Bash con ejemplos, parte 1

Introducción

Te puedes llegar a preguntar por qué deberías aprender programación en Bash. Bueno, aquí hay un par de razones que te presionan para hacerlo:

Ya la estás corriendo

Si lo chequeas, probablemente encontrarás que ya estás corriendo bash en este momento. Aún si has cambiado tu intérprete de comandos ("shell") por defecto, bash está probablemente corriendo aún en algún lugar de tu sistema, porque es el intérprete de comandos de Linux estándar y es usada para una gran variedad de propósitos. Porque bash ya está corriendo, cualquier número de guiones adicionales que corras serán intrínsecamente eficientes en cuanto al uso de memoria porque la comparten con algún proceso de bash que ya se encuentra corriendo. ¿Por qué cargar un intérprete de 500K si ya estás corriendo algo que puede hacer el trabajo, y hacerlo bien?

Ya la estás usando

No solo ya estás corriendo bash, sino que además estás interactuando con bash diariamente. Siempre está allí, así que tiene sentido aprender cómo usarla en su máximo potencial. Hacerlo hará tu experiencia con bash más divertida y productiva. Pero... ¿Por qué deberías aprender programación en bash? Fácil, porque ya piensas en términos de comandos, copiando archivos, y usando las tuberías y redireccionando salidas. ¿No deberías aprender un lenguaje que te permita trabajar y construir a partir de estas poderosas herramientas que ya sabes utilizar? Los intérpretes de comandos dan libertad al potencial de los sistemas UNIX, y bash es el intérprete de comandos de Linux. Es el "pegamento" de alto nivel entre tú y la máquina. Crece en tu conocimiento sobre bash y automáticamente incrementarás tu productividad bajo Linux y UNIX -- es así de simple.

Confusión con bash

Aprender bash del modo equivocado puede ser un proceso muy confuso. Muchos usuarios novatos escriben man bash para ver la página del manual de bash ("man page"), solo para ser confrontados con una muy concisa y técnica descripción de la funcionalidad del intérprete de comandos. Otros intentan con info bash (para ver la documentación que provee GNU info), causando que la página del manual sea reimpresa o, si tienen suerte, verán a lo sumo una página de documentación escasamente más amigable.

Aunque esto puede ser algo entristecedor para los novatos, la documentación estándar de bash no puede ser todas las cosas para toda la gente, y se orienta hacia aquellos ya familiarizados con la programación del intérprete de comandos en general. Hay definitivamente un montón de información técnica excelente en la página del manual ("man page"), pero su utilidad para los principiantes es limitada.

Allí es donde esta serie entra en el juego. En ella, te mostraré cómo usar las construcciones de bash en realidad, para que te encuentres preparado para escribir tus propios guiones. En vez de descripciones técnicas, te proveeré explicaciones en tu idioma, para que sepas no solo qué es lo que algo hace, sino además cuándo deberías

usarlo. Hacia el final esta serie de tres partes, serás capaz de escribir tus propios guiones complejos para bash, y de estar al nivel en el que podrás usar bash confortablemente y aumentar tus conocimientos leyendo (y entendiendo!) la documentación estándar de bash. Comencemos.

Variables de entorno

Bajo bash y bajo casi cualquier intérprete de comandos, el usuario puede definir variables de entorno, que son guardadas internamente como cadenas de caracteres ASCII. Una de las cosas más prácticas acerca de las variables de entorno es que son una parte estándar del modelo de proceso de UNIX. Esto significa que las variables de entorno no son exclusivas de los guiones del intérprete de comandos, sino que también pueden ser usadas por programas compilados de manera estándar. Cuando "exportamos" una variable de entorno bajo bash, cualquier programa subsecuente que corramos podrá leer lo que le asignamos, sea un guión del intérprete de comandos o no. Un buen ejemplo es el comando vipw, que normalmente permite al superusuario root editar el archivo con la clave ("password") del sistema. Ajustando la variable de entorno EDITOR con el nombre de tu editor de texto favorito, puedes configurar a vipw para que lo use en lugar de vi, algo bastante práctico si estás acostumbrado a xemacs y realmente no te gusta vi.

La manera estándar de definir una variable de entorno bajo bash es:

Listado de Código 1.1: Definir una variable de entorno

```
$ myvar='This is my environment variable!'
```

El comando de arriba definió una variable de entorno llamada "myvar" que contiene la cadena "This is my environment variable!". Hay algunas cosas a las que es necesario prestarle atención en lo anterior: primero, no hay ningún espacio rodeando al signo "="; cualquier espacio allí resultaría en un error (pruébalo y confírmalo). La segunda cosa a tener en cuenta es que, aunque pudimos haber omitido las comillas si se tratase de una sola palabra, son necesarias cuando el valor de la variable de entorno es más de una palabra (contiene espacios o tabs).

