The Basics of the `bash` shell

Phil Spector

September 4, 2009

The `bash` shell is the program that Linux and Mac systems use to actually communicate with the computer. When you type a command into the shell, it first performs some very important services, so it's essential to understand what the shell does in order to use the computer effectively. Some of the more important services that the shell provides are explained in the following sections.

1 Command Path

When you type a command into the shell, it will search several different directories to find it, stopping at the first directory with a command of the specified name. The directories it searches, and the order of searching is controlled by the command path. You can see what the current command path is by examining the `PATH` shell variable:

```
echo $PATH
```

Suppose you want to make sure that the first directory that the shell looks at is a directory called `mycommands` in your home directory. You could modify the `PATH` variable as follows:

```
export PATH=~/mycommands:$PATH
```

First, note that the previous command path will still be searched after your local directory, because `$PATH` (the old value) is included as part of the new path. It is usually a very bad idea to overwrite the entire command path – usually one or two directories will be appended or prepended to the path as shown in the example above.

The `export` command insures that other shells created by the current shell (for example, to run a program) will inherit the modified path; such a variable is sometimes known as an environmental variable. Without the `export` command, any shell variables that are set will only be modified within the current shell.

2 Aliases

Aliases allow you to use an abbreviation for a command, to create new functionality or to insure that certain options are always used when you call an existing command. For example,

?	a single character
*	zero or more characters
[c_1c_2...]	matches c_1, c_2, ...
[^c_1c_2...]	matches anything but c_1, c_2, ...
[c_1-c_2]	matches the range from c_1 to c_2
{string-1,string-2,$\cdots$}	uses each string in turn
~	your home directory
~user	user's home directory

Table 1: Wildcards for the bash shell

suppose that you would rather type bye instead of logout when you terminate a session. You could type the command

```
alias bye=logout
```

and you could then type bye in that shell to logout.

As another example, suppose you find the -F option of ls (which displays / after directories, * after executable files and @ after links) to be very useful. The command

```
alias ls='ls -F'
```

will insure that the -F option will be used whenever you use ls.

If you need to use the unaliased version of something for which you've created an alias, precede the name with a backslash (\). For example, to use the normal version of ls after you've created the alias described above, just type

```
\ls
```

at the bash prompt.

Of course, the real utility of aliases is only achieved when they are automatically set up whenever you log in to the computer. To achieve that goal with aliases (or any other bash shell commands), simply insert the commands in the file .bashrc in your home directory.

3 Filename expansion with wildcards

The shell will expand certain special characters to match patterns of file names, before passing those filenames on to a program. Note that the programs themselves don't know anything about wildcards – it is the shell that does the expansion, so that programs don't see the wildcards.

Table 1 shows some of the special characters that the shell uses for expansion:

Here are some examples of using wildcards:

control-A	beginning of line
control-E	end of line
control-N	next command
control-P	previous command
control-F	one character forward
control-B	one character backward
control-D	delete character
control-K	delete to end of line
control-W	delete to beginning of line

Table 2: Control keys to navigate command history

List all files ending with a digit:	`ls *[0-9]`
Make a copy of `filename` in `filename.old`	`cp filename{,.old}`
Remove all files beginning with `a` or `z`	`rm [az]*`

The `echo` command can be used to verify that a wildcard expansion will do what you think it will. If you want to suppress the special meaning of a wildcard in a shell command, precede it with a backslash (\).

4 Tab Completion

When working in the shell, it is often unnecessary to type out an entire command or file name, because of a feature known as tab completion. When you are entering a command or filename in the shell, you can, at any time, hit the tab key, and the shell will try to figure out how to complete the name of the command or filename you are typing. If there is only one command in the search path and you're using tab completion with the first token of a line, then the shell will display its value and the cursor will be one space past the completed name. If there are multiple commands that match the partial name, hitting tab twice will display a list of choices, and redisplay the partial command line for further editing. Similar behavior with regard to filenames occurs when tab completion is used on anything other than the first token of a command.

5 Command History

The shell provides two mechanisms for re-editing previously submitted commands. The first uses either the arrow keys on the keyboard or a set of control key/letter key combinations to navigate and edit previous commands. Some of the available control key/letter combinations are displayed in Table 2 After positioning the cursor, you can then edit the commands and resubmit them by hitting enter. If you ever encounter a state where your keystrokes are not recognized, hitting the escape key will usually restore things to normal.

`!!`	previous command
`!-`*n*	*n*th previous command
`!$`	last token from previous line
`^`*old*`^`*new*`^`	previous command with *old* substituted with *new*
`!!-`*n*`:s/`*old*`/`*new*`/`	*n*th previous command with *old* substituted with *new*

Table 3: History Substitution

There are many other operations that can be performed on the command history. Since these features are provided by the `readline` library, you can find more information by searching for that library.

The other mechanism that can be used to manipulate the command history is known as history substitution. Table 3 shows some of the commands that the shell will interpret specially with regard to command history.'

If you're not sure what the result of using history substitution will be, you can append "`:p`" at the end of the substitution, and the result will be printed, but not executed. You can then use the arrow key to bring back the statement of editing/execution.

