
BASH Unidad 7

Manejo de errores y consejos

Tabla de contenidos

Manejo de errores

La diferencia entre un buen programa y uno malo se mide, entre otros aspectos, por su robustez. Es decir, en la habilidad del programa para manejar situaciones en las que algo vaya mal.

En esta práctica vamos a trabajar el manejo de errores durante la ejecución de nuestros scripts.

Estado de salida

Todos los programas —o al menos los que están bien escritos— retornan un estado de salida cuando finalizan su ejecución:

- **= 0**: Si el programa **finaliza con éxito**.
- **≠ 0**: Si el programa **ha fallado de alguna manera**.

Por lo tanto:

- Es muy importante verificar el estado de salida de los comandos ejecutados.
- Es importante que nuestros scripts devuelvan un estado de salida con sentido, siguiendo las reglas indicadas, de manera que informe de lo que ha ocurrido durante la ejecución.

Supongamos que tenemos un script como este:

```
# Ejemplo de una muy mala idea
cd "$directorio_inexistente"
rm *.log
```

¿Por qué es mala idea hacer esto?

- En principio no es mala idea si nada va mal. Estas dos líneas cambian el directorio actual a `$directorio_inexistente` y elimina los archivos ".log" que haya en ese directorio.
- Pero **¿qué ocurre si el directorio `$directorio_inexistente` no existe?** En ese caso el comando **cd** va a fallar y el script ejecutará el comando **rm** en el directorio actual. **Obviamente, este no es el comportamiento esperado**.

Como habrás supuesto, el problema con este script es que no comprueba el estado de salida del comando cd antes de llamar al comando rm.

Comprobando el estado de salida

Hay varias formas en las que se puede obtener el estado de salida de un programa.

Primero se puede examinar el contenido de la variable de entorno $?. Esta variable contiene el estado de salida del último comando ejecutado:

```
yo@mihost:~$ true; echo $?
0
yo@mihost:~$ false; echo $?
1
```

Los comandos **true** y **false** son programas que no hacen otra cosa que devolver un estado de salida de 0 y de 1 respectivamente. Usándolos podemos ver un ejemplo de cómo la variable $? contiene el estado de salida del programa ejecutado previamente.

Por lo tanto, podríamos reescribir el script anterior de la siguiente manera:

```
cd "$directorio_inexistente"
if [ "$?" = "0" ]; then
    rm *.log
else
    echo "¡No puedo cambiar al directorio!" >&2
    exit 1
fi
```

En esta versión examinamos el estado de salida del comando y si no es 0, imprimimos un mensaje de error en la salida de error y terminamos el script con código de salida 1.

Mientras que la anterior solución funciona, hay mecanismos más sofisticados que pueden ahorrarnos tiempo. La siguiente aproximación incluye el comando que queremos comprobar dentro del propio **if**, ya que este evalúa el estado de salida del comando.

```
if cd "$directorio_inexistente"; then
    rm *.log
else
    echo "¡No puedo cambiar al directorio!" >&2
    exit 1
fi
```

Función de salida con error

Como vamos a verificar errores con frecuencia en nuestros programas, tiene sentido escribir una función que muestre mensajes de error.

```
# Una función de salida con error

error_exit()
{
     echo "$1" >&2
     exit 1
}

# Usando error_exit
if cd "$directorio_inexistente"; then
    rm *.log
else
    error_exit "¡No puedo cambiar al directorio! Abortando..."
fi
```

Listas AND y OR

Podemos simplificar aún más nuestro script haciendo uso de las listas AND y OR, denotadas por los operadores "&&" y "||", respectivamente.

Una lista **AND** tiene la forma:

```
comando1 && comando2
```

El comando2 se ejecuta si y solo si el comando1 devuelve un valor de retorno de 0.

Una lista **OR** tiene la forma:

```
comando1 || comando2
```

El comando2 se ejecuta si y solo si el comando1 devuelve un valor de retorno distinto de 0.

