8.Substituciones

Substitución de Variables

La substitución de variables permite al programador del shell manipular el valor de una variable basado en su estado. La substitución de variables se clasifica en las siguientes categorías:

- Acciones tomadas cuando la variable tiene valor
- Acciones tomadas cuando la variable no tiene valor

Forma	Descripción
${parameter:-word}	Si parameter es nulo o no está definido, word es substituido por parameter. El valor de parameter no cambia.
${parameter:=word}	Si parameter es nulo o no está definido, parameter es seteado al valor de word.
${parameter:?message}	Si parameter es nulo o no está definido, message es impreso a la salida de error estándar (stdout). Chequea que la variable esté correctamente definida.
${parameter:+word}	Si parameter está definido, word es reemplazado por parameter. El valor de parameter cambia.

Substituyendo por un Valor por Defecto

La primera forma permite substituir por un valor por defecto cuando la variable no está definida o es nula.
Un simple ejemplo de uso es:

```
PS1=${HOST:-localhost}"$ " ; export PS1 ;
```

Asignando un Valor por Defecto

Para setear el valor de una variable, se debe usar la segunda forma.

En el ejemplo anterior queda:

```
PS1=${HOST:=`uname -n`}"$ " ; export PS1 HOST ;
```

Abortando por Error en una Variable

A veces substituir un valor por defecto puede ocultar problemas, para esto el shell soporta la tercera forma de substitución que permite imprimir un error.
Esto se usa comúnmente un script requiere que ciertas variable estén definidas para funcionar correctamente. Por ejemplo , el siguiente comando termina si la variable $HOME no está definida:

```
${HOME:?"No está definido el directorio home."}
```

La última forma de substitución de variables es usada para substituir cuando una

variable está definida.
El uso común de esta variable es para indicar que el script está ejecutando en modo debug:

echo ${DEBUG:+"Debug is active."}

Substitución de Comandos y Expresiones Aritméticas

El shell provee dos formas de substitución adicionales:

- Substitución de comandos
- Substituciones aritméticas

La substitución de comandos permite capturar la salida de un comando y substituirla en otro comando, mientras que la substitución aritmética permite hacer operaciones matemáticas simples sobre enteros.

La substitución de comandos se utiliza generalmente para asignar la salida de un comando a una variable. Los siguientes ejemplos muestran la substitución de comandos:

```
DATE=`date`
USERS=`who | wc -l`
UP=`date ; uptime`
grep `id -un` /etc/passwd
```

Substitución Aritmética

En bash, el shell permite realizar operaciones aritméticas sobre enteros. Esto evita tener que ejecutar un programa adicional cómo expr o bc para hacer operaciones matemáticas en un script del shell.
La substitución aritmética se realiza cuando se escribe un comando de la siguiente forma:

```
$((expression))
```

```
foo=$(( ((5 + 3*2) - 4) / 2 ))
```

Resumen

En este capítulo se vieron cuatro formas principales de substitución disponibles en el shell:

- Substitución de nombre de archivos
- Substitución de variables
- Substitución de comandos
- Substituciones aritméticas

A medida que se escriban scripts y se utilice el shell para resolver problemas se va a encontrar la utilidad de estas substituciones.

9.Quoting

En el capitulo anterior, se vieron las substituciones del shell, que suceden automáticamente cuando se ingresa un wildcard o el símbolo $. La forma en el shell interpreta estos y otros caracteres especiales generalmente es útil, pero algunas veces es necesario deshabilitar las substituciones. La deshabilitación del significado especial de los caracteres se llama “quoting”, y se puede hacer de 3 formas:

- Uso de la barra invertida (\)
- Uso de la comilla simple (')
- Uso de la comilla doble (")

Quoting con Barras Invertidas

Vamos a utilizar el comando echo que muestra sus argumentos en la siguiente línea. Por ejemplo,

```
echo Hola mundo
```

muestra el siguiente mensaje en pantalla:

```
Hola mundo
```

Esta es una lista con la mayoría de los caracteres que el shell interpreta como especiales:

