Combining commands with pipes and redirection

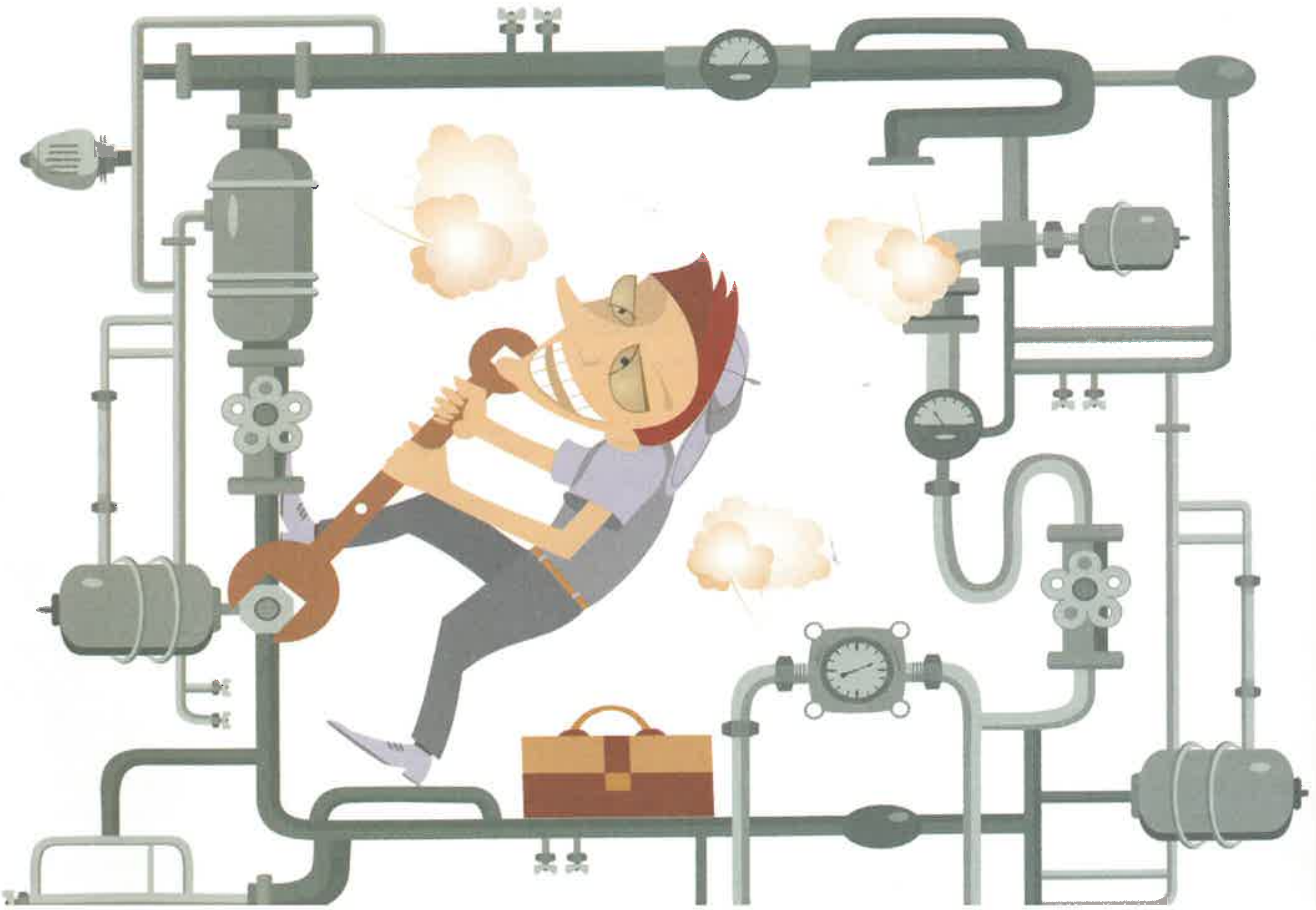
Pipe Time

Special tools in the shell help you combine commands to create impromptu applications. *By Martin Streicher*

The Linux command line provides hundreds of small utilities to read, write, parse, and analyze data. With just a few extra keystrokes, you can combine those utilities into innumerable impromptu applications. For example, imagine you must extract an actor's lines. That is to say, given the text shown in Listing 1, you must produce *That's what they call a sanity clause* for Groucho. The `grep` command can find substrings, strings, and patterns in a text file. You can use `grep` to find all lines that begin with *GROUCHO*. Then you can use `cut` to divide the matching lines into pieces and combine the two commands with a pipe (`|`):

```
$ grep -i -E '^Groucho' marx.txt | ⮐
  cut -d ':' -f 2
That's what they call a sanity clause.
```

The `grep` clause searches the file `marx.txt` for all occurrences of "Groucho" that appear at the beginning of a line (`-E '^Groucho'`), ignoring differences in case (`-i`). The *cut* clause separates the line into fields delimited by a colon (`-d ':'`) and selects the second field



(-f 2). The pipe operator turns the output of the grep clause into the input of the *cut* clause.

