Exploring filters and pipes

Posted at 10:49pm on Tuesday March 31st 2009

When many newbies first encounter Linux, the 'cool stuff' that often gets their attention is the incredible array of command line tools, and something called a pipe that allowed you to connect them together. Together, these provide an incredibly powerful component-based architecture designed to process streams of text-based data.

If you've never dabbled with filters and pipes before, or perhaps you've just been too scared, we want to help you out, so read on to learn how you can make powerful Linux commands just by stringing smaller bits together...

A filter is a program that reads a single input stream, transforms it in some way, and writes the result to a single output stream, as shown in Figure 1, below. By default, the output stream (called standard output or just stdout) is connected to the terminal window that the program is running in, and the input stream (standard input, or just stdin) is connected to the keyboard, though in practice filters are rarely used to process data typed in manually at the keyboard.

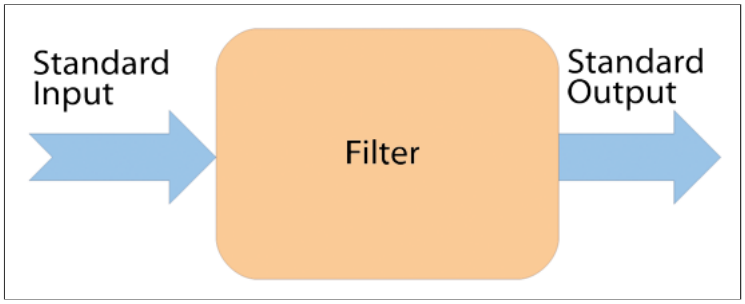

Figure 1: A filter has a single input stream and a single output stream.

If a filter is given a file name as a command line argument, it will open that file and read from it instead of reading from stdin, as shown in Figure 2 below. This is a much more common arrangement. There are a few sample commands in the box opposite. By themselves, many individual filters don't do anything exciting. It's what you can do with them in combination that gets interesting.

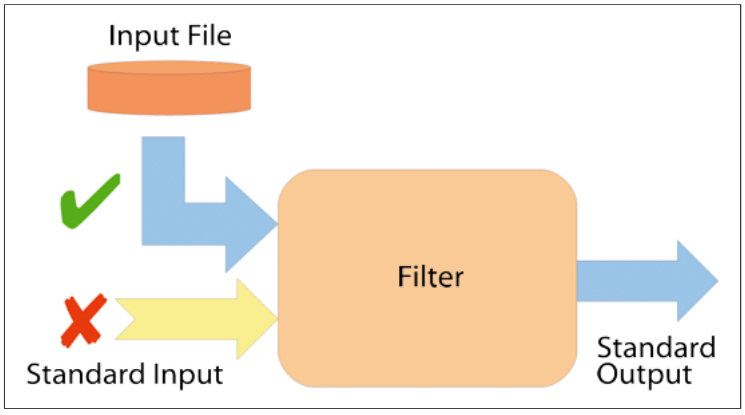

Figure 2: Given a filename, a filter reads it, ignoring its standard input.

Combining filters: pipes

A pipe allows data to flow from an output stream in one process to an input stream in another process. Usually, it's used to connect a standard output stream to a standard input stream. Pipes are very easy to create at the shell command line using the | (vertical bar) symbol. For example, when the shell sees this command line:

```
$ sort foo.txt | tr '[A-Z]' '[a-z]'
```

it runs the programs sort and tr in two separate, concurrent processes, and creates a pipe that connects the standard output of sort (the 'upstream' process) to the standard input of tr (the 'downstream' process). The

plumbing for this is shown in Figure 3, below.

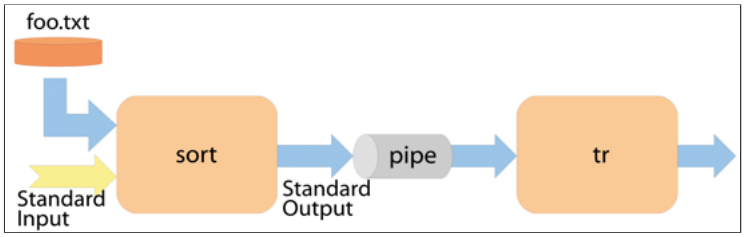

Figure 3: Plumbing for the command "sort foo.txt | tr '[A-Z]' '[a-z]'".