El valor de salida de las listas AND y OR es el valor de salida del último comando ejecutado en la lista. Podemos usar los comandos **true** y **false** para ver cómo funciona esto:

```
yo@mihost:~$ true || echo "echo ejecutado"
yo@mihost:~$ false || echo "echo ejecutado"
echo ejecutado
yo@mihost:~$ true && echo "echo ejecutado"
echo ejecutado
yo@mihost:~$ false && echo "echo ejecutado"
yo@mihost:~$
```

Usando esta técnica podemos escribir una versión incluso más simple de nuestro script:

```
cd "$directorio_inexistente" ||
    error_exit "¡No puedo cambiar al directorio!"
rm *.log
```

Si no se requiere salir del script en caso de error, se puede hacer lo siguiente:

```
cd "$directorio_inexistente" && rm *.log
```

🚨Es importante aclarar que incluso con la defensa contra errores que hemos introducido en nuestro ejemplo de uso del comando **cd**, este código sigue siendo vulnerable a un error común de programación: **¿qué ocurre si escribimos mal el nombre de la variable que contiene el nombre del directorio?** En ese caso, la shell considerará la variable como vacía y el comando ejecutado, **cd ""**, terminará con éxito, aunque no cambiará de directorio de trabajo porque el primer argumento es una cadena vacía. El resultado obvio es que se borrarán los archivos del directorio actual, así que... ¡Mucho cuidado!

Mejorando la función de salida con error

Hay una serie de mejoras que podemos hacer a la función **error_exit**. Por ejemplo, es bueno incluir el nombre del programa en el mensaje que se muestra para dejar claro quién

---

está produciendo el mensaje de error. Esto empieza a ser importante cuando los programas crecen en complejidad y tienes scripts que llaman a otros scripts. Fíjate también en la inclusión de la variable de entorno LINENO que ayudará a identificar la línea exacta del script donde ha ocurrido el error.

```
#!/usr/bin/env bash

# La variable PROGNAME contiene el nombre del programa que está
# siendo ejecutado. Se puede obtener extrayendo el nombre de
archivo
# de la ruta en el valor del primer parámetro posicional ($0)

PROGNAME=$(basename $0)

error_exit()
{

#
----------------------------------------------------------------
#     Función para salir en caso de error fatal
#          Acepta 1 argumento:
#               cadena conteniendo un mensaje descriptivo del
error
#
----------------------------------------------------------------

    echo "${PROGNAME}: ${1:-Error desconocido}" >&2
    exit 1
}

# Ejemplo de llamada a la función error_exit. Nótese la inclusión
# de la variable de entorno LINENO que contiene el número de
# línea actual

echo "Ejemplo de error con mensaje y número de línea"
error_exit "$LINENO: Ha ocurrido un error."
```

El uso de las llaves dentro de la función "error_exit" es un ejemplo de expansión de parámetros:

- Se puede rodear el nombre de una variable con llaves (como en `${PROGNAME}`) si necesitas estar seguro de que el nombre se separará del texto escrito a su alrededor.
- Algunas personas siempre utilizan este mecanismo, como un buen hábito, independientemente de que haya texto alrededor del nombre de la variable o no.

- El segundo uso `${1:-"Error desconocido"}` significa que si el parámetro 1 (`$1`) no está definido, se sustituye por la cadena `"Error desconocido"`.

  💡 Usando estas expansiones de parámetros es posible realizar ciertas manipulaciones de cadena muy útiles. Puedes leer más acerca de estas expansiones en la sección "EXPANSIONS" del manual de bash.

Consejos para evitar problemas

En este apartado vamos a ver algunos **consejos que todo desarrollador de scripts en BASH debería seguir** para evitar problemas y **cómo depurar nuestros scripts** cuando descubramos algún error.

Deteniendo la ejecución del script en el primer error

Como hemos comentado anteriormente, el problema con el siguiente ejemplo `trouble1.bash` es que BASH no se detiene en caso de encontrar un error, por lo que lo correcto sería comprobar si **cd** tiene éxito antes de ejecutar **rm**:

```
#!/usr/bin/env bash

directorio_inexistente="/ruta/no/valida"

cd "$directorio_inexistente"
rm *.log
```

Sin embargo, en scripts muy sencillos es común que nos olvidemos de hacer estas comprobaciones, mientras que en scripts muy complejos seguramente no hagamos todas las que son necesarias.