- ? [ ] ' " \ $; & ( ) | ^ < > new-line space tab

Que sucede su se agrega uno de estos caracteres al comando echo:

```
echo Hola; mundo
```

Ahora da el siguiente error:

```
Hola
bash: mundo: Command not found
```

```
echo Debe $1250
```

Esto parece una simple sentencia echo, pero la salida no es la esperada porque $1 es una variable especial del shell:

```
Debe 250
```

El signo $ es uno de los caracteres especiales, por lo tanto debe ser “quoteado” para impedir que el shell lo maneje de manera especial:

```
echo Debe \$1250
```

Ahora se obtiene la salida:

Debe $1250

Usando Comillas Simples

Este es un ejemplo del comando echo que debe ser modificado porque contiene caracteres especiales:

```
echo <-$1250.**>; (actualizar?) [s|n]
```

Poniendo la barra invertida adelante de cada caracter especial es tedioso y hace la línea difícil de leer:

```
echo \<-\$1250.\*\*\>\; \(actualizar\?\) \[s\|n\]
```

Hay una forma más simple de “quotar” un grupo de caracteres grande. Poniendo un caracter (') al comienzo y final de la línea:

```
echo '<-$1250.**>; (actualizar?) [s|n]'
```

Cualquier carácter entre comillas simples son “quoteados” como si tuviesen la barra invertida adelante. Ahora el comando echo imprime correctamente.

Si la comilla simple aparece en el medio de una cadena, no se debe poner la cadena entre comillas simples:

```
echo 'It's Friday'
```

Esto falla mostrando solamente el siguiente carácter, mostrando el cursor para que se ingrese más datos:

```
> _
```

La solución correcta para este caso sería:

echo It\'s Friday

Usando Comillas Dobles

Las comillas simples a veces puede deshabilitar demasiadas de las ventajas especiales del shell. Por ejemplo la siguiente sentencia echo contiene muchos caracteres especiales que deben ser “quoteados” para utilizarlos literalmente:

```
echo '$USER debe <-$1250.**>; [ a la fecha de (`date +%m/%d`) ]'
```

La salida utilizando comillas simples es fácil de predecir:

```
$USER debe <-$1250.**>; [ a la fecha de (`date +%m/%d`) ]
```

Las comillas simples también evitan la substitución de comandos (Capítulo 8), de esta forma cuando se quiere insertar el día y mes actual usando el comando date, falla.

Las comillas dobles son la solución al problema.

Las comillas dobles invalidan el significado especial de todos los caracteres excepto los siguientes:

- $ para substitución de parámetros.
- ` comillas invertidas para habilitar substitución de comandos.
- \$ para habilitar el literal del signo dolar.
- \` para habilitar la comilla invertida literal.
- \" para habilitar las comillas dobles embebidas.
- \\ para habilitar las barras invertidas embebidas.
- Todos los caracteres precedidos por \ son literales (no especiales).

Que sucede si se encierra con comillas dobles:

```
echo "$USER debe <-$1250.**>; [ a la fecha de (`date +%m/%d`) ]"
```

Queda:

```
ltsp debe <-250.**>; [ a la fecha de (12/21) ]
```

Caracter	***Efecto***
Comillas simples	*Todos los caracteres entre comillas simples pierden su significado especial.*
Comillas dobles	*La mayoría de los caracteres especiales pierden su significado especial con excepción de:* • *$* • *`* • *\$* • *\`* • *\"* • **
Barra invertida	*Cualquier carácter que es precedido por la barra invertida pierde su significado especial.*

Combinando Distintos Estilos

Se puede combinar cualquiera de las 3 alternativas libremente en el mismo comando. Este ejemplo contiene comillas simples, barra invertida y comillas dobles:

```
echo La variable '$USER' contiene este valor \> "|$USER|"
```

La salida del comando si el usuario es ltsp es el contenido de $USER:

La variable $USER contiene este valor > |ltsp|

Resumen

En este capítulo se vieron 3 forma de quoting y como usarlas:

- Barra invertida
- Comillas simples
- Comillas dobles

10.Control de Flujo

El orden en que los comandos son ejecutados en un script de shell se llama flujo del script. En los scripts que que se vieron hasta ahora, el flujo fue siempre el mismo porque el mismo conjunto de comandos siempre se ejecutaba en el mismo orden.

