News, tips, partners, and perspectives for the Oracle Linux operating system and upstream Linux kernel work

PERSPECTIVES, TECHNOLOGIES

May 27, 2010

The top 10 tricks of Perl one-liners

Guest Author

I'm a recovering perl hacker. Perl used to be far and away my language of choice, but these days I'm more likely to write new code in Python, largely because far more of my friends and coworkers are comfortable with it.

I'll never give up perl for quick one-liners on the command-line or in one-off scripts for munging text, though. Anything that lasts long enough to make it into git somewhere usually gets rewritten in Python, but nothing beats perl for interactive messing with text.

Perl, never afraid of obscure shorthands, has accrued an impressive number of features that help with this use case. I'd like to share some of my favorites that you might not have heard of.

One-liners primer

We'll start with a brief refresher on the basics of perl one-liners before we begin. The core of any perl one-liner is the *-e* switch, which lets you pass a

snippet of code on the command-line:
perl -e 'print "hi\n"' prints "hi" to the console.

The second standard trick to perl one-liners are the *-n* and *-p* flags. Both of these make perl put an implicit loop around your program, running it once for each line of input, with the line in the *$_* variable. *-p* also adds an implicit *print* at the end of each iteration.

Both of these use perl's special "ARGV" magic file handle internally. What this means is that if there are any files listed on the command-line after your *-e*, perl will loop over the contents of the files, one at a time. If there aren't any, it will fall back to looping over standard input.

perl -ne 'print if /foo/' acts a lot like *grep foo*, and *perl -pe 's/foo/bar/'* replaces *foo* with *bar*

Most of the rest of these tricks assume you're using either *-n* or *-p*, so I won't mention it every time.

The top 10 one-liner tricks

Trick #1: *-l*
Smart newline processing. Normally, perl hands you entire lines, including a trailing newline. With *-l*, it will strip the trailing newline off of any lines read, and automatically add a newline to anything you *print* (including via *-p*).

Suppose I wanted to strip trailing whitespace from a file. I might naïvely try something like

```
perl -pe 's/\s*$//'
```

The problem, however, is that the line ends with *"\n"*, which is whitespace, and so that snippet will also remove all newlines from my file! *-l* solves the problem, by pulling off the newline before handing my script the line, and then tacking a new one on afterwards:

```
perl -lpe 's/\s*$//'
```

Trick #2: *-0*

Occasionally, it's useful to run a script over an entire file, or over larger chunks at once. *-0* makes *-n* and *-p* feed you chunks split on *NULL* bytes instead of newlines. This is often useful for, e.g. processing the output of *find -print0*. Furthermore, *perl -0777* makes perl not do any splitting, and pass entire files to your script in *$_*.

```
find . -name '*~' -print0 | perl -0ne unlink
```

Could be used to delete all ~-files in a directory tree, without having to remember how *xargs* works.

Trick #3: *-i*

-i tells perl to operate on files in-place. If you use *-n* or *-p* with *-i*, and you pass perl filenames on the command-line, perl will run your script on those files, and then replace their contents with the output. *-i* optionally accepts an backup suffix as argument; Perl will write backup copies of edited files to names with that suffix added.

```
perl -i.bak -ne 'print unless /^#/' script.sh
```

Would strip all whole-line commands from *script.sh*, but leave a copy of the original in *script.sh.bak*.

Trick #4: The *..* operator

Perl's *..* operator is a stateful operator -- it remembers state between evaluations. As long as its left operand is false, it returns false; Once the left hand returns true, it starts evaluating the right-hand operand until that becomes true, at which point, on the next iteration it resets to false and starts testing the other operand again.

What does that mean in practice? It's a range operator: It can be easily used to act on a range of lines in a file. For instance, I can extract all GPG public keys from a file using:

```
perl -ne 'print if /-----BEGIN PGP PUBLIC KEY BLOCK-----/../-----END PGP
PUBLIC KEY BLOCK-----/' FILE
```

Trick #5: *-a*

-a turns on autosplit mode – perl will automatically split input lines on whitespace into the *@F* array. If you ever run into any advice that accidentally escaped from 1980 telling you to use awk because it

automatically splits lines into fields, this is how you use perl to do the same thing without learning another, even worse, language.