Para evitar este tipo de riesgos, solo tenemos que activar la opción "-e" usando **set -e**. De forma que si el comando **cd** —o **rm** o cualquier otro del script— falla, el script es detenido inmediatamente.

```
#!/usr/bin/env bash

set -e

directorio_inexistente="/ruta/no/valida"

# Borrar los archivos .log en $directorio_inexistente
cd "$directorio_inexistente"
rm *.log
```

Si queremos volver en cualquier momento al comportamiento normal de BASH, solo tenemos que utilizar el comando **set +e**. Sin embargo, la recomendación es activar "-e" en las primeras líneas del script y no desactivarlo nunca.

Por otro lado, la opción "-e" no tiene efecto en algunos casos:

- **Tuberías (|)**: En las tuberías el script solo se detiene si el comando que falla es el último de la tubería, ya que ese es el que determina el estado de salida de la sentencia.
- **Listas de comandos (&& y ||)**: Igual que con las tuberías, los comandos unidos por **&&** y **||** en una lista, no provocan la detención del script si fallan. Solo si el estado final de toda la sentencia es un error.
- **Condicionales (if, while y until)**: El comando en la condición de **if**, **while** o **until** no provoca la detención del script.
- **Asignación de variables**: Un comando de asignación a una variable, como `var=$(comando_que_falla)` no detiene el script.

Por tanto, con **set -e** se pueden seguir usando **if**, **&&**, **||** y expresiones similares para detectar y manejar los errores manualmente, con la ventaja de que si algún caso se nos escapa, el programa se detendrá sin mayores problemas.

Por ejemplo, el posible error con el comando **cd** se puede seguir manejando manualmente, mientras dejamos en manos de "-e" un posible error del comando **rm**.

```
#!/usr/bin/env bash

set -e

# Borrar los archivos .log en $directorio_inexistente
cd "$directorio_inexistente" ||
    error_exit "$LINENO: Error accediendo a $directorio_inexistente"
rm *.log
```

Personalizando el manejo de errores con trap

Aunque "-e" detiene el script cuando ocurre un error, por defecto no proporciona información detallada sobre qué comando falló o en qué línea. Para obtener información más útil durante la depuración, podemos combinar "-e" con el comando **trap**.

El comando **trap** permite ejecutar código cuando se reciben las señales indicadas:

```
trap [acción] [señal]
```

En particular, podemos usar la señal ERR para ejecutar código personalizado cada vez que un comando falla:

```
#!/usr/bin/env bash

set -e
trap 'echo "Error $? en la línea $LINENO de: $BASH_COMMAND"' ERR

# Borrar los archivos .log en $directorio_inexistente
cd "$directorio_inexistente"
rm *.log
```

De forma que si ejecutamos el script, obtendríamos una salida similar a:

```
./trouble1.bash
yo@mihost:~$ Error 1 en la línea 7 de: cd "$directorio_inexistente"
```

En este caso, el manejo de errores también se puede mejorar usando funciones:

```
#!/usr/bin/env bash

PROGNAME=$(basename $0)

error_handler()
{
    local exit_code=$?
    local line_number="$1"
    local command="$2"
    local message="'${command}' falló con el código ${exit_code}"

    echo "${PROGNAME}: ERROR: línea ${line_number}: ${message}"
>&2
    exit $exit_code
}

set -e
trap 'error_handler $LINENO "$BASH_COMMAND"' ERR

# Borrar los archivos .log en $directorio_inexistente
cd "$directorio_inexistente"
rm *.log
```

Variables no inicializadas

Veamos la siguiente variante del script que estamos usando como ejemplo `trouble1.bash`:

```
#!/usr/bin/env bash
```

```
# Configuramos el directorio donde vamos a trabajar
directorio_base="/home/usuario/documentos"

# ...

# Borrar los archivos .log en $directorio_base
cd "$directrio_base"
rm *.log
```

Observa que hemos cometido un error tipográfico en el comando **cd**, usando por error la variable `$directrio_base`, que nunca hemos inicializado. Por tanto, BASH sustituirá la expansión por una cadena vacía y el comando **cd** no hará nada, pero tampoco fallará. Es decir, que los archivos serán borrados en el directorio de trabajo actual.

