity to replace the standard HTTP processing with processing for some other protocol (e.g., POP3, FTP, etc.).

Apache will continue executing all handlers registered for this phase until the first handler returns something other than `Apache::DECLINED`.

The `PerlProcessConnectionHandler` directive may appear in the main configuration files and within `<VirtualHost>` sections.

The *process_connection* handler can be written in two ways. The first way is to manipulate bucket brigades, in a way very similar to the filters. The second, simpler way is to bypass all the filters and to read from and write to the connection socket directly.

A *process_connection* handler accepts a connection record object as its only argument:

```
sub handler {
    my ($c) = @_;
    # ...
    return Apache::OK;
}
```

Now let's look at two examples of connection handlers. The first uses the connection socket to read and write the data, and the second uses bucket brigades to accomplish the same thing and allow the connection filters to do their work.

Socket-based protocol module. To demonstrate the workings of a protocol module, we'll take a look at the `Book::Eliza` module, which sends the data read from the client as input to `Chatbot::Eliza`, which in turn implements a mock Rogerian psychotherapist and forwards the response from the psychotherapist back to the client. In this module we will use the implementation that works directly with the connection socket and therefore bypasses any connection filters.

A protocol handler is configured using the `PerlProcessConnectionHandler` directive, and we will use the `Listen` and `<VirtualHost>` directives to bind to the nonstandard port 8084:

```
Listen 8084
<VirtualHost _default_:8084>
    PerlModule                  Book::Eliza
    PerlProcessConnectionHandler Book::Eliza
</VirtualHost>
```

`Book::Eliza` is then enabled when starting Apache:

```
panic% httpd
```

And we give it a whirl:

```
panic% telnet localhost 8084
Trying 127.0.0.1...
Connected to localhost (127.0.0.1).
Escape character is '^]'.
Hello Eliza
How do you do. Please state your problem.

How are you?
Oh, I?

Why do I have core dumped?
You say Why do you have core dumped?

I feel like writing some tests today, you?
I'm not sure I understand you fully.

Good bye, Eliza
Does talking about this bother you?

Connection closed by foreign host.
```

The code is shown in Example 25-3.

Example 25-3. Book/Eliza.pm

```
package Book::Eliza;

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

use Apache::Connection ();
use APR::Socket ();

require Chatbot::Eliza;

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

use constant BUFF_LEN => 1024;

my $eliza = new Chatbot::Eliza;

sub handler {
    my $c = shift;
    my $socket = $c->client_socket;

    my $buff;
    my $last = 0;
    while (1) {
        my($rlen, $wlen);
        $rlen = BUFF_LEN;
        $socket->recv($buff, $rlen);
        last if $rlen <= 0;
```

Example 25-3. Book/Eliza.pm (continued)

```
        # \r is sent instead of \n if the client is talking over telnet
        $buff =~ s/[\r\n]*$//;
        $last++ if $buff =~ /good bye/i;
        $buff = $eliza->transform( $buff ) . "\n\n";
        $socket->send($buff, length $buff);
        last if $last;
    }

    Apache::OK;
}
1;
```

The example handler starts with the standard package declaration and, of course, `use strict;`. As with all `Perl*Handlers`, the subroutine name defaults to `handler`. However, in the case of a protocol handler, the first argument is not a `request_rec`, but a `conn_rec` blessed into the `Apache::Connection` class. We have direct access to the client socket via `Apache::Connection`'s `client_socket()` method, which returns an object blessed into the `APR::Socket` class.

Inside the read/send loop, the handler attempts to read `BUFF_LEN` bytes from the client socket into the `$buff` buffer. The `$rlen` parameter will be set to the number of bytes actually read. The `APR::Socket::recv()` method returns an APR status value, but we need only check the read length to break out of the loop if it is less than or equal to 0 bytes. The handler also breaks the loop after processing an input including the "good bye" string.

Otherwise, if the handler receives some data, it sends this data to the `$eliza` object (which represents the psychotherapist), whose returned text is then sent back to the client with the `APR::Socket::send()` method. When the read/print loop is finished the handler returns `Apache::OK`, telling Apache to terminate the connection. As mentioned earlier, since this handler is working directly with the connection socket, no filters can be applied.

