Tema 5. Shell

Bash Scripting Cheatsheet
O **shell** *traduce comandos en ordes* ao SO.
O **bash** é un shell que ten todas as características de *sh* (que era o shell que implementaba as primeiras versións de linux) e ademáis inclúe algunhas características avanzadas de *C*.ç

Funcionamento do shell

Hai **comandos internos** (`cd`, `alias`, `echo`, ...), que se executan no *mesmo proceso*, e **comandos externos** (`cp`, `cat`, `mkdir`, ...), que inician un *novo proceso* e cambian a súa imaxe pola do arquivo executable.
Para ver o tipo dun comando: `type [comando]`

Liña de comandos

Estrutura dun comando: `COMANDO [OPCIONS] [PARAMETROS]`
Os comandos execútanse en **primer plano** (foreground), onde o shell espera a que termine o comando antes de aceptar outro, ou en **segundo plano** (background).
Para **pausar** un coamndo: `CTRL-Z`; Para **terminar** un comando: `CTRL-C`
Para **ver os comandos en background**: `jobs`
Para **traer un comando a primer plano**: `fg %job_id`
Para **levar un comando a segundo plano**: `bg %job_id`
Para especificar varios ficheiros como parámetros pódense empregar **comodíns**:

Parámetros

Caracter	Correspondencia
*	0 ou + caracteres
?	1 caracter
[ ]	un dos caracteres entre corchetes
[!] ou [^]	calquera caracter que non estea entre corchetes

Hai algúns **caracteres especiais** que son recoñecidos polo shell como tal (&, *, ?, $, ...). Para *eliminar ese significado especial* temos varias opcións:

- Comilla simple ('): ignora tódolos caracteres especiais
- Comilla dobre ("): ignora tódolos caracteres especiais excepto $, \, '
- Barra invertida (\ ): ignora o caracter especial que sigue a \

Bash permite outras **expansións para parámetros**:

- Chaves `{}` para xerar strings (Ex: `echo a{1,2,3}b` --> `a1b a2b a3b`)
- Tilde `~` para o directorio de usuario (Ex: `cat ~/.bash_history`)
- Aritmética `$[expr]` para avaliar expersións (Ex: `echo $[(4+11)/3]`)

Variables

En shell pódense **crear variables** dende a liña de comandos:
`$> variable='algo'` ou `$> read variable (ENTER)` = scanf

Para **acceder ao contido das variables**: `$variable` (para mostralo como string: `echo`)

Hai dous **tipos de variables**: *variables locais*, que son visibles só dende o shell actual e se poñen en minúsculas normalmente; e *variables de entorno*, que van sempre en maiúsculas e hai que empregar `export VARIABLE` para que sexa visible polos shells fillos.
*Para **ver o listado de variables**: `$> printenv`

Nome	*Propósito*
`HOME`	directorio base do usuario
`SHELL`	executable da shell
`USERNAME`	nome de usuario
`PWD`	directorio actual
`PATH`	path dos executables

Redirección E/S

Toda E/S realízase a través de **ficheiros**, e cada proceso ten asociado 3 ficheiros E/S:

Descriptor	*Nome*	*Destino*
0	strandard input (`stdin`)	teclado
1	standard output (`stdout`)	pantalla
2	standard error (`stderr`)	pantalla

Podemos **redireccionar** a E/S con:

- `<`: redirección de *entrada* estándar (`cat < listado` lee o contido de listado e o pon por pantalla)
- `>`: redirección de *saída* estándar a un arquivo (`ls > listdo` garda a saída de `ls` no arquivo listado)
- `<<`: redirección de *entrada* estándar *heredoc*. Permite definir un bloque de entrada ata atopar unha *palabra clave*:

```
$ cat << EOF
Lee todo o texto
ata atopar un
EOF
```

- `>>`: redirección de saída estándar a un arquivo (en modo concatenación, polo que engade o contido ao final do arquivo sen sobreescribilo)
- `|`: **Pipe**. Redirixe a *saída dun comando á entrada doutro*
  * Para redirección do `stderr`: `2>`

Avaliación de comandos

Os comandos teñen un **código de saída** que se almacena en `$?`: 0 se terminou ben; >0 se terminou cun erro
Pódense executar **varios comandos seguidos condicionalmenvarios comandos seguidos condicionalmentete** con `&&` ou `||`: `cmnd1 && cmnd2`
Orde de avaliación de comandos:

1. *Redirección E/S*
2. Expansión de *variables*
3. Expansion de *nomes de ficheiro*

Comandos para o procesamento de texto:

Chámanselle **FILTROS**: `cat`, `head`, `tail`, `grep`, `find`

- Permiten manipular ficheiros e empregan a `stdin` e a `stdout`, aínda que se poden canalizar e redireccionar
  `grep` ("Global Regular Expression Print"): serve para buscar patróns de texto e ten a seguinte forma:
  `grep [opcions] "patrón" arquivo`

CUESTIÓN:

