

Curso de Shell Script

Papo de Botequim - Parte II

Nossos personagens voltam à mesa do bar para discutir expressões regulares e colocar a “mão na massa” pela primeira vez, construindo um aplicativo simples para catalogar uma coleção de CDs. **POR JÚLIO CÉSAR NEVES**

Garçom! Traz um “chops” e dois “pastel”. O meu amigo hoje não vai beber porque está finalmente sendo apresentado a um verdadeiro sistema operacional, e ainda tem muita coisa a aprender!

- E então, amigo, tá entendendo tudo que te expliquei até agora?
- Entendendo eu tô, mas não vi nada prático nisso...
- Calma rapaz, o que te falei até agora serve como base ao que há de vir daqui pra frente. Vamos usar essas ferramentas que vimos para montar programas estruturados. Você verá porque até na TV já teve programa chamado “O Shell é o Limite”. Para começar vamos falar dos comandos da família *grep*
- *Grep*? Não conheço nenhum termo em inglês com este nome...
- É claro, *grep* é um acrônimo (sigla) para *Global Regular Expression Print*, que usa expressões regulares para pesquisar a ocorrência de cadeias de caracteres na entrada definida.

Por falar em expressões regulares (ou *regex*), o Aurélio Marinho Jargas escreveu dois artigos [1 e 2] imperdíveis para a Revista do Linux sobre esse assunto e também publicou um livro [3] pela Editora Novatec. Acho bom você ler esses artigos, eles vão te ajudar no que está para vir.

Eu fico com *grep*, você com gripe

Esse negócio de gripe é brincadeira, só um pretexto para pedir umas caipirinhas. Eu te falei que o *grep* procura cadeias de caracteres dentro de uma entrada definida, mas o que vem a ser uma “entrada definida”? Bem, existem várias formas de definir a entrada do comando *grep*. Veja só. Para pesquisar em um arquivo:



```
$ grep franklin /etc/passwd
```

Pesquisando em vários arquivos:

```
$ grep grep *.sh
```

Pesquisando na saída de um comando:

```
$ who | grep carvalho
```

No 1º exemplo, procurei a palavra *franklin* em qualquer lugar do arquivo */etc/passwd*. Se quisesse procurar um nome de usuário, isto é, somente no início dos registros desse arquivo, poderia digitar `$ grep '^franklin' /etc/passwd`.

“E para que servem o circunflexo e os apóstrofos?”, você vai me perguntar. Se tivesse lido os artigos que mencionei, saberia que o circunflexo serve para limitar a pesquisa ao início de cada linha e os apóstrofos servem para o Shell não interpretar esse circunflexo, deixando-o passar incólume para o comando *grep*.

No 2º exemplo mandei listar todas as

linhas que usavam a palavra *grep*, em todos os arquivos terminados em *.sh*. Como uso essa extensão para definir meus arquivos com programas em Shell, malandramente, o que fiz foi listar as linhas dos programas que poderia usar como exemplo do comando *grep*.

Olha que legal! O *grep* aceita como entrada a saída de outro comando, redirecionado por um *pipe* (isso é muito comum em Shell e é um tremendo acelerador da execução de comandos). Dessa forma, no 3º exemplo, o comando *who* listou as pessoas “logadas” na mesma máquina que você (não se esqueça jamais: o Linux é multiusuário) e o *grep* foi usado para verificar se o Carvalho estava trabalhando ou “coçando”.

O *grep* é um comando muito conhecido, pois é usado com muita frequência. O que muitas pessoas não sabem é que existem três comandos na família *grep*: *grep*, *egrep* e *fgrep*. A principais diferenças entre os 3 são:

- *grep* - Pode ou não usar expressões regulares simples, porém no caso de não usá-las, o *fgrep* é melhor, por ser mais rápido.
- *egrep* (“e” de *extended*, estendido) - É muito poderoso no uso de expressões regulares. Por ser o mais poderoso dos três, só deve ser usado quando for necessária a elaboração de uma expressão regular não aceita pelo *grep*.
- *fgrep* (“f” de *fast*, rápido) - Como o nome diz, é o ligeirinho da família, executando o serviço de forma muito veloz (por vezes é cerca de 30% mais rápido que o *grep* e 50% mais que o *egrep*), porém não permite o uso de expressões regulares na pesquisa.