Bucket brigade–based protocol module. Now let's look at the same module, but this time implemented by manipulating bucket brigades. It runs its output through a connection output filter that turns all uppercase characters into their lowercase equivalents.

The following configuration defines a `<VirtualHost>` listening on port 8085 that enables the `Book::Eliza2` connection handler, which will run its output through the `Book::Eliza2::lowercase_filter` filter:

```
Listen 8085
<VirtualHost _default_:8085>
    PerlModule                   Book::Eliza2
    PerlProcessConnectionHandler Book::Eliza2
    PerlOutputFilterHandler      Book::Eliza2::lowercase_filter
</VirtualHost>
```

As before, we start the *httpd* server:

```
panic% httpd
```

and try the new connection handler in action:

```
panic% telnet localhost 8085
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
Hello Eliza!
hi. what seems to be your problem?

Problem? I don't have any problems ;)
does that trouble you?

Not at all, I don't like problems.
i'm not sure i understand you fully.

I said that I don't like problems.
that is interesting. please continue.

You are boring :(
does it please you to believe i am boring?

Yes, yes!
please tell me some more about this.

Good bye!
i'm not sure i understand you fully.

Connection closed by foreign host.
```

As you can see, the response, which normally is a mix of upper- and lowercase words, now is all in lowercase, because of the output filter. The implementation of the connection and the filter handlers is shown in Example 25-4.

Example 25-4. Book/Eliza2.pm

```
package Book::Eliza2;

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

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

require Chatbot::Eliza;

use APR::Const -compile => qw(SUCCESS EOF);
use Apache::Const -compile => qw(OK MODE_GETLINE);

my $eliza = new Chatbot::Eliza;
```

Example 25-4. Book/Eliza2.pm (continued)

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

    my $bb_in  = APR::Brigade->new($c->pool, $c->bucket_alloc);
    my $bb_out = APR::Brigade->new($c->pool, $c->bucket_alloc);
    my $last = 0;

    while (1) {
        my $rv = $c->input_filters->get_brigade($bb_in,
                                                Apache::MODE_GETLINE);

        if ($rv != APR::SUCCESS or $bb_in->empty) {
            my $error = APR::strerror($rv);
            unless ($rv == APR::EOF) {
                warn "[eliza] get_brigade: $error\n";
            }
            $bb_in->destroy;
            last;
        }

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

            $bucket->remove;

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

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

            if ($data) {
                $data =~ s/[\r\n]*$//;
                $last++ if $data =~ /good bye/i;
                $data = $eliza->transform( $data ) . "\n\n";
                $bucket = APR::Bucket->new($data);
            }

            $bb_out->insert_tail($bucket);
        }

        my $b = APR::Bucket::flush_create($c->bucket_alloc);
        $bb_out->insert_tail($b);
        $c->output_filters->pass_brigade($bb_out);
        last if $last;
    }

    Apache::OK;
}
```

Example 25-4. Book/Eliza2.pm (continued)

```
use base qw(Apache::Filter);
use constant BUFF_LEN => 1024;

sub lowercase_filter : FilterConnectionHandler {
    my $filter = shift;

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

    return Apache::OK;
}

1;
```

For the purpose of explaining how this connection handler works, we are going to simplify the handler. The whole handler can be represented by the following pseudocode:

```
while ($bb_in = get_brigade( )) {
    while ($bucket_in = $bb_in->get_bucket( )) {
        my $data = $bucket_in->read( );
        $data = transform($data);
        $bucket_out = new_bucket($data);

