A bag of tricks for the productive Perl enthusiast

Tricks and Treats

If you are a frequent shell user who navigates, searches for text, or installs CPAN modules, you will definitely appreciate these helper scripts and modules to take some typing off your hands. *By Mike Schilli*

I recently moved to a new developer desktop and took that opportunity not just to tidy up my overflowing home directory, but to rebuild it completely. Hundreds of partially obsolete helper scripts had accumulated over the years. To introduce some kind of order into this chaos, I decided to start from scratch and install any script I really missed in my daily work – and in a reproducible way, of course, to take the pain out of the next move.

Repo Links

All of my scripts have moved to the subdirectories of various Git repositories for versioning and replication. To allow the user to call the helpers without specifying a path, symlinks from the `bin` path in my home directory point to the actual checked-in scripts. Another script, `binlinks` (Listing 1), maps the scripts checked into the Git repository to links in the user's local `bin` directory in its `DATA` area. For example, the `logtemp` script, which I use to query my temperature sensor [2], stays in the `articles` Git repository; whereas my handwritten repository converter tool, `cvs2git`, is better off in the experimental Schilli Labs `sandbox` repository.

When a new script needs to be added to the `bin` path, the developer just needs to append it to the `binlinks` `DATA` area and then call the `binlinks` script. The script then checks all the entries to see if the required link already exists in `~/bin` and creates a link if this is not the case. That `binlinks` itself resides in a Git repository should be self-explanatory. It uses the Sysadm::Install CPAN module, just because of the `mkd` function, which creates new directories at arbitrary depths without further ado and generates informative Log4perl output to document the process.

Follow the Link

As you can see, many scripts that are called now are actually symlinks. If a symlink points to a file in another directory, developers will want to change into that directory for further development work. The `lcd` command, with the link as a parameter, takes care of this (see Figure 1).

Old-school Unix users will, of course, know that a shell script can't change the user's current directory. Scripts are running in subshells, and when they terminates, there are no notable side effects for whatever called it. For this reason, `lcd` is defined as a Bash function in the `.bashrc` startup file of the Bash shell:

```
function lcd () { cd `symlinkdir $1`; \
                  pwd; ls; }
```

MIKE SCHILLI

Mike Schilli works as a software engineer with Yahoo! in Sunnyvale, California. He can be contacted at *mschilli@perl-meister.com*. Mike's homepage can be found at *http://perlmeister.com*.

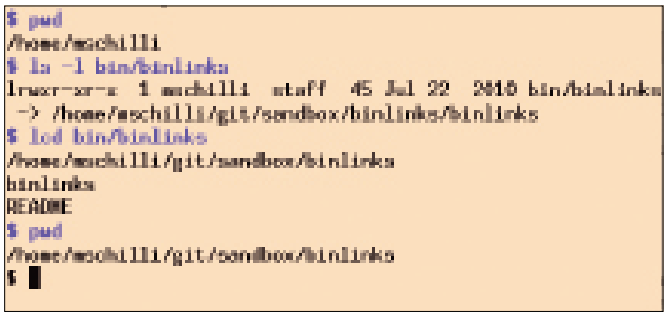

Figure 1: The `lcd` function changes to the directory containing the script to which a symlink points.

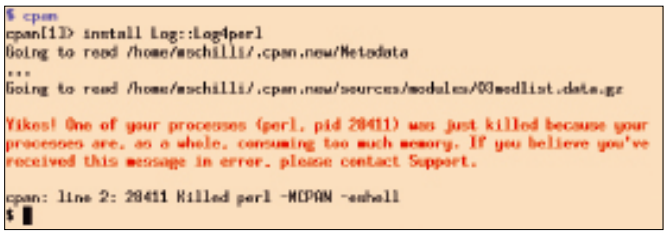

Figure 2: Too much for cheap hosters: Using a CPAN shell to install a Perl module is tantamount to pulling the ripcord.

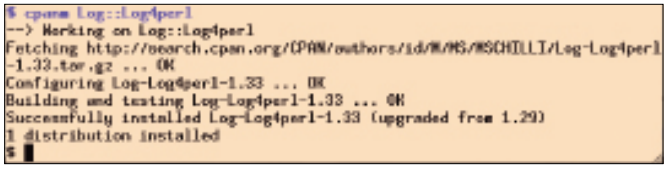

Figure 3: The frugal App::cpanminus CPAN module and its helper script `cpanm` install the required module without any trouble.

