Bash Scripting

Alessandro Barenghi

Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano

alessandro.barenghi - at - polimi.it

March 16, 2016

Introduction

The `bash` command shell

- Bash stands for "Bourne-Again shell": it is a GNU reimplementation of the Unix r7 shell by S. Bourne
- Derived from a long line of other shells (sh, Bourne shell, Korn Shell,...)
- The most common shell – and the one you get by default on most Linux systems
- Other shells differ from `bash` in terms of syntactic sugar

Introduction

Shell scripts

- A shell script is simply a ASCII text file, which is fed to a command interpreter
- The script should begin with a shebang[a] symbol (`#!`) followed by the pathname of the interpreter
- Any system binary can be used as an interpreter
  - The script pathname is simply passed to it as the first parameter
- In case no interpreter is specified, the default interpreter (i.e., typically `/bin/sh`) is used

[a] also called a sha-bang, hashbang, pound-bang, hash-exclam, or hash-pling

Shell Variables

“typesystem”

- By default, the variables have global visibility in the script
- The basic variable type in `bash` is the textual string
- Any C-like variable name is fine, but the common convention is to have ALL UPPERCASE names
- The $ operator substitutes a variable with its content

Shell Variables

Commandline parameters

- Commandline parameters of a script can be obtained via implicit variables as:
  - `$0` , `$1` , `$2` and so on, to obtain one of them (as per C's `argv`)
  - `$*` yields a single variable with the concatenated parameters
  - `$@` yields a list of the parameters, as tokenized by bash
- `$#` Yields the number of parameters
- In case you need it `$_` recalls the value of the last variable expansion

Shell Variables

Environment Variables

- The bash interpreter, as all the binaries, runs with *environment defined variables*
- Environment variables can be accessed by running programs (e.g. through C's `getenv` function)
- You can add a variable for a single command execution prefixing its declaration to the command
- The `export` command allows you to export the variable in the current bash environment
- To remove a variable from the environment, use the `unset` command

Non-builtins

Some helpful commands

- All the environment variables can be output invoking `env`
- Your current working directory is obtained invoking `pwd`
- `seq` is a simple program generating sequence of strings, with a variable integer value
- Common syntax: `seq <beginning> [step] <end>`
- the `-f=format` allows you to use a format string to print out the values
- `-w` pads the numbers to a constant lengths with zeros
- `-s=separator` changes the different line ending from `\n`

Concatenation, splitting

`cat`, `split` and others

- `cat` (a shorthand for conCATenate) is a command concatenating the files passed as inputs to stdout
- `split` splits an input file into multiple output ones, with a given discipline (e.g. maximum size)
- `head` and `tail` print to stdout the first or last lines of a file
- `echo` simply prints out its first parameter on the output

Control Flow Basics

Straight line code

- The statements in a `bash` script are executed in program order by default
- `;` can be used as a statement separator between statements on a single line: you can turn any script in a one liner (not advised)
- A `#` not followed by a `!` is the beginning of a single line comment
- The `exec` command explicitly executes a `fork-exec` call pair to run the following commands
- The `echo` command prints its arguments on the screen
- The `exit <number>` command exits the script with return value `number`

Control Flow Basics

`if` construct

- The `if` statement syntax is:
  - `if <condition>; then <statements> else <statements> fi`
- The output of a command can be used as a condition (successful execution is true, error is false)
- `true` and `false` commands are available for base cases
- The `test` command allows you to test common stuff
  - `test -e <filename>` returns true if the file exists
  - `test ! -e <filename>` returns true if the file doesn't exist
  - `test <string1> = <string2>` returns true if the strings match
- Enclosing the condition in `[ ]` is equivalent to invoking `test`

Control Flow Basics

`for` construct

- The `for` statement syntax is:
  - `for <variable> in <list>; do <statements> done`
- `for` splits `<list>` according to the value of the IFS[a] variable and runs the loop body assigning each element to `variable`
- Typically the `<list>` is obtained as the output of executing a command via command substitution:
  - Enclose something in ` `` `, `bash` will substitute it with the output of its execution
- Common command substitutions include either `ls` or `seq`