- Agora que você já conhece as diferenças entre os membros da família, me diga: o que você acha dos três exemplos que eu dei antes das explicações?

- Achei que o *fgrep* resolveria o teu problema mais rapidamente que o *grep*.
- Perfeito! Tô vendo que você está atento, entendendo tudo que estou te explicando! Vamos ver mais exemplos

Quadro 1 - Listando subdiretórios

```
$ ls -l | grep '^d'
```

drwxr-xr-x	3	root	root	4096	Dec 18	2000	doc
drwxr-xr-x	11	root	root	4096	Jul 13	18:58	freeciv
drwxr-xr-x	3	root	root	4096	Oct 17	2000	gimp
drwxr-xr-x	3	root	root	4096	Aug 8	2000	gnome
drwxr-xr-x	2	root	root	4096	Aug 8	2000	idl
drwxrwxr-x	14	root	root	4096	Jul 13	18:58	locale
drwxrwxr-x	12	root	root	4096	Jan 14	2000	lyx
drwxrwxr-x	3	root	root	4096	Jan 17	2000	pixmaps
drwxr-xr-x	3	root	root	4096	Jul 2	20:30	scribus
drwxrwxr-x	3	root	root	4096	Jan 17	2000	sounds
drwxr-xr-x	3	root	root	4096	Dec 18	2000	xine
drwxr-xr-x	3	root	root	4096	Jun 19	2000	xpls

para clarear de vez as diferenças de uso entre os membros da família.

Eu sei que em um arquivo qualquer existe um texto falando sobre Linux, só não tenho certeza se está escrito com *L* maiúsculo ou minúsculo. Posso fazer uma busca de duas formas:

```
egrep (Linux | linux) arquivo.txt
```

ou então:

```
grep [Ll]linux arquivo.txt
```

No primeiro caso, a expressão regular complexa (*Linux* | *linux*) usa os parênteses para agrupar as opções e a barra vertical (|) é usada como um “ou” (*or*, em inglês) lógico, isto é, estou procurando *Linux* ou *linux*.

No segundo, a expressão regular *[Ll]linux* significa: começado por *L* ou *l* seguido de *linux*. Como esta é uma expressão simples, o *grep* consegue resolvê-la, por isso é melhor usar a segunda forma, já que o *egrep* tornaria a pesquisa mais lenta.

Outro exemplo. Para listar todos os subdiretórios do diretório corrente, basta usar o comando `$ ls -l | grep '^d'`. Veja o resultado no Quadro 1.

No exemplo, o circunflexo (^) serviu para limitar a pesquisa à primeira posição da saída do *ls* longo (parâmetro

-l). Os apóstrofos foram usados para o Shell não “ver” o circunflexo. Vamos ver mais um. Veja na Tabela 1 as quatro primeiras posições possíveis da saída de um *ls -l* em um arquivo comum (não é diretório, nem link, nem ...).

Para descobrir todos os arquivos executáveis em um determinado diretório eu poderia fazer:

```
$ ls -la | egrep '^-..(x|s)'
```

novamente usamos o circunflexo para limitar a pesquisa ao início de cada linha, ou seja, listamos as linhas que começam por um traço (-), seguido de qualquer coisa (o ponto), novamente seguido de qualquer coisa, e por fim um *x* ou um *s*. Obteríamos o mesmo resultado se usássemos o comando:

```
$ ls -la | grep '^-..[xs]'
```

e além disso, agilizaríamos a pesquisa.

A “CDteca”

Vamos começar a desenvolver programas! Creio que a montagem de um banco de dados de músicas é bacana para efeito didático (e útil nestes tempos de downloads de arquivos MP3 e queimadores de CDs). Não se esqueça que, da mesma forma que vamos desenvolver um monte de programas para organizar os seus CDs de música, com pequenas adaptações você pode fazer o mesmo para organizar os CDs de software que vêm com a Linux Magazine e outros que você compra ou queima, e disponibilizar esse banco de software para todos os que trabalham com você (o Linux é multiusuário, e como tal deve

ser explorado).