LISTING 1: binlinks

```
01 #!/usr/local/bin/perl -w
02 ############################
03 # binlinks - Link git-ver-
04 # sioned scripts to bin dir
05 # Mike Schilli, 2011
06 # (m@perlmeister.com)
07 ############################
08 use strict;
09 use Log::Log4perl qw(:easy);
10 use File::Basename;
11 use Sysadm::Install qw(mkd);
12
13 Log::Log4perl->easy_init(
14  $DEBUG);
15
16 my ($home) = glob "~";
17 my $home_bin = "$home/bin";
18
19 while (<DATA>) {
20  chomp;
21
22  my ($linkbase, $src) =
23    split ' ', $_;
24
25  $src = "$home/$src";
26  my $binpath =
27    "$home_bin/$linkbase";
28
29  if (-l $binpath) {
30   DEBUG
31    "$binpath already exists";
32   next;
33  } elsif (-e $binpath) {
34   ERROR
35   "$binpath already exists,",
36   " but not a link!";
37   next;
38  }
39
40  INFO
41   "Linking $binpath -> $src";
42
43  symlink $src, $binpath
44    or LOGDIE "Cannot link " .
45       "$binpath->$src ($!)";
46 }
47
48 __DATA__
49 logtemp git/articles/temper/eg/
           logtemp
50 cvs2git git/sandbox/cvs2git/cvs2git
```

If somebody calls `lcd bin/cvs2git`, the Bash function passes the `bin/cvs2git` argument to the `symlinkdir` script and then calls the `cd` command with the printed directory. Listing 2 shows the implementation of `symlinkdir`.

The script uses the `readlink()` system call to follow the link passed in as parameter and repeats this until the result is no longer a link. The `dirname()` function from the File::Basename module extracts the directory from the resulting path and line 19 prints it on standard output, where the Bash `lcd()` function picks it up, changes to the corresponding directory, outputs the directory with `pwd`, and then calls `ls` to list its entries.

Frugal CPAN Installer

Hardly a single Perl column does without installing additional CPAN modules. This is normally a short and painless process thanks to a CPAN shell, which you can call either as `perl -MCPAN -eshell` or by using the `cpan` command that accompanies moderately recent Perl distributions. However, because the CPAN shell is not exactly frugal with its use of resources, this process can quickly lead to a developer's account being shut down by cheap hosting providers.

Figure 2 shows what happens on the shared hosting provider DreamHost, even before the CPAN shell can load the desired module from CPAN. Allegedly, it uses too much memory, and to avoid other shared accounts suffering, DreamHost pulls the plug – a bit too early for my liking.

This is where the CPAN module App::cpanminus enters the scene. Figure 3 shows the terse output of this unimposing jack of all trades. It is so frugal in its use of resources that even cheap hosting providers with their strict rules don't notice a strain on their resources and let it proceed unencumbered.

Just like its bigger sibling `CPAN.pm`, `cpanminus` can also handle local module

LISTING 2: symlinkdir

```
01 #!/usr/local/bin/perl -w
02 use strict;
03 use File::Basename;
04
05 my ($link) = @ARGV;
06
07 die "No link specified"
08   unless $link;
09 die
10   "$link not a symbolic link"
11   unless -l $link;
12
13 while (-l $link) {
14  $link = readlink($link);
15 }
16
17 $link = dirname($link)
18   unless -d $link;
19 print "$link\n";
```

paths. To allow users with nonprivileged accounts to install CPAN modules, and to avoid messing up the package manager's well-organized system, experts strongly recommend the use of local::lib if your choice of Linux distribution doesn't include a Perl module you need in its package repository.

Working as the administrator root (for the last time), you can employ the package manager to install local::lib. On Ubuntu, you would do:

```
sudo apt-get install liblocal-lib-perl
```

If a hosting provider doesn't permit root access and does not allow you to install the very useful local::lib, you can download the tarball from CPAN, unpack it, and type the following:

```
perl Makefile.PL --bootstrap
make install
```

Then, you can append the following to your Bash shell startup file (typically `.bashrc`):

```
eval $(perl -I$HOME/perl5/lib/perl5 ➥
            -Mlocal::lib)
```

This code will set the `PERL_MM_OPT` and `PERL5LIB` variables so that any modules you now install with a CPAN shell (or `cpanminus`) end up in the `perl5` directory below the nonprivileged user's home directory.

