Curso de Shell Script

Papo de botequim III

Um chopinho, um aperitivo e o papo continua. Desta vez vamos aprender alguns comandos de manipulação de cadeias de caracteres, que serão muito úteis na hora de incrementar nossa "CDteca". POR JULIO CEZAR NEVES

Garçon! traga dois chopes por favor que hoje eu vou ter que falar muito. Primeiro quero mostrar uns programinhas simples de usar e muito úteis, como o *cut*, que é usado para cortar um determinado pedaço de um arquivo. A sintaxe e alguns exemplos de uso podem ser vistos no Quadro 1:

Como dá para ver, existem quatro sintaxes distintas: na primeira (*-c 1-5*) especifiquei uma faixa, na segunda (*-c -6*) especifiquei todo o texto até uma posição, na terceira (*-c 4-*) tudo de uma determinada posição em diante e na quarta (*-c 1,3,5,7,9*), só as posições determinadas. A última possibilidade (*-c -3,5,8-*) foi só para mostrar que podemos misturar tudo.

Mas não pense que acabou por aí! Como você deve ter percebido, esta forma de cut é muito útil para lidar com arquivos com campos de tamanho fixo, mas atualmente o que mais existe são arquivos com campos de tamanho variável, onde cada campo termina com um delimitador. Vamos dar uma olhada no arquivo *musicas* que começamos a preparar na última vez que viemos aqui no botequim. Veja o Quadro 2.

Então, recapitulando, o layout do arquivo é o seguinte: *nome do álbum^intérprete1~nome da música1:...: intérpreten~nome da músican*, isto é, o nome do álbum será separado por um circunflexo (^) do resto do registro, que é formado por diversos grupos compostos pelo intérprete de cada música do CD e a respectiva música interpretada. Estes grupos são separados entre si por dois-pontos (:) e o intérprete será separado do nome da música por um til (~).

Então, para pegar os dados referentes a todas as segundas músicas do arquivo musicas, devemos digitar:

```
$ cut -f2 -d: musicas
Artista2~Musica2
Artista4~Musica4
Artista6~Musica5
Artista8~Musica8@10_L:
```

Ou seja, cortamos o segundo campo z(*-f* de *field*, campo em inglês) delimitado (*-d*) por dois-pontos (*:*). Mas, se quisermos somente os intérpretes, devemos digitar:

```
$ cut -f2 -d: musicas | cut ➥
-f1 -d~
Artista2
Artista4
Artista6
Artista8
```

Para entender melhor isso, vamos analisar a primeira linha de musicas:

```
$ head -1 musicas
album 1^Artista1~Musica1: ➥
Artista2~Musica2
```

Então observe o que foi feito:

```
album 1^Artista1~Musica1: ➥
Artista2~Musica2
```

Desta forma, no primeiro cut o primeiro campo do delimitador (*-d*) dois-pontos (*:*) é *album 1^Artista1~Musica1* e o segundo, que é o que nos interessa, é *Artista2~Musica2*. Vamos então ver o que aconteceu no segundo cut:

```
Artista2~Musica2
```

Agora, primeiro campo do delimitador (*-d*) til (~), que é o que nos interessa, é *Artista2* e o segundo é *Musica2*. Se o raciocínio que fizemos para a pri-

Quadro 1 – O comando cut

A sintaxe do cut é: *cut -c PosIni-PosFim [arquivo]*, onde *PosIni* é a posição inicial, e *PosFim* a posição final. Veja os exemplos:

```
$ cat numeros
1234567890
0987654321
1234554321
9876556789

$ cut -c1-5 numeros
12345
09876
12345
98765

$ cut -c-6 numeros
123456
098765
123455
987655

$ cut -c4- numeros
4567890
7654321
4554321
6556789

$ cut -c1,3,5,7,9 numeros
13579
08642
13542
97568

$ cut -c -3,5,8- numeros
1235890
0986321
1235321
9875789
```