A pipe connects any two commands, and you can construct a long chain of commands with many pipes. For example, if you want to count the number of words Groucho speaks, you can append the clause `| wc -w` to the previous command.

The pipe is just one form of *redirection*. Redirection tools can change the source or the destination of a process's data. The shell offers other forms of redirection, too, and learning how to apply these tools is key to mastering the shell.

Data In, Data Out

If you run grep by itself, it reads data from the *standard input device* (*stdin*) and emits results to the *standard output device* (*stdout*). Errors are sent to a third channel called the *standard error device* (*stderr*).

Typically, the data for stdin is provided by you via the keyboard, and by default, both stdout and stderr are sent to the terminal connected to your shell. However, you can redirect any or all of those conduits. For instance, you can redirect stdin to read data from a file instead of the keyboard. You can also redirect stdout and stderr (separately) to write data somewhere other than the terminal window. As shown previously, you can also redirect the stdout of one command to become the stdin of a subsequent command.

The syntax for redirection depends on the shell you use, but almost all shells support the following operations:

- `< input_file` redirects stdin to read data from the named file.
- `> output_file` redirects stdout, sending the results of a command or a pipe (but not the errors) to a named file. If the file does not exist, it is created; if the file exists, its contents are overwritten with the results.
- `>> output_file` is similar to > but appends stdout to the named file. If the file does not exist, it is created; however, if the file exists, its contents are preserved and amended with the results.
- `>& output_file` works like >, but it captures stdout *and* stderr in the specified file, creating the file if necessary, and overwriting the contents if it previously existed.

A few examples are shown in Listing 2.

In Listing 2, the first command should look familiar. The addition of `> groucho.txt` saves the output of the command line to the file `groucho.txt`. The second command appends the string *I started work on Nov 2 at 9 am.* to the file `timecard.txt`. The third command runs the Ruby script `myapp.rb`. Input is taken from the file named `data` and the stdout and stderr are captured in `log`.

LISTING 1: marx.txt

```
GROUCHO: That's what they call a sanity clause.
CHICO: Ah, you fool wit me. There ain't no Sanity Claus!
```

LISTING 2: Redirection Examples

```
$ # First example
$ grep -i -E '^Groucho' marx.txt | cut -d ':' -f 2  > groucho.txt
$ cat groucho.txt
That's what they call a sanity clause.

$ # Second example
$ cat timecard.txt
I started work on Nov 1 at 8.15 am.
I finished work on Nov 1 at 5 pm.

$ echo 'I started work on Nov 2 at 9 am.' >> timecard.txt

$ cat timecard.txt
I started work on Nov 1 at 8.15 am.
I finished work on Nov 1 at 5 pm.
I started work on Nov 2 at 9 am.

$ # Third example
$ ruby myapp.rb < data >& log
```

Advanced Use of Pipes

Consider the following command-line combination:

```
$ find /path/to/files ↩
  -type f | xargs grep -H -I -i  -n string
```

This command enumerates all plain files in the named path, searches each one for occurrences of the given string, and generates a list of files that contain the string, including the line number and the specific text that matched. The `find` clause searches the entire hierarchy rooted at `/path/to/files`, looking for plain files (`-type f`). Its output is the list of plain files.

The `xargs` clause is special: `xargs` launches a command – here, `grep` plus everything to the end of the line – once for each file listed by `find`. The options `-H` and `-n` preface each match with the file name and line number of each match, respectively. The option `-i` ignores case. `-I` skips binary files.

Assuming that the directory `/path/to/src` contains files `a`, `b`, and `c`, using `find` in combination with `xargs` is the equivalent of:

```
$ find /path/to/src
a
b
c
$ grep -H -I -i -n string a
$ grep -H -I -i -n string b
$ grep -H -I -i -n string c
```

In fact, searching a collection of files is so common that grep has its own option to recurse a file system hierarchy. Use -d recurse or its synonyms -R or -r. For example, the command

```
grep -H -I -i -n -R string /path/to/src
```

works as well as the combination of find and xargs. However, if you need to be selective and pick specific kinds of files, use find.