        $bb_out->insert_tail($bucket_out);
    }
    $bb_out->insert_tail($flush_bucket);
    pass_brigade($bb_out);
}
```

The handler receives the incoming data via bucket bridages, one at a time, in a loop. It then processes each brigade, by retrieving the buckets contained in it, reading in the data, transforming that data, creating new buckets using the transformed data, and attaching them to the outgoing brigade. When all the buckets from the incoming bucket brigade are transformed and attached to the outgoing bucket brigade, a flush bucket is created and added as the last bucket, so when the outgoing bucket brigade is passed out to the outgoing connection filters, it will be sent to the client right away, not buffered.

If you look at the complete handler, the loop is terminated when one of the following conditions occurs: an error happens, the end-of-stream bucket has been seen (i.e., there's no more input at the connection), or the received data contains the string "good bye". As you saw in the demonstration, we used the string "good bye" to terminate our shrink's session.

We will skip the filter discussion here, since we are going to talk in depth about filters in the following sections. All you need to know at this stage is that the data sent from the connection handler is filtered by the outgoing filter, which transforms it to be all lowercase.

HTTP Request Phases

The HTTP request phases themselves have not changed from mod_perl 1.0, except the `PerlHandler` directive has been renamed `PerlResponseHandler` to better match the corresponding Apache phase name (*response*).

The only difference is that now it's possible to register HTTP request input and output filters, so `PerlResponseHandler` will filter its input and output through them. Figure 25-3 depicts the HTTP request cycle, which should be familiar to mod_perl 1.0 users, with the new addition of the request filters. From the diagram you can also see that the request filters are stacked on top of the connection filters. The request input filters filter only a request body, and the request output filters filter only a response body. Request and response headers can be accessed and modified using the `$r->headers_in`, `$r->headers_out`, and other methods.

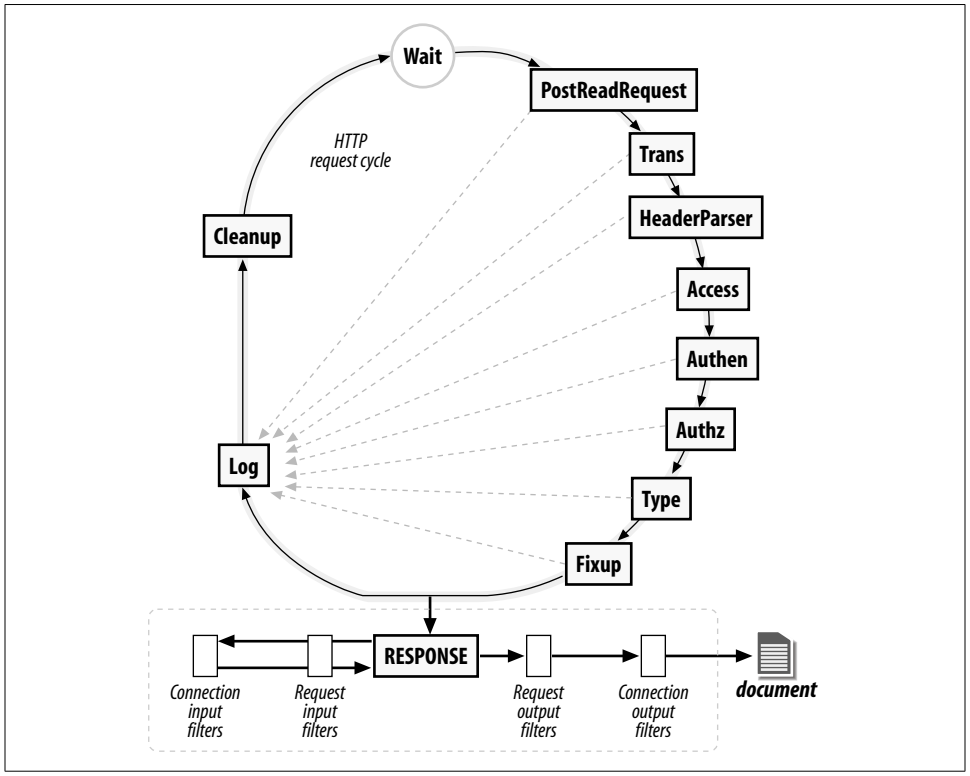

Figure 25-3. mod_perl 2.0 HTTP request cycle

I/O Filtering

Now let's talk about a totally new feature of mod_perl 2.0: input/output filtering.

As of this writing the mod_perl filtering API hasn't been finalized, and it's possible that it will change by the time the production version of mod_perl 2.0 is released. However, most concepts presented here won't change, and you should find the discussion and the examples useful for understanding how filters work. For the most up-to-date documentation, refer to *http://perl.apache.org/docs/2.0/user/handlers/filters.html*.

I/O Filtering Concepts

Before introducing the mod_perl filtering API, there are several important concepts to understand.

Two methods for manipulating data

As discussed in the last chapter, 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 the bucket brigades.

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