Nota: Para información extremadamente detallada sobre cómo deben ser usadas las comillas en bash, es probable que la sección "QUOTING" de la "man page" de bash te resulte útil. La existencia de secuencias especiales de caracteres que son "expandidas" (reemplazadas) por otros valores complica el modo en que las cadenas son manejadas en bash. Solo cubriremos las funciones más usadas/importantes de las comillas en esta serie.

En tercer lugar, mientras que normalmente podemos usar comillas dobles en vez de comillas simples, hacerlo en el ejemplo anterior hubiera causado un error. ¿Por qué? Porque el usar comillas simples desactiva una de las características de bash llamada "expansión", donde caracteres y secuencias de caracteres especiales son reemplazados por valores. Por ejemplo, el carácter "!" es el carácter de expansión del historial, que bash normalmente reemplaza por un comando previamente escrito. (No cubriremos la expansión del historial en esta serie de artículos, porque no es usada frecuentemente en la programación en bash. Para más información sobre eso, mira la sección "HISTORY EXPANSION" en la página del manual ("man page") de bash.) Aunque este comportamiento al estilo "macro" puede ser muy práctico, en esta ocasión queremos un signo de exclamación literal al final del valor de nuestra variable de entorno, en vez de un "macro".

Ahora, echémosle una mirada a cómo uno usa en realidad una variable de entorno. Aquí hay un ejemplo:

Listado de Código 1.2: Usar variables de entorno

```
$ echo $myvar
This is my environment variable!
```

Precediendo el nombre de nuestra variable de entorno con un $, podemos hacer que bash la reemplace por el valor de myvar. En la terminología de bash, esto se llama "expansión de variable" ("variable expansion"). Pero, si probamos lo siguiente:

Listado de Código 1.3: Primer intento de uso de expansión de una variable

```
$ echo foo$myvarbar
foo
```

Queríamos que esto imprima "fooThis is my environment variable!bar", pero no funcionó. ¿Cuál fue el error? En pocas palabras, la facilidad de expansión de variable de bash se encontró confundida. No pudo dilucidar si queríamos expandir la variable $m, $my, $myvar, $myvarbar, etc. ¿Cómo podemos ser más explícitos y claramente decirle a bash a qué variables nos estamos refiriendo? Intenta esto:

Listado de Código 1.4: Expansión de variable; segundo intento.

```
$ echo foo${myvar}bar
fooThis is my environment variable!bar
```

Como puedes ver, podemos encuadrar el nombre de nuestra variable de entorno entre llaves cuando no se encuentra claramente separada del texto que la rodea. Mientras que $myvar es más rápido de escribir y funcionará en la mayoría de las ocasiones, ${myvar} puede ser comprendida correctamente en casi cualquier situación. Más allá de eso, ambas hacen lo mismo, y verás ambas formas de expansión de variable en el resto de la serie. Querrás recordar que tienes que usar la forma más explícita (con las llaves) cuando tu variable de entorno no se encuentre aislada del texto que la rodea mediante algún espacio en blanco (espacio o tabs).

Recuerda que también mencionamos que podemos "exportar" variables. Cuando exportamos una variable de entorno, esta se encuentra automáticamente disponible en el entorno de cualquier guión o ejecutable que se corra subsecuentemente. Los guiones del intérprete de comandos pueden acceder a la variable de entorno usando el soporte que brinda bash intrínsecamente, mientras que los programas en C pueden usar la función getenv(). Aquí hay un código en C de ejemplo que deberías escribir y compilar -- nos permitirá entender las variables de entorno desde la perspectiva de C:

Listado de Código 1.5: myvar.c -- un programa en C de ejemplo para las variables de entorno

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  char *myenvvar=getenv("EDITOR");
  printf("The editor environment variable is set to %s\n",myenvvar);
}
```

Guarda el código anterior en un archivo llamado myenv.c, y luego compílalo usando el

siguiente comando:

Listado de Código 1.6: Compilar el código de arriba

```
$ gcc myenv.c -o myenv
```

Ahora habrá un programa ejecutable en tu directorio que, al ser corrido, imprimirá el valor de la variable de entorno EDITOR, si es que lo tiene. Esto es lo que pasa cuando lo corro en mi máquina:

Listado de Código 1.7: Correr el programa de arriba

```
$ ./myenv
The editor environment variable is set to (null)
```

Hmmm... Como la variable de entorno EDITOR no poseía ningún valor, el programa en C recibe una cadena nula. Intentemos nuevamente dándole un valor específico:

Listado de Código 1.8: Probar con un valor específico