En la mayoría de los scripts, se necesita cambiar los comandos que se ejecutan dependiendo de cierta condición provista por el usuario o detectada por el script. Cuando se cambia el comando que se ejecuta basado en alguna condición, se está cambiando el flujo del script. Por este motivo los comandos discutidos en este capítulo se denominan **comandos de control de flujo**.

Los mecanismos disponibles en el shell son:

- La sentrencia if
- La sentencia case

La Sentencia *if*

La sentencia if ejecuta acciones dependiendo de si determinada condición es verdadera o falsa. Dado que el código de retorno de un comando es true (0) o false (!0), uno de los usos más comunes de la sentencia if es para chequear errores.

La sintaxis básica de una sentencia if es:

```
if list1
then
        list2
elif list3
then
        list4
else
        list5
fi
```

Tanto la sentencia elif como else son opcionales. Si se tiene una sentencia elif, no se necesita una sentencia else y viceversa. Una sentencia if puede ser escrita con cualquier número de sentencias elif.

El flujo de control general de la sentencia if es el siguiente:

1. se evalúa list1.
2. Si el código de salida de list1 es 0, indica una condición true, list2 es evaluado y la sentencia if termina.
3. En caso contrario, se ejecuta list3 y se chequea su código de salida.
4. Si list3 retorna 0,se ejecuta list4 y la sentencia if termina.

5. Si list3 no retorna 0, se ejecuta list5.

Tests de Archivos

Las expresiones de test de archivo verifican si un archivo cumple determinado criterio. La sintaxis general para testeo de archivos es

test opción archivo

o

[opción archivo] Importante: los espacios después y antes de []!!!!

En este caso opción es alguno de los mostrados en la Tabla y archivo es el nombre del archivo o directorio.

Opciones para Testing de Archivos con el Comando test

Opción	Descripción
-b archivo	Verdadero si archivo existe y es un archivo especial de bloques.
-c archivo	Verdadero si archivo existe y es un archivo especial de caracteres.
-e archivo	Verdadero si archivo existe.
-f archivo	Verdadero si archivo existe y es un archivo regular.
-g archivo	Verdadero si archivo existe y tiene el SGID bit seteado.
-h archivo	Verdadero si archivo existe y es un link simbólico.
-k archivo	Verdadero si archivo existe y tiene el "sticky" bit seteado.
-r archivo	Verdadero si archivo existe y es legible.

Los siguientes ejemplos muestran algunos de los usos de la sentencia if usanso el comando test sobre archivos.

Considerar la siguiente sentencia if:

```
$ if [ -d /home/stfp/bin ] ; then PATH="$PATH:/home/stfp/bin" ; fi
```

En este ejemplo se verifica si el directorio /home/stfp/bin existe o no. Si es verdadero, agrega el directorio a la variable PATH. Sentencias de este tipo se encuentran generalmente en los scripts de inicialización del shell como el .profile.

Comparación de Strings

El comando test también soporta comparaciones simples de de strings. Exsiten 2 formas principales:

1. Verificar cuando un string es vacío
2. Verificar cuando 2 strings son iguales

Un string no puede ser comparado con una expresión usando el comando test. La sentencia case, revisada más adelante debe utilizarse en estos casos.

Opciones del comando test para Comparación de Strings

Opción	Descripción
-z string	Verdadero si string tiene longitud 0.
-n string	Verdadero si string tiene longitud distinta de 0.
string1 = string2	Verdadero si los strings son iguales.
string1 != string2	Verdadero si los strings no son iguales.