Y si resulta que también olvidamos poner las comillas:

```
cd $directrio_base
```

el borrado de los archivos ".log" tendría lugar en el directorio personal del usuario.

Incluso si no es un error tipográfico en el nombre de la variable, sino que esperamos que la variable a veces no valga nada, tenemos que tener cuidado. Por ejemplo, el siguiente script `trouble2.bash` se ejecuta sin problema si tiene un valor asignado:

```
#!/usr/bin/env bash

# ...

if [ $number = "1" ]; then
    echo "Número igual a 1"
else
    echo "Número no es igual a 1"
fi
```

Pero falla con un error similar al siguiente si `number` no ha sido inicializado:

```
yo@mihost:~$ ./trouble2.bash
./trouble2.bash: línea 5: [: =: se esperaba un operador unario
Número no es igual a 1
```

Esto ocurre porque cuando la variable `number` está vacía, BASH ve lo siguiente después de la expansión de esa línea.

```
if [ = "1" ]; then
```

que obviamente es un error porque el operador "=" necesita dos operandos, uno antes y otro después —no es un operador unario, como nos dice el error—.

Para resolver el problema basta con cambiar la línea por la siguiente:

```
if [ "$number" = "1" ]; then
```

Ya que así, después de que la shell haga la expansión, verá lo siguiente:

```
if [ "" = "1" ]; then
```

que es una condición falsa, expresando correctamente nuestra intención.

Por tanto, siempre **es conveniente tener presente qué ocurriría con el programa si una variable estuviera vacía**.

Detectando variables no inicializadas

Actualmente, **es mucho más seguro pedir a BASH que detenga el script si detecta que se intentan usar variables que no han sido inicializadas previamente**. Para eso BASH proporciona la opción `-u` (también conocida como `nounset`):

```
set -u
```

Por ejemplo, el siguiente script es detenido en la línea del comando **cd** al detectar que se intenta acceder a una variable que no existe, debido al error tipográfico `$directrio_base`:

```
#!/usr/bin/env bash

set -u

# Configuramos el directorio donde vamos a trabajar
directorio_base="/home/usuario/documentos"

# ...

# Borrar los archivos .log en $directorio_base
cd "$directrio_base"
rm *.log
```

Gracias a **set -u**, el script se detendrá con un mensaje de similar al siguiente:

```
yo@mihost:~$ ./trouble1.bash
./trouble1.bash: line 11: directrio_base: unbound variable
```

Es recomendable combinar **set -u** con **set -e** para crear scripts más robustos:

```
#!/usr/bin/env bash

set -e -u        # También es válido usar set -eu

directorio_base="/home/usuario/documentos"
archivo_importante="datos.txt"

cd "$directorio_base" || exit 1
cp "$archivo_importante" backup/
```

Olvidarse de las comillas

Volviendo al script `trouble2.bash`:

```
#!/usr/bin/env bash

number=1

if [ $number = "1" ]; then
    echo "Número igual a 1"
else
    echo "Número no es igual a 1"
fi
```

¿Qué ocurre si editamos la línea 6 y eliminamos la comilla del final de la línea?:

```
echo "Número igual a 1
```

Si ejecutas el script de nuevo deberías obtener algo así:

```
yo@mihost:~$ ./trouble2.bash
./trouble2.bash: línea 8: EOF inesperado mientras se buscaba un
`"' coincidente
./trouble2.bash: línea 11: error sintáctico: no se esperaba el
final del fichero
```

Lo que ocurre es que una vez se han abierto las comillas, la shell busca las comillas de cierre para saber donde termina la cadena, pero antes de que eso ocurra se encuentra con el final del archivo —el End-Of-File o EOF—.

