ITEDES

Educación Digital

Módulo:

Fundamentos de Ingeniería de Software

Segmento:

Algoritmos y Estructuras de Datos

Tema:

Programación en BASH

Prof. Germán C. Basisty
german.basisty@itedes.com

Índice de Contenidos

Programación Avanzada en BASH

BASH (acrónimo de Bourne Again Shell) es un intérprete de comandos de **UNIX** con características avanzadas de programación que permite crear scripts complejos. La aplicación principal de las capacidades de scripting de **BASH** es sobre tareas de gestión del sistema - tales como automatización de procesos, tratamiento de archivos, etc. - aunque no está limitado solamente a eso.

En este apéndice (que estará constantemente en evolución) se explicarán características avanzadas de scripting en **BASH**.

Estructura básica

La estructura básica de un script es:

```
#!SHEBANG

VARIABLE DE CONFIGURACIÓN

DEFINICIÓN DE FUNCIONES

CÓDIGO PRINCIPAL
```

El SHEBANG

Si es posible use un shebang. Tenga cuidado con **/bin/sh** El argumento de que "*en Linux **/bin/sh** es **Bash***" no siempre es correcto y es técnicamente irrelevante.

El shebang tiene dos propósitos:

- Especifica el intérprete que se utilizará cuando se llame directamente al archivo de script: si codifica Bash, especifique bash.

- Documenta el intérprete deseado (entonces: use bash cuando escriba un guión Bash, use sh cuando escriba un guión Bourne / POSIX general, etc.).

Variables de configuración

Las variables de configuración son las que están destinadas a ser cambiadas por el usuario.
Haga que sean fáciles de encontrar (directamente en la parte superior del script o inmediatamente antes de utilizarlas). Nómbrelas de forma significativa y eventualmente añada un breve comentario.

Definiciones de funciones

A menos que existan razones para no hacerlo, todas las definiciones de funciones se deben declarar antes de que se ejecute el código del script principal. Esto proporciona una visión general mucho mejor y garantiza que todos los nombres de funciones se conozcan antes de ser utilizados.

Dado que una función no se analiza antes de que se ejecute, generalmente no tiene que asegurarse de que estén en un orden específico.

Es una buena práctica escribir las funciones en archivos separados e incluirlos luego en el código principal mediante:

```
. nombre_archivoFunciones
```

Scripteando con estilo

Estas son algunas pautas de codificación que ayudarán a producir código más robusto.

Esto no es una biblia, por supuesto. Pero he visto tanto código feo y terrible (no solo en shell) durante todos los años, que estoy 100% convencido de que debe haber un poco de diseño y estilo al codificar. Un buen diseño de código ayuda a leer su código, haciendo que entre más fácil por los ojos.

Directrices de indentado

La indentación no es nada que técnicamente influya en un script, es solo para nosotros los humanos.

Estoy acostumbrado a ver / usar la indentación de cuatro caracteres de espacio (aunque muchos pueden preferir 2 espacios) configurados directamente contra la tabulación:

- Es fácil y rápido escribir.
- Es lo suficientemente amplio como para dar un salto visual y lo suficientemente pequeño como para no perder demasiado espacio en la línea.

Rompiendo líneas

Siempre que necesite romper líneas de código largo, debe seguir una de estas dos reglas:

Indentación usando el ancho del comando:

```
activar some_very_long_option \
        some_other_option
```

Indentación usando dos espacios:

```
activar some_very_long_option \
  some_other_option
```

Personalmente, con algunas excepciones, prefiero la primera porque apoya la impresión visual de "estas cosas juntas". Para cortar una línea de código, utilizar el caracter “\” al final de la línea en donde se desee aplicar el corte.

Estructuras de control de flujo

Los comandos de control de flujo forman las estructuras de programación que hacen que un shell script sea diferente de una simple enumeración de instrucciones. Por lo general, contienen una especie de "encabezado" y un "cuerpo" que contiene un bloque de código. Este tipo de comando compuesto es relativamente fácil de indentar.

Buenas prácticas de estilo:

- Poner la palabra clave de presentación y la lista de comandos o parámetros iniciales en una línea ("encabezado")
- Poner la palabra clave "introducir cuerpo" en la línea siguiente
- La lista de comandos del "cuerpo" en líneas separadas, correctamente indentadas
- Poner la palabra clave de cierre en una línea separada, con indentado como la palabra clave introductoria inicial.

Ejemplo:

```
HEAD_KEYWORD parámetros
BODY_BEGIN
        BODY_COMMANDS
BODY_END
```

if / then / elif / else

Esta construcción es un poco especial, porque tiene palabras clave (elif, else) en el medio. La forma visualmente atractiva es indentarlas así:

```
if ...
then
        ...
elif …
then
        ...
else
        ...
fi
```

for

```
for ((i = 0; i < 100; i++)) {
        ...
}
```

while / do while / until

```
while ((...))
do
        …
done

while :
do
        …
        ((...)) || break
done

until ((...))
do
        …
done
```

case

La construcción del case podría necesitar un poco más de discusión, ya que su estructura es un poco más compleja.