Even though you don't have to work with bucket brigades directly, since you can write filters using the simplified, streaming filter interface (which works with 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, and that the end-of-stream indicator (EOS) is sometimes sent in a separate bucket brigade, so it shouldn't be a surprise if the filter is invoked even though no real data went through.

You will also need to understand how to manipulate bucket brigades if you plan to implement protocol modules, as you have seen earlier in this chapter.

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 the request and response. Of course, if a different protocol is served over the connection (e.g., IMAP), the data could have a completely different pattern than the HTTP protocol (headers and body).

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

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 the upstream filter or content provider.

For example, if a content-generation handler sends a string, and then forces a flush, following with 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 that 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 no more will be data sent:

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

In our example the filter will be invoked three times. Notice that sometimes the EOS bucket comes attached to the last bucket brigade with data and sometimes in its own bucket brigade. This should be transparent to the filter logic, as we will see shortly.

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 when coding a filter is to never assume that the filter is always going to be invoked once, or a fixed number of times. You can't make assumptions about the way the data is going to come in. Therefore, a typical filter handler may need to split its logic into three parts, as depicted in Figure 25-4.

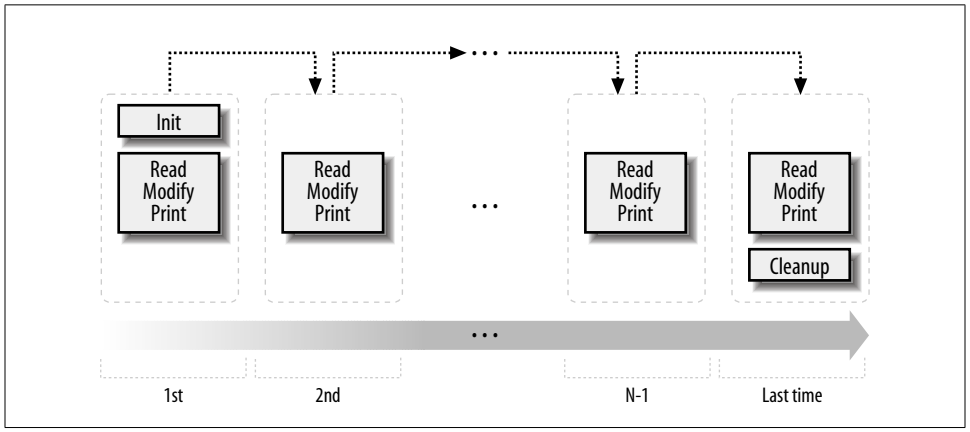

Figure 25-4. mod_perl 2.0 filter logic

Jumping ahead, we will show some pseudocode that represents all three parts. This is what a typical filter looks like:

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

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

    # runs on all invocations
    process($filter);

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

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

Let's examine the parts of this pseudofilter:

1. Initialization

   During the initialization, the filter runs all the code that should be performed only once across multiple invocations of the filter (during a single request). The filter context is used to accomplish this task. For each new request, the filter context is created before the filter is called for the first time, and it's destroyed at the end of the request. When the filter is invoked for the first time, `$filter->ctx` returns `undef` and the custom function `init( )` is called:

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

   This function can, for example, retrieve some configuration data set in *httpd.conf* or initialize some data structure 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:

   ```
   $filter->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 $filter = shift;