This particular tr command, if you're interested, replaces all characters in the set [A-Z] by the corresponding character from the set [a-z]; that is, it translates upper-case to lower-case.

Capture standard output

Most of our examples assume that standard output is connected to the terminal window, which is the default behaviour. However, it's easy to send stdout to a file instead using the shell's > operator. For example:

```
$ sort foo.txt | tr '[A-Z]' '[a-z]' > sorted.txt
```

...runs the same pipeline as before but redirects the stdout from tr into the file sorted.txt. Note: it's the shell that performs this re-arrangement of the plumbing; the tr command simply writes to stdout and neither knows nor cares where this is directed.

Semi-filters

Linux has many command-line tools that aren't really filters, but which nonetheless write their results to stdout and so can be used at the head of a pipeline. Examples include ls, ps, df, du, but there are lots of others. So for example,


```
$ ps aex | wc
```

will count the number of processes running on the machine, and

```
$ ls -l | grep '^l'
```

will show just the symbolic links in the current directory. (Regular expression ^l matches those lines that begin with the letter 'l'.)

The other case of a semi-filter (that is, programs that read stdin but don't write to stdout) is less common. The only ones that come to mind are the browsing program less, the printing utility lpr, and the command-line mailer mail. Such programs can be used at the downstream end of a pipeline, for example:

```
grep syslog /var/log/messages | less
```

...searches for entries in /var/log/messages relating to syslog, and browses the output using less. Using less on the downstream end of a pipeline in this way is very common.

You can put perform complex and powerful actions by trying out some of the filter commands in the table below.

Commonly used filters

Filter	What it does

cat	Copies input direct to output.
head	Shows beginning of a file (default 10 lines).
tail	Shows end of a file (default 10 lines).
wc	Counts characters, words and lines.
sort	Sorts input lines.
grep	Shows lines that match a regular expression.
tr	Translates or deletes specified character sets.
sed	Stream editor.
uniq	Discards all but one of successive identical lines.
awk	Highly-programmable field-processing.

sed - the stream editor

The stream editor sed supports automated editing of text and is a more flexible filter than most. It streams its input (either stdin, or a specified input file) line by line, applies one or more editing operations, and writes the resulting lines to stdout.

Actually, sed has quite a repertoire of commands, but the most useful is the substitute command. As a simple example to get us started, let's use sed to perform a simple substitution on text entered directly from the keyboard (stdin):

```
$ sed 's/rich/poor/g'
He wished all men as rich as he
He wished all men as poor as he
And he was as rich as rich could be
And he was as poor as poor could be
^D
$
```

A little explanation may be in order. The sed command used here replaces all occurrences of the string 'rich' by the string 'poor'. The /g suffix on the command causes the replacement to be "global" - that is, if there is more than one occurrence of the original string within the line, they will all be replaced. (Without the /g suffix, only the first one would be replaced.) Following the sed command we see two pairs of lines. The first of each pair is the text we typed in on the keyboard, and the second of each pair is the edited line that sed has written to its standard output.

Here's a more useful example, in which we use a regular expression to strip out all fields, except the first, from the file /etc/passwd:

```
$ sed 's/:.*//' /etc/passwd
root
daemon
bin
sys
... more lines deleted ...
```

Here, the text we're replacing is the regular expression :.*, which matches from the first colon to the end of the line. The replacement string (between the second and third slashes) is empty, so that whatever the regular expression matches is deleted from the line. In this case, we end up with a list of the account names in the passwd file.

Let's be clear about one thing, though - sed is not actually making any changes to /etc/passwd. It is simply reading the file as input and writing the modified lines to stdout.

awk

Named after its inventors (Aho, Weinberger and Kernighan) awk is really in a league of its own, being a fully-fledged programming language with variables, loops, branches, functions, and more. An awk program consists of one or more pattern-action pairs like this:

```
pattern { action }
pattern { action }
```

The pattern is some sort of test that is applied to each line; if the pattern matches, the corresponding action is taken. If the pattern is omitted, the action is taken for every line. If the action is omitted, the default action is to print the entire line. Awk excels at processing textual data that's organised into fields (ie columns); it reads its input line by line and automatically splits it into fields, making them available in the special variables $1, $2, $3 etc.