En general, cada nueva "capa" obtiene un nuevo nivel de sangría:

```
case $var1 in
        Hola)
                echo "dijiste hola"
                ;;
        adiós)
                echo "dijiste adiós"
                ;;
        *)
                echo "Dijiste algo raro ..."
                ;;
esac
```

Algunas notas sobre la estructura *case*:

- Si no es 100% necesario, el paréntesis izquierdo opcional en el patrón no se usa.
- El terminador de acción (;;) se indenta en el mismo nivel que el bloque de código.
- Aunque opcional, se da el último terminador de acción.

Pautas de sintaxis y codificación

Construcciones crípticas

Construcciones crípticas, todos las conocemos, todos las amamos. Si no son 100% necesarios, evítelos, ya que nadie, excepto usted, podrá descifrarlos. Es, al igual que en C, el término medio entre inteligente, eficiente y legible. Si necesita usar un constructo críptico, incluya un comentario que explique lo que hace su "monstruo".

Variables

Nombres de variables

Dado que todas las variables reservadas son MAYÚSCULAS, la forma más segura es usar solo nombres de variables en minúsculas. Esto es cierto para leer entradas de usuario, variables de recuento de bucles, etc.

Los nombre de variables deben escribirse en minúsculas, y si tiene que unir dos palabras, utilice ***camelCase***

Inicialización de variables

Como en C, siempre es una buena idea inicializar las variables, sin embargo, el intérprete de comandos inicializará las variables nuevas (mejor: las variables desvinculadas generalmente se comportarán como variables que contienen una cadena nula).

No es un problema pasar una variable de entorno al script. Si asume ciegamente que todas las variables que usa por primera vez están vacías, cualquiera puede inyectar contenido en una variable pasándola a través del entorno.

La solución es simple y efectiva: declare e inicialice las variables:

```
declare myInput = ""
declare -a myArray=()
declare -i myNumber = 0
```

El comando **declare** si bien en el bloque de programación principal es optativo, ayuda a mejorar la legibilidad del código. En las funciones, utilizar **declare** hace que las variables sean locales (solo en las funciones, ya que las variables declaradas en el bloque de código principal automaticamente las convierte en globales).

Algúnos modificadores útiles de **declare** son:

- **declare -i** convierte el tipo de datos de una variable a entero. Si se ingresan datos alfanuméricos, serán reemplazados por cero.
- **declare -a** especifica que la variable a declarar es un vector. Puede combinarse con **-i** para especificar un vector de enteros.
- **declare -r** sirve para declarar constantes, es análogo al comando **readonly**

Expansión de variables

A menos que esté realmente seguro de lo que está haciendo, entrecomille cada expansión de variables. Hay algunos casos en que esto no es necesario desde un punto de vista técnico, pero entrecomillar nunca es un error.

```
if [[ $variable == “hola” ]]
then
        …
fi
```

es mucho mejor si

```
if [[ “$variable” == “hola” ]]
then
        …
fi
```

Sin embargo hay situaciones en donde entrecomillar es técnicamente incorrecto:

```
list="uno dos tres"

# NO SE DEBE entrecomillar $list aquí
for palabra in $list
do
...
done
```

Funciones

Nombres de funciones

Los nombres de las funciones deben ser en minúscula (o **camelCase** en su defecto) y significativos. Los nombres de las funciones deben ser legibles por humanos. Una función llamada f1 puede ser fácil y rápida de escribir, pero para la depuración y especialmente para otras personas, no revela nada. Los buenos nombres ayudan a documentar su código sin usar comentarios adicionales.

No use nombres de comando para sus funciones. Poner de nombre “vim” a una función colisionará con el comando de UNIX.

A menos que sea absolutamente necesario, solo use caracteres alfanuméricos y el guión bajo para los nombres de las funciones. / bin / ls es un nombre de función válido en Bash, pero no es una buena idea.

Sustitución de comandos

La sustitución de comandos sirve para ejecutar un comando y utilizar como valor de retorno el resultado de esa ejecución. Esta técnica es especialmente útil cuando se desea que una función devuelva un valor. Utilizar la notación **$ (...)**

```
#!/bin/bash

function greet() {
        echo “Hola!!”
}

declare myVariable

myVariable=$(greet)

echo “$myVariable” <= Mostrará Hola!!

exit 0
```

Comportamiento y robustez

Fallar temprano

Fallar temprano, esto suena mal, pero por lo general es bueno. Fallar temprano significa erradicar lo antes posible cuando las comprobaciones indican un error o una condición no cumplida. Fallar temprano significa un error antes de que su script comience su trabajo en un estado potencialmente roto.

Disponibilidad de comandos

Si usa comandos externos que pueden no estar presentes en el PATH, o no están instalados, verifique su disponibilidad y luego diga al usuario que faltan.

Ejemplo

```
declare my_needed_commands="sed awk lsof who"
declare -i missing_counter=0

for needed_command in $my_needed_commands
do
        if ! hash "$needed_command" >/dev/null 2>&1
        then
                echo “Comando no encontrado en el PATH: $needed_command"
                ((missing_counter++))
        fi
done

if ((missing_counter > 0))
then
        echo “Requerimientos del mínimos del sistema insatisfechos. Abortando…”
        exit 1
fi
```

Salir de manera significativa

El código de salida es su única forma de comunicarse directamente con el proceso de llamadas sin ninguna disposición especial. Si su script sale, proporcione un código de salida significativo. Eso mínimamente significa:

- exit 0 (cero) si todo está bien
- exit 1 - en general distinto de cero - si hubo un error

Esto, y solo esto, permitirá que el componente llamante verifique el estado de operación de su script.