Verificando si un String es Vacio

La sintaxis de la primer forma es

test opción string

or

[opción string]

En este caso opción es - z o - n y string es cualquier string válido para el shell. La opción - z (zero) verifica si la logitud de un string es 0, mientras la opción - n (nonzero) se usa para verificar si la longitud de un string es distinto de 0.
Por ejemplo, el siguiente ejemplo

```
if [ -z "$AUTOS" ] ; then
        echo "No tenemos ningún auto" ;
else
        echo "Tenemos los siguientes autos: $AUTOS"
fi
```

genera el string

No tenemos ningún auto

Si el string contenido $$AUTOS tiene longitud 0.

El siguiente es un ejemplo sencillo de comparación de string:

```
if [ "$AUTO" = ferrari ] ; then
        echo "La ferrari de Carlo."
else
        echo "Este auto $AUTO no es la ferrari de Carlo."
fi
```

Si el operador != se en lugar de =, test devuelve verdadero si los strings son distintos. Usando el operador !=, se pude reescribir el comando anterior como:

```
if [ "$AUTO" != ferrari ] ; then
        echo "El auto $AUTO no es la ferrari de Carlo."
else
        echo "La ferrari de Carlo."
fi
```

Comparaciones Numéricas

El comando test permite compara números enteros. La sintaxis básica es

```
test int1 operador int2
```

o

```
[ int1 operador int2 ]
```

Una de las tareas más comunes en un script es ejecutar programas y verificar el estado de retorno. Usando los operadores de comparación numérica, se puede verificar el estado de salida de un comando y tomar diferentes acciones cuando un comando es exitoso y cuando no lo es.

Por ejemplo:

```
ln -s /home/lstp/bin/bash /usr/contrib/bin
```

Si se ejecuta el comando en la línea de comandos, se puede ver un mensaje de error e intervenir para corregir el problema. En un script, el mensaje de error es ignorado y el script continúa la ejecución. Por este motivo, es necesario verificar si un programa termino en forma exitosa.

Como se vio con el comando test, la salida con 0 es exitosa, mientras valores distintos de 0 indican algún tipo de error. El estado de terminación del último comando se guarda en la variable $?, entonces se puede verificar si un comando terminó exitosamente haciendo:

```
if [ $? -eq 0 ] ; then
        echo "Comando exitoso." ;
else
        echo "Se encontró un error."
        exit
fi
```

Si el comando termina con el código 0, se emite un mensaje de "éxito"; en caso contrario, se emite un mensaje de error y se termina. Esto puede simplificarse de la siguiente manera:

```
if [ $? -ne 0 ] ; then
        echo "Se encontrón un error."
        exit
fi
echo "El comando fue exitoso."
```

En este ejemplo se verifica si el comando falló o no.

Operadores para Comparaciones Aritméticas con el Comando test

Operador	Descripción
int1 -eq int2	Verdadero si int1 es igual aqual int2.
int1 -ne int2	Verdadero si int1 no es igual a int2.
int1 -lt int2	Verdadero si int1 es menor que int2.
int1 -le int2	Verdadero si int1 es menor o igual a int2.
int1 -gt int2	Verdadero si int1 es mayor que int2.
int1 -ge int2	Verdadero si int1 es o igual a int2.

Expresiones Compuestas

Hasta ahora se vieron expresiones individuales, sin embargo muchas veces se necesita combinar expresiones. Cuando una o más expresiones se combinan, el resultado se llama expresión compuesta.

Se pueden crear expresiones compuestas usando operadores del comando test, o se pueden utilizar los operadores de ejecución condicional, && y ||. También se puede crear una expresión compuesta que es la negación de otra expresión usando el operador !.

Operadores para Crear Expresiones Complejas

Operador	Descripción
! expr	Verdadero si expr es falso. La expresión expr puede se cualquier comando test válido.
expr1 -a expr2	Verdadero si expr1 y expr2 son verdaderos.
expr1 -o expr2	Verdadero si alguno expr1 o expr2 son verdaderos.

Usando los operadores de de test – La sintaxis para crear expresiones compuestas es

```
test expr1 operador expr2
```

o

```
[ expr1 operador expr2 ]
```

En este ejemplo expr1 y expr2 son cualquier expresión test válida, y operador es cualquiera de -a (and) or -o (or).

