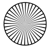
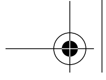
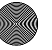


CHAPTER 18
mod_perl Data-Sharing Techniques

In this chapter, we discuss the ways mod_perl makes it possible to share data between processes or even between different handlers.

Sharing the Read-Only Data in and Between Processes

If you need to access some data in your code that's static and will not be modified, you can save time and resources by processing the data once and caching it for later reuse. Since under mod_perl processes persist and don't get killed after each request, you can store the data in global variables and reuse it.

For example, let's assume that you have a rather expensive function, `get_data( )`, which returns read-only data as a hash. In your code, you can do the following:

```
...
use vars qw(%CACHE);
%CACHE = get_data( ) unless %CACHE;
my $foo = $CACHE{bar};
...
```

This code creates a global hash, `%CACHE`, which is undefined when the code is executed for the first time. Therefore, the `get_data( )` method is called, which hopefully populates `%CACHE` with some data. Now you can access this data as usual.

When the code is executed for the second time within the same process, the `get_data( )` method will not be called again, since `%CACHE` has the data already (assuming that `get_data( )` returned data when it was called for the first time).

Now you can access the data without any extra retrieval overhead.

If, for example, `get_data( )` returns a reference to a list, the code will look like this:

```
....
use enum qw(FIRST SECOND THIRD);
use vars qw($RA_CACHE);
```

```
$RA_CACHE = get_data( ) unless $RA_CACHE;
my $second = $RA_CACHE->[SECOND];
...
```

Here we use the enum pragma to create constants that we will use in accessing the array reference. In our example, FIRST equals 0, SECOND equals 1, etc. We have used the RA_ prefix to indicate that this variable includes a reference to an array. So just like with the hash from the previous example, we retrieve the data once per process, cache it, and then access it in all subsequent code re-executions (e.g., HTTP requests) without calling the heavy get_data( ) method.

This is all fine, but what if the retrieved data set is very big and duplicating it in all child processes would require a huge chunk of memory to be allocated? Since we assumed that the data is read-only, can we try to load it into memory only once and share it among child processes? There is a feasible solution: we can run the get_data( ) method during server startup and place the retrieved data into a global variable of some new package that we have created on the fly. For example, let's create a package called Book::Cache, as shown in Example 18-1.

Example 18-1. Book/Cache.pm

```
package Book::Cache;

%Book::Cache::DATA = get_data( );
sub get_data {
    # some heavy code that generates/retrieves data
}
1;
```

And initialize this module from *startup.pl*:

```
use Book::Cache ( );
```

Now when the child processes get spawned, this data is available for them all via a simple inclusion of the module in the handler's code:

```
use Book::Cache ( );
...
$foo = $Book::Cache::DATA{bar};
...
```

Be careful, though, when accessing this data. The data structure will be shared only if none of the child processes attempts to modify it. The moment a child process modifies this data, the copy-on-write event happens and the child copies the whole data structure into its namespace, and this data structure is not shared anymore.

Sharing Data Between Various Handlers

Sometimes you want to set some data in one of the early handler phases and make it available in the latter handlers. For example, say you set some data in a TransHandler and you want the PerlHandler to be able to access it as well.

To accomodate this, Apache maintains a "notes" table (tables are implemented by the `Apache::Table` module) in the request record. This table is simply a list of key/value pairs. One handler can add its own key/value entry to the notes table, and later the handler for a subsequent phase can retrieve the stored data. Notes are maintained for the life of the current request and are deleted when the transaction is finished. The `notes()` method is used to manipulate the notes table, and a note set in one Apache module (e.g., mod_perl) can later be accessed in another Apache module (e.g., mod_php).

The `notes()` method accepts only non-reference scalars as its values, which makes this method unfit for storing non-scalar variables. To solve this limitation mod_perl provides a special method, called `pnotes()`, that can accept any kind of data structure as its values. However, the data set by `pnotes()` is accessible only by mod_perl.

The note gets set when the key/value pair is provided. For example, let's set a scalar value with a key *foo*:

```
$r->notes("foo" => 10);
```

and a reference to a list as a value for the key *bar*:

```
$r->pnotes("bar" => [1..10]);
```

Notes can be retrieved in two ways. The first way is to ask for the value of the given key:

```
$foo = $r->notes("foo");
```

and:

```
@bar = @{ $r->pnotes("bar") || [ ] };
```

Note that we expect the note keyed as *bar* to be a reference to a list.

The second method is to retrieve the whole notes table, which returns a hash reference blessed into the `Apache::Table` class:

```
$notes = $r->notes();
$foo   = $notes->{foo};
```

and:

```
$pnotes = $r->pnotes();
@bar    = @{ $pnotes->{bar} || [ ] };
```

Apache modules can pass information to each other via the notes table. Here is an example of how a mod_perl authentication handler can pass data to a mod_php content handler:

```
package Book::Auth;
...
sub handler {
    my $r = shift;
...
    $r->notes('answer',42);
```

```
...
}
1;
```

The mod_php content handler can retrieve this data as follows:

```
...
$answer = apache_note("answer");
...
```

You can use notes along with the subrequest methods lookup_uri( ) and lookup_filename( ), too. To make it work, you need to set notes in the subrequest object. For example, if you want to call a PHP subrequest from within mod_perl and pass it a note, you can do it in the following way:

```
my $subr = $r->lookup_uri('wizard.php');
$subr->notes('answer' => 42);
$subr->run;
```

As of the time of this writing you cannot access the parent request tables from a PHP handler; therefore, you must set this note for the subrequest. If the subrequest is running in the mod_perl domain, however, you can always keep the notes in the parent request notes table and access them via the main( ) method:

```
$r->main->notes('answer');
```

Similarly to the notes, you may want or need to use the Apache environment variables table to pass the information between different handlers.

If you know what environment variables you want to set before the server starts and you know their respective values, you can use SetEnv and PerlSetEnv in *httpd.conf*, as explained in Chapter 4. These settings will always be the same for all requests.

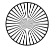
However, if you want to change or add some of the environment variables during the processing of a request, because some other handler that will be executed later relies on them, you should use the subprocess_env( ) method.

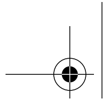
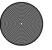
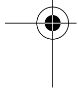
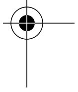
```
<!--#if expr="$hour > 6 && $hour < 12" -->
Good morning!
<!--#elif expr="$hour >= 12 && $hour <= 18" -->
Good afternoon!
<!--#elif expr="$hour > 18 && $hour < 22" -->
Good evening!
<!--#else -->
Good night!
<!--#endif -->
```

and you have the following code in your mod_perl handler:

```
my $hour = (localtime)[2];
$r->subprocess_env(hour => $hour);
```

The page will nicely greet the surfer, picking the greeting based on the current time. Of course, the greeting will be correct only for users located in the same time zone as the server, but this is just a simple example.



References

- mod_include, an Apache module that provides Server-Side Includes (SSI): *http://httpd.apache.org/docs/mod/mod_include.html*