Resolver este tipo de problemas suele ser complicado en un script de gran tamaño. **Este es uno de los motivos por los que deberías comprobar frecuentemente los scripts mientras los escribes**. Además, usar editores de texto con coloreado de sintaxis hace que sea más sencillo encontrar este tipo de errores.

Aislando los problemas

Encontrar bugs en un programa a veces es una tarea muy difícil. Así que a continuación vamos a comentar algunas técnicas que pueden resultar útiles.

Aislar bloques de código mediante comentarios

Este truco involucra poner caracteres de comentario al comienzo de ciertas líneas para evitar que la shell las interprete. Frecuentemente, deberás hacer eso a un bloque de código para comprobar si un problema concreto desaparece. Así puedes aislar la porción del programa que está causando —o no está causando— un problema.

Por ejemplo, si estuviéramos intentando resolver nuestro problema anterior con las comillas, podríamos haber hecho lo siguiente:

```
#!/usr/bin/env bash

number=1

if [ $number = "1" ]; then
    echo "Número igual a 1
#else
#   echo "Número no es igual a 1"
fi
```

Comentando la cláusula **else** y ejecutando el script veríamos como el problema no está en dicha cláusula, como el error emitido por BASH no sugería antes.

Usar comandos echo para verificar tus suposiciones

Los bugs no siempre están donde esperamos encontrarlos. El motivo es que con frecuencia hacemos falsas suposiciones acerca de la ejecución de nuestro programa. Para evitarlo podemos usar el comando **echo**, mientras depuramos, para producir mensajes que nos confirmen que el programa está haciendo lo que esperamos.

Hay dos tipos de mensajes que podemos estar interesados en insertar:

- El primer tipo simplemente es para anunciar que hemos llegado a cierto punto del código del programa. Esto lo vimos en semanas anteriores, cuando explicamos el *stubbing* como técnica para comprobar que nuestras funciones estaban siendo llamadas correctamente:

  ```
  system_info()
  ```

```
{
        # Función de stub temporal
        echo "función system_info"
}
```

- El segundo tipo sirve para mostrar el valor de una variable o variables usadas en un cálculo o una condición. Esto es muy útil porque con frecuencia te encontrarás con que una porción de tu programa fallará porque estás asumiendo que algo previo, que has asumido que es correcto, de hecho no funciona correctamente y está provocando un fallo en una porción posterior.

Observar la ejecución tu script

Es posible hacer que BASH muestre lo que está haciendo mientras se ejecuta un script. Para ello solo es necesario activar la opción "-x" así:

```
set -x
```

Por ejemplo, al ejecutar el siguiente script:

```
#!/usr/bin/env bash

number=1

set -x
if [ $number = "1" ]; then
    echo "Número igual a 1"
else
    echo "Número no es igual a 1"
fi
```

BASH mostrará cada línea antes de ejecutarla—con todas las expansiones realizadas—, a partir del punto donde se ejecutó **set -x**:

```
yo@mihost:~$ ./trouble2.bash
+ '[' 1 = 1 ']'
+ echo 'Número igual a 1'
Número igual a 1
```

Esta técnica se llama *tracing* o rastreo. Alternativamente, en cualquier punto del programa se puede utilizar **set +x** para desactivarlo, si solo nos interesa hacer el rastreo en una porción del script, porque es ahí donde creemos que está el problema:

```
#!/usr/bin/env bash

number=1
```

```
set -x
if [ $number = "1" ]; then
    echo "Número igual a 1"
else
    echo "Número no es igual a 1"
fi
set +x

# El resto del script...
```

Creando archivos temporales

Muchos programas requieren el uso de archivos temporales. Es decir, archivos que se utilizan momentáneamente por el script y que pueden ser borrados al terminar su ejecución.