Usando operadores condicionales, la sintaxis para crear expresiones compuestas es

```
test expr1 operador test expr1
```

or

```
[ expr1 ] operador [ expr2 ]
```

En este ejemplo expr1 y expr2 son cualquier expresión test válida, y operator es alguno de && (and) o || (or).

Dos ejemplos equivalentes de expresiones compuestas que demuestran como crear una expresión compuesta:

```
if [ -z "$DTHOME" ] && [ -d /usr/dt ] ; then DTHOME=/usr/dt ; fi
if [ -z "$DTHOME" -a -d /usr/dt ] ; then DTHOME=/usr/dt ; fi
```

Negando una expresión

El último tipo de expresión compuesta es la expresión de negación. La negación invierte el resultado de una expresión. Expresiones true son tratadas como expresiones false y vice versa. La sintaxis básica del operador de negación es

```
test ! expr
```

o

```
[ ! expr ]
```

En este caso expr es cualquier expresión test válida. Un ejemplo es el siguiente comando:

```
$ if [ ! -d $HOME/bin ] ; then mkdir $HOME/bin ; fi
```

La Sentencia case

La sentencia case es otra de las formas de control de flujo disponibles en el shell. En esta sección se describe su uso.

La sintaxis básica es

```
case word in
        pattern1)
```

```
            list1
            ;;
        pattern2)
            list2
            ;;
esac
```

En este caso el string word es comparado con cada pattern hasta que uno que matchea se encuentra. La lista de comandos list a continuación del pattern que matchea es ejecutada. Si ninguno concuerda, la sentencia case termina sin ejecutar ninguna acción. No hay límites en el número máximo de patterns, pero debe haber al menos uno.

Cuando una lista de comandos se ejecuta, el comando ;; indica que el flujo del programa debe saltar hasta el final de la sentencia case.

Ejemplo de una Sentencia case

Considerar la siguiente declaración de variable y sentencia case:

```
AUTO=ferrari
case "$AUTO" in
        audi) echo "El audi está muy bien." ;;
        bmw) echo "El bmw es bastante caro." ;;
        ferrari) echo "Carlo es famoso por su ferrari." ;;
esac
```

Usando Patterns

En ejemplo previo, se usaron strings fijos como pattern. Si se utiliza de esta forma la sentencia case se convierte en una sentencia if glorificada. Por ejemplo, la sentencia if

```
if [ "$AUTO" = audi ] ; then
        echo "A Carlo le gusta el audi."
elif [ "$AUTO" = bmw ] ; then
        echo "A carlo le gusta el bmw."
elif [ "$FRUIT" = ferrari ] ; then
        echo "Carlo es famoso por su ferrari."
fi
```

es más verborragico, pero el pode real de la sentencia case no recide en simplificar sentencias if. El poder reside en el hecho de que utiliza patterns para realizar el la comparación.

Un pattern es un string que consiste de caracteres regulares y wildcards especiales . EL pattern determina cuando existe un matcheo.

Los patterns pueden utilizar los mismo caracteres especiales que los utilizados para expandir nombres de archivos más el operador “or”, |. Algunas acciones por

defecto pueden realizarse dando el pattern *, que matchea con cualquier cosa.

Un ejemplo de sentencia case que utiliza patterns es

```
case "$TERM" in
        *term)
                TERM=xterm ;;
        network|dialup|unknown|vt[0-9][0-9][0-9])
                TERM=vt100 ;;
esac
```

En este caso el string contenido en $TERM se compara contra 2 patterns. Si el string termina en term, a $TERM se le assigna el valor xterm. De otra forma, se compara $TERM contra los strings network, dialup, unknown, y vtXXX, donde XXX algún número de 3 dígitos, como 102. Si alguno de estos strings matchea, se setea $TERM con vt100.

Resumen

En este capítulo, se cubrieron dos mecanismos de flujos de control principales disponibles en el shell. Se vieron los siguientes temas relacionados a la sentencia if:

- Tests de archivos
- Comparaciones de strings
- Comparaciones numéricas
- Uso de expresiones compuestas

Además se vio el uso de la sentencia case y el uso en combinación con patterns.