6 Redirection

UNIX programs operate by reading input from a stream known as standard input, and writing their results to a stream known as standard output. In addition, a third stream known as standard error receives error messages, and other information that's not part of the command's results. In the usual interactive session, standard output and standard error default to your screen, and standard input defaults to your keyboard. You can change the place from which programs read and write through redirection. Notice that, once again, it's the shell providing this service, not the individual programs, so redirection will work for all programs. Table 4 shows some examples of redirection.

Operations where output from one command is used as input to another command (via the `|` operator) are known as pipes; they are made especially useful by the convention that many UNIX commands will accept their input through the standard input stream when no file name is provided to them. Table 5 shows some examples of these programs, which are sometimes refered to as filters.

A closely related, but subtly different capability is offered by the use of backticks (`` ` ``). When the shell encounters a command surrounded by backticks, it runs the command and replaces the backticked expression with the output from the command; this allows something similar to a pipe, but is appropriate when a command reads its arguments directly from the command line instead of through standard input.

For example, suppose we are interested in searching for the word "graphics" in the last 10 files which were modified in the current directory. We can find the names of the last 10 files which were modified using

command >*file*	send *command*'s standard output to *file*, overwriting its contents
command >>*file*	send *command*'s standard output to *file*, appending to existing contents
command >*file* 2>*errors*	send *command*'s standard output to *file*, and its standard error to *errors*
command &>*file*	send *command*'s standard output and its standard error to the same location
command < *file*	execute *command*, reading standard input from *file*
command < *file* > *output*	execute *command*, reading standard input from *file* and sending standard output to *output*
command << XXX	execute *command* using the following lines, until a line with only XXX (arbitrary) is found
command-1 | *command-2*	use standard output from *command-1* as standard input to *command-2*

Table 4: Redirection examples

`head`	shows first few lines of a file
`tail`	shows last few lines of a file
`wc`	counts lines, words and characters of a file
`grep`	finds patterns in a file
`sort`	sorts the lines of a file
`less`	displays a file one screen at a time

Table 5: Filters

```
ls -t | head -10
```

and we can search for the required pattern using `grep`. Putting these together with the backtick operator we can solve the problem using

```
grep graphics `ls -t | head -10`
```

Note that piping the output of the `ls` command into `grep` would not achieve the desired goal, since `grep` reads its filenames from the command line, not standard input.

7 Job Control

When you run a command in a shell by simply typing its name, you are said to be running in the foreground. When a job is running in the foreground, you can't type additional commands into that shell, but there are two signals that can be sent to the running job through the keyboard. To interrupt a program running in the foreground, use control-c (i.e. holding down the control key and the c key at the same time); to quit a program, use control-\.

While modern windowed systems have lessened the inconvenience of tying up a shell with foreground processes, there are some situations where running in the foreground may not be the best approach. The primary need for an alternative to foreground processing arises when you wish to have jobs continue to run after you log off the computer. In cases like this your can run a program in the background by simply terminating the command with an ampersand (`&`). However, before putting a job in the background, you should consider how you will access its results, since the default standard output stream is not preserved when you log off from the computer. Thus, redirection (including redirection of standard error) is essentially when running jobs in the background.

As a simple example, suppose that you wish to run a `matlab` program, and you don't want it to terminate when you log off. The following command execute the command in `matlab.in`, and would route both the normal output and any errors to the file `matlab.out`:

```
matlab < matlab.in &> matlab.out
```

while the following command would send the output to `matlab.out` and the errors to `matlab.err`:

```
matlab < matlab.in > matlab.out 2> matlab.err
```

If you forget to put a job in the background when you first execute it, you can do it while it's running in the foreground in two steps. First, suspend the job using the control-Z signal. After receiving the signal, the program will stop executing, but will still have access to all files and other resources. Next, issue the `bg` command, which will put the stopped job in the background.

Since only foreground jobs will accept signals through the keyboard, if you want to terminate a background job you must first determine the unique process id (PID) for the process

you wish to terminate through the use of the `ps` command. (The method described here also works for foreground jobs, although directly signalling is ofter simpler.) For example, to see all the R jobs running on a particular computer, you could use a command like:

```
ps -aux | grep R
```

Among the output there might appear a line that looks like this:

```
spector  11998  0.4 10.6 141312 108956 pts/20  S+   15:22   0:13 /usr/local/linux/R-2.4.0/bin/exec/R
```

In this example, the `ps` output tells us that this R job has a PID of `11998`. You could then issue the command:

```
kill 11998
```

or, if that doesn't work

```
kill -9 11998
```

to terminate the job.

Another useful command in this regard is `killall`, which accepts a program name instead of a process id, and will kill all instances of the named program. Of course, it will only kill the jobs that belong to you, so it will not affect the jobs of other users.

The most important thing to keep in mind when managing your running jobs is that the `ps` and `kill` commands only apply to the particular computer on which they are executed, not to the entire computer network. Thus, if you start a job on one machine, you must log back into that same machine in order to manage your jobs.