
Dave Hamilton - www.sxc.hu

Curso de Shell Script

Papo de Botequim VI

Blocos de código e laços (ou *loops*, como preferem alguns) são o tema do mês em mais uma lição de nosso curso de Shell Script. Garçom, salta uma boa redondinha, que tô a fim de refrescar o pensamento! POR JULIO CEZAR NEVES

Fala, cara! E aí, já tá sabendo tudo do comando *for*? Eu te deixei um exercício para treinar, se não me engano era para contar a quantidade de palavras de um arquivo... Você fez?

– Claro! Tô empolgadão com essa linguagem! Eu fiz da forma que você pediu, olha só...
– Êpa! Peraí que eu tô sequinho pra tomar um chope. Aê Chico, traz dois por favor. Um sem colarinho!
– Como eu ia dizendo, olha como eu fiz. É muito fácil...

```
$ cat contpal.sh
#!/bin/bash
# Script meramente pedagógico
# cuja função é contar a
# quantidade de palavras de
# um arquivo. Supõe-se que as
# palavras estão separadas
# entre si por espaços, <TAB>
# ou <ENTER>.
if  [ $# -ne 1 ]
then
    echo uso: $0 /caminho/do/➥
arquivo
    exit 2
fi
Cont=0
for Palavra in $(cat $1)
do
    Cont=$((Cont+1))
done
echo O arquivo $1 tem $Cont ➥
palavras.
```

Ou seja, o programa começa, como sempre, verificando se a passagem de parâmetros foi correta; em seguida o comando *for* se incumbe de pegar cada uma das palavras (lembre-se que o *$IFS* padrão é branco, *TAB* e *ENTER*, que é exatamente o que desejamos para separar as palavras), incrementando a variável *$Cont*. Vamos relembrar como é o arquivo *ArqDoDOS.txt*.

```
$ cat ArqDoDOS.txt
Este arquivo
foi gerado pelo
DOS/Rwin e foi
baixado por um
ftp mal feito.
```

Agora vamos testar o programa passando esse arquivo como parâmetro:

```
$ contpal.sh ArqDoDOS.txt
O arquivo ArqDoDOS.txt tem 14
palavras.
```

Funcionou legal! Se você se lembra, em nossa última aula mostramos o loop *for* a seguir:

```
for ((; i<=9;))
do
    let i++
    echo -n "$i "
done
```

Uma vez que chegamos neste ponto, creio ser interessante citar que o Shell trabalha com o conceito de "Expansão Aritmética" (*Arithmetic Expansion*), que é acionada por uma construção da forma *$((expressão))* ou *let expressão*.

No último loop *for* usei a expansão aritmética das duas formas, mas não podemos seguir adiante sem saber que a expressão pode ser uma das listadas na tabela 1.

Mas você pensa que o papo de loop (ou laço) se encerra no comando *for*? Ledo engano, amigo, vamos a partir de agora ver mais dois comandos.

O comando *while*

Todos os programadores conhecem este comando, porque é comum a todas as linguagens. Nelas, o que normalmente ocorre é que um bloco de comandos é executado, enquanto (enquanto, em inglês, é "while") uma determinada condição for verdadeira.

Tabela 1: Expressões no Shell

Expressão	Resultado
id++ id--	pós-incremento e pós-decremento de variáveis
++id --id	pré-incremento e pré-decremento de variáveis
**	exponenciação
* / %	multiplicação, divisão, resto da divisão (módulo)
+ -	adição, subtração
<= >= < >	comparação
== !=	igualdade, desigualdade
&&	E lógico
||	OU lógico

Pois bem, isso é o que acontece nas linguagens caretas! Em programação Shell, o bloco de comandos é executado enquanto um comando for verdadeiro. E é claro, se quiser testar uma condição, use o comando *while* junto com o comando *test*, exatamente como você aprendeu a fazer no *if*, lembra? Então a sintaxe do comando fica assim:

```
while comando
do
    cmd1
    cmd2
    ...
    cmdn
done
```

e dessa forma, o bloco formado pelas instruções *cmd1, cmd2,...* e *cmdn* é executado enquanto a execução da instrução *comando* for bem sucedida.