[a] internal field separator

Control Flow Basics

`case` construct

- The `case` statement syntax is:
  `case <variable> in <list>; <pattern>) <statements> ;; esac`
- The `case` statements performs in a similar fashion to the `switch-case` in C, i.e., match a variable against alternatives
- Main difference: since the variables are string, you can specify a POSIX regular expression[a] to be matched
  - The `*` metacharacter indicates any (zero or more) characters
  - `[ ]` can enclose character classes, e.g., `[0-9]`
  - Concatenation is simply concatenating patterns

[a]Caveat: POSIX regular expressions do not match the expressive power of common regular expression, e.g., they can count

Arithmetic interpretation

Learning to do math

- All the variables are by default strings in `bash` , i.e. no arith operations are available
    - e.g., trying to print the variable `SUM=1+1` will result in `1+1` being printed
- It is possible to ask for an arithmetic interpretation of a string via double braces `(( ))` → `echo $((6*7))` will print `42`
- It is not possible to enclose multiple statements in `(( ))` with the only following exception:
    - `for (( i=0 ; $i<10 ; i=$i+1 )) do ... ; done`

Functions

...or somehow that looks like them

- Bash provides minimal support for code refactoring in the form of simplified functions
- The syntax for a function declaration is:
  ```
  <function_name> () { <function_body> }
  ```
- All the functions are assumed to have variable arguments, accessible via `$@`
- Function invocation is simply done as
  ```
  function_name arg1 arg2 arg3 ...
  ```
- Warning: the function body is not parsed until the function is invoked: latent syntax errors may hide there

Pipes and File Descriptors

Data flow among commands

- Proper `bash` plumbing and flow redirection is a basic skill to redirect stuff around the system
- Practically, `bash` provides the user with constructs to
  - Concatenate the standard output files (`stdout,stderr`) of a command into the standard input of another one
  - Feed any of them with/to a file on the disk
  - Duplicate data flows or merge them
- Fits the UNIX paradigm: every command does one precise thing, combine them to obtain complex effects

Pipes and File Descriptors

Files and file descriptors

- Bash allows the redirection of information into a command's `stdin` or from a command's `stdout, stderr`
- Many commands under Linux assume something is piped into their `stdin` to work properly
- `bash` allows explicit referencing to `stdin,stdout, stderr` employing the `0,1,2` token respectively
- More filedescriptor (fds from now on) can be opened, and referenced with integers > 2

Pipes and File Descriptors

Redirection operators

- `fd > file` and `fd < file` redirects data from a fd, into a file, and viceversa. Redirect a command output, redirects `stdout` only
- `fd >> file` appends to the file instead of overwriting
- `fd1 >& fd2` merges the data from `fd1` into `fd2`
- `command1 | command2` redirects the `stdout` of `command1` into the stdin of `command2` (this is not bidirectional!)
- The operators and commands can be concatenated at will

Pipes and File Descriptors

Useful redirection targets

- the stdin of `less` : `less` is a text pager, which allows you to read long texts
- the stdin of `sort` : sorts by line the text passed as input
- the stdin of `uniq` : returns unique occurrences of the lines
- `/dev/null` : a fake device discarding anything written into it
- `tee <filename>` : copies whatever it is redirected into its `stdin` both to its `stdout` and into the `<filename>`

Pipes and flows

Managing Filedescriptors

- Executing `filedescriptor<> filename` opens your file for reading and writing (e.g. `exec 6<>`)
- To close a fd, redirect it to `-` (e.g. `exec 6>&-` closes 6)
- `read -n number_of_chars variable_name` reads, by default from the stdin fd, n characters into the variable
- `read -n number_of_chars variable_name <&3` reads from fd 3, n characters into the variable
- A simple `echo ‘‘foo’’` may be used to write into an open fd through redirection (e.g. `echo ‘‘foo’’>3`)

Pipes and flows

Pipes

- Employing a pipe character (i.e. `|`) chains the standard output of the leading command to the trailing of the last one
- The `stdbuf` command allows to specify how many bytes should the input (`-i`), output (`-o`), and error(`-e`) buffers be long (changing `stderr` buffer is not advised!)
- Comes in terribly handy when employing `nc` or whenever you want a stream to be pushing characters down the pipe instantly
  - Simply prepend a `stdbuf -i0 -o0 -e0` and enjoy!
- Removing the buffering is *critical* if you're juggling with `socat,nc` and the likes to send something via network