Quadro 2 – O arquivo *musicas*

```
$ cat musicas
album 1^Artista1~Musica1:➥
Artista2~Musica2
album 2^Artista3~Musica3:➥
Artista4~Musica4
album 3^Artista5~Musica5:➥
Artista6~Musica5
album 4^Artista7~Musica7:➥
Artista8~Musica8
```

meira linha for aplicado ao restante do arquivo, chegaremos à resposta anteriormente dada. Outro comando muito interessante é o *tr* que serve para substituir, comprimir ou remover caracteres. Sua sintaxe segue o seguinte padrão:

```
tr [opções] cadeia1 [cadeia2]
```

O comando copia o texto da entrada padrão (*stdin*), troca as ocorrência dos caracteres de *cadeia1* pelo seu correspondente na *cadeia2* ou troca múltiplas ocorrências dos caracteres de *cadeia1* por somente um caracter, ou ainda caracteres da *cadeia1*. As principais opções do comando são mostradas na Tabela 1.

Primeiro veja um exemplo bem bobo:

```
$ echo bobo | tr o a
baba
```

Isto é, troquei todas as ocorrências da letra *o* pela letra *a*. Suponha que em determinado ponto do meu script eu peça ao operador para digitar *s* ou *n* (sim ou não), e guardo sua resposta na variável *$Resp*. Ora, o conteúdo de *$Resp* pode conter letras maiúsculas ou minúsculas, e desta forma eu teria que fazer diversos testes para saber se a resposta dada foi *S, s, N* ou *n*. Então o melhor é fazer:

```
$ Resp=$(echo $Resp | tr SN sn)
```

e após este comando eu teria certeza de que o conteúdo de *$Resp* seria um *s* ou um *n*. Se o meu arquivo *ArqEnt* está todo em letras maiúsculas e desejo passá-las para minúsculas eu faço:

```
$ tr A-Z a-z < ArqEnt > / tmp/$$
$ mv -f /tmp/$$ ArqEnt
```

Note que neste caso usei a notação A-Z para não escrever ABCD...YZ. Outro tipo de notação que pode ser usada são as *escape sequences* (como eu traduziria? Seqüências de escape? Meio sem sentido, né? Mas vá lá...) que também são reconhecidas por outros comandos e também na linguagem C, e cujo significado você verá na Tabela 2:

Deixa eu te contar um "causo": um aluno que estava danado comigo resolveu complicar minha vida e como resposta a um exercício prático, valendo nota, que passei ele me entregou um script com todos os comandos separados por ponto-e-vírgula (lembre-se que o ponto-e-vírgula serve para separar diversos comandos em uma mesma linha). Vou dar um exemplo simplificado, e idiota, de um script assim:

```
$ cat confuso
echo leia Programação Shell ➥
Linux do Julio Cezar Neves ➥
> livro;cat livro;pwd;ls;rm ➥
-f livro2>/dev/null;cd ~
```

Eu executei o programa e ele funcionou:

```
$ confuso
leia Programação Shell Linux ➥
do Julio Cezar Neves
/home/jneves/LM
confuso  livro  musexc  musicas
musinc  muslist  numeros
```

Mas nota de prova é coisa séria (e nota de dólar é mais ainda) então, para entender o que o aluno havia feito, o chamei e em sua frente digitei:

```
$ tr ";" "\n" < confuso
echo leia Programação Shell ➥
Linux do Julio Cezar Neves
pwd
cd ~
ls -l
rm -f lixo 2>/dev/null
```

O cara ficou muito desapontado, porque em dois ou três segundos eu desfiz a gozação que ele perdeu horas para fazer. Mas preste atenção! Se eu estivesse em uma máquina Unix, eu teria digitado:

```
$ tr ";" "\012" < confuso
```

Agora veja a diferença entre o resultado de um comando date executado hoje e outro executado há duas semanas:

```
Sun Sep 19 14:59:54  2004
Sun Sep  5 10:12:33  2004
```

Notou o espaço extra após o "Sep" na segunda linha? Para pegar a hora eu deveria digitar:

```
$ date | cut -f 4 -d ' '
14:59:54
```

