Beginner’s Guide to Bash

11th Sep 2015

Table of Contents

Goals for this session

This session is aimed at beginners in the world of Bash scripting and Unix in general. If you don’t know what Bash is, or if you’ve just used other people’s Bash scripts but their internal workings remain mysterious, or you’ve tried to use Bash and ended up confused by its idiosyncrasies then this session will hopefully be helpful.

By the end of it you will:

- know roughly what Bash is
- know where to read the docs and be aware of the some of the odd naming conventions in shell-land
- know how to automate Bash’s behaviour by writing scripts
- understand when it’s appropriate to use shell scripting
- understand the limitiations of shell scripting
- know where to find documentation

What is Bash?

Bash is an example of a family of programs, known as “shells” which take commands from the user and execute them. Bash is probably the most common shell and is usually the default on modern OSs that need one.

It’s main job is to execute other programs on your behalf, and this follows the Unix philosphy of having small programs that do one thing well, and then giving you the ability to glue these programs together to achieve larger tasks.

For example, the shell will provide you with the ability to `pipe' the output of one command into the input of another, as in the command line below: it uses the `wc` program

to count the number of words in each file in the current directory, and then pipe this to the `sort` command:

```
wc -w * | sort -n
```

Most commands you run, even the most basic ones, are external programs to which the shell is able to delegate its work. To see this in action, you can type `which ls`: this will show you where the binary that implements the `ls` command is held.

Exercise 1: Getting help

1. Type `man bash` to see the documentation for the Bash shell.

   Once in, as well as the arrow keys, you can press `space` to move forward one page, and `b` to move backwards one page.

2. Press `/` to begin a search and type `PS1` (use capital letters) followed by enter. Pressing `n` will move forwards from result to result, and `q` will cancel the search.

   These search results should give you some hints about how to change the prompt by setting a variable.

   Press `<` to jump back to the start, and search for `PROMPTING`. Take a look at the escape codes you can use within the prompt.

3. Google "bash man page" and see if you prefer to use that instead.

4. Press `q` to exit the documentation viewer and get back to the Bash prompt. Change the prompt (which in the example below is `$`) by typing:

```
$ PS1="\d :> "
Fri Sep 11 :>
```

Most Unix-like systems, incuding OSX, come with a set of documentation files call man pages, which document in great detail everything you could ever want to know about a command on the system. They can be accessed by typing `man` followed by the name of the command you want to learn about, and the best place to start is perhaps the man page about the man command itself, and you can search man pages with the `apropos` command:

```
$ apropos compress
gzip (1)             - compress or expand files
[...]
```

The man pages are also published on line and this can be a more convenient way to view them. Just be aware of differences that can crop up between different OSs and distributions (using the man command should give you the answer for the specific system you are using).

Exercise 2: Locating files

Compress all of the files in a directory that don't match the `*.gz` suffix.

In the `test-data` directory of the repo will find a number of files, some of which are compressed, and some of which aren't.

1. In that directory, type `ls *.gz` to list all of the which already have this extension.
2. Look in the man page and search for "Pathname Expansion" and read about the pattern matching options.
3. Try to invert the previous pattern by trying `ls !(*.gz)`.

   If you get an error, you will need to enable the extended globbing option with `shopt -s extglob`.

```
$ ls
a.gz  b  c.gz
$ ls !(*.gz)
bash: !: event not found
$ shopt -s extglob
$ ls !(*.gz)
b
```

This exercise is an example of a task that is well-suited to shell scripting.

Bash (and shells in general) makes the assumption that what you are trying to do is execute system commands, and so that is it's default interpretation of what you want to do, with everything else as just glue that sticks those elements together.

As a counter example, one way we might solve this exercise in Scala would be a program like this:

```
import java.io.File

def process(f: File, indent: Int = 0): Unit = {
  if(f.isDirectory)
    for(child <- f.listFiles) process(child)
  else if(!f.getName.endsWith(".gz"))
    /* ... either call out to `gzip` or use the JDK library... */
    Runtime.getRuntime.exec(Array("gzip", f.getPath))
}

process(new File("."))
```

However, with we achieve this in a single line in bash:

1. Look at the `gzip` man page, you can see that we can pass multiple files to it.
2. Type `gzip !(*.gz)`. The wildcard pattern will get expanded to a list of files and these will be passed to gzip as its arguments.