Bit Bucket

As you've seen, most commands emit output of one kind or another. Most command-line commands use stdout and stderr to show progress and error messages, in that order. If you want to ignore that sort of output – which is useful, because it often interferes with working at the command line – redirect your output to the "bit bucket," /dev/null. Bits check in, but they don't check out.

Listing 3 shows a simple example. If you redirect the standard output of cat to /dev/null, nothing is displayed. (All the bits are thrown into the virtual vertical file.) However, if you make a mistake, error messages, which are emitted to standard error, *are* displayed. If you want to ignore all output, use the >& operator to send stdout and stderr to the bit bucket.

You can also use /dev/null as a zero-length file to empty existing files or create new, empty files (Listing 4).

Other Tricks

In addition to redirection, the shell offers many other tricks to save time and effort.

The "back tick" or "back quote" operator (` ... `) expands commands in place. A phrase between back ticks runs first, while the shell interprets the command line, and its output replaces the original phrase. You can use back ticks to yield, for example, a file name or a date:

```
$ ps > state.`date '+%F'`
$ ls state*
state.2009-11-21
$ cat state.2009-11-21
13842 ttys001    0:00.54 -bash
30600 ttys001    1:57.15 ruby ./script/server

$ cat `ls state.*`
13842 ttys001    0:00.54 -bash
30600 ttys001    1:57.15 ruby ./script/server
```

The first command line captures the list of running processes in a file named something like state.YYYY-MM-DD, where the date portion of the name is generated by the command date '+%F'. The single quotes around the argument prevent the shell from interpreting + and %. The last command shows another example of the back tick. The evaluation of ls state.* yields a file name.

Speaking of capturing results, if you want to capture the output of a series of commands, you can combine them within braces ({ ... }):

```
$ { ps; w } > state.`date '+%F'`
```

In the preceding command, ps runs, followed by w (which shows who is using the machine), and the collected output is captured in a file.

You can also embed a sequence of commands in parentheses to achieve the same re-

LISTING 3: The Bit Bucket

```
$ ls
secret.txt
$ cat secret.txt
I am the Walrus.
$ cat secret.txt > /dev/null
$ cat socrates.txt > /dev/null
cat: socrates.txt: No such file or directory
$ cat socrates.txt >& /dev/null
$ echo Done.
Done.
```

LISTING 4: Empty Files

```
$ cat secret.txt
Anakin Skywalker is Darth Vader.
$ cp /dev/null secret.txt
$ cat secret.txt

$ echo "The moon is made of cheese!" > secret.txt
$ cat secret.txt
The moon is made of cheese!
$ cat /dev/null > secret.txt
$ cat secret.txt

$ cp /dev/null newsecret.txt
$ cat newsecret.txt

$ echo Done.
Done.
```

sult, with one important difference: The series of commands collected in parentheses runs in a *subshell* and does not affect the state of the current shell. For example, you might expect the commands

```
{ cd $HOME; ls -1 }; pwd
(cd $HOME; ls); pwd
```

to produce the same output. Note, however, that the commands in braces change the working directory of the current shell. The latter technique is inert.

The decision to use a combination or a subshell depends on your intentions, although the subshell is a much more powerful tool. You can use a subshell to expand a command in place, just as you can with back ticks. Better yet, a subshell can contain another subshell, so expansions can be nested. The two commands

```
$ { ps; w } > state.$(date '+%F')
$ { ps; w } > state.`date '+%F'`
```

are identical. The `$( )` runs the commands within the parentheses and then replaces itself with the output. In other words, `$( )` expands in place, just like back ticks; however, unlike back ticks, `$( )` can be very complex and can even include other `$( )` expansions:

```
$ (cd $(grep strike /etc/passwd | ⮐
    cut -f6 -d':'); ls)xw
```

This command searches the system password file to find an entry for user `strike`, clips the home directory field (field six, if you count from zero), changes to that directory, and lists its contents. The output

```
grep /etc/passwd strike | cut -f6 -d':'
```

is expanded in place before any other operation. Because the subshell has so many uses, you might prefer to use it instead of the `{ }` or the back tick operators. ●●●

THE AUTHOR

Martin Streicher is the founder of Locomotive, a creative coding cooperative based on Ruby on Rails. When not writing prose or code, he dreams of becoming a famous comic book author. You can reach Martin at *martin.streicher@gmail.com*.