       my $ctx = $filter->ctx;
       $ctx->{invoked}++;
       $filter->ctx($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`, so that mod_perl will pass the bucket brigades to the next filter. If this handler returns `Apache::OK`, the data will simply be lost.

2. Processing

   The next part:

   ```
   process($filter);
   ```

   is unconditionally invoked on every filter invocation. This is 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 $filter = shift;
       while ($filter->read(my $data, READ_SIZE)) {
   ```

```
            $filter->print(lc $data);
        }
    }
```

Here the filter operates on only 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, and the filter will have to store the remainder of the unprocessed data in the filter context and then reuse it in the next invocation. Another good example is a filter that performs data compression (compression usually is effective only when applied to relatively big chunks of data)—if a single bucket brigade doesn't contain enough data, the filter may need to buffer the data in the filter context until it collects enough of it.

1. 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, it can check whether this is the last time it's invoked using the `$filter->seen_eos` method:

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

   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 the EOS token) and sometimes it comes all alone (the last invocation gets only the EOS token). 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 the filter won't function properly.

   Filters that directly manipulate bucket brigades have to look for a bucket whose type is EOS for the same reason.

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).

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 it's 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 next filter.

Input and output filters differ in the ways they acquire the bucket brigades (which include the data that they filter). Although the difference can't be seen when a streaming API is used, it's important to understand how things work underneath.

When an input filter is invoked, it first asks the upstream filter for the next bucket brigade (using the `get_brigade( )` call). That upstream filter in turn asks for the bucket brigade from the next upstream filter in the chain, and so on, until the last filter that reads from the network (called `core_in`) 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 processes the data and sends it to its downstream filter, and so on, until it reaches the very first filter that asked for the data. (In reality, some other handler triggers the request for the bucket brigade (e.g., the 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.)

Figure 25-5 depicts a typical input filter chain data flow, in addition to the program control flow. The arrows show when the control is switched from one filter to another, and the black-headed arrows show the actual data flow. The diagram includes some pseudocode, both 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 walls in the diagram), waiting for the call to return. When this call returns, all upstream filters have already completed their filtering tasks.

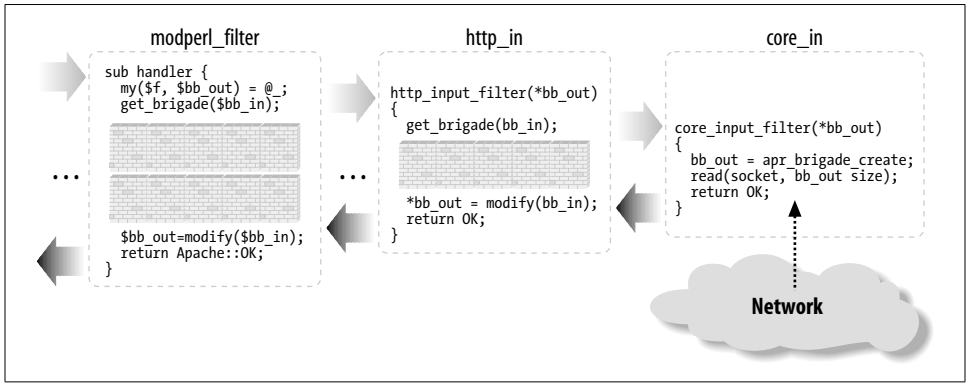

Figure 25-5. mod_perl 2.0 input filter program control and data flow

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

Figure 25-5 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 filter chain. 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), then it applies any modifications and passes 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, and so on, all the way down to the last filter (called `core`), which writes the data to the network, via the socket to which the client is listening. 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, because they have to wait for the `pass_brigade( )` call to return.

Figure 25-6 depicts a typical output filter chain data flow in addition to the program control flow. As in the input filter chain diagram, the arrows show the program control flow, and the black-headed arrows show the data flow. Again, the diagram uses Perl pseudocode for the mod_perl filter and C pseudocode for the Apache filters, and the brick walls represent the blocking. The diagram shows only part of the real HTTP response filter chain; `...` stands for the omitted filters.

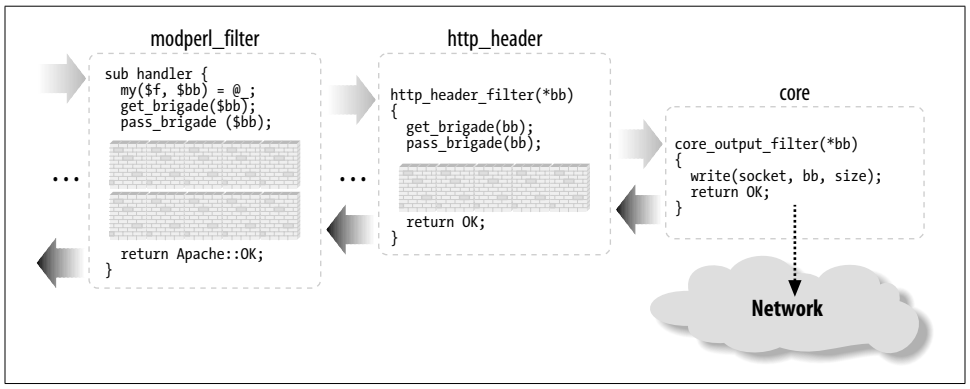

Figure 25-6. mod_perl 2.0 output filter program control and data flow

Filter Configuration

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

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

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

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

1;
```