There are various situations in which Bash will replace special text with some computed values, and this is usually referred to in the docs as `EXPANSION`, e.g parameter expansion, filename expansion etc. Not suprisingly the latter is for finding and inserting files that match a wildcard patterns (this also has the charming moniker `globbing` in shell parlance).

By default you are limited to a couple of simple wildcards, but when the `extglob` (for extended globbing) setting is enabled you have significantly more options available.

Note

Why is this behind an option? Backwards compatibility: if you try to use the pattern that you need without setting this option, you will get an error caused by the fact that it already has another meaning.

Optional: recursive wildcards

The Scala example is descending the directory recursively. One way to achieve this in Bash might be to combinine smaller tools to get the job done. For example, you've got `find` (http://linux.die.net/man/1/find) which will do this for you, and you can usefully combine it with `xargs` (http://linux.die.net/man/1/xargs).

Or, again you can delve into the many options of Bash:

```
$ ls *.gz
a.gz  c.gz
$ ls */*.gz
dir1/a.gz
$ shopt -s globstar
$ ls **/*.gz
a.gz  c.gz  dir1/a.gz  dir1/dir2/c.gz
```

Exercise 3: Writing scripts

Everything that you write on the Bash command line can be put into a file and executed as a script. The simplest way to run it is just to pass it along as the first argument to a new instance of Bash.

1. Open up a text editor and write the commands from the previous exercise in a file.
2. Run `bash <filename>` to execute your script.

Another way to do it is to add the execute permission and add a special line at the top which tells the OS to pass the file to Bash:

If `hello.sh` contains:

```
#!/bin/bash
echo "hello world"
```

we can:

```
$ chmod u+x hello.sh
$ ./hello.sh
hello world
$ mv hello.sh hello
$ ./hello
hello world
```

In both of these instances, a new copy of Bash will be run which will house the script while it runs. This means that it won’t be able to access the variables and other things that are owned by the starting instance.

In that case, you can either use the `source` command to execute the commands from the file in the same environment exactly as if you typed them at the command line, or you can `export` those resource to child processes.

```
$ ./hello
hello world
$ bash ./hello.sh
Hello
$ cat hello.sh
echo "Hello $NAME"
$ NAME="Superman"
$ bash hello.sh
Hello
$ source hello.sh
Hello Superman
$ export NAME=Spiderman
$ bash hello.sh
Hello Spiderman
```

Exercise 4: Doing maths in Bash is painful…

If we want to add up the total size of all of the files in a directory, we need to do some arithmetic which is quite fiddly in Bash, because of it’s default assumption that you are just typing commands and thier arguments.

1. Search for `Arithmetic Expansion` in the man page to find the way to add numbers together.

2. Use the `$(( ))` interpolation option to insert arithmetic:

```
$ echo $((10 + 5))
15
```

1. Use the `stat` command if it is available, or `ls` and manipulate the output.

2. Create a new script and use a `for` loop to look up this information for each file:

```
COUNT=0
for f in *
do
  SIZE=$(stat -c %s $f)
  COUNT=$(($COUNT + $SIZE))
done
echo $SIZE
```

Note

we are also using `$()` to interpolate the output of a command, which is separate from `$(())`. Confusing, right?!

This exercise provides a good example of why shell scripts can often be very fiddly, because things which are easy in traditional programming languages can be difficult in Bash as it requires you to jump through hoops to shift into `programming mode`.

Conclusion

We have looked at what Bash is, and just as importantly what it **isn’t**. Hopefully you have also gained some signposts that will point the way to getting documention on things that are new when reading or modifying existing scripts. We have also seen some of the subtleties that can confuse beginners with unexpected behaviour when running scripts.

There are of course a lot more commands and other odds and ends that make up Bash’s full toolset, and we will cover these in later sessions.