A Caching CPAN Proxy

by Paul Miller
jettero@cpan.org

The Comprehensive Perl Archive Network (CPAN) is easily the coolest part of Perl. Over the years I've become addicted to so many modules on CPAN that installing a new Linux distribution typically means an hour or two of me installing Perl modules somewhere down the road. Since I maintain a few dozen machines, I find myself installing the same modules over and over again. I also find myself typing the same CPAN shell commands over and over again.

In the `CPAN.pm` shell, the setting that causes me the most grief is the urllist. It's a bit of a project for me to select a mirror site to download the modules from. For me to remember remember which machine had that good mirror, `ssh`-ing over there and copy/pasting it to the new machine quickly becomes tedious.

On my machines, I tend not to update modules when everything is working relatively well. Sometimes many months go by before I open a CPAN shell to install new modules (or update old ones). As a result, I find that each time I open the shell I have to pick a mirror.

I brainstormed one day that I could solve both problems by making my own CPAN mirror. I could simply set all my CPAN shells to point to my mirror and, once they all pointed to the same place, I'd only have to change my mirror settings in one place when I wanted to use a different mirror. I could also pull the same modules over and over from many machines without feeling like I'm wasting public resources.

Initially, I had looked at building an actual mirror. This is usually done with rsync and therefore creates a relatively heavy load on the server. It also takes up quite a bit of space (around 1GB), when I really only need a couple megs worth of files. Next, I looked at `CPAN::Mini`. It is definitely closer to what I had in mind, since it only pulls the newest versions and it allows users to filter out namespaces that aren't likely to be needed.

I never did build a mirror with `CPAN::Mini`. I only wanted to mirror the modules I was actually going to use. The problem with filtering is that I don't know know which modules I am going to use because the fifty or so I'm likely to install all have dependencies of their own.

Then it occurred to me that what I really wanted was a simple caching proxy. Furthermore, I wanted this to work via plain old vanilla CGI so I could install it on my shared hosting provider without having to jump through a lot of hoops.

■ My first proxy----------

So, first things first. Using Apache and the `$ENV{PATH_INFO}` environment variable, it's simple enough for me to construct a CGI script that can deal with paths deeper than the CGI:

```
use strict;
use warnings;
use CGI;
use CGI::Carp qw(fatalsToBrowser);

my $cgi = new CGI;

print $cgi->header("text/plain");
print $cgi->path_info, "\n";
```

Now, if I surf to *http://localhost/test.pl/extra1/extra2/extra3*, I get just the portion of the URL after the script name, which is the `path_info`:

```
/extra1/extra2/extra3
```

All I have to do to fetch files from a mirror is re-write the URL and pull the right file, although it would be best if I could pull the MIME type through so the CPAN shell gets the MIME type it expects.

LWP does everything a savvy coder needs for all web-related things, this case being no exception. In just a few lines I can get the HTTP status from the CGI header, the content type (also from the header) and the file contents:

```
use LWP::UserAgent;
use HTTP::Request;

my $mirror = "http://cpan.cs.utah.edu/";
my $ua     = new LWP::UserAgent;

my $path_info = $cgi->path_info;
$path_info =~ s/^\///;

my $new_url  = "$mirror$path_info";
my $request  = HTTP::Request->new(
  GET => $new_url);
my $response = $ua->request($request);
my $status   = $response->status_line;

print $cgi->header(-status=>$status,
  -type=>$response->header(
     'content-type' ));

  if( $response->is_success ) {
    print $response->content;
```

```
} else {
  print $status;
}
```

With the tiny program above, I can now set all my CPAN shells to point to it and satisfy one of my goals. If I change the mirror in one place all my hosts feel the change.

Of course, for really big files this CGI script will take up a large amount of RAM for no real reason. To avoid this, I initially decided to write the data to a temporary file, get the response headers, print them, and then print the contents of the temporary file:

```
use File::Temp qw( tempfile );
```

The really nice thing about `File::Temp` is that it decides the location of the temporary file automatically, opens it in read/write and cleans up after itself. So all I have to do is change these couple lines around a little:

```
my $temp_file = tempfile();

# my $response  = $ua->request($request);
my $response  = $ua->request($request,
  sub { my $chunk = shift;
    print $temp_file $chunk });
```

`LWP::UserAgent` uses the anonymous sub passed as the second argument to `request()` as a chunk handler. It sends each block through the sub as they come in rather than storing them all in memory as one huge string. Then below, instead of printing the response content (which is now empty because of that chunk handler), I just seek back to the top of the temporary file and print the contents. I do this little by little to avoid the memory problem I was solving.

```
# print $response->content;
seek $temp_file, 0, 0 or
  die "can't get to top of tempfile: $!";

my $buf;
while( read $temp_file, $buf, 4096 ) {
  print $buf;
  }
```

■ Adding caching ----------

I used the program as written for a few months before it started to nag me that I could do better. At work, I build huge file-packs full of my favorite Perl modules and the packs usually don't build right on the first try.

So not only am I downloading the same modules over and over on many machines, but I'm now downloading the same modules over and over on the *same* machine. I usually feel a pang of guilt when I'm selfishly consuming free resources needlessly.