```
$ EDITOR=xemacs
$ ./myenv
The editor environment variable is set to (null)
```

Aunque pudiste haber esperado que myenv imprima el valor "xemacs", no funcionó, porque no exportamos la variable de entorno EDITOR. Esta vez lo haremos funcionar:

Listado de Código 1.9: Mismo programa luego de exportar la variable

```
$ export EDITOR
$ ./myenv
The editor environment variable is set to xemacs
```

Así, has visto con tus propios ojos que otro proceso (en este caso nuestro programa en C) no puede ver la variable de entorno hasta que esta es exportada. Incidentalmente, si tú quieres, puedes definir y exportar una variable de entorno usando una sola línea, del siguiente modo:

Listado de Código 1.10: Definir y exportar una variable de entorno en un solo comando

```
$ export EDITOR=xemacs
```

Funciona idénticamente a la versión de dos líneas. Este sería un buen momento para enseñar cómo borrar una variable de entorno usando unset:

Listado de Código 1.11: Borrar una variable de entorno

```
$ unset EDITOR
$ ./myenv
The editor environment variable is set to (null)
```

Vistazo a los cortes de cadenas

Cortar cadenas -- esto es, separar una cadena original en más pequeños y separados trozos -- es una de esas tareas que es llevada a cabo diariamente por tu guión del intérprete de comandos tipo. Muchas veces, los guiones del intérprete de comandos necesitan tomar una ruta completa, y encontrar el archivo o directorio final. Aunque es

posible (y divertido!) programar esto en bash, el ejecutable estándar de UNIX basename hace esto extremadamente bien:

Listado de Código 1.12: Usar basename

```
$ basename /usr/local/share/doc/foo/foo.txt
foo.txt
$ basename /usr/home/drobbins
drobbins
```

basename es una muy práctica herramienta para cortar cadenas. Su acompañante, llamado dirname, devuelve la "otra" parte de la ruta que basename desecha:

Listado de Código 1.13: Usar dirname

```
$ dirname /usr/local/share/doc/foo/foo.txt
/usr/local/share/doc/foo
$ dirname /usr/home/drobbins/
/usr/home
```

Nota: Ambos dirname y basename no miran ningún archivo ni directorio en el disco; son puramente comandos de manipulación de cadenas.

Sustitución de comandos

Una cosa bastante práctica de saber es cómo crear una variable de entorno que contenga el resultado de un comando ejecutable. Esto es muy fácil de hacer:

Listado de Código 1.14: Crear una variable de entorno con el resultado de un comando

```
$ MYDIR=`dirname /usr/local/share/doc/foo/foo.txt`
$ echo $MYDIR
/usr/local/share/doc/foo
```

Lo que hicimos arriba se llama sustitución de comando ("command substitution"). Varias cosas merecen ser notadas en este ejemplo. En la primera línea, simplemente encuadramos el comando que queríamos ejecutar entre esas comillas especiales. Esas comillas no son las comillas simples estándar, sino que son las que se imprimen al oprimir una de las teclas del teclado que normalmente se encuentra por encima de la tecla Tab. Podemos hacer exactamente lo mismo con la sintaxis alternativa para la sustitución de comandos de bash:

Listado de Código 1.15: Sintaxis alternativa para la sustitución de comandos

```
$ MYDIR=$(dirname /usr/local/share/doc/foo/foo.txt)
$ echo $MYDIR
/usr/local/share/doc/foo
```

Como puedes ver, bash provee múltiples maneras de realizar exactamente la misma cosa. Usando la sustitución de comandos, podemos ubicar cualquier comando o tuberías con comandos entre ` ` o $( ) y asignar su resultado a una variable de entorno. ¡Qué cosa práctica! Aquí hay un ejemplo de cómo usar tuberías con la sustitución de comandos:

Listado de Código 1.16: Tuberías y sustitución de comandos

```
$ MYFILES=$(ls /etc | grep pa)
```

```
$ echo $MYFILES
pam.d passwd
```

Cortando cadenas como un profesional

Si bien basename y dirname son herramientas grandiosas, hay momentos en los que podemos necesitar realizar operaciones de corte de cadenas de una manera más avanzada que solo manipulaciones estándar con las rutas. En esos casos, podemos aprovechar la característica intrínseca avanzada de bash de expansión de variable. Ya hemos usado la manera estándar de expansión de variable, que se ve como esto: ${MYVAR}. Pero bash puede también realizar cortes de cadenas prácticos por sí mismo. Échale una mirada a estos ejemplos:

Listado de Código 1.17: Ejemplos de corte de cadenas