– Péra aí! De onde eu vou receber os dados dos CDs?

– Vou mostrar como o programa pode receber parâmetros de quem o estiver executando e, em breve, ensinarei a ler os dados da tela ou de um arquivo.

Passando parâmetros

Veja abaixo a estrutura do arquivo contendo a lista das músicas:

```
nomedoálbum^intérprete1~nome?
damúsical:...:intérpreten~nome?
damúsican
```

Isto é, o nome do álbum será separado por um circunflexo do resto do registro, formado por diversos grupos compostos pelo intérprete de cada música do CD e a música interpretada. Estes grupos são separados entre si por dois pontos (:) e, internamente, o intérprete será separado por um til (~) do nome da música.

Quero escrever um programa chamado *musinc*, que incluirá registros no meu arquivo músicas. Passarei cada álbum como parâmetro para o programa:

```
$ musinc “álbum^interprete~?
musica:interprete~musica:...”
```

Desta forma, *musinc* estará recebendo os dados de cada álbum como se fosse uma variável. A única diferença entre um parâmetro recebido e uma variável é que os primeiros recebem nomes numéricos (o que quis dizer é que seus nomes são formados somente por um algarismo, isto é, \$1, \$2, \$3, ..., \$9). Vamos, fazer mais alguns testes:

```
$ cat teste
#!/bin/bash
#Teste de passagem de parametros
echo “1o. parm -> $1”
echo “2o. parm -> $2”
echo “3o. parm -> $3”
```

Agora vamos rodar esse programinha:

```
$ teste passando parametros para ?
testar
bash: teste: cannot execute
```

Ops! Esqueci-me de tornar o script executável. Vou fazer isso e testar novamente o programa:

Tabela 1

Posição	Valores possíveis
1ª	-
2ª	r ou -
3ª	w ou -
4ª	x,s(suid) ou -

```
$ chmod 755 teste
$ teste passando parametros para 3
testar
1o. parm -> passando
2o. parm -> parametros
3o. parm -> para
```

Repare que a palavra *testar*, que seria o quarto parâmetro, não foi listada. Isso ocorreu porque o programa *teste* só lista os três primeiros parâmetros recebidos. Vamos executá-lo de outra forma:

```
$ teste "passando parametros" 3
para testar
1o. parm -> passando parametros
2o. parm -> para
3o. parm -> testar
```

As aspas não deixaram o Shell ver o espaço em branco entre as duas primeiras palavras, e elas foram consideradas como um único parâmetro. E falando em passagem de parâmetros, uma dica: veja na Tabela 2 algumas variáveis especiais. Vamos alterar o programa *teste* para usar as novas variáveis:

```
$ cat teste
#!/bin/bash
# Programa para testar passagem
de parametros (2a. Versao)
echo 0 programa $0 recebeu $#
parametros
echo "1o. parm -> $1"
echo "2o. parm -> $2"
echo "3o. parm -> $3"
echo Para listar todos de uma
\"tacada\" eu faco $*
```

Listagem 1: Incluindo CDs na "CDteca"

```
$ cat musinc
#!/bin/bash
# Cadastra CDs (versao 1)
#
echo $1 >> musicas
```

Listagem 2

```
$ cat musinc
#!/bin/bash
# Cadastra CDs (versao 2)
#
echo $1 >> musicas
sort -o musicas musicas
```

Execute o programa:

```
$ teste passando parametros para testar
0 programa teste recebeu 4
parametros
1o. parm -> passando
2o. parm -> parametros
3o. parm -> para
Para listar todos de uma
"tacada" eu faco passando
parametros para testar
```

Repare que antes das aspas usei uma barra invertida, para escondê-las da interpretação do Shell (se não usasse as contrabarras as aspas não apareceriam).