11.Loops

En este capítulo se verá como se utilizan los loops (bucles) en un script. Los loops permiten ejecutar una serie de comandos repetidas veces. Existen 3 tipos principales de loops:

- El loop while
- El loop for
- El loop select

El loop while permite ejecutar un conjunto de comandos en forma repetida hasta que cierta condición ocurra. Generalmente se usa cuando se necesita manipular el valor de una variable en forma repetida.

El loop for permite ejecutar un conjunto de comandos en forma repetida por cada ítem en una lista. Uno de los usos comunes es ejecutar el mismo conjunto de comandos para un número grande de archivos.

El loop select provee una forma fácil de crear un menú numerado del cual los usuarios pueden seleccionar una acción. Es útil cuando se necesita preguntar al usuario que elija uno o más ítems de una lista de opciones.

El Loop while

La sintaxis del loop while es

```
while command
do
        list
done
```

command es un comando simple a ejecutar, mientras que list es un conjunto de uno o más comandos a ejecutar. Si bien command puede ser cualquier comando Linux válido,generalmente es utilizada una expresión test.

La ejecución de un loop while procede de acuerdo a los siguientes pasos:

1. Ejecutar command.
2.1. Si el código de salida de command es distinto de 0, termina el loop.
2.2. Si el código de salida de command es 0, ejecuta list.
3. Cuando list termina de ejecutar, vuelve al Paso 1.

Si command y list son cortos, el loop se suele escribir en una sola línea:

```
while command ; do list ; done
```

El siguiente es un ejemplo de while que imprime los número de 0 a 9:

```
x=0

while [ $x -lt 10 ]
do
        echo $x
        x=$((x + 1))
done
```

La salida debería ser:

```
0
1
2
3
4
5
6
7
8
9
```

Cada vez que se ejecuta el loopp, se verifica la variable x para ver si es menor que 10. Si se cumple la condición se imprime el valor actual de x y se incrementa en 1.

Whiles Anidados

Es posible usar un while en el cuerpo de otro de la siguiente manera:

```
while command1 ; # this is loop1, the outer loop
do
        list1
        while command2 ; # this is loop2, the inner loop
        do
                list2
        done
l       list3
done
```

No existen restricciones respecto a la profundidad de los anidamientos, sin embargo es conveniente evitar loops anidados de más de 3 o 4 niveles. Esto hace difícil la comprensión del script y encontrar errores.

El siguiente ejemplo agrega un loop anidado al ejemplo anterior:

```
x=0
while [ "$x" -lt 10 ] ; # loop1
do
        y="$x"
        while [ "$y" -ge 0 ] ; # loop2
        do
```

```
                echo -n "$y"
                y=$(($y - 1))
        done
        echo
        x=$(($x + 1))
done
```

Que produce la siguiente salida

```
0
1 0
2 1 0
3 2 1 0
4 3 2 1 0
5 4 3 2 1 0
6 5 4 3 2 1 0
7 6 5 4 3 2 1 0
8 7 6 5 4 3 2 1 0
9 8 7 6 5 4 3 2 1 0
```

Ejemplo: Validación de la Entrada del Usuario

Suponemos que necesitamos un script que pida que se ingrese el nombre de un directorio. El script continúa hasta que se ingresa un nombre de directorio válido. Esto se resume en la siguiente secuencia de pasos:

1. Preguntar al usuario el nombre del directorio.
2. Leer la respuesta del usuario.
3. Verificar que el usuario ingresó un directorio válido.
4. *Validar la respuesta.*
5. Si la respuesta es inválida setear la variable a null. Esto permite que el while continúe.
6. Si la respuesta es válida, el valor de la variable mantiene la respuesta del usuario y al no ser null termina.

```
RESPONSE=
while [ -z "$RESPONSE" ] ;
do
        echo -n "Ingrese el path donde se encuentran sus archivos: "
        read RESPONSE
        if [ ! -d "$RESPONSE" ] ; then
                echo "ERROR: Ingresar un directorio válido."
                RESPONSE=
        fi
done
```