To demonstrate awk we'll use a small sample of geographic data for the accuracy of which I put my trust in Collins Complete World Atlas. The data shows countries, areas, populations (in thousands), languages spoken, and currency. For brevity we're restricting ourselves to four lines of data that look like this:

```
Country      Area    Population  Languages              Currency
Albania      28748   3130        Albanian,Greek         Lek
Greece       131957  11120       Greek                  Euro
Luxembourg   2586    465         German,French          Euro
Switzerland  41293   7252        German,French,Italian  Franc
```

The data is in the file geodata.

Many awk programs are simple enough to enter as a command line argument to awk. Here's an example:

```
$ awk '{ print $1, $5 }' geodata
Country Currency
Albania Lek
Greece Euro
Luxembourg Euro
Switzerland Franc
```

This awk program has just a single pattern-action. The pattern is missing, so the action is taken on every line. The action is to print the first and fifth fields; ie, the country name and the currency.

Next, let's find those countries that use Euros. We can use a pattern to make a test on the value in the fifth field, like this:

```
$ awk ' $5=="Euro" { print $1 }' geodata
Greece
Luxembourg
```

What about calculating the total population? This needs two pattern-actions, one that fires on every line to accumulate the running sum of the population (the values in the third field), and one that fires right at the end to print out the result. Although it would be possible to enter this program directly on the command line, as we have in the examples so far, it's getting a little longer so it's more convenient to put it into a separate file, which I called totalpop.awk. It looks like this:

```
{ sum += $3 }
END { print sum }
```

The first action has no pattern, so it takes place on every line. The variable sum is just a name I made up. Variables do not need to be pre-declared in awk; they spring into existence (and are initialised to zero) at the mere mention of their name. The second action uses the special pattern 'END' that fires just once, after all the input has been processed, to print out the result.

Now, I can run awk and have it take its program from the file totalpop.awk like this:

```
$ awk -f totalpop.awk geodata
21967
```

Notice the use of the -f flag in the first line above to specify the file containing the awk program. How would we set about finding those countries in which a specified language is spoken? This is a little harder because each country has a comma-separated list of one or more languages and we will have to split this list apart. Fortunately, awk provides a built-in function for doing just that. Here's the complete program, which I called language.awk:

```
{ NL = split($4, langs, ",");
  for (i=1; i<=NL; i++)
    if ( langs[i] == "Greek")
      print $1
}
```

There's only one action, and it has no associated pattern (so it fires on every line); however the action is starting to look a bit more like a conventional program, with a function call, a couple of variables, a loop, and an if test. Here's a sample run:

```
$ awk -f language.awk geodata
Albania
Greece
```

Back to a one-liner to show that awk can do arithmetic. This example shows all entries that have a population density greater than 150 per square kilometre:

```
$ awk '$3*1000/$2 > 150' geodata
Luxembourg   2586   465       German,French          Euro
Switzerland  41293  7252      German,French,Italian  Franc
```

Notice that this awk program has a pattern but no action. As you can see, the default action is to print the entire line. Awk has lots more features than shown here, and many people write awk programs much longer than four lines! My mission, though, it not to show how long an awk program can get, but rather to show how short it can get, and still do something useful.

A worked example

Let's bring several of the tools we've discussed together in one final example. Our mission is to count the word frequencies in a sample of text, and our text this morning is taken from Mark, Chapter 6 (The King James version, downloadable from the Electronic Text Center of the University of Virginia Library - see http://etext.virginia.edu/kjv.browse.html).

Now, It's possible, in theory, to type the entire solution interactively at the command prompt, but we'll build it up as a shell script called wordfreq.sh, adding processing steps one by one and showing what the output looks like at each stage.

The file, as I retrieved it from the University of Virginia website, consists of lines of text, one for each bible verse, with the verse number and a colon at the start. So for example, the line for verse 42 looks like:

```
42: And they did all eat, and were filled.
```

I placed this text into a file called mark.txt.