Para hacerlo correctamente debemos observar algunas recomendaciones:

- La tradición en UNIX dicta que se debe usar un directorio llamado /tmp para almacenar estos archivos. El problema es que cualquier usuario puede escribir en ese directorio.
- La solución ideal sería usar un directorio donde solo el usuario tenga permisos. Por eso **en los sistemas modernos cada usuario tiene un directorio temporal propio. La variable de entorno $TMPDIR suele indicar su ruta**.
- **Es conveniente asegurarse que los archivos temporales de nuestro script no entran en conflicto ni sobreescriban los archivos temporales de otros scripts o programas.**

Un buen nombre de archivo podría ser el siguiente:

```
TEMP_FILE=${TMPDIR:-/tmp}/miscript.$$.$RANDOM
```

- La variable `$TMPDIR` contiene la ruta del directorio temporal. Si no está definida, se usa la ruta "/tmp".
- Es una práctica común incluir el nombre del programa en el nombre del archivo, por ejemplo "miscript".
- Después se usa la variable `$$` para incorporar el identificador de procesos (PID) del programa. Esto ayuda a identificar qué proceso es el responsable del archivo, pero no es lo bastante impredecible como para que el nombre del archivo sea seguro.
- Finalmente, usamos la variable `$RANDOM` para incorporar un número aleatorio al nombre del archivo.

Una forma similar es haciendo uso del comando **mktemp**:

```
TEMP_FILE=$(mktemp -t tmp.XXXXXXXXXX.miscript.$$)
```

```
echo $TEMP_FILE
/tmp/tmp.RXVMdsa2LY.miscript.169
```

Esta forma es la recomendada de obtener un archivo temporal, puesto que **mktemp** se asegura de escoger un nombre que aún no esté en uso y crea un archivo vacío con dicho nombre, para evitar que otro proceso escoja el mismo en el futuro.

Actividad 5

Es hora de parar un momento y trabajar sobre un ejercicio propuesto.

Mejorar el manejo de errores

Incorpora a tu script la función de salida con error **error_exit**. Úsala para salir con error si se pasa al script una opción no soportada, después de mostrar la ayuda sobre el uso del script.

Además, del **while** que proceso la línea de comandos, comprobar si los comandos **df**, **du** y **uptime** existen. Si no es así, saldrá con un error indicando que falta un programa básico para el funcionamiento del script. Recuerda que la ruta del archivo que implementa un comando se puede obtener ejecutando **type -P uptime**.

Actividad 6

Es hora de parar un momento y trabajar sobre un ejercicio propuesto.

Vamos a mejorar nuestra función **home_space** con algunas características adicionales.

Recordemos que si el usuario actual es el root, procesaremos CADA directorio en /home por separado. Mientras que si no es root, procesaremos solo el directorio personal del usuario.

Teniendo esto en cuenta, cambiaremos la salida de **home_space** para que muestre información adicional de cada directorio así:

```
ARCHIVOS  DIRECTORIOS  USADO    DIRECTORIO
     123            4  1092345  vmunoz
     43             4  923450   jttoledo
     67             5  282334   jmtorres
```

...

La primera columna adicional muestra el número total de archivos en el directorio, incluyendo archivos en subdirectorios en cualquier nivel de profundidad. La segunda columna adicional es el número total de subdirectorios, en cualquier nivel de profundidad —no solo subdirectorios directos—.

Para obtener los valores de estas columnas adicionales se puede usar el comando **find**. Mientras que para dejarlo bonito, se pueden usar los comandos printf o **column** para que las columnas queden bien alineadas.

El comando printf es igual que la sentencia printf de AWK que vimos en la Unidad 3, con la diferencia de que no se utilizan comas "," para separar los argumentos:

```
yo@mihost:~$ printf "|%10s|%10s|\n" "Apellido" "Nombre"
|  Apellido|    Nombre|
yo@mihost:~$ printf "|%10s|%10s|\n" "Pedro" "Afonso"
yo@mihost:~$ n=123.4567
yo@mihost:~$ printf "|%9.3f|\n" $n
|  123.457|
yo@mihost:~$ printf "|%11.3E|\n" $n
|  1.235E+02|
```