Como disse, os parâmetros recebem números de 1 a 9, mas isso não significa que não posso usar mais de nove parâmetros. Significa que só posso endereçar nove. Vamos testar isso:

```
$ cat teste
#!/bin/bash
# Programa para testar passagem
de parametros (3a. Versao)
echo 0 programa $0 recebeu $#
parametros
echo "1lo. parm -> $11"
shift
echo "2o. parm -> $1"
shift 2
echo "4o. parm -> $1"
```

Execute o programa:

```
$ teste passando parametros para 3
testar
0 programa teste recebeu 4
parametros que são:
1lo. parm -> passando1
2o. parm -> parametros
4o. parm -> testar
```

Duas coisas muito interessantes aconteceram neste script. Para mostrar que os nomes dos parâmetros variam de \$1 a \$9 digitei *echo \$11* e o que aconteceu? O Shell interpretou como sendo \$1 seguido do algarismo 1 e listou *passando1*;

O comando *shift*, cuja sintaxe é *shift n*, podendo o *n* assumir qualquer valor numérico, despreza os *n* primeiros parâmetros, tornando o parâmetro de ordem *n + 1*.

Bem, agora que você já sabe sobre passagem de parâmetros, vamos voltar à nossa "cdteca" para fazer o script de

inclusão de CDs no meu banco chamado *musicas*. O programa é muito simples (como tudo em Shell). Veja a Listagem 1.

O script é simples e funcional; limito-me a anexar ao fim do arquivo *musicas* o parâmetro recebido. Vamos cadastrar 3 álbuns para ver se funciona (para não ficar "enchendo lingüiça," suponho que em cada CD só existem duas músicas):

```
$ musinc "album3^Artista5~
~Musica5:Artista6~Musica5"
$ musinc "album1^Artista1~
~Musical:Artista2~Musica2"
$ musinc "album 2^Artista3~
~Musica3:Artista4~Musica4"
```

Listando o conteúdo do arquivo *musicas*:

```
$ cat musicas
album3^Artista5~Musica5:Artista6~
~Musica6
album1^Artista1~Musical:Artista2~
~Musica2
album2^Artista3~Musica3:Artista4~
~Musica4
```

Podia ter ficado melhor. Os álbuns estão fora de ordem, dificultando a pesquisa. Vamos alterar nosso script e depois testá-lo novamente. Veja a listagem 2. Simplesmente inseri uma linha que classifica o arquivo *musicas*, redirecionando a saída para ele mesmo (para isso serve a opção *-o*), após cada álbum ser anexado.

```
$ cat musicas
album1^Artista1~Musical:Artista2~
~Musica2
albu2^Artista3~Musica3:Artista4~
~Musica4
album3^Artista5~Musica5:Artista6~
~Musica6
```

Oba! Agora o programa está legal e quase funcional. Ficará muito melhor em uma nova versão, que desenvolveremos após aprender a adquirir os dados da tela e formatar a entrada.

Tabela 2: Variáveis especiais

Variável	Significado
\$0	Contém o nome do programa
\$#	Contém a quantidade de parâmetros passados
\$*	Contém o conjunto de todos os parâmetros (muito parecido com @\$)

Ficar listando arquivos com o comando *cat* não está com nada, vamos fazer um programa chamado *muslist* para listar um álbum, cujo nome será passado como parâmetro. Veja o código na Listagem 3:

Vamos executá-lo, procurando pelo álbum 2. Como já vimos antes, para passar a string *album 2* é necessário protegê-la da interpretação do Shell, para que ele não a interprete como dois parâmetros. Exemplo:

```
$ muslist "album 2"
grep: can't open 2
musicas: album1^Artista1~Musical2
:Artista2~Musica2
musicas: album2^Artista3~Musica3
:Artista4~Musica4
musicas:album3^Artista5~Musica5
:Artista6~Musica6
```

Que lambança! Onde está o erro? Eu tive o cuidado de colocar o parâmetro passado entre aspas para o Shell não o dividir em dois! É, mas repare como o *grep* está sendo executado:

```
grep $1 musicas
```

Mesmo colocando *álbum 2* entre aspas, para que fosse encarado como um único parâmetro, quando o *\$1* foi passado pelo Shell para o comando *grep*, transformouse em dois argumentos. Dessa forma, o conteúdo da linha que o *grep* executou foi o seguinte:

```
grep album 2 musicas
```

Como a sintaxe do *grep* é:

Listagem 3 - muslist

```
$ cat muslist
#!/bin/bash
# Consulta CDs (versao 1)
#
grep $1 musicas
```

Listagem 4 muslist melhorado

```
$ cat muslist
#!/bin/bash
# Consulta CDs (versao 2)
#
grep -i "$1" musicas
```

```
grep <cadeia de caracteres> [arq1, arq2, ..., arqn]
```

O *grep* entendeu que deveria procurar a cadeia de caracteres *album* nos arquivos 2 e *musicas*. Como o arquivo 2 não existe, *grep* gerou o erro e, por encontrar a palavra *album* em todos os registros de *musicas*, listou a todos.