No seguinte código:

```
$> ls | more
$> pipe=\|
$> ls $pipe more
```

Por que `ls | more` funciona e `ls $pipe more` non?
Porque ao declarar `|` como unha variable, a shell interpreta o pipe como unha cadea de texto, xa que cree que `|` e `more` son args para `ls`. Poderíamos facer: `eval "ls $pipe more"`

Programación de scripts en shell

Un **script** é un *ficheiro* de texto que contén unha *secuencia de comandos* que se executan secuencialmente.
Un script *comeza* sempre polo **shebang** (`#!`), que lle indica o *intérprete de comandos que empregar* polo script. Por exemplo `#!/bin/bash` indica que se execute meidante bash

A **execución** dun script dende a liña de comandos (`$> ./script.sh`) require de *permisos de execución*:
`$> chmod +x script.sh`
En cambio, se lle indicamos a **shell como arg**, só precisa de *permisos de lectura*: `$> bash script.sh`

Paso de parámetros ao script

Pódenselle pasar argumentos *dende a liña de comandos*: `$> ./script.sh [PARAMS]`
Estes **parámetros se almacenan por orde** nas variables:

- `$0`: *nome* do script
- `$1` a `$9`: *parámetros do 1 ao 9*
- `${10}`, `${11}`, … : parámetros por encima de 9
- `$#`: *número* de parámetros
- `$*`, `$@`: *todos* os parámetros

Tamén se poden empregar as **variables de entorno** ou **outras variables** como `$?` (para o código de saída do comando) ou `$$` (PID do script actual)

Estruturas de control de fluxo

Comparación de valores

Para **comparar valores** podemos empregar `[ expresion ]` ou `test expresion`, que *devolve 0 se é correcto e 1 se é falso* (ollo! Ao revés que C). Expresións de `test`:

- `-d`: Comproba que o parámetro é un *directorio*
- `-f`: Comproba que o parámetro é un *ficheiro regular*
- `n1 -eq n2`, `n1 -gt n2`, … : Compara *números enteiros*
- `S1 = S2`, `S1 != S2`: Compara/verifica *cadeas de texto*
  *Ollo! Se empregamos `[ ]`, debe haber un **espacio** despois de `[` e antes de `]`

Operadores lóxicos:

- `!`: NOT
- `-a`: AND
- `-o`: OR
- `\( expr \)`: agrupación de expresións

Comando test extendido:

A partir da versión 2.02, pódese empregar `[[expr]]` para realizar comparacións: permite empregar os operadores `&&` e `||` para unir expresións e non precisa de escapar os paréteses.
Ex: `if [[ $? -gt 0 && ($USER = 'root' || $USER = ‘admin’) ]]`

Estrutura if ... then ... fi:

```
if comando1 then
    Codigo
elif comando2 then
    Codigo
else
    Codigo
fi
```

*O `if` só comproba a saída do comando (`$?`)

Estrutura case ... in:

```
case expresion in
    case1)
        bloque de comandos
        ;;
    case2)
        bloque de comandos
        ;;
    *)
        bloque de comandos por defecto
        ;;
esac
```

Estrutura for:

```
for variable in lista
do
    bloque de comandos empregando $variable
done
```

*Podemos empregar e expandir variables para crear a lista
Sintaxe alternativa:

```
for ((a=0; a < 5; a++))
do
    sufixo="0$a"
    touch servidor_${sufixo}.data
done
```

*Ollo! co *dobre paréntese*

Estrutura while:

```
while comando
do
    bloque de comandos
done
```

Estrutura until:

```
until comando
do
    bloque de comandos
done
```

Coas estruturas `for`, `while` e `until` podemos empregar **`break`** e **`continue`** para *saír dun lazo* ou *saltar á seguinte iteración*. *Con `break n` podemos saír de varios lazos á vez.

Redireccións

Para **gardar** a saída do script nun ficheiro: `echo $resultado > ficheiroSaida`
Para **ler** o contido dun ficheiro: `read variable < ficheiro` (só le a primeira liña)
Se queremos **ler varias liñas** podemos empregar unha estrutura de control:

```
while read BUFFER
do
    echo "$BUFFER" >> $2 # Garda as liñas que le no segundo arg
done < $1 # O ficheiro de entrada é o primeiro arg
```

Funcións

Podemos organizar o código en **funcións** da seguinte forma:

```
funcion(){
    comandos
}
```

Os **parámetros** se almacenan por *orde* nas variables `$n` (igual que os parámetros do script) e `${FUNCNAME[0]}` contén o nome da función, xa que `$0` segue a conter o nome do script.

Ao igual que en C, o **código de saída** especifícase cun `return` (por defecto devólvese o código do último comando) e se almacena en `$?` xusto despois de chamar a función.

Misc

A **sustitución dun comando** permite *asignar a saída dun comando a unha variable*, empregando ` (comilla aguda) ou `$()`. Ex: `x=$(pwd); echo $x`

En canto ao **rendemento**, o shell é bastante ineficiente realizar *traballos pesados* como empregar bucles para ler ficheiros.