Tema 12: Introducción a la programación Shell

Contenido

1. Interprete de comandos: Shell

El intérprete de comandos es el programa que recibe lo que se escribe en la terminal y lo convierte en instrucciones para el sistema operativo.

En otras palabras el objetivo de cualquier intérprete de comandos es ejecutar los programas que el usuario teclea en el prompt del mismo. El prompt es una indicación que muestra el intérprete para anunciar que espera una orden del usuario. Cuando el usuario escribe una orden, el intérprete ejecuta dicha orden. En dicha orden, puede haber programas internos o externos: Los programas internos son aquellos que vienen incorporados en el propio intérprete, mientras que los externos son programas separados (ej: aplicaciones de /bin,/usr/bin,...).

En el mundo Linux/Unix existen tres grandes familias de Shells como se muestra en la tabla a continuación. Estas se diferencian entre sí básicamente en la sintaxis de sus comandos y en la interacción con el usuario.

Tipo de Shell	Shell estándar	Clones libres
AT&T Bourne shell	sh	ash, bash, bash2
Berkeley "C" shell	csh	tcsh
AT&T Korn shell	ksh	pdksh, zsh
Otros interpretes	--	esh, gush, nwsh

Interpretes de comandos en Linux/Unix

1.1. Sintaxis de los comandos

Los comandos tienen la siguiente sintaxis:

```
# programa arg1 arg2 ... argn
```

Se observa que, en la ``línea de comandos'', se introduce el programa seguido de uno o varios argumentos. Así, el intérprete ejecutará el programa con las opciones que se hayan escrito.

Cuando se quiere que el comando sea de varias líneas, se separa cada línea con el carácter barra invertida (). Además, cuando se quiere ejecutar varios comandos en la misma línea, los separa con punto y coma (;). Por ejemplo:

```
# make modules ; make modules_install
```

En los comandos, también se puede utilizar los comodines:

- El asterisco () es equivalente a uno o más caracteres en el nombre de un archivo.

Ejm: ls *.c lista todos los archivos con extensión c.

• El signo de interrogación (?) es equivalente a un único carácter.

Ejm: ls curso.te? lista el archivo curso.tex completando el último carácter.

• Un conjunto de caracteres entre corchetes es equivalente a cualquier carácter del conjunto.

Ejm: ls curso_linux.t[aeiou]x lista curso_linux.tex selecionando la e del conjunto. .

2. Variables de entorno

Una variable de entorno es un nombre asociado a una cadena de caracteres.

Dependiendo de la variable, su utilidad puede ser distinta. Algunas son útiles para no tener que escribir muchas opciones al ejecutar un programa, otras las utiliza el propio shell (PATH, PS1,...). La tabla muestra la lista de variables más usuales.

Variable	Descripción
DISPLAY	Donde aparecen la salidas de X-Windows.
HOME	Directorio personal.
HOSTNAME	Nombre de la máquina.
MAIL	Archivo de correo.
PATH	Lista de directorios donde buscar los programas.
PS1	Prompt.
SHELL	Intérprete de comandos por defecto.
TERM	Tipo de terminal.
USER	Nombre del usuario.

Variables de entorno más usuales

La forma de definir una variable de entorno cambia con el interprete de comandos, se muestra tcsh y bash siendo los dos más populares en el ámbito Linux:

bash:

export VARIABLE=Valor

tcsh:

setenv VARIABLE Valor

3. Programación shell

La programación del shell es una de las herramientas más apreciadas por todos los administradores y muchos usuarios de Linux/Unix ya que permite automatizar tareas complejas, comandos repetitivos y ejecutarlas con un solo llamado al script o hacerlo automáticamente a horas escogidas sin intervención de personas.

Los programas de shell no necesitan ser complicados, como ocurre en otros lenguajes, y son ejecutados línea a línea por lo que a estos programas se les conoce con el nombre de shell script.

La programación shell en Unix/Linux es, en cierto sentido, equivalente a crear archivos .BAT en DOS. La diferencia es que en Unix/Linux es mucho mas potente. Estos scripts pueden usar un sinnúmero de herramientas como:

- Comandos del sistema Linux/Unix (ejm: ls, cut)
- Funciones intrínsecas del shell (ejm: kill, nice)
- Lenguaje de programación del shell (ejm: if/then/else/fi) (ver tabla de comandos)
- Programas y/o lenguajes de procesamiento en línea. (ejm: awk, sed, Perl)
- Programas propios del usuario escritos en cualquier lenguaje.

El lenguaje de programación de cada shell provee de una amplia gama de estructuras de control como se muestra en la tabla de commandos de la shell

- for name [in word;] do list ; done
- select name [in word ;] do list ; done
- case word in [pattern [| pattern]\ldots) list ;;]\ldots esac
- if list then list [elif list then list]\ldots [else list] fi
- $while list do list done
- $until list do list done
- [function] name () { list; }

Instrucciones bash para programación shell tbl_instr_bash Un sencillo ejemplo es realizar un backup de solo ciertos directorios (prog_dir1 y prog_dir2), luego comprimirlos usando bzip2 y enviarlos a un area de almacenamiento (digamos una unidad ZIP previamente montada en /mnt/zipdrive), y además con que el nombre del archivo contenga la fecha del día. Suena difícil? Realmente no lo es.