É melhor ignorarmos maiúsculas e minúsculas na pesquisa. Resolveremos os dois problemas com a Listagem 4.

Nesse caso, usamos a opção *-i* do *grep* que, como já vimos, serve para ignorar maiúsculas e minúsculas, e colocamos o *\$1* entre aspas, para que o *grep* continuasse a ver a cadeia de caracteres resultante da expansão da linha pelo Shell como um único argumento de pesquisa.

```
$ muslist "album 2"
album2^Artista3~Musica3:Artista4
~Musica4
```

Agora repare que o *grep* localiza a cadeia pesquisada em qualquer lugar do registro; então, da forma que estamos fazendo, podemos pesquisar por álbum, por música, por intérprete e mais. Quando conhecermos os comandos condicionais, montaremos uma nova versão de *muslist* que permitirá especificar por qual campo pesquisar.

Ah! Em um dia de verão você foi à praia, esqueceu os CDs no carro, aquele "solzinho" de 40 graus empenou seu disco favorito e agora você precisa de uma ferramenta para removê-lo do banco de dados? Não tem problema, vamos desenvolver um script chamado *musexc*, para excluir estes CDs.

Antes de desenvolver o "bacalho", quero te apresentar a uma opção bastante útil da família de comandos *grep*. É a opção *-v*, que quando usada lista todos os registros da entrada, exceto o(s) localizado(s) pelo comando. Exemplos:

```
$ grep -v "album 2" musicas
album1^Artista1~Musical:Artista2
~Musica2
album3^Artista5~Musica5:Artista6
~Musica6
```

Conforme expliquei antes, o *grep* do exemplo listou todos os registros de *musicas* exceto o referente a *album 2*, porque atendia ao argumento do co-

Listagem 5 - musexc

```
$ cat musexc
#!/bin/bash
# Exclui CDs (versao 1)
#
grep -v "$1" musicas > /tmp/mus$$
mv -f /tmp/mus$$ musicas
```

mando. Estamos então prontos para desenvolver o script para remover CDs empenados da sua "CDteca". Veja o código da Listagem 5.

Na primeira linha mandei para */tmp/mus\$\$* o arquivo *musicas*, sem os registros que atendessem a consulta feita pelo comando *grep*. Em seguida, movi */tmp/mus\$\$* por cima do antigo *musicas*. Usei o arquivo */tmp/mus\$\$* como arquivo de trabalho porque, como já havia citado no artigo anterior, o *\$\$* contém o PID (identificação do processo) e, dessa forma, cada um que editar o arquivo *musicas* o fará em um arquivo de trabalho diferente, evitando colisões.

Os programas que fizemos até aqui ainda são muito simples, devido à falta de ferramentas que ainda temos. Mas é bom praticar os exemplos dados porque, eu prometo, chegaremos a desenvolver um sistema bacana para controle dos seus CDs. Na próxima vez que nos encontrarmos, vou te ensinar como funcionam os comandos condicionais e aprimoraremos mais um pouco esses scripts. Por hoje chega! Já falei demais e estou de goela seca! Garçom! Mais um sem colarinho! ■

INFORMAÇÕES

- [1] <http://www.revistadoinux.com.br/ed/003/ferramentas.php3>
- [2] <http://www.revistadoinux.com.br/ed/007/ereg.php3>
- [3] <http://www.aurelio.net/er/livro/>

SOBRE O AUTOR

Julio Cezar Neves é Analista de Suporte de Sistemas desde 1969 e trabalha com Unix desde 1980, quando fez parte da equipe que desenvolveu o SOX, um sistema operacional similar ao Unix, produzido pela Cobra Computadores.