Mas há duas semanas ocorreria o seguinte:

```
$ date | cut -f 4 -d ' '
5
```

Isto porque existem 2 caracteres em branco antes do 5 (dia). Então o ideal seria transformar os espaços em branco consecutivos em somente um espaço para poder tratar os dois resultados do comando *date* da mesma forma, e isso se faz assim:

```
$ date | tr -s " "
Sun Sep 5 10:12:33 2004
```

E agora eu posso cortar:

```
$ date | tr -s " " | cut -f 4 ➥
-d " "
10:12:33
```

Olha só como o Shell está quebrando o galho. Veja o conteúdo de um arquivo baixado de uma máquina Windows:

```
$  cat -ve ArqDoDOS.txt
Este arquivo^M$
foi gerado pelo^M$
DOS/Win e foi^M$
baixado por um^M$
ftp mal feito.^M$
```

Dica: a opção *-v* do cat mostra os caracteres de controle invisíveis, com a notação ^*L*, onde ^ é a tecla *Control* e *L* é a respectiva letra. A opção *-e* mostra o final da linha como um cifrão (*$*).

Isto ocorre porque no DOS o fim dos registros é indicado por um *Carriage Return* (\r – Retorno de Carro, CR) e um *Line Feed* (\f – Avanço de Linha, ou LF). No Linux porém o final do registro é indicado somente pelo *Line Feed*. Vamos limpar este arquivo:

```
$ tr -d '\r' < ArqDoDOS.txt > /tmp/$$
$ mv -f /tmp/$$ ArqDoDOS.txt
```

Agora vamos ver o que aconteceu:

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

Bem a opção *-d* do *tr* remove do arquivo todas as ocorrências do caractere especificado. Desta forma eu removi os caracteres indesejados, salvei o texto em um arquivo temporário e posteriormente renomeei-o para o nome original. Uma observação: em um sistema Unix eu deveria digitar:

```
$ tr -d '\015' < ArqDoDOS.➥
txt > /tmp/$$
```

Uma dica: o problema com os terminadores de linha (CR/LF) só aconteceu porque a transferência do arquivo foi feita no modo binário (ou image), Se antes da transmissão do arquivo tivesse sido estipulada a opção *ascii* do ftp, isto não teria ocorrido.

– Olha, depois desta dica tô começando a gostar deste tal de shell, mas ainda tem muita coisa que não consigo fazer.

– Pois é, ainda não te falei quase nada sobre programação em shell, ainda tem muita coisa para aprender, mas com o que aprendeu, já dá para resolver muitos problemas, desde que você adquira o "modo shell de pensar". Você seria capaz de fazer um script que diga quais pessoas estão "logadas" há mais de um dia no seu servidor?

– Claro que não! Para isso seria necessário eu conhecer os comandos condicionais que você ainda não me explicou como funcionam. - Deixa eu tentar mudar um pouco a sua lógica e trazê-la para o "modo shell de pensar", mas antes é melhor tomarmos um chope.

Agora que já molhei a palavra, vamos resolver o problema que te propus. Veja como funciona o comando *who*:

```
$ who
jneves   pts/➥
1        Sep 18 13:40
rtorres  pts/➥
0        Sep 20 07:01
rlegaria pts/➥
1        Sep 20 08:19
lcarlos  pts/➥
3        Sep 20 10:01
```

Tabela 1 – O comando tr

Opção	Significado
-s	Comprime n ocorrências de cadeia1 em apenas uma
-d	Remove os caracteres de cadeia1

E veja também o date:

```
$ date
Mon Sep 20 10:47:19 BRT 2004
```

Repare que o mês e o dia estão no mesmo formato em ambos os comandos. Ora, se em algum registro do *who* eu não encontrar a data de hoje, é sinal que o usuário está "logado" há mais de um dia, já que ele não pode ter se "logado" amanhã... Então vamos guardar o pedaço que importa da data de hoje para depois procurá-la na saída do comando *who*:

```
$ Data=$(date | cut -f 2-3 ➥
-d' ')
```

Eu usei a construção *$(...)*, para priorizar a execução dos comandos antes de atribuir a sua saída à variável *Data*. Vamos ver se funcionou:

```
$ echo $Data
Sep 20
```

Beleza! Agora, o que temos que fazer é procurar no comando who os registros que não possuem esta data.