Se crea un archivo texto con cualquier nombre, por ejemplo mibackup que contenga las instrucciones que se desea ejecutar.

```
#!/bin/sh
```

```
#
echo "---------- Captura fecha -----------"
fecha=‘date +%Y%m%d‘
#
echo "---------- Haciendo Tar  -----------"
tar cvf backup$fecha.tar prog_dir1 prog_dir2
#
echo "----------- Comprimiendo -----------"
bzip2 backup$fecha.tar
#
echo "---------- Enviándolos a zip -------"
cp ./backup$fecha.tar /mnt/zipdrive
#
echo "----------- Limpiando --------------"
rm -f ./backup$fecha.tar
#
echo "----------- Final      -------------"
```

Luego, se le asigna permisos de ejecución con el comando

chmod +x mibackup

y está listo para ser ejecutado.

4. Conceptos básicos

4.1.- Variables

Como en cualquier lenguaje de programación, las variables se utilizan para poder guardar información y a partir de ella poder tomar decisiones o realizar operaciones. Lógicamente, las variables no pueden tener el nombre de ninguna palabras reservada (p.e echo) y hay dos formas diferentes de utilizarla dependiendo de si queremos asignarle un valor o operar con ellas. A continuación vamos a ver un ejemplo en el que se le asigna a la variable numero un valor y luego se muestra por pantalla.

```
#!/bin/bash
numero=5
echo "el valor de la variable es "$numero
Como se puede ver en el ejemplo cuando se quiere acceder al valor de
la variable se utiliza el símbolo $.
```

4.2.- Paso de parámetros

A menudo es necesario que nuestros scripts reciban parámetros desde la línea de comandos para hacerlos más versátiles. Los parámetros se pueden usar dentro del script como cualquier otra variable de shell. Los parámetros dentro del shell script son accesibles utilizando las variables: $0 es el nombre del programa, $1 es el primer parámetro, $2, es el segundo parámetro, etc. Además, se utiliza la variable $# que establece el número de parámetros que ha recibido el shell. A continuación vamos a ver un ejemplo:

```
#!/bin/bash
echo "El nombre del programa es "$0
echo "El primer parámetro recibido es "$1
echo "El segundo parámetro recibido es "$2
echo "..."
echo "En total se han recibido "$#" parámetros"
```

Las siguientes variables son muy útiles al programar los guiones shells:

- *$?*: esta variable contiene el valor de salida de la última orden ejecutada. Es útil para saber si una orden ha finalizado con éxito o ha tenido problemas. Un '0' indica que no ha habido errores, otro valor indica que sí ha habido errores.
- *$!*: identificador de proceso de la última orden ejecutada en segundo plano.
- *$$*: el identificador de proceso (PID) de este shell, útil para incluirlo en nombres de ficheros para hacerlos únicos.
- *$-*: las opciones actuales suministradas para esta invocación del shell.
- *$**: todos los argumentos del shell comenzando por el *$1*. Cuando la expansión ocurre entre comillas dobles, se expande a una sola palabra con el valor de cada parámetro separado por el primer carácter de la variable especial *IFS*. Esto es, .$*. es equivalente a .$1c$2c. . . .., donde c es el primer carácter del valor de la variable *IFS*. Si *IFS* no está definida, los parámetros se separan por espacios. Si *IFS* es la cadena vacía, los parámetros se juntan sin ningún separador.
- *$@*: igual que el anterior, excepto cuando va entrecomillado. Cuando la expansión ocurre dentro de comillas dobles, cada parámetro se expande a una palabra separada. Esto es, .$@. es equivalente a .$1. .$2.. . .

5.- Entrada y salida de datos

5.1.- E/S por consola

Como ya hemos visto antes, la salida de datos se realiza con el comando echo y la entrada de datos, además de poder realizarla con el paso de parámetros se realiza con el comando read. A continuación se muestra un ejemplo:

```
#!/bin/bash
echo -n "Introduce el valor de la variable: "
#el parámetro -n se utiliza para evitar el salto de línea
read numero
echo "El valor introducido es: "$numero
```

Además, la entrada y salida de datos se puede realizar a través de ficheros o del resultado de la ejecución de un comando, pero eso lo veremos más adelante

6.- Operaciones aritmético lógicas

Como cualquier lenguaje de programación se pueden realizar operaciones aritmético y lógicas sobre las variables. Para realizar operaciones se utiliza el comando expr y para realizar comparaciones se utiliza el comando test. A continuación vamos a ver a fondo cada uno de los comandos.