```
$ MYVAR=foodforthought.jpg
$ echo ${MYVAR##*fo}
rthought.jpg
$ echo ${MYVAR#*fo}
odforthought.jpg
```

En el primer ejemplo, escribimos ${MYVAR##*fo}. ¿Qué significa esto exactamente? Básicamente, dentro de ${ }, escribimos el nombre de la variable de entorno, dos ##s, y una "wildcard" (comodín, representación de caracteres o cadenas de caracteres no explicitados), "*fo". Luego, bash tomó MYVAR, encontró la más larga sub-cadena desde el comienzo de la cadena "foodforthought.jpg" que encajaba con la "wildcard" "*fo", y realizó el corte desde el comienzo de la cadena principal. Esto es un poco difícil de asimilar a la primera, así que para entender cómo funciona esta opción especial de "##", sigamos los pasos que realizó bash para completar esta expansión. Primero, comenzó buscando las sub-cadenas desde el principio de "foodforthought.jpg" que encajaran con la "wildcard" "*fo". Aquí están las sub-cadenas que chequeó:

Listado de Código 1.18: Sub-cadenas siendo chequeadas

```
f
fo              MATCHES *fo
foo
food
foodf
foodfo          MATCHES *fo
foodfor
foodfort
foodforth
foodfortho
foodforthou
foodforthoug
foodforthought
foodforthought.j
foodforthought.jp
foodforthought.jpg
```

Luego de buscar sub-cadenas que encajaran (puedes ver que bash encontró dos) seleccionó la más larga, la eliminó del inicio de la cadena original, y devolvió el resultado.

La segunda forma de expansión de variable mostrada arriba es casi idéntica a la primera, excepto que usa solo un "#" -- y bash realiza un proceso muy muy similar. Chequea las mismas sub-cadenas que chequeó en nuestro primer ejemplo, solo que bash ahora quita la más corta de nuestra cadena original, y devuelve el resultado.

Entonces, tan pronto como determina que la subcadena "fo" es lo que buscaba, elimina "fo" de nuestra cadena y devuelve "odforthought.jpg".

Esto puede parecer extremadamente críptico, así que te mostraré una manera fácil de recordar estas herramientas. Cuando buscamos la sub-cadena más larga, usamos ## (porque ## es más largo que #). Cuando buscamos la más corta, usamos #. ¿Ves? ¡No es tan difícil de recordar! Espera. ¿Cómo recordamos que debemos usar el carácter '#' para eliminar desde el "principio" de una cadena? ¡Simple! Notarás que en un teclado americano, shift-4 es "$", que es el carácter de expansión de variable de bash. En el teclado, inmediatamente a la izquierda de "$" está "#". Entonces, puedes ver que "#" está "al principio" de "$", y así (de acuerdo a nuestra regla mnemotécnica), "#" elimina caracteres desde el principio de la cadena. Te puedes llegar a preguntar cómo eliminamos caracteres ubicados al final de la cadena. Si adivinaste que usamos el carácter inmediatamente a la derecha de "$" en el teclado americano ("%"), estás en lo cierto! Aquí hay algunos ejemplos rápidos sobre cómo cortar porciones finales de cadenas:

Listado de Código 1.19:

```
$ MYFOO="chickensoup.tar.gz"
$ echo ${MYFOO%%.*}
chickensoup
$ echo ${MYFOO%.*}
chickensoup.tar
```

Como puedes ver, las opciones de expansión de variable % y %% funcionan del mismo modo que # y ##, excepto que eliminan la "wildcard" del final de la cadena. Nota que no estás obligado a usar el carácter "*" si quieres eliminar una sub-cadena específica del final:

Listado de Código 1.20: Cortar sub-cadenas del final

```
MYFOOD="chickensoup"
$ echo ${MYFOOD%%soup}
chicken
```

En este ejemplo, no importa si usas "%%" o "%", ya que solo una sub-cadena puede encajar. Y recuerda, si te olvidas si debes usar "#" o "%", mira las teclas 3, 4, y 5 en tu teclado y te darás cuenta.

Podemos usar otra forma de expansión de variable para seleccionar una sub-cadena específica, basándonos en un punto de inicio y una longitud. Intenta escribir las siguientes líneas en bash:

Listado de Código 1.21: Seleccionar una sub-cadena específica

