

Shell scripting: repaso y nuevos conceptos

Explicación de práctica 1

Sistemas Operativos

Facultad de Informática
Universidad Nacional de La Plata

2020



1 Repaso

- Definiciones

- Elementos de una shell

- Funcionamiento

- En *nix todo es un archivo

- Comandos útiles

- Ejemplos

2 Pipes y Netcat

- Pipes

 - Unnamed Pipes

 - Named Pipes

- Netcat

- Netcat con Pipes y Redirecciones



1 Repaso

- Definiciones

- Elementos de una shell

- Funcionamiento

- En *nix todo es un archivo

- Comandos útiles

- Ejemplos

2 Pipes y Netcat

- Pipes

 - Unnamed Pipes

 - Named Pipes

- Netcat

- Netcat con Pipes y Redirecciones



- Una *shell* es un intérprete de comandos
- Interactiva o *batch*
- En *nix: shells configurables e intercambiables
 - sh
 - bash (nos basaremos en esta para la materia)
 - dash
 - csh
 - zsh
 - ...entre otras...



- Embebidos en la shell
- `man bash` para listarlos
- `help COMANDO` para ver su documentación
- Hemos visto en ISO:
 - `cd, source, for, if, while, continue, break, echo, exit, return, export, bg, fg, help, kill, let, local, pwd, read, test (o [], o [[]])`



- Cualquier binario que instalemos
- ¡Cualquier script que hagamos nosotros mismos!
- `man COMANDO` para ver su documentación ->
/usr/share/doc (texto plano, HTML, gzip)
- Parámetro `--help`
- Hemos visto en ISO:
 - `grep, cat, cut, ln, ls, find, expr, rm, mv, cp, whereis`



- Comienzan con numeral

```
# Esto es un comentario  
ls /etc # Y esto también
```

- Son útiles para documentar nuestro código para futura referencia



- Son comandos internos de bash
 - if
 - for
 - while
 - ...
- if y while responden a *true* y *false*
- Ejemplo:

```
if [ 4 -gt 3 ]  
then  
    echo 4 es mayor que 3  
else  
    echo algo esta roto  
fi
```



- Escapan la tokenización (*\$IFS* y *;*)
- **Strings**
 - `hola`
 - `"hola"`
 - `'hola'`
 - `"string con espacios que reemplaza valores"`
 - `'string con espacios que no reemplaza valores'`



- **Arreglos**

- `()`
- `(esto es un arreglo con 7 elementos)`
- `(este tiene "tres elementos")`



- **Asignación (=)**

- `mi_variable=valor`
- `variable='un texto'`
- `arreglo=(uno dos tres)`

- **Acceso (\$)**

- `echo $mi_variable`
- `echo "imprimo $variable en pantalla"`
- `echo 'imprimo $variable en pantalla'`



- **Asignación (=)**
 - `arreglo=(1 2 3)`
 - `arreglo[3]=cuatro`
 - `arreglo[1]=dos`
 - `unset arreglo[0]`
- **Acceso (\$)**
 - `echo ${arreglo[0]}`
 - `echo ${arreglo[*]}`
 - `echo $#arreglo[*]}`



- Por defecto, todas las variables son **globales** al script
- Se pueden definir variables *locales* mediante `local`
- Existen variables globales a la sesión, llamadas **variables de entorno**:
 - `export MI_VARIABLE="su valor"`
 - Ejemplos: `$HOME`, `$PATH`, `$PWD`, `$UID`, `$EUID`, `$IFS`
 - Se pueden consultar las variables de entorno definidas mediante `export` o `env`
 - Se pueden modificar
 - Bash utiliza `$HOME/.bashrc` para inicializar la shell. Por ejemplo: variable `$PATH`, `$PS1`, etc.



- Son read-only y pueden ser de mucha utilidad:
 - `$!`: Mantiene el *PID* del último comando ejecutando en *background*.
 - `$_`: Mantiene el último parámetro del último comando ejecutado.
 - `$$`: Mantiene el *PID* del proceso actual, es decir el script en si mismo.