6.1.- Expr

El comando expr se utiliza principalmente para realizar operaciones aritméticas simples y, en menor medida para manipular cadenas. La sintaxis de expr es: expr arg1 op arg2 [op arg3...] En la tabla 1 podemos ver los diferentes operaciones de expr A continuación vamos a ver un ejemplo de uso de operadores aritméticos:

```
#!/bin/bash
echo -n "Introduce un valor: "
read var1
echo -n "Introduce un valor: "
read var2
resultado=$(expr $var1 \* $var2)
echo "El resultado de la multiplicación es "$resultado
```

Otra funcionalidad adicional de la función expr y que resulta muy interesante a la hora de programar es la generación de números aleatorios:

```
#/bin/bash
numero=$(expr $RANDOM % 100)
echo $numero
```

Tabla 1. **Operadores de expr**

Operador	**Comentario**
Operadores aritméticos	
+	Suma
-	Resta
*	Multiplicación. El operador * va precedido de \ porque * ya tiene un significado en GNU/Linux
/	División
%	Resto de la división
Operadores relacionales	
=	Igualdad
!=	Diferentes
>	Mayor
>=	Mayor o igual
<	Menor
<=	Menor o igual
Operadores lógicos	
|	Or lógico
&	And lógico

6.2.- Test

El comando test permite evaluar tres tipos de elementos: archivos, cadenas y números. Su sintaxis es:

```
test - opcion archive

 test [ expresión ]
```

En la tabla 2 podemos ver las diferentes opciones que nos permite utilizar el comando test según el tipo de datos.

Tabla 2. **Opciones del comando test**

Opción	Descripción
Archivos o directorios	
-f	Devuelve verdadero (0) si el archivo existe y es un archivo regular (no es un directorio ni un archivo de dispositivo)
-s	Devuelve verdadero (0) si el archivo existe y si su tamaño es mayor que 0.
-r	Devuelve verdadero (0) si el archivo existe y tiene permisos de lectura.
-w	Devuelve verdadero (0) si el archivo existe y tiene permisos de escritura.
-x	Devuelve verdadero (0) si el archivo existe y tiene permisos de ejecución.
-d	Devuelve verdadero (0) si existe y es un directorio.
Valores numéricos	
-lt	Menor que
-le	Menor o igual que
-qt	Mayor que
-qe	Menor o igual que
-eq	Igual a
-ne	No igual a
Conectores	
-o	OR

-a	AND
!	NOT

A continuación podemos ver un ejemplo de cada tipo:

- Evaluación de ficheros o directorios:

```
test -f /etc/passwd
```

- Evaluación de cadenas

```
test [cadena1 = cadena2]
test [cadena1 != cadena2]
```

- Evaluación numérica. Las evaluaciones numéricas son sólo válidas para números enteros y su sintaxis es:

```
test [ numero operador numero ]
```

```
Por ejemplo:
```

```
Test [ 20 –lt 40 ]
```

¡OJO! Los espacios en blanco entre la expresión y los corchetes son necesarios.

7. - Estructuras de control

Las estructuras de control permiten cambiar el flujo de programa, en función del estado de las variables.

7.1.- Condición simple (IF)

Las condiciones simples (if) nos permiten que en caso de cumplirse una determinada condición se ejecute un determinado código. La sintaxis de las sentencias if es:

if condición

```
  then
     comandos
  else
     comandos
fi
```

A continuación podemos ver un ejemplo

```
#!/bin/bash
echo -n "Introduce un valor: "
read var
```

```
if (( var < 10 ))
then
   echo "Es menor que 10"
else
   echo "Es mayor que 10"
fi
```

7.2.- Condiciones múltiples (CASE)

Cuando queremos realizar muchas condiciones sobre un mismo valor (p.e. en un menú) la mejor opción es utilizar case. Su estructura es:

```
case  $variable in
   valor1)   commando
             ..
             commando;;
   valor2)   commando
             ..
             commando;;

…

   *)        commando
             ..
             commando;;
esac
```

A continuación se puede ver un ejemplo sencillo:

```
#!/bin/bash
echo -n "Introduce un valor: "
read var1
case $var1 in
 1) echo " uno ";;
 2) echo " dos ";;
 3) echo " tres ";;
 4) echo " cuatro ";;
 *) echo "opcion incorrecta ";;
esac
```

7.3.- Bucle for

El bucle for se utiliza para ejecutar un código un determinado número de veces. Su sintaxis es:

```
for (( expr1; expr2; expr3 )) do

  ...
Done
```

A continuación podemos ver un ejemplo que muestra los números del 0 al 5:

```
#!/bin/bash
for ((  i = 0 ;  i <= 5;  i++  ))
do
 echo " $i "
```

```
done
```

También se puede utilizar el for para moverse en una lista de elementos

for variabe in lista_elementos

do

```
 echo " $variable "
done
```

7.4.- Bucle while

El bucle while permite ejecutar un código hasta que no se cumpla una determinada condición de salida. Su sintaxis es:

while [condición]

do

```
   comando1
   comando2
   ...
done
```

A continuación podemos ver un ejemplo muy sencillo:

```
#/bin/bash
limite=5
i=0;
while (test $limite -gt $i)
do
   echo “Valor $i”
   let i=$i+1
done
```