www.sxc.hu

Redirecting input and output

DATA FLOW

Working in the shell has many benefits. Pipelines, redirectors, and chains of commands give users almost infinite options.

BY HEIKE JURZIK

Combining shell commands lets you leverage the power of the command line. This not only includes linking individual programs, but also forwarding command output. For example, if the information you need disappears from the screen too fast for you to read, you can simply redirect it to a file or use a pager.

Just a couple of basic items are all it takes to link individual commands in bash. Instead of saying

```
$ mkdir folder
$ cd folder
$ cp ../folder2/* .
```

you can combine these steps and use a semicolon to tell the interpreter to run these commands in quick succession:

```
$ mkdir ↵
folder ; cd folder ; ↵
cp ../folder2/* .
```

To exercise more control, you can pass a condition to bash. For example, you could execute a second (or third or *n*th) command only if another command has completed successfully or failed.

To make sure a file exists before deleting, you could add a simple test condition to a command line with *rm*:

```
test -w file && rm file
```

In this case, the test program simply checks to see whether the file exists and whether it is writable (parameter *-w*); if so, *rm* deletes the file. If you often need to build software from the source code and use the normal *./configure; make; make install* bunch of commands, you can combine the three using *&&* to ensure that the each successive command is only executed if the previous command was error free:

```
./configure && make && ↵
make install
```

Besides the *-w* option, the *test* command has a number of other practical flags up its sleeve. For example, *-d* lets you check to see whether a directory exists and create the directory if not:

```
test -d folder || mkdir folder
```

The two pipe characters between the commands act as a logical OR.

No fewer than three channels exist for command input and output in the shell: Programs read data from their standard input (STDIN, channel 0) or from a file, the program output is sent to standard output (STDOUT, channel 1), and error messages are written to the standard error output (STDERR, channel 2):

```
command < input > output 2> ↵
error
```

The < and > operators indicate the direction; if standard input does not originate at the keyboard, < tells *command* to read it from a file. To redirect the output to a file, you need the > operator. Error output also uses >; however, you need to specify the channel by adding a file description (*2*>). Table 1 gives an overview of typical redirection scenarios.

Redirecting Output

As I mentioned previously, the > operator redirects the output from a program to a file. Instead of >, you could also

write *1 >* because this option specifies the first channel; however, this is not strictly necessary because the shell will assume standard output if you do not say otherwise:

```
ls /etc > etc_content.txt
```

If the file behind the operator already exists, the shell will simply overwrite it. The test condition I looked at earlier prevents this from happening:

```
test -w etc_content.txt || ls ➘
/etc > etc_content.txt
```

Or you can double the operator:

```
ls /etc >> etc_content.txt
```

The two "greater than" signs (> >) mean that the shell should append the output from the *ls* command to the *etc_content.txt* file if the file already exists; if not, the shell should create a new file and write the output to that file.

Capturing Error Messages

To redirect the second channel, use the number *2* followed by the > operator. This method is good to use if a program generates so many error messages that they prevent you from reading the program's actual output:

```
$ find /home -name "*.tex"
find: /home/lost+found: ➘
No permissions
find: /home/petronella/data: ➘
No permissions
/home/huhn/book/book.tex
/home/huhn/book/chap01.tex
...
```

The command

```
find /home -name "*.tex" 2>➘
/dev/null
```

lets you send error messages to */dev/null* on your machine to avoid them cluttering up standard output.

Two in One Blow

A clever combination of operators allows you to redirect two channels at the same time. If you want to write the standard output from the *find* command in the previous example to a file, but without logging all the error messages, use:

```
find /home -name ➘
"*.tex" > ➘
findoutput 2> ➘
/dev/null
```

The double operator > >, which will create files that don't exist or append to files that do, is also useful in this scenario. The previous example showed you how to do this for standard output. For standard error output, you can use the double arrow to the same effect if you need to write error messages to a file:

```
find /home -name "*.tex" > ➘
findoutput 2>> error
```

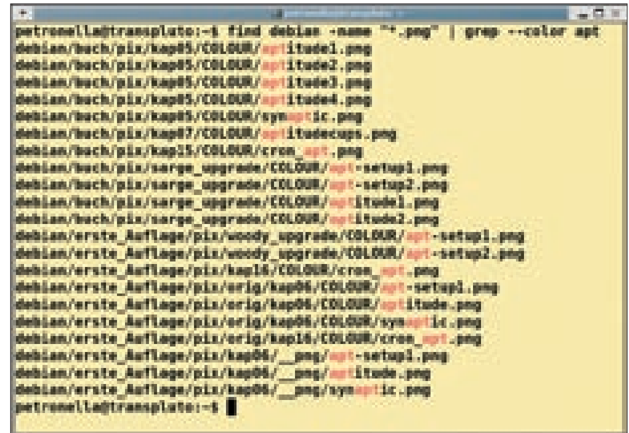

Figure 1: The output from the find command is sent straight to the grep tool without detouring via the shell.

Pipe System

Pipes often save you extra steps by directing the output from one program directly to another, without needing to detour via a file. The pipe character (|) separates the individual commands, as the following example demonstrates.

```
ls /etc | less
```

This sends the output from the *ls* command to the *less* pager instead of directly to the terminal. The pager shows the output screen-by-screen and supports scrolling. The pipe is very common in combination with grepping output for specific strings; for example,

```
find debian -name "*.png" | ➘
grep --color apt
```

searches the *debian* directory for all files that end in *.png* and passes the output straight to *grep*. Grep searches for the *apt* string and highlights the hits in red, thanks to the *--color* option (Figure 1).

You can use multiple pipes. The following command lists the content of your home directory line-by-line, passes the results to the *grep* tool, searches for the *.jpg* string, and counts the matches:

```
$ ls -1 ~ | grep .jpg | wc -l
12
```

When I used this command, I found 12 JPG files in my home directory without long-winded searching.

Time for Tee?

To branch off between the individual pipe components, you can use the Tee program. The command expects data from standard input and writes the data both to a file and to the screen.

A "Tee junction" can occur at the end of a command line or between the individual commands:

```
command1 | tee output.txt | ➘
command2
```

To search for PNG files, starting in the current directory, then log this output in the *images.txt* file while displaying it on the screen and searching for the *book* string with *grep*, use:

```
find . -name "*.png" | ➘
tee images.txt | grep book
```

By default, Tee will overwrite the file if it exists. To append output to an existing file, you simply add the *-a* parameter:

```
find . -name "*.png" | tee ➘
-a images.txt | grep book
```

The various operators, along with the pipe and Tee commands, thus support extremely flexible combinations of commands. It is typically not worthwhile to create a script simply for quick searching, for example.

With regular practice, you will soon be discovering even more new uses and new combinations. ■