- Encapsulan lógica
- Una vez definidas, funcionan como nuevos comandos
- Para definirlas:

```
function imprimir() {  
    echo $*  
}
```

```
imprimir Bienvenidos a SO
```



- Son unidireccionales
- Toman la salida de un comando y la pasan a un archivo
- Pueden ser destructivas o no
- Por ejemplo:

```
echo hola > /tmp/salida.txt  
echo chau >> /tmp/salida.txt
```



- Son unidireccionales
- Toman el contenido de un archivo y lo pasan a la entrada de un comando
- Por ejemplo:

```
mysql basededatos < dump.sql
```



- Son unidireccionales
- Toman la salida de un comando y se la hacen entrada del siguiente
- Son un mecanismo de **IPC**
- Utilizan *buffering*
- Por ejemplo:

```
cat /etc/passwd | cut -d: -f2 | grep a | wc -l
```



- Primer línea (opcional) de un script
- Indica qué comando se usará para ejecutar el script
- Pueden definir parámetros para la invocación
 - `#!/bin/bash`
 - `#!/usr/bin/env ruby`



- Se los accede posicionalmente, a partir del índice 1:
 - \$1 a \$n
 - \$0
 - \$*
 - \$#



- Denotan si la operación realizada por un comando o una función fue exitosa o no
 - 0 en caso de éxito
 - 1 a 255 en caso de error
- Se retornan con comandos internos:
 - `exit` en el contexto de un comando/script
 - `return` en el contexto de una función
- Se consultan con una variable especial: `$?`



- Reemplaza un comando por su salida estándar
- Literalmente coloca la salida de un comando en el lugar que este ocupa
- Se realiza con cadenas especiales:
 - ``:echo "mi nombre es `whoami`"`
 - `$():echo "mi nombre es $(whoami)"`



- Si se tienen permisos de ejecución:
 - `./script.sh`
- Especificando el binario a utilizar
 - `bash script.sh`
- En modo de depuración
 - `bash -x script.sh`



- Cada comando tiene *al menos* 3 archivos abiertos:
 - `stdin (0)`: entrada estándar
 - `stdout (1)`: salida estándar
 - `stderr (2)`: salida de error estándar
- Como son archivos, podemos leer de y escribir en ellos
 - `read variable` lee de `stdin`
 - `read variable < ARCHIVO` escribe el contenido de `ARCHIVO` en `stdin` para el comando `read`
 - `echo Un mensaje` escribe en `stdout` (equivale a `echo Un mensaje >&1`)
 - `echo Un mensaje >&2` escribe en `stderr`



- Imprimir texto

```
echo Shell scripting es sencillo
```



- Imprimir el contenido de un archivo

```
cat archivo
```



- Leer de entrada estándar (teclado) en una variable

```
read variable
```



- Leer el primer campo delimitado por :

```
cut -d: -f1 # lee de stdin
```

```
cut -d: -f1 archivo # lee desde archivo
```



Traducir (intercambiar) caracteres

- Traducir (intercambiar) caracteres

```
echo hola | tr a-z A-Z # traduce las minúsculas  
por mayúsculas
```



- Contar la cantidad de líneas

```
wc -l # de stdin  
wc -l archivo # de archivo
```



- Buscar recursivamente archivos por su nombre en el directorio actual (.)

```
find . -name "*.sh"
```



- Buscar un patrón

```
grep shell # en stdin  
grep shell archivo # en archivo  
grep shell * # en todos los archivos del  
directorio actual
```



- Empaquetado de archivos

```
tar -cf destino.tar origen1 origen2 # empaqueta en  
destino.tar  
tar -xf destino.tar # desempaqueta destino.tar
```



- Compresión de archivos

```
gzip destino.tar # comprime destino.tar en destino
tar .tar.gz
gzip -d destino.tar.gz # descomprime destino.tar.
gz
```