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

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

```
PerlModule Book::FilterRequestFoo
PerlModule Book::NiceResponse
<Location /filter_foo>
    SetHandler modperl
    PerlResponseHandler     Book::NiceResponse
    PerlInputFilterHandler  Book::FilterRequestFoo::input
    PerlOutputFilterHandler Book::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 Book::FilterConnectionBar;
use base qw(Apache::Filter);

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

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

1;
```

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

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

    PerlInputFilterHandler  Book::FilterConnectionBar::input
    PerlOutputFilterHandler Book::FilterConnectionBar::output
```

```
        <Location />
            SetHandler modperl
            PerlResponseHandler Book::NiceResponse
        </Location>

    </VirtualHost>
```

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

Input Filters

We looked at how input filters call each other in Figure 25-5. Now let's look at some examples of input filters.

Bucket brigade–based connection input filter

Let's say that we want to test how our handlers behave when they are requested as `HEAD` requests rather than `GET` requests. 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 input filter handler that does that by directly manipulating the bucket brigades is shown in Example 25-5.

Example 25-5. Book/InputFilterGET2HEAD.pm

```
package Book::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($filter, $bb, $mode, $block, $readbytes) = @_;

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

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

Example 25-5. Book/InputFilterGET2HEAD.pm (continued)

```
    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
            $filter->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 basic task of any input filter handler is to request the bucket brigade from the upstream filter, and return it to the 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, and return bucket brigades:

- 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.
- 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, alhough the second technique is more efficient since it doesn't have the overhead of creating 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 `POST`ed data that may match `/^GET/` in one of the buckets). We use `$filter->ctx` as a flag here. When it's undefined, the filter knows that it hasn't done the required substitution; 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 $filter->ctx;
```

mod_perl then calls `get_brigade( )` internally, which passes the bucket brigade to the downstream filter. Alternatively, the filter could do:

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

but this is a bit less efficient.

If the job hasn't yet been done, 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 it finds it, a new bucket is created with the modified data `s/^GET/HEAD/`, and that bucket is 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 the bucket that we don't need any more:

```
$b->insert_after($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. Consider the response handler shown in Example 25-6.

Example 25-6. Book/RequestType.pm

```
package Book::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);
```

Example 25-6. Book/RequestType.pm (continued)

```
    Apache::OK;
}

1;
```

This handler 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 will still set the correct Content-Length header, which will be 24 in case of a GET request and 25 for HEAD. Therefore, if this response handler is configured as:

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

the response's body is:

```
the request type was GET
```

and the Content-Length header is set to 24.

However, if we enable the Book::InputFilterGET2HEAD input connection filter:

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

    <Location />
        SetHandler modperl
        PerlResponseHandler +Book::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. If Apache wasn't discarding the body of responses to HEAD requests, 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.

Bucket brigade–based HTTP request input filter

Let's look at the request input filter that lowers the case of the text in the request's body, `Book::InputRequestFilterLC` (shown in Example 25-7).

Example 25-7. Book/InputRequestFilterLC.pm