– Ah! Eu acho que estou entendendo! Você falou em procurar e me ocorreu o comando grep, estou certo?

– Certíssimo! Só que eu tenho que usar o grep com aquela opção que ele só lista os registros nos quais ele não encontrou a cadeia. Você se lembra que opção é essa?

– Claro, a *-v*...

– Isso! Tá ficando bom! Vamos ver:

```
$  who | grep -v "$Data"
jneves   pts/➥
1        Sep 18 13:40
```

Se eu quisesse um pouco mais de perfumaria eu faria assim:

```
$  who | grep -v "$Data" |➥
 cut -f1 -d ' '
jneves
```

Viu? Não foi necessário usar comando condicional, até porque o nosso comando condicional, o famoso if, não testa condição, mas sim instruções. Mas antes veja isso:

```
$ ls musicas
musicas
$ echo $?
0
$ ls ArqInexistente
ls: ArqInexistente: No such ➥
file or directory
$ echo $?
1
$ who | grep jneves
jneves   pts/1        Sep 18 ➥
13:40 (10.2.4.144)
$ echo $?
0
$ who | grep juliana
$ echo $?
1
```

O que é que esse *$?* faz aí? Algo começado por cifrão ($) parece ser uma variável, certo? Sim é uma variável que contém o código de retorno da última instrução executada. Posso te garantir que se esta instrução foi bem sucedida, *$?* terá o valor zero, caso contrário seu valor será diferente de zero. O que nosso comando condicional (if) faz é testar esta variável. Então vamos ver a sua sintaxe:

```
if cmd
then
    cmd1
    cmd2
    cmdn
else
    cmd3
    cmd4
    cmdm
fi
```

Ou seja, caso comando *cmd* tenha sido executado com sucesso, os comandos do bloco *do then* (*cmd1, cmd2 e cmdn*) serão executados, caso contrário, os comandos do bloco opcional *do else* (*cmd3, cmd4 e cmdm*) serão executados. O bloco do *if* é terminando com um *fi*.

Vamos ver na prática como isso funciona, usando um script que inclui usuários no arquivo */etc/passwd*:

```
$ cat incusu
#!/bin/bash
# Versão 1
if  grep ^$1 /etc/passwd
then
    echo Usuario \'$1\'➥
```

Tabela 2

Seqüência	Significado	Octal
\t	Tabulação	\011
\n	Nova linha <ENTER>	\012
\v	Tabulação Vertical	\013
\f	Nova Página	\014
\r	Início da linha <^M>	\015
\\	Uma barra invertida	\0134

```
    já existe
else
    if  useradd $1
    then
        echo Usuário \'$1\' ➥
incluído em /etc/passwd
    else
        echo "Problemas no ➥
cadastramento. Você é root?"
    fi
fi
```

Repare que o *if* está testando direto o comando *grep* e esta é a sua finalidade. Caso o *if* seja bem sucedido, ou seja, o usuário (cujo nome está em *$1*) foi encontrado em */etc/passwd*, os comandos do bloco *do then* serão executados (neste exemplo, apenas o echo). Caso contrário, as instruções do bloco *do else* serão executadas, quando um novo *if* testa se o comando *useradd* foi executado a contento, criando o registro do usuário em */etc/passwd*, ou exibindo uma mensagem de erro, caso contrário.