We're going to use awk's associative arrays to perform the word count, but the input could do with a little tidying first. For starters, we need to get rid of those verse numbers. We can easily do this with sed's substitute command, which is the first line to go into our wordfreq.sh script:

```
#!/bin/bash
sed 's/^[0-9]*:\ //' $1
```

The first line is part of the shell scripting machinery - it tells Linux to use the bash shell as the interpreter for the script. The second line is a classic use of sed. This use of the substitute command should be clear from our earlier examples; the 'old pattern' uses regular expressions to say "beginning of line, zero or more digits, colon, literal space" and the 'new pattern' (between the second and third forward slashes) is empty.

Thus, those leading verse numbers are stripped away. The $1 at the end of the line is more shell scripting machinery - it will be substituted by the command-line argument we supply to the script. This is the file we want sed to process. With our two line script created, we can turn on execute permission:

```
chmod u+x wordfreq.sh
```

Now we can run the script, specifying the input file name as argument:

```
./wordfreq.sh mark.txt
```

Our sample verse 42 now looks like this:

```
And they did all eat, and were filled.
```

Next, I decided to make the word counting case-insensitive. The easiest way to do this is to convert all upper-case letters in the input to lower-case. This is easily done with tr. So now our script is three lines long:

```
#!/bin/bash
sed 's/^[0-9]*:\ //' $1 | \
tr '[A-Z]' '[a-z]'
```

We have appended a pipe symbol and a backslash on line 2, allowing us to continue the command on line 3. (There is no need to split the pipeline across multiple lines, I just figured it would be easier to read that way.) The tr command on the third line is another classic, it says "replace each character in the set A-Z by the corresponding character in the set a-z". If we run our new, three-line script, our sample verse 42 looks like this:

```
and they did all eat, and were filled.
```

The next step is to get rid of those pesky punctuation characters. Again, this is easily done with tr, using the -d option. So now our script looks like this:

```
#!/bin/bash
sed 's/^[0-9]*:\ //' $1 | \
tr '[A-Z]' '[a-z]' | \
tr -d '[.,;:]'
```

The last line simply removes all characters in the set [.,;:]. After applying this version of the script, our sample verse looks like this:

```
and they did all eat and were filled
```

Now we have something fit for awk to process. This is where we do the actual word frequency counting. The basic idea is to loop over each individual word in the document, using the word itself as an index into an associative array, and simply incrementing the corresponding element of the array for each word.

Once the entire document has been processed, we can print out the index and value of each element of the array - that is, the word and the number of times that word occurs. I decided to put the awk program into a separate file called wordfreq.awk, so now we have two scripts to deal with, the shell script wordfreq.sh and the awk program wordfreq.awk

The shell script now looks like this:

```
sed 's/^[0-9]*:\ //' $1 | \
tr '[A-Z]' '[a-z]' | \
tr -d '[.,;:]' | \
awk -f wordfreq.awk
```

...and the awk program looks like this:

```
{ for (i=1; i<=NF; i++)
    w[$i]++
}
END { for (word in w)
        print word, w[word]
}
```

This awk program has two actions. Easier to spot is the first, which is taken on all input lines (there is no selection pattern), loops over all fields in the input line (ie all words in the input) and increments the corresponding element of the associative array w.

The variable names i and w were my own choice, the variable NF is a built-in awk variable and contains the number of fields in the current line. The statement w[$i]++, which increments the appropriate element of the associative array w, is really the core of this solution, Everything else that is there just exists to allow this statement to work.

The second action in this awk program only occurs once, at the end, after all input has been processed. It simply loops over the elements of the w array, printing out the index into the array (that is, the word) and the content of the array (that is, a count of how many times that word occurred).

The output from our script is now fundamentally transformed and looks like this:

```
themselves 4
would 3
looked 1
taken 1
of 27
sit 1
privately 1
abroad) 1
name 1
and 134
```

...with a further 400-odd lines deleted.

