UNIX: Shell scripts

Estrutura geral de um script

Os arquivos de script permitem construir esquemas de execução complexos a partir dos comandos básicos do shell. A forma mais elementar de arquivo de script é apenas um conjunto de comandos em um arquivo texto, com permissões de execução habilitadas. O arquivo `backup.sh`, cujo conteúdo é mostrado a seguir, é um exemplo de script:

backup.sh

```
echo "Iniciando backup..."

# montar o diretório do servidor de backup
mount fileserver.ufpr.br:/backup /backup

# efetuar o backup em um arquivo tar compactado
tar czf /backup/home.tar.gz /home

# desmontar o diretório do servidor de backup
umount /backup

echo "Backup concluido !"
```

Quando o script backup for executado, os comandos do arquivo serão executados em sequência pelo shell corrente (de onde ele foi lançado). Assim, se o usuário estiver usando o shell bash, os comandos do script serão executados por esse shell. Como isso poderia gerar problemas em scripts usados por vários usuários, é possível forçar a execução do script com um shell específico (ou outro programa que interprete os comandos do arquivo). Para isso é necessário informar ao sistema operacional o programa a ser usado, na primeira linha do arquivo do script:

backup.sh

```
#!/bin/bash --noprofile

# A opção --noprofile inibe a leitura dos arquivos de inicialização
# do shell, tornando o lançamento do script muito mais rápido.
# comandos de um script em Bash-Shell

server=fileserver.ufpr.br
backdir=/backup
...
exit 0
```

Com isso, será lançado um shell Bash separado (um novo processo), somente para interpretar as instruções do script. O novo processo será terminado pelo comando `exit`, cujo parâmetro é devolvido ao shell anterior através da variável `$?`. Esse procedimento pode ser usado para lançar scripts para outros shells, ou mesmo outras linguagens interpretadas, como Perl, Awk, Sed, Php, Python, etc.



Embora não seja obrigatório, é um bom hábito usar a extensão `.sh` nos arquivos contendo scripts de shell, para deixar claro para o usuário seu conteúdo.

Parâmetros de entrada

Os argumentos da linha de comando são passados para o shell através da variável local $BASH_ARGV. Os campos individuais dessa variável podem ser acessados como em uma variável local qualquer. Além disso, uma série de atalhos é definida para facilitar o acesso a esses parâmetros:

- `$0` : o nome do script
- `$n` : o n-ésimo argumento da linha de comando
- `$*`, `$@` : todos os argumentos da linha de comando
- `$#` : número de argumentos
- `$?` : status do último comando executado (status ≠ 0 indica erro)
- `$$` : número de processo (PID) do shell que executa o script

Eis um exemplo através do script `listaparams.sh`:

listaparams.sh

```
#!/bin/bash
# exemplo de uso dos parâmetros de entrada
echo "Nome do script : $0"
echo "Primeiro parâmetro : $1"
echo "Todos os parâmetros : $*"
echo "Numero de parametros : $#"
echo "Numero deste processo : $$"
exit 0
```

Chamando o script acima com alguns parâmetros se obtém a seguinte resposta:

```
$ listaparams.sh banana tomate pessego melao pera uva
Nome do script : listaparams
Primeiro parâmetro : banana
Todos os parâmetros : banana tomate pessego melao pera uva
Numero de parametros : 6
Numero deste processo : 2215
```

Controle de fluxo

Existem diversos construtores de controle de fluxo que podem ser usados em scripts BASH. Os principais são descritos a seguir.