Executar o programa e passe como parâmetro um usuário já cadastrado:

```
$ incusu jneves
jneves:x:54002:1001:Julio Neves:➥
/home/jneves:/bin/➥
bash
Usuario 'jneves' ja existe
```

No exemplo dado, surgiu uma linha indesejada, ela é a saída do comando grep. Para evitar que isso aconteça, devemos desviar a saída para */dev/null*. O programa fica assim:

```
$ cat incusu
#!/bin/bash
# Versão 2
if  grep ^$1 /etc/passwd > ➥
/dev/null
then
    echo Usuario \'$1\' já ➥
existe
else
    if  useradd $1
    then
        echo Usuário \'$1\' ➥
incluído em /etc/passwd
    else
        echo "Problemas no ➥
cadastramento. Você é root?"
    fi
fi
```

Vamos testá-lo como um usuário normal :

```
$ incusu ZeNinguem
./incusu[6]: useradd: not found
 Problemas no cadastramento. ➥
 Você é root?
```

Aquela mensagem de erro não deveria aparecer! Para evitar isso, devemos redirecionar a saída de erro (*stderr*) do comando *useradd* para */dev/null*. A versão final fica assim:

```
$ cat incusu
#!/bin/bash
# Versão 3
if  grep ^$1 /etc/passwd > ➥
/dev/null
then
    echo Usuario \'$1\' já ➥
existe
else
    if  useradd $1 2> /dev/null
    then
        echo Usuário \'$1\' ➥
incluído em /etc/passwd
    else
        echo "Problemas no ➥
cadastramento. Você é root?"
    fi
fi
```

Depois disso, vejamos o comportamento do programa, se executado pelo root:

```
$ incusu botelho
Usuário 'botelho' incluido em ➥
/etc/passwd
```

E novamente:

```
$ incusu botelho
Usuário 'botelho' já existe
```

Lembra que eu falei que ao longo dos nossos papos e chopes os nossos programas iriam se aprimorando? Então vejamos agora como podemos melhorar o nosso programa para incluir músicas na "CDTeca":

```
$ cat musinc
#!/bin/bash
# Cadastra CDs (versao 3)
#
if  grep "^$1$" musicas > ➥
/dev/null
then
    echo Este álbum já está ➥
cadastrado
else
    echo $1 >> musicas
    sort musicas -o musicas
fi
```

Como você viu, é uma pequena evolução em relação à versão anterior. Antes de incluir um registro (que na versão anterior poderia ser duplicado), testamos se o registro começa (^) e termina ($) de forma idêntica ao parâmetro *álbum* passado ($1). O circunflexo (^) no início da cadeia e cifrão ($) no fim, servem para testar se o parâmetro (o álbum e seus dados) é exatamente igual a algum registro já existente. Vamos executar nosso programa novamente, mas desta vez passamos como parâmetro um álbum já cadastrado, pra ver o que acontece:

```
$ musinc "album 4^Artista7~➥
Musica7:Artista8~Musica8"
Este álbum já está cadastrado
```

E agora um não cadastrado:

```
$ musinc "album 5^Artista9~➥
Musica9:Artista10~Musica10"
$ cat musicas
album 1^Artista1~Musica1:➥
Artista2~Musica2
album 2^Artista3~Musica3:➥
Artista4~Musica4
album 3^Artista5~Musica5:➥
Artista6~Musica5
album 4^Artista7~Musica7:➥
Artista8~Musica8
album 5^Artista9~Musica9:➥
Artista10~Musica10
```

Como você viu, o programa melhorou um pouquinho, mas ainda não está pronto. À medida que eu te ensinar a programar em shell, nossa CDteca vai ficar cada vez melhor.

– Entendi tudo que você me explicou, mas ainda não sei como fazer um if para testar condições, ou seja o uso normal do comando.

– Para isso existe o comando test, que testa condições. O comando if testa o comando test. Como já falei muito, preciso de uns chopes para molhar a palavra. Vamos parar por aqui e na próxima vez te explico direitinho o uso do test e de diversas outras sintaxes do if.

– Falou! Acho bom mesmo porque eu também já tô ficando zonzo e assim tenho tempo para praticar esse monte de coisas que você me falou hoje.

Para fixar o que você aprendeu, tente fazer um scriptizinho para informar se um determinado usuário, cujo nome será passado como parâmetro, está "logado" no sistema ou não.

– Aê Chico! traz mais dois chopes pra mim por favor...