```
package Book::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($filter, $bb, $mode, $block, $readbytes) = @_;

    my $c = $filter->c;
    my $bb_ctx = APR::Brigade->new($c->pool, $c->bucket_alloc);
    my $rv = $filter->next->get_brigade($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 with this request or whether it has already done something. It's a stateless handler, since it has to lowercase everything that passes through it. Notice that this filter can't be used as a 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 `Book::Dump` response handler we developed earlier 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    +Book::Dump
    PerlInputFilterHandler +Book::InputRequestFilterLC
</Location>
```

when issuing a `POST` request:

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

we get a response like this:

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

We can see that our filter lowercased the `POST`ed body before the content handler received it, and the query string wasn't changed.

Stream-based HTTP request input filter

Let's now look at the same filter implemented using the stream-based filtering API (see Example 25-8).

Example 25-8. Book/InputRequestFilterLC2.pm

```
package Book::InputRequestFilterLC2;

use strict;
use warnings;
```

Example 25-8. Book/InputRequestFilterLC2.pm (continued)

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

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

use constant BUFF_LEN => 1024;

sub handler : FilterRequestHandler {
    my $filter = shift;

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

    Apache::OK;
}
1;
```

You've probably asked yourself why we had to go through the bucket-brigade filters when all this can be done so much more easily. The reason is that we wanted you to understand how the filters work underneath, which will help you when you need to debug filters or optimize their speed. Also, in certain cases a bucket-brigade filter may be more efficient than a stream-based one. For example, if the filter applies a transformation to selected buckets, certain buckets may contain open file handles or pipes, rather than real data. When you call `read()` the buckets will be forced to read in that data, but if you don't want to modify these buckets, you can pass them as they are and let Apache use a faster technique for sending data from the file handles or pipes.

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

`read()` populates `$buffer` to a maximum of `BUFF_LEN` characters (1,024 in our example). Assuming that the current bucket brigade contains 2,050 characters, `read()` will get the first 1,024 characters, then 1,024 characters more, and finally the remaining two characters. Notice that even though the response handler may have sent more than 2,050 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 showed 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 possible to get more than one bucket brigade from the same filter handler invocation only if the filter is not using the streaming interface—simply call `get_brigade()` as many times as needed or until the EOS token is received.

The configuration section is pretty much identical:

```
<Location /lc_input2>
     SetHandler modperl
     PerlResponseHandler    +Book::Dump
     PerlInputFilterHandler +Book::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 like this:

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

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

Output Filters

Earlier, in Figure 25-6, we saw how output filters call each other. Now let's look at some examples of output filters.

Stream-based HTTP request output filter

The PerlOutputFilterHandler handler registers and configures output filters.

The example of a stream-based output filter that we are going to present is simpler than the one that directly manipulates bucket brigades, although internally the stream-based interface is still manipulating the bucket brigades.

Book::FilterROT13 implements the simple Caesar-cypher encryption that replaces each English letter with the one 13 places forward or back along the alphabet, so that "mod_perl 2.0 rules!" becomes "zbq_crey 2.0 ehyrf!". Since the English alphabet consists of 26 letters, the ROT13 encryption is self-inverse, so the same code can be used for encoding and decoding. In our example, Book::FilterROT13 reads portions of the output generated by some previous handler, rotates the characters and sends them downstream.

The first argument to the filter handler is an Apache::Filter object, which as of this writing provides two methods, read() and print(). The read() method reads a chunk of the output stream into the given buffer, returning the number of characters read. An optional size argument may be given to specify the maximum size to read into the buffer. If omitted, an arbitrary number of characters (which depends on the size of the

bucket brigade sent by the upstream filter or handler) will fill the buffer. The `print()` method passes data down to the next filter. This filter is shown in Example 25-9.

Example 25-9. Book/FilterROT13.pm

```
package Book::FilterROT13;

use strict;