- Empaquetado y compresión con `tar`

```
tar -cfz destino.tar.gz origen1 origen2 # invoca  
gzip automáticamente  
tar -xzf destino.tar.gz # descomprime y  
desempaqueta destino.tar.gz
```



- Listado de procesos

```
ps -aux # lista los procesos en ejecución de todos  
los usuarios
```



- `https://github.com/unlp-so/shell-scripts`



1 Repaso

Definiciones

Elementos de una shell

Funcionamiento

En *nix todo es un archivo

Comandos útiles

Ejemplos

2 Pipes y Netcat

Pipes

Unnamed Pipes

Named Pipes

Netcat

Netcat con Pipes y Redirecciones



- Son los que se utilizaron para realizar los scripts en ISO (|)
- ¿Cuántos procesos actúan en la siguiente línea ejecutada en una terminal? (*subshell*¹)

```
ls | grep x
```

```
echo "Subshell_1 \${BASH_SUBSHELL}=\${BASH_SUBSHELL} \
  \${BASHPID}=\${BASHPID} \${PPID}=\${PPID}" | (read
  v; echo "$v - Subshell_2 \${BASH_SUBSHELL}=\
  \${BASH_SUBSHELL} \${BASHPID}=\${BASHPID} \${PPID}=\
  \${PPID}")
```

¹<http://www.tldp.org/LDP/abs/html/subshells.html>



- Son los que se utilizaron para realizar los scripts en ISO (|)
- ¿Cuántos procesos actúan en la siguiente línea ejecutada en una terminal? (*subshell*¹)

```
ls | grep x
```

```
echo " Subshell_1 \${BASH_SUBSHELL}=${BASH_SUBSHELL} \
  \${BASHPID}=${BASHPID} \${PPID}=${PPID}" | (read
  v; echo "$v - Subshell_2 \${BASH_SUBSHELL}=${BASH_SUBSHELL} \
  \${BASHPID}=${BASHPID} \${PPID}=${PPID}")
```

¹<http://www.tldp.org/LDP/abs/html/subshells.html>



- También conocidas con el nombre de *FIFO*
- Se representan mediante archivos
- Se crean con el comando *mkfifo* o *mknode*
- Se identifican con la letra *p*

```
$ ls -l  
prw-r--r--  1 root  root    0 Jan 22 23:11  
    named_pipe
```



- Si en una terminal ejecutamos lo siguiente

```
mkfifo pipe_so; ls -l > pipe_so
```

- Y en otra terminal ejecutamos lo siguiente

```
cat < pipe_so
```

- ¿Qué sucede?



- Es una utilidad de red destacada
- Permite leer y escribir datos a través de conexiones *TCP*

```
# En el servidor  
nc -l port_number
```

```
# En el cliente  
nc host_name port_number
```

- También permite utilizar *UDP* (parametro *-u*)
- Tunneling
- Escaneo de puertos (parametro *-z*)

```
nc -zv host_name port_number
```



- Transferencia de datos, como el contenido de un archivo

```
# En el servidor
```

```
nc -l -p port_number
```

```
# En el cliente
```

```
nc port_number -q 1 < /home/so/archivo_a_enviar
```

- ¿Cuál es el comportamiento el siguiente código?

```
# En el servidor
```

```
cat named_pipe | nc -l -p 1500 > named_pipe
```

```
# En el cliente
```

```
nc localhost 1500
```



- Responder las preguntas en base a la ejecución del siguiente *script*:
 - `https://github.com/unlp-so/shell-scripts/blob/master/12-subshells-scope-variables.sh`
- ¿Por qué?:
 - ① El valor de "i" es "1" y por qué es "2" en determinados casos
 - ② El valor de "SUBSHELL" es "0" y por qué es "1" en determinados casos
 - ③ Dentro de la función "f1" existe ">&2"



¿Preguntas?