Suponha a seguinte cena: tinha uma tremenda gata me esperando e eu estava preso no trabalho sem poder sair porque o meu chefe, que é um pé no saco (aliás chefe-chato é uma redundância, né?), ainda estava na sala dele, que fica bem na minha passagem para a rua. Ele começou a ficar cabreiro depois da quinta vez que passei pela sua porta e olhei para ver se já havia ido embora. Então voltei para a minha mesa e fiz, no servidor, um script assim:

```
$ cat logaute.sh
#!/bin/bash
# Espero que a Xuxa não tenha
# copyright de xefe e xato :)
while who | grep xefe
do
    sleep 30
done
echo O xato se mandou, não ➥
hesite, dê exit e vá à luta
```

Neste scriptzinho, o comando *while* testa o pipeline composto pelos comandos *who* e *grep*, que será verdadeiro enquanto o *grep* localizar a palavra *xefe* na saída do comando *who*. Desta forma, o script dormirá por 30 segundos enquanto o chefe estiver logado (Argh!). Assim que ele se desconectar do servidor, o fluxo do script sairá do loop e te mostrará a tão ansiada mensagem de liberdade. Mas quando executei o script, adivinha o que aconteceu?

```
$ logaute.sh
xefe  pts/0   Jan  4 08:46 ➥
(10.2.4.144)
xefe  pts/0   Jan  4 08:46 ➥
(10.2.4.144)
...
xefe  pts/0   Jan  4 08:46 ➥
(10.2.4.144)
```

Isto é, a cada 30 segundos a saída do comando grep seria enviada para a tela, o que não é legal, já que poluiria a tela do meu micro e a mensagem tão esperada poderia passar despercebida. Para evitar isso, já sabemos que a saída do pipeline tem que ser redirecionada para o dispositivo */dev/null*.

```
$ cat logaute.sh
#!/bin/bash
# Espero que a Xuxa não tenha
# copyright de xefe e xato :)
while who | grep xefe > /dev/null
do
    sleep 30
done
echo O xato se mandou, não ➥
hesite, dê exit e vá a luta
```

Agora quero montar um script que receba o nome (e eventuais parâmetros) de um programa que será executado em background e que me informe do seu término. Mas, para você entender este exemplo, primeiro tenho de mostrar uma nova variável do sistema. Veja estes comandos executados diretamente no prompt:

```
$ sleep 10&
[1] 16317
$ echo $!
16317
[1]+  Done                    sleep 10
$ echo $!
16317
```

Isto é, criei um processo em background que dorme por 10 segundos, somente para mostrar que a variável *$!* guarda o PID (*Process ID*) do último processo em background. Mas observe a listagem e repare, após a linha do *Done*, que a variável reteve o valor mesmo após o término desse processo.

Bem, sabendo isso, já fica mais fácil monitorar qualquer processo em background. Veja só como:

```
$cat monbg.sh
#!/bin/bash
# Executa e  monitora um
# processo em background
$1 &     # Coloca em backgroud
while ps | grep -q $!
do
    sleep 5
done
echo Fim do Processo $1
```

Esse script é bastante similar ao anterior, mas tem uns macetes a mais, veja só: ele tem que ser executado em background para não prender o prompt mas o *$!* será o do programa passado como parâmetro, já que ele foi colocado em background após o *monbg.sh* propriamente dito. Repare também na opção *-q* (quiet) do grep, que serve para fazê-lo "trabalhar em silêncio". O mesmo resultado poderia ser obtido com a linha: *while ps | grep $! > /dev/null*, como nos exemplos que vimos até agora.

Vamos melhorar o nosso velho *musinc*, nosso programa para incluir registros no arquivo *musicas*, mas antes preciso te ensinar a pegar um dado da tela, e já vou avisando: só vou dar uma pequena dica do comando read (que é quem pega o dado da tela), que seja o suficiente para resolver este nosso problema. Em uma outra rodada de chope vou te ensinar tudo sobre o assunto, inclusive como formatar tela, mas hoje estamos falando sobre loops. A sintaxe do comando read que nos interessa por hoje é a seguinte:

```
$ read -p "prompt de leitura" var
```

Onde "prompt de leitura" é o texto que você quer que apareça escrito na tela. Quando o operador teclar tal dado, ele será armazenado na variável *var*. Por exemplo:

```
$ read -p "Título do Álbum: " Tit
```

Bem, uma vez entendido isso, vamos à especificação do nosso problema: faremos um programa que inicialmente lerá o nome do álbum e em seguida fará um loop de leitura, pegando o nome da música e o artista. Esse loop termina quando for informada uma música com nome vazio, isto é, quando o operador

Dica

Leitura de arquivo significa ler um a um todos os registros, o que é sempre uma operação lenta. Fique atento para não usar o while quando for desnecessário. O Shell tem ferramentas como o sed e a família grep, que vasculham arquivos de forma otimizada sem que seja necessário o uso do while para fazê-lo registro a registro.

der um simples <ENTER>. Para facilitar a vida do operador, vamos oferecer como default o mesmo nome do artista da música anterior (já que é normal que o álbum seja todo do mesmo artista) até que ele deseje alterá-lo. Veja na listagem 1 como ficou o programa.