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

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

use constant BUFF_LEN => 1024;

sub handler {
    my $filter = shift;

    while ($filter->read(my $buffer, BUFF_LEN)) {
        $buffer =~ tr/A-Za-z/N-ZA-Mn-za-m/;
        $filter->print($buffer);
    }

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

Let's say that we want to encrypt the output of the registry scripts accessed through a */perl-rot13* location using the ROT13 algorithm. The following configuration section accomplishes that:

```
PerlModule Book::FilterROT13
Alias /perl-rot13/ /home/httpd/perl/
<Location /perl-rot13>
    SetHandler perl-script
    PerlResponseHandler ModPerl::Registry
    PerlOutputFilterHandler Book::FilterROT13
    Options +ExecCGI
    #PerlOptions +ParseHeaders
</Location>
```

Now that you know how to write input and output filters, you can write a pair of filters that decode ROT13 input before the request processing starts and then encode the generated response back to ROT13 on the way back to the client.

The request output filter can be used as the connection output filter as well. However, HTTP headers will then look invalid to standard HTTP user agents. The client should expect the data to come encoded as ROT13 and decode it before using it. Writing such a client in Perl should be a trivial task.

Another stream-based HTTP request output filter

Let's look at another example of an HTTP request output filter—but first, let's develop a response handler that sends two lines of output: the numerals 1234567890 and the English alphabet in a single string. This handler is shown in Example 25-10.

Example 25-10. Book/SendAlphaNum.pm

```
package Book::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 newline characters in their places. Since we want to reverse characters only in the response body, without breaking the HTTP headers, we will use an HTTP request output filter.

The first filter implementation (Example 25-11) uses the stream-based filtering API.

Example 25-11. Book/FilterReverse1.pm

```
package Book::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 $filter = shift;
```

Example 25-11. Book/FilterReverse1.pm (continued)

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

    Apache::OK;
}
1;
```

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

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

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

```
1234567890
abcdefghijklmnopqrstuvwxyz
```

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

```
0987654321
zyxwvutsrqponmlkjihgfedcba
```

The `Apache::Filter` module loads the `read()` and `print()` methods that encapsulate the stream-based 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 (1,024 in our example), then it prints each line reversed while preserving the newline control characters at the end of each line. Behind the scenes, `$filter->read()` retrieves the incoming brigade and gets the data from it, and `$filter->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 EOS token is received.

So as not to distract the reader from the purpose of the example, we've used oversimplified code that won't correctly handle input lines that are longer than 1,024 characters or use a different line-termination token (it could be "\n", "\r", or "\r\n", depending on the platform). Moreover, a single line may be split across two or even more bucket brigades, so we have to store the unprocessed string in the filter context so that it can be used in 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 newline token. On the next handler invocation, this data is then prepended to the next chunk that is read. When the filter is invoked for the last time, it unconditionally reverses and flushes any remaining data.

Bucket brigade-based HTTP request output filter

The filter implementation in Example 25-12 uses the bucket brigades API to accomplish exactly the same task as the filter in Example 25-11.

Example 25-12. Book/FilterReverse2.pm

```
package Book::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($filter, $bb) = @_;
```

Example 25-12. Book/FilterReverse2.pm (continued)

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

    Apache::OK;
}
1;
```

Here's the corresponding configuration:

```
PerlModule Book::FilterReverse2
PerlModule Book::SendAlphaNum
<Location /reverse2>
    SetHandler modperl
    PerlResponseHandler     Book::SendAlphaNum
    PerlOutputFilterHandler Book::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-based one. The handler receives the incoming bucket brigade `$bb` as its second argument. Because when it is completed, the handler must pass a brigade to the

next filter in the stack, we create a new bucket brigade, into which we 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` one at a time, reading the data from each bucket, reversing the data, and then putting it into a newly created bucket, which is inserted at the end of the new bucket brigade. If we see a bucket that designates the end of the stream, we insert that bucket at 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.

As in the original version of `Book::FilterReverse1::handler`, this filter is not smart enough to handle incomplete lines. The trivial exercise of making the filter foolproof by porting a better matching rule and using the `$leftover` buffer from the previous section is left to the reader.