```
$ EXCLAIM=cowabunga
$ echo ${EXCLAIM:0:3}
cow
$ echo ${EXCLAIM:3:7}
abunga
```

Esta forma de corte de cadena puede sernos muy útil; simplemente especifica el carácter a partir del cual iniciar y la longitud de la sub-cadena, todo separado por dos puntos.

Poniendo en práctica el corte de cadenas

Ahora que hemos aprendido todo acerca del corte de cadenas, escribamos un pequeño y simple guión del intérprete de comandos. Nuestro guión aceptará un solo archivo como argumento, e imprimirá si parece ser un "tarball" o no. Para determinar si se trata de un "tarball", se fijará en el patrón ".tar" al final del archivo. Aquí está:

Listado de Código 1.22: mytar.sh -- un guión de ejemplo

```
#!/bin/bash

if [ "${1##*.}" = "tar" ]
then
       echo This appears to be a tarball.
else
       echo At first glance, this does not appear to be a tarball.
fi
```

Para correr este guión, transcríbelo dentro a un archivo llamado mytar.sh, y luego escribe chmod 755 mytar.sh para hacerlo ejecutable. Luego, pruébalo con un "tarball" del siguiente modo:

Listado de Código 1.23: Probar el guión

```
$ ./mytar.sh thisfile.tar
This appears to be a tarball.
$ ./mytar.sh thatfile.gz
At first glance, this does not appear to be a tarball.
```

OK, funciona, pero no es muy funcional. Antes de que lo hagamos más útil, echémosle una mirada a la construcción "if" usada arriba. En ella, tenemos una expresión booleana. En bash, el operador de comparación "=" chequea la igualdad de cadenas. En bash, todas las expresiones booleanas son encuadradas en corchetes. ¿Pero qué es lo que la expresión booleana prueba en realidad? Echémosle una mirada a la parte izquierda. De acuerdo con lo que hemos aprendido sobre corte de cadenas, "${1##*.}" eliminará la sub-cadena más larga que encaje con "*." del principio de nuestra cadena contenida en la variable de entorno "1", devolviendo el resultado. Esto causará que se devuelva todo lo escrito luego del último ".". Obviamente, si el archivo termina en ".tar", tendremos "tar" como resultado, y la condición se cumplirá.

Te puedes estar preguntando qué representa la variable de entorno "1". Muy simple -- $1 es el primer argumento recibido por el guión desde la línea de comandos, $2 es el segundo, etc. OK, ahora que hemos repasado la función, podemos echar una primera mirada a las instrucciones "if".

Instrucciones "if"

Como la mayoría de los lenguajes, bash tiene su propia implementación de condicionales. Cuando la uses, ajústate al formato de arriba; esto es, mantén el "if" y el "then" en líneas separadas, y procura que el "else" y el "fi" final (y requerido) estén alineados horizontalmente con los primeros. Esto hace que el código sea más fácil de entender y posibilita encontrar los errores más rápidamente. Además de la forma de "if,else", hay algunas otras formas de instrucciones "if":

Listado de Código 1.24: Forma básica de la instrucción if

```
if      [ condition ]
```

```
then
        action
fi
```

Esta realiza una acción solo si la condición es verdadera; caso contrario, no realiza ninguna acción y continúa ejecutando cualquier línea luego del "fi".

Listado de Código 1.25: Chequear condiciones antes de seguir después del fi

```
if [ condition ]
then
        action
elif [ condition2 ]
then
        action2
.
.
.
elif [ condition3 ]
then

else
        actionx
fi
```

La forma "elif" de arriba probará consecutivamente cada condición y ejecutará la acción correspondiente a la primera condición verdadera. Si ninguna de las condiciones es verdadera, ejecutará la acción del "else", si la hay, y luego continuará ejecutando las líneas que sigan a la instrucción entera de"if,elif,else".

La próxima vez

Ahora que hemos cubierto lo más básico de bash, es tiempo de poner manos a la obra y escribir algunos guiones reales. En el próximo artículo, cubriré las construcciones de bucles (iteraciones), funciones, "namespace" (ámbito de variables), y otros temas esenciales. Luego, estaremos listos para escribir algunos guiones más complicados. En el tercer artículo, nos enfocaremos casi exclusivamente en guiones y funciones más complejos, como también en varias opciones de diseño de guiones en bash. ¡Nos vemos en el siguiente!

Sumario: Aprendiendo cómo programar en el lenguaje de guiones de bash, tu interacción del "día a día" con Linux se hará más divertida y productiva, y serás capaz de construir a partir de lo que brinda UNIX de manera estándar (como las tuberías y redirecciones) que ya conoces y amas. En esta serie de tres partes, Daniel Robbins te enseñará cómo programar en bash mediante ejemplos. Cubrirá los aspectos absolutamente básicos (haciendo de esta una excelente serie para principiantes) y mostrará características más avanzadas de bash a medida que la serie avance.