Nosso exemplo começa com a leitura do título do álbum. Caso ele não seja informado, terminamos a execução do programa. Em seguida um *grep* procura, no início (^) de cada registro de músicas, o título informado seguido do separador (^) (que está precedido de uma contrabarra [\] para protegê-lo da interpretação do Shell).

Para ler os nomes dos artistas e as músicas do álbum, foi montado um loop *while* simples, cujo único destaque é o fato de ele armazenar o nome do intérprete da música anterior na variável *$oArt*, que só terá o seu conteúdo alterado quando algum dado for informado para a variável *$Art*, isto é, quando não for teclado um simples *ENTER* para manter o artista anterior.

O que foi visto até agora sobre o *while* foi muito pouco. Esse comando é muito utilizado, principalmente para leitura de arquivos, porém ainda nos falta bagagem para prosseguir. Depois que aprendermos mais sobre isso, veremos essa instrução mais a fundo.

O comando *until*

Este comando funciona de forma idêntica ao *while*, porém ao contrário. Disse tudo mas não disse nada, né? É o seguinte: ambos testam comandos; ambos possuem a mesma sintaxe e ambos atuam em loop; porém, o *while* executa o bloco de instruções do loop enquanto um comando for bem sucedido; já o until executa o bloco do loop até que o comando seja bem sucedido. Parece pouca coisa, mas a diferença é fundamental. A sintaxe do comando é praticamente a mesma do while. Veja:

```
until comando
do
    cmd1
    cmd2
    ...
    cmdn
done
```

e dessa forma o bloco de comandos formado pelas instruções *cmd1, cmd2,...* e *cmdn* é executado até que a execução da instrução *comando* seja bem sucedida.

Como eu te disse, *while* e *until* funcionam de forma antagônica, e isso é muito fácil de demonstrar: em uma guerra, sempre que se inventa uma arma, o inimigo busca uma solução para neutralizá-la. Foi baseado nesse principio belicoso que meu chefe desenvolveu, no mesmo servidor em que eu executava o *logaute.sh*, um script para controlar o meu horário de chegada.

Um dia tivemos um problema na rede. Ele me pediu para dar uma olhada no micro dele e me deixou sozinho na sala. Resolvi bisbilhotar os arquivos – guerra é guerra – e veja só o que descobri:

```
$cat chegada.sh
#!/bin/bash
until who | grep julio
do
    sleep 30
done
echo $(date "+ Em %d/%m às ➥
%H:%Mh") > relapso.log
```

Olha que safado! O cara estava montando um log com os meus horários de chegada, e ainda por cima chamou o arquivo de *relapso.log*! O que será que ele quis dizer com isso?

Nesse script, o pipeline *who | grep julio*, será bem sucedido somente quando *julio* for encontrado na saída do comando *who*, isto é, quando eu me "logar" no servidor. Até que isso aconteça, o comando *sleep*, que forma o bloco de instruções do *until*, colocará o programa em espera por 30 segundos. Quando esse loop encerrar-se, será enviada uma mensagem para o arquivo *relapso.log*. Supondo que no dia 20/01 eu me "loguei" às 11:23 horas, a mensagem seria a seguinte:

Listagem 1

```
$ cat musinc.sh
#!/bin/bash
# Cadastra CDs (versao 4)
#
clear
read -p "Título do Álbum: " Tit
[ "$Tit" ] || exit 1 # Fim da execução se título vazio
if grep "^$Tit\^" musicas > /dev/null
then
    echo "Este álbum já está cadastrado"
    exit 1
fi
Reg="$Tit^"
Cont=1
oArt=
while true
do
    echo "Dados da trilha $Cont:"
    read -p "Música: " Mus
    [ "$Mus" ] || break       # Sai se vazio
    read -p "Artista: $oArt // " Art
    [ "$Art" ] && oArt="$Art" # Se vazio Art anterior
    Reg="$Reg$oArt~$Mus:"     # Montando registro
    Cont=$((Cont + 1))
    # A linha anterior tb poderia ser ((Cont++))
done
echo "$Reg" >> musicas
sort musicas -o musicas
```

Em 20/01 às 11:23h

Voltando à nossa CDteca, quando vamos cadastrar músicas seria ideal que pudéssemos cadastrar diversos CDs de uma vez só. Na última versão do programa isso não ocorre: a cada CD cadastrado o programa termina. Veja na listagem 2 como melhorá-lo.