I had previously flirted with the `Cache::Memcached` module. The whole idea of a distributed memory cache daemon is just fantastic and I had a lot of fun with it. Usually when I play around with a module, I go source diving to see what it does and how it does it. I was familiar with the layout of the `Cache::*` tree and it felt like a natural fit for my mirror. Clearly `Cache::Memcached` is a poor fit, but `Cache::File` certainly isn't. In fact, it's perfect.

When I looked at creating a CPAN mirror, I decided it would take up too much space on my servers and would ultimately hurt the CPAN mirror I was trying to help. Ideally, the mirror should only pull and mirror modules that I am actually going to be use immediately.

The problem really isn't a lot different than the one I was solving before. Instead of using `File::Temp` to store just the content, I'm going to use a `Cache::File`, and I have to also store the HTTP status and the content type of the result.

To guarantee that the cache results work the same as regular `LWP` results, I chose a two stage model. First, I see if there's a cache entry for the URL and use it if there is. Or, if it's not there, I download the file, build a cache entry and return to the first step to display it from the cache:

```
my $new_url = "$mirror$path_info";

my $cache   = Cache::File->new(
     cache_root   => "/tmp/testc/",
   default_expires => "2 day"
   );
```

The `$cache` object takes care of all the file and directory management. I just need to tell it where to put the directory and it does all the heavy lifting for me:

```
my $only_once = 1;

MAIN_LOOP: {
  if( $cache->exists($new_url) and
    $cache->exists("$new_url.hdr") ) {
    my $VAR1;
    my $response =
      eval $cache->get("$new_url.hdr");

    die "problem finding cache entry\n"
       if $@;
```

I store the `HTTP::Response` object as the output from `Data::Dumper` so when I evaluate it I can use it just like any other `HTTP::Response` object. Of course, if something is wrong with the cache entry, the `eval()` will fail and set `$@`. That really shouldn't come up very often unless there are some larger problems with the system the cache is running on.

The rest of this block looks just like it did before except for the way my program opens the input file handle:

```
my $status = $response->status_line;

  print $cgi->header(
    -status => $status,
    -type   => $response->header(
      'content-type' )
      );

  if( $response->is_success ) {
    my $fh = $cache->handle(
      $new_url, "<" ) or die
        "problem finding cache entry\n";

    my $buf;

    while( read $fh, $buf, 4096 ) {
      print $buf;
      }
    close $fh;

    } else {
      print $status;
    }
```

Not all of the different `Cache::*` modules can return file handles for keys. The ones that can are all usable for this purpose. The ease with which `Cache::File` drops into place is really quite amazing:

```
    } elsif( $only_once ) {
      # NOTE: without this, the handle()
      # won't create the key
      $cache->set( $new_url, 1 );

      my $temp_file = $cache->handle(
                  $new_url, ">" );
      my $request   = HTTP::Request->new(
        GET => $new_url);

      my $response  = $ua->request(
        $request,
        sub {
          my $chunk = shift;
          print $temp_file $chunk }
          );

      close $temp_file;
```

This all works just like before, too, except that I push the data to the cache instead of printing it to the temporary file. Writing the `LWP` results to the cache is not very different from writing it to the temporary file.

The biggest difference is using the `Cache::File::handle()` method instead of writing to a temporary file handle. There is one other difference; I need to also cache the document type and the HTTP status returned by `LWP`.

```
  $cache->set(
           "$new_url.hdr",
           Dumper($response)
           );
```

I have chosen to dump the entire `HTTP::Response` object to a cache entry so that I can continue to use ordinary `LWP` syntax in the top of the if block. Arguably, it makes sense to store only the bits I actually need like this instead:

```
    $cache->set( "$new_url.hdr",
      Dumper({
        status => $response->status_line,
        mime   => $response->header(
         'content-type' )),
        })
        );
```

Perhaps that's a task for another day. It doesn't save me an `eval()` since I still have to stringify the data, so perhaps not. It appears to be a matter of preference.

```
            $only_once = 0;
            redo MAIN_LOOP;
        } else {
            die "problem with $new_url";
        }
    }
```

This global `$only_once` variable exists to prevent my CGI from ever looping infinitely from some unforeseen problem.

■ Conclusion

This program has evolved a bit from the form in this article. I retooled it as a module and released it on CPAN. Now it only takes a couple seconds to set up my proxy using `CPAN::CachingProxy`.

The moral of the story is obvious. Nearly everything I needed to make the caching proxy was already on CPAN. And now the caching proxy itself is there, too. CPAN sometimes seems to contain *too many* choices, but there's rarely a need to start from scratch on any project.

In this case, I barely wrote a hundred lines of glue and everything worked the way I wanted.

■ References

All the modules used in the article, and their documentation, are available on CPAN.

Information about creating a CPAN rsync mirror can be found in the FAQ on *http://www.cpan.org/misc/cpan-faq.html.*

■ About the author

Paul Miller (*jettero@cpan.org*) received his Bachelor's degree from Western Michigan University in Computer Science (Computational). He's been writing Perl programs for work and play, pretty much every day, since 1997. Obsessions not including Perl: his wife, his infant son, skydiving, and Go.