Condições

Como na maioria das linguagens, no shell Bash, testes de condições são realizados por estruturas do tipo *if-then-else*. As condições testadas são os status de saída da execução de comandos (o valor inteiro retornado pela chamada de sistema `exit()` do comando). Caso o status seja zero (0), a condição é considerada verdadeira:

```
if cmp $file1 $file2 >/dev/null   # testa o status do comando cmp
then
   echo "os arquivos são iguais"
else
   echo "os arquivos são distintos"
fi
```

Essa lógica “ao contrário” pode causar uma certa confusão aos iniciantes. Assim, para simplificar a programação de scripts, é definido um operador `test` *condition*, que também pode ser representado por `[` *condition* `]` e retorna zero (0) se a condição testada for verdadeira:

```
if test $n1 -lt $n2         # $n1 é menor que $n2?
then
   echo "$n1 é menor que $n2"
fi

# ou

if [ $n1 -lt $n2 ]          # $n1 é menor que $n2?
then
   echo "$n1 é menor que $n2"
fi
```

Os principais operadores de teste disponíveis são:

Operador *if-then*

```
if comando
then
   ...
fi

# testa o status do comando cmp
if cmp file1 file2 >/dev/null
then
   echo "Os arquivos são iguais"
fi
```

Operador *if-then-else*

```
if comando
then
   ...
else
   ...
fi

# testa a existência de $file1
if [ -e "$file1" ]
then
   echo "$file1 existe"
else
   echo "$file1 não existe"
fi
```

Operador *if-then-elif-else*

```
if comando 1
then
   ...
elif comando 2
then
   ...
else
   ...
fi

# compara as variáveis $n1 e $n2
if [ $n1 -lt $n2 ]
then
   echo "$n1 < $n2"
elif [ $n1 -gt $n2 ]
then
   echo "$n1 > $n2"
else
   echo "$n1 = $n2"
fi
```

Operador *case*

```
case variável in
   "string1")
      ...
      break;;
   "string2")
      ...
      break;;
   *):
      ...
      break;;
esac

case $opt in
   "-c")
         complete=1 ;;
   "-s")
         short=1 ;
         name="" ;;
   *)
         echo "opção $opt desconhecida" ;
         exit 1 ;;
esac
```

Os principais tipos de teste disponíveis são:

- Comparações entre números
  - `-eq` : igual a
  - `-ne` : diferente de
  - `-gt` : maior que
  - `-ge` : maior ou igual a
  - `-lt` : menor que
  - `-le` : menor ou igual a
  - `-a` : AND binário (bit a bit)

    - `-o` : OR binário (bit a bit)
- Comparações entre strings usando `[ ]`
    - `=` : igual a
    - `!=` : diferente de
    - `-z` : string de tamanho zero
- Comparações entre strings usando `[[ ]]`
    - `<=` : menor ou igual a (lexicográfico)
    - `>=` : maior ou igual a (lexicográfico)
- Associações entre condições
    - `&&` : AND lógico
    - `||` : OR lógico

Os operadores de teste em arquivos permitem verificar propriedades de entradas no sistema de arquivos. Eles são usados na forma -op, onde op corresponde ao teste desejado. Os principais testes são:

- `e` : a entrada existe
- `r` : a entrada pode ser lida
- `w` : a entrada pode ser escrita
- `O` : o usuário é o proprietário da entrada
- `s` : tem tamanho maior que zero
- `f` : é um arquivo normal
- `d` : é um diretório
- `L` : é um link simbólico
- `b` : é um dispositivo orientado a bloco
- `c` : é um dispositivo orientado a caractere
- `p` : é um named pipe (fifo)
- `S` : é um socket special file
- `u` : tem o bit SUID habilitado
- `g` : tem o bit SGID habilitado
- `G` : grupo da entrada é o mesmo do proprietário
- `k` : o stick bit está habilitado
- `x` : a entrada pode ser executada
- `nt` : Verifica se um arquivo é mais novo que outro
- `ot` : Verifica se um arquivo é mais velho que outro
- `ef` : Verifica se é o mesmo arquivo (link)

Eis um exemplo de uso de testes em arquivos:

```
arquivo='/etc/passwd'

if [ -e $arquivo ]
then
   if [ -f $arquivo ]
   then
      if [ -r $arquivo ]
      then
         source $arquivo
      else
         echo "Nao posso ler o arquivo $arquivo"
      fi
   else
      echo "$arquivo não é um arquivo normal"
   fi
else
   echo "$arquivo não existe"
fi
```