Nesta versão, um loop maior foi adicionado antes da leitura do título, que só terminará quando a variável *$Para* deixar de ser vazia. Caso o título do álbum não seja informado, a variável *$Para* receberá um valor (coloquei 1, mas poderia ter colocado qualquer coisa) para sair desse loop, terminando o programa. No resto, o script é idêntico à versão anterior.

Atalhos no loop

Nem sempre um ciclo de programa, compreendido entre um *do* e um *done*, sai pela porta da frente. Em algumas oportunidades, temos que colocar um comando que aborte de forma controlada esse loop. De maneira inversa, algumas vezes desejamos que o fluxo de execução do programa volte antes de chegar ao *done*. Para isso, temos respectivamente os comandos *break* (que já vimos rapidamente nos exemplos do comando *while*) e *continue*, que funcionam da forma mostrada na figura 1.

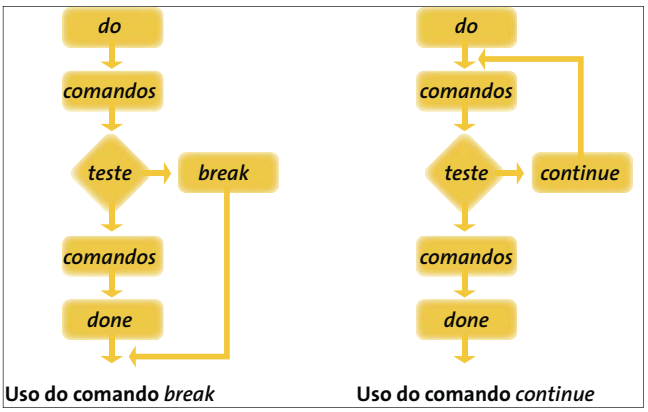

Figura 1: A estrutura dos comandos *break* e *continue*, usados para controlar o fluxo de execução em loops.

O que eu não havia dito anteriormente é que nas suas sintaxes genéricas eles aparecem da seguinte forma:

```
break [qtd loop]
```

e também:

```
continue [qtd loop]
```

Onde *qtd loop* representa a quantidade dos loops mais internos sobre os quais os comandos irão atuar. Seu valor por *default* é 1.

Duvido que você nunca tenha apagado um arquivo e logo após deu um tabefe na testa se xingando porque não devia tê-lo removido. Pois é, na décima vez que fiz esta besteira, criei um script para simular uma lixeira, isto é, quando mando remover um (ou vários) arquivo(s), o programa "finge" que deletou, mas no duro o que ele fez foi mandá-lo(s) para o diretório /tmp/LoginName_do_usuario. Chamei esse programa de *erreeme* e no arquivo */etc/profile* coloquei a seguinte linha, que cria um "apelido" para ele:

```
alias rm=erreeme
```

Veja o programa na listagem 3. Como você pode ver, a maior parte do script é formada por pequenas críticas aos parâmetros informados, mas como o script pode ter recebido diversos arquivos a remover, a cada arquivo que não se encaixa dentro do especificado há

Listagem 2

```
$ cat musinc.sh
#!/bin/bash
# Cadastra CDs (versao 5)
#
Para=
until [ "$Para" ]
do
    clear
    read -p "Título do Álbum: " Tit
    if  [ ! "$Tit" ] # Se titulo vazio...
    then
        Para=1     # Liguei flag de saída
    else
        if grep "^$Tit\^" musicas > /dev/null
        then
            echo "Este álbum já está cadastrado"
            exit 1
        fi
        Reg="$Tit^"
        Cont=1
        oArt=
        while [ "$Tit" ]
        do
            echo Dados da trilha $Cont:
            read -p "Música: " Mus
            [ "$Mus" ] || break       # Sai se vazio
            read -p "Artista: $oArt // " Art
            [ "$Art" ] && oArt="$Art" # Se vazio Art anterior
            Reg="$Reg$oArt~$Mus:"     # Montando registro
            Cont=$((Cont + 1))
            # A linha anterior tb poderia ser ((Cont++))
        done
        echo "$Reg" >> musicas
        sort musicas -o musicas
    fi
done
```