As an example, you could print a list of files along with their link counts using

```
ls -l | perl -lane 'print "$F[7] $F[1]"'
```

Trick #6: *-F*

-F is used in conjunction with *-a*, to choose the delimiter on which to split lines. To print every user in */etc/passwd* (which is colon-separated with the user in the first column), we could do:

```
perl -F: -lane 'print $F[0]' /etc/passwd
```

Trick #7: \K

\K is undoubtedly my favorite little-known-feature of Perl regular expressions. If *\K* appears in a regex, it causes the regex matcher to drop everything before that point from the internal record of "Which string did this regex match?". This is most useful in conjunction with *s///*, where it gives you a simple way to match a long expression, but only replace a suffix of it.

Suppose I want to replace the *From:* field in an email. We could write something like

```
perl -lape 's/(^From:).*/$1 Nelson Elhage <nelhage\@ksplice.com>/'
```

But having to parenthesize the right bit and include the *$1* is annoying and error-prone. We can simplify the regex by using *\K* to tell perl we won't want to replace the start of the match:

```
perl -lape 's/^From:\K.*/ Nelson Elhage <nelhage\@ksplice.com>/'
```

Trick #8: $ENV{}

When you're writing a one-liner using *-e* in the shell, you generally want to quote it with ', so that dollar signs inside the one-liner aren't expanded by the shell. But that makes it annoying to use a ' inside your one-liner, since you can't escape a single quote inside of single quotes, in the shell.

Let's suppose we wanted to print the username of anyone in */etc/passwd* whose name included an apostrophe. One option would be to use a standard shell-quoting trick to include the ':

```
perl -F: -lane 'print $F[0] if $F[4] =~ /'"'"'/' /etc/passwd
```

But counting apostrophes and backslashes gets old fast. A better option, in my opinion, is to use the environment to pass the regex into perl, which lets you dodge a layer of parsing entirely:

```
env re="'" perl -F: -lane 'print $F[0] if $F[4] =~ /$ENV{re}/'
/etc/passwd
```

We use the *env* command to place the regex in a variable called *re*, which we can then refer to from the perl script through the *%ENV* hash. This way is slightly longer, but I find the savings in counting backslashes or quotes to be worth it, especially if you need to end up embedding strings with more than a single metacharacter.

Trick #9: BEGIN and END
BEGIN { ... } and *END { ... }* let you put code that gets run entirely before or after the loop over the lines.

For example, I could sum the values in the second column of a CSV file using:

```
perl -F, -lane '$t += $F[1]; END { print $t }'
```

Trick #10: -MRegexp::Common
Using *-M* on the command line tells perl to load the given module before running your code. There are thousands of modules available on CPAN, numerous of them potentially useful in one-liners, but one of my favorite for one-liner use is Regexp::Common, which, as its name suggests, contains regular expressions to match numerous commonly-used pieces of data.

The full set of regexes available in Regexp::Common is available in its documentation, but here's an example of where I might use it:

Neither the *ifconfig* nor the *ip* tool that is supposed to replace it provide, as far as I know, an easy way of extracting information for use by scripts. The *ifdata* program provides such an interface, but isn't installed everywhere. Using perl and Regexp::Common, however, we can do a pretty decent job of extracing an IP from *ips output:*

```
ip address list eth0 | \
  perl -MRegexp::Common -lne 'print $1 if /($RE{net}{IPv4})/'
```

So, those are my favorite tricks, but I always love learning more. What tricks have you found or invented for messing with perl on the command-line? What's the most egregious perl "one-liner" you've wielded, continuing to tack on statements well after the point where you should have dropped your code into a real script?

~nelhage

Join the discussion

Comments (6)

Recent Content

TECHNOLOGIES

Announcing Oracle VirtIO Drivers 1.1.5 for Microsoft Windows

We are pleased to announce Oracle VirtIO Drivers for Microsoft Windows release 1.1.5. The Oracle VirtIO Drivers for Microsoft Windows are...

TECHNOLOGIES

Announcing Oracle Linux 7 Update 8 Beta Release

We are pleased to announce the availability of the Oracle Linux 7 Update 8 Beta release for the 64-bit

Intel and AMD (x86_64) and 64-bit...