Laços

Laço *for*

```
for variável in lista de valores
do
   ...
done

for i in *.c
do
   echo "compilando $i"
   cc -c $i
done
```

Laço *while*

```
while condição
do
   ...
done

i=0
while [ $i -lt 10 ]
do
   echo $i
   let i++
done
```

Operador *select*

```
select variável in lista de valores
do
   ...
done

select f in "abacate" "pera" "uva" "banana" "morango"
do
   echo "Escolheu $f"
done
```

Além das estruturas acima, algumas outras podem ser usadas para executar comandos em situações específicas:

- `` `comando` `` : substitui a expressão entre crases pelo resultado (`stdout`) da execução do comando. Por exemplo, a linha de comando abaixo coloca na variável `arqs` os nomes de arquivos retornados pelo comando `find`:

```
arqs=`find /etc -type f -iname '???'`
```

- `comando1; comando2; comando3` : executa sequencialmente os comandos indicados

Operadores aritméticos

Variáveis contendo números inteiros podem ser usadas em expressões aritméticas e lógicas. A atribuição do resultado de uma expressão aritmética a uma variável pode ser feita de diversas formas. Por exemplo, as três expressões a seguir têm o mesmo efeito:

```
i=$((j + k))

let i=j+k

i=`expr $j + $k`
```

Os principais operadores aritméticos disponíveis são:

- + - * / : aritmética básica
- ** : potenciação
- % : módulo (resto)
- += -= *= /= %= : aritmética e atribuição (como em C)
- <> : deslocamento de bits
- <>= : deslocamento e atribuição
- & | : AND e OR binários
- &= |= : AND e OR binários com atribuição
- ! : NOT binário
- ^ : XOR binário
- && || : AND e OR lógicos

Exercícios

1. Analise e descreva o que faz o script abaixo, passo a passo. Em seguida, copie-o no arquivo `meuscript.sh`, em sua área de trabalho, e teste-o.

   meuscript.sh

   ```
   #!/bin/bash --noprofile

   # testar se ha um so parametro de entrada
   if [ $# != 1 ]
   then
      echo ''Erro na chamada''
      echo ''Uso: criadir numero de diretorios''
      exit 1
   fi

   num=0

   while [ $num -lt $1 ]
   do
      echo ''Criando diretorio $num''
      mkdir dir$num
      let num++
   done

   echo ''Acabei de criar $1 diretorios''
   ```

```
exit 0
```

2. Escreva um script chamado `clean.sh` para limpar seu diretório `$HOME`, removendo todos os arquivos com extensão `bak` ou `~` que não tenham sido acessados há pelo menos 3 dias. Dica: use os comandos `find` e `rm` e a avaliação por crases.
3. Escreva um script para criar diretórios com nome `DirXXX`, onde XXX varia de 001 a 299. Dica: use o comando `printf` para gerar o nome dos diretórios a criar.
4. Escreva um conjunto de scripts para gerenciar o apagamento de arquivos. O script `del.sh` deve mover os arquivos passados como parâmetros para um diretório lixeira; o script `undel.sh` deve mover arquivos da lixeira para o diretório corrente e o script `lsdel.sh` deve listar o conteúdo da lixeira. O diretório `lixeira` deve ser definido através da variável de ambiente `$LIXEIRA`.
5. Funda os scripts do exercício anterior em um só script `del.sh`, com os demais (`undel.sh` e `lsdel.sh`) sendo links simbólicos para o primeiro. Como fazer para que o script saiba qual a operação desejada quando ele for chamado, sem precisar informá-lo via parâmetros?
6. Escreva um script para verificar quais hosts de uma determinada rede IP estão ativos. Para testar se um host está ativo, use o comando `ping`. A rede deve ser informada via linha de comando, no formato `x.y.z`, e o resultado deve ser enviado para um arquivo com o nome `x.y.z.log`. Deve ser testada a acessibilidade dos hosts de `x.y.z.1` a `x.y.z.254`.
7. Escreva um script que recebe o nome de um arquivo como parâmetro e o copia para o diretório `/tmp`
8. Escreva um script que recebe 2 parâmetros. O primeiro é um diretório e o segundo um arquivo. O script deve mover o arquivo para o diretório.
9. Escreva um script que utiliza o comando `for` para criar arquivos com nomes que começam com teste e terminam com um número de 0 a 9 (DICA: `for (i=0; i<=9; i++); do`)
10. Veja o que faz o script abaixo:

```
for i in *; do echo $i; done
```