Listagem 3: erreeme.sh

```
$ cat erreeme.sh
#!/bin/bash
#
# Salvando cópia de um arquivo antes de removê-lo

# Tem de ter um ou mais arquivos a remover
if  [ $# -eq 0 ]
then
    echo "Erro -> Uso: erreeme arq [arq] ... [arq]"
    echo "O uso de metacaracteres e' permitido. Ex.➥
erreeme arq*"
    exit 1
fi

# Variável do sistema que contém o nome do usuário.
MeuDir="/tmp/$LOGNAME"
# Se não existir o meu diretório sob o /tmp...
if  [ ! -d $MeuDir ]
then
    mkdir $MeuDir      # Vou criá-lo
fi

# Se não posso gravar no diretório...
if  [ ! -w $MeuDir ]
then
    echo "Impossivel salvar arquivos em $MeuDir. ➥
Mude as permissões..."
    exit 2
fi

# Variável que indica o cod. de retorno do programa
Erro=0
# Um for sem o in recebe os parametros passados
for Arq
do
# Se este arquivo não existir...
    if  [ ! -f $Arq ]
    then
        echo "$Arq nao existe."
        Erro=3
        continue       # Volta para o comando for
    fi

# Cmd. dirname informa nome do dir de $Arq
    DirOrig=`dirname $Arq`
# Verifica permissão de gravacao no diretório
    if  [ ! -w $DirOrig ]
    then
        echo "Sem permissão no diretorio de $Arq"
        Erro=4
        continue         # Volta para o comando for
    fi

# Se estou "esvaziando a lixeira"...
    if  [ "$DirOrig" = "$MeuDir" ]
    then
        echo "$Arq ficara sem copia de seguranca"
        rm -i $Arq      # Pergunta antes de remover
        # Será que o usuário removeu?
        [ -f $Arq ] || echo "$Arquivo removido"
        continue
    fi

# Guardo no fim do arquivo o seu diretório original➥
para usá-lo em um script de undelete
    cd $DirOrig
    pwd >> $Arq
    mv $Arq $MeuDir  # Salvo e removo
    echo "$Arq removido"
done

# Passo eventual número do erro para o código
# de retorno
exit $Erro
```

um *continue*, para que a seqüência volte para o loop do *for* de forma a receber outros arquivos.

Quando você está no Windows (com perdão da má palavra) e tenta remover aquele monte de lixo com nomes esquisitos como HD04TG.TMP, se der erro em um dos arquivos os outros não são removidos, não é? Então, o *continue* foi usado para evitar que uma impropriedade dessas ocorra, isto é, mesmo que dê erro na remoção de um arquivo, o programa continuará removendo os outros que foram passados.

– Eu acho que a esta altura você deve estar curioso para ver o programa que restaura o arquivo removido, não é? Pois então aí vai vai um desafio: faça-o em casa e me traga para discutirmos no nosso próximo encontro aqui no boteco.
– Poxa, mas nesse eu acho que vou dançar, pois não sei nem como começar...
– Cara, este programa é como tudo o que se faz em Shell: extremamente fácil. É para ser feito em, no máximo, 10 linhas. Não se esqueça de que o arquivo está salvo em */tmp/$LOGNAME* e que sua última linha é o diretório em que ele residia antes de ser "removido". Também não se esqueça de criticar se foi passado o nome do arquivo a ser removido.
– É eu vou tentar, mas sei não...
– Tenha fé, irmão, eu tô te falando que é mole! Qualquer dúvida é só passar um email para julio.neves@gmail.com. Agora chega de papo que eu já estou de goela seca de tanto falar. Me acompanha no próximo chope ou já vai sair correndo para fazer o script que passei?
– Deixa eu pensar um pouco...
– Chico, traz mais um chope enquanto ele pensa!

SOBRE O AUTOR

Julio Cezar Neves é Analista de Suporte de Sistemas desde 1969 e trabalha com Unix desde 1980, quando participou do desenvolvimento do SOX, um sistema operacional similar ao Unix produzido pela Cobra Computadores. Pode ser contatado no e-mail julio.neves@gmail.com