Finally, we can make the output more useful by doing a reverse numeric sort on the second field, (so that the most common words appear at the top of the list) and using head to pick off the first ten lines. Our final script looks like this:

```
#!/bin/bash
sed 's/^[0-9]*:\ //' $1 | \
tr '[A-Z]' '[a-z]' | \
tr -d '[.,;:]' | \
awk -f wordfreq.awk | \
sort -nr -k2 | \
head
```

Notice the flags on the sort command. -n forces a numeric sort , -r reverses the results of the sort, and -k2 sorts on the second field. Now we have the word frequency report we were looking for:

```
and 134                         of 27
the 64                      him 26
he 38                       unto 23
they 31                         to 22
them 31                         his 21
```

There's more than one way to do it

As with most things in life, there are other ways to solve this particular problem. Rather than using awk's associative arrays, we can split the file up as one word per line (using tr) then sort it alphabetically using sort (so that repeated instances of the same word will appear on successive lines) then use uniq to report how many times each line appears, then do a reverse numeric sort, then pick off the first ten lines. The script looks like this:

```
#!/bin/bash
sed 's/^[0-9]*:\ //' $1 | \
tr '[A-Z]' '[a-z]' | \
tr -d '[.,;:]' | \
tr ' ' '\n' | \
sort | \
uniq -c | \
sort -nr | \
head
and the output looks like this:
    134 and                 27 of
```

```
     64 the                  26 him
     38 he                           23 unto
     31 they                 22 to
     31 them                 21 his
```

...which is the same as we had before, except the fields are in the opposite order. To understand how this solution works, try building it up step by step using a sample of input of your choice, and observing the output at each stage.

Simple regular expressions

Command Line	What it does
head /etc/passwd	Show the first ten lines of /etc/passwd.
grep '/bin/bash$' \ /etc/passwd	Show the lines in /etc/passwd relating to accounts that use bash as their login shell.
sort /etc/services	Sort the services from /etc/services in alphabetical order.
wc /etc/* 2> /dev/null	Count the lines, words and characters of all the files in /etc; throw away any error messages.

First published in Linux Format magazine

You should follow us on Identi.ca or Twitter

Your comments

Pretty clear!
Gullit (not verified) - April 1, 2009 @ 7:50am

When i first met Bash's features what got my attention was pipelining, it's very useful when writing shell scripts (with help of RE).

Very informative!
Anonymous Penguin (not verified) - April 1, 2009 @ 8:18am

I have to commend you on a very well written, concise and properly advanced article! Very useful, as it seem to be an ever increasing demand for filtering a shitload of logs at work, be it apache, postfix or syslog. Thanks!

SSH Hack Attempt Pipeline
Anonymous Penguin (not verified) - April 1, 2009 @ 5:01pm

Check it out!

cat /var/log/messages | grep ssh | grep failure | cut -f 13-13 -d " " | sort | uniq

shows me the IP's of people that tried to hack my SSH server!
Correct me if I'm wrong, also f could be different for each distro.

"""" sed 's/^[0-9]*:\ //' $1

phuongvd (not verified) - April 2, 2009 @ 9:47am

"""
sed 's/^[0-9]*:\ //' $1 | \
tr '[A-Z]' '[a-z]' | \
tr -d '[.,;:]' | \
awk -f wordfreq.awk
and the awk program looks like this:
{ for (i=1; i<=NF; i++)
w[$i]++
}
END { for (word in w)
print word, w[word]
}
"""

The above should be in two seperated box, not one like in the article.

And thank you for such a nice article + good example ;)

Sed

FlatCap (not verified) - April 2, 2009 @ 11:57am

Nice article. A good demonstration of the power of awk.

I'd just suggest clearing up one thing about sed. For a global replace, it's a command of g, not a suffix of /g

The slash is the terminator of the search/replace and is always required.

FlatCap

well done

Roy (not verified) - April 4, 2009 @ 9:43am

great article! I love all your articles...! very well written and hugely informative!

Thats amazing .....

Jacki (not verified) - April 6, 2009 @ 11:40am

not comparable !! thanks a lot to writer and tuxrudar.com

Very nice article!

Kody (not verified) - April 11, 2009 @ 3:34am

Thanks for putting this article together! Very well written!

Sort of late but need help

OldMan (not verified) - May 8, 2009 @ 10:59pm

{ for (i=1; i<=NF; i++)
w[$i]++
}
END { for (word in w)
print word, w[word]
}

Can someone be helpful enough to explain in layman terms what the above is about.

I am sort of learning computer from scratch. thanks.

Comment viewing options

Threaded list - expanded ▾ | Date - oldest first ▾ | 50 comments per page ▾ | Save settings

Select your preferred way to display the comments and click "Save settings" to activate your changes.