11. Modifique o script acima para listar os atributos de todos os arquivos.
12. Implemente o script abaixo passo a passo e descubra o que faz cada comando:

```
#!/bin/bash
set -x

temph=`date | cut -c13-14`
dat=`date +"%A %d de %B de %Y (%r)"`

if [ $temph -lt 12 ]; then
   mess="Bom dia $LOGNAME, tenha um bom dia!"
elif [ $temph -gt 12 -a $temph -le 18 ]; then
   mess="Boa tarde $LOGNAME"
elif [ $temph -gt 18 ]; then
   mess="Boa noite $LOGNAME"
fi
echo -e "$mess\nHoje e $dat"
```

13. Implemente o script abaixo passo a passo e descubra o que faz cada comando:

```
#!/bin/bash
#set -x

# Esse script fornece informacoes sobre um arquivo.

FILENAME="$1"

echo "Propriedades de $FILENAME:"
```

```
if [ -f $FILENAME ]; then
  echo "Tamanho: $(ls -lh $FILENAME | awk '{ print $5 }')"
  echo "Tipo: $(file $FILENAME | cut -d":" -f2 -)"
  echo "Numero de Inode: $(ls -i $FILENAME | cut -d" " -f1 -)"
  echo "$(df -h $FILENAME | grep -v Mounted | awk '{ print "Em",$1", \
que esta montado como particao ",$6}')"
else
  echo "Arquivo nao existe."
fi
```

14. Escreva um script de backup que salve uma cópia de um diretório em outro diretório. O script deve verificar se ambos diretórios existem e criar o destino caso não exista. O nome do arquivo de backup deve conter a palavra “manha”, “tarde”, “noite”, dependendo de quando foi executado.
15. Escreva um script que pergunte a idade do usuário. Caso ele tenha mais que 18 anos, imprima uma mensagem informando que ele está autorizado a beber álcool. Caso ele tenha menos que 18 anos, informe-o quantos anos terá que aguardar até poder beber.
    - Adicionalmente, calcule quantos litros de cerveja seu usuário já bebeu na vida, assumindo que um adulto bebe em média 100 litros por ano, e imprima esta informação ao usuário
16. Escreva um script que encontra todos arquivos com determinada extensão em um determinado diretório e, baseado nos parâmetros enviados, faça cópias dos arquivos com uma nova extensão, renomeie os arquivos para uma nova extensão, ou remova os arquivos. O script terá quatro parâmetros (três em caso de remoção):

    ```
    diretorio [ -c | -m | -r ] ext1 ext2
    ```

    onde as extensões ext1 e ext2 são strings que podem ou não conter o caractere “.”. A opção -c denota copiar, -m denota mover/renomear e -r denota remover. O script deve imprimir uma linha contendo

    ```
    nome antigo => nome novo
    ```

    para cada arquivo copiado ou renomeado. Em caso de remoção, imprimir apenas o nome do arquivo. Se o diretório não existir ou o usuário não tiver permissões para alterá-lo, imprimir uma mensagem de erro.

From:
https://wiki.inf.ufpr.br/maziero/ - **Prof. Carlos Maziero**

Permanent link:
https://wiki.inf.ufpr.br/maziero/doku.php?id=unix:shell_scripts

Last update: **2023/11/23 15:33**