This happens when a user types `make install`. At the same time, Perl scripts that list `use SomeModule` in their code to bind a Perl module will be able to find them in the local path.

If a script (such as a cronjob) doesn't have access to these environment variables from `.bashrc`, an explicit `use local::lib` entry inserted into the program code before loading the required locally installed modules will do the trick, too.

LISTING 3: perltidyrc

```
01 # perltidy Options for Perl scripts
     in Linux Magazine
02
03 -l=29   # line width
04 -i=2    # 2 cols indent
05 -ci=2   # 2 cols continuation indent
06 -ce     # cuddled else
07 -vt=2   # vertical tightness
08 -nbbc   # no blank lines before
             whole-line comments
```

Finding Text

Quite often, developers search for a specific text string they know is contained somewhere within the various files of a project's source code. If the text "blabla" is hiding somewhere in a file below the current path, in the shell, you could run the `find` command:

```
find . -type file -exec grep blabla {} ➥
     /dev/null \;
```

But this does mean a huge amount of typing, and you really have to think – especially when it comes to the trick with `/dev/null`, which also shows the file names for single matches, and the masked semicolon, which is strangely necessary to tell the `-exec` option that the command passed to it is now complete. I used to use a `findgrep` script to launch a recursive text search, but with `ack` [3], users can simply load the powerful command from CPAN (`cpanm ack`), type

```
$ ack blabla
```

and be done with it. Having said this, `ack` is fairly strict about file types. It will only search files that look like text on the basis of the name suffix; if you want to search all files, you need to type `ack -a blabla`. If performance is important to you, the often overlooked

```
$ git grep blabla
```

is a better choice in a Git repository. Because `git` saves the files it manages in an index, it doesn't need to walk the file trees for a recursive search. This process will win hands down against less sophisticated approaches, especially for files that are not yet in the operating system's buffer cache and that reside in deeply nested folders.

Automatic Formatting

To make sure your homegrown Perl scripts comply with existing standards, well-behaved programmers will run them through the `perltidy` beautifier when they are done. The script is available as a CPAN module (`cpanm perltidy`) and supports a plethora of configurations that will match any style. Where are you placing the curly brackets – in the `if` line or the line following it? Does `else` directly follow the closing curly bracket ("cuddled else"), or does it follow after a line break? Do empty lines occur between round brackets, separating function calls and their arguments? And most importantly, what is the maximum line length, and when should the formatter split long lines of code?

The `perltidy` manual page lists options for all these styles and describes their effects. Listing 3 shows the configuration for Perl listings in *Linux Magazine*. The line width is a rather challenging 29 characters, and the formatter indents lines in blocks by two characters (`-i=2`). If it splits a line, it indents the rest of the line by two characters (`-ci=2`) in the next line. The `else` instructions follow the closing curly bracket of the `if` block directly, without a line break ("cuddled else," `-ce`). Because space is at a premium in the magazine and the editors don't like to waste it, "vertical tightness" is set to the maximum value, `-vt=2`. Using this option, the formatter saves line breaks wherever it can. Finally, to save even more space, `-nbbc` specifies that there are no empty lines in front of full-line comments.

To tell the formatter to remodel a Perl script using the defined options, developers need to call it as follows:

```
perltidy -pro=path/perltidyrc scriptname
```

If the script is syntactically correct, the result is a `scriptname.tdy` file with the right formatting. If you prefer, you can do this

```
:nnoremap <buffer> <silent> X
:w<Enter>1GdG\
:.!perltidy -pro=path/perltidyrc
<%<Enter>
```

to create a vim command that handles the formatting in the editor when you press *X*. Now that's what I call convenient. ■■■

INFO

[1] Listings for this article: *http://www.linux-magazine.com/Resources/Article-Code*

[2] "RRDtool" by Michael Schilli, *Linux Magazine*, November 2010, pg. 62

[3] "ack": *http://betterthangrep.com*