Curso de Shell Script

Papo de Botequim

Você não agüenta mais aquele seu amigo usuário de Linux enchendo o seu saco com aquela história de que o sistema é fantástico e o Shell é uma ferramenta maravilhosa? A partir desta edição vai ficar mais fácil entender o porquê deste entusiasmo...

POR JULIO CEZAR NEVES

Diálogo entreouvido em uma mesa de um botequim, entre um usuário de Linux e um empurrador de mouse:

- Quem é o Bash?
- É o filho caçula da família Shell.
- Pô cara! Estás a fim de me deixar maluco? Eu tinha uma dúvida e você me deixa com duas!
- Não, maluco você já é há muito tempo: desde que decidiu usar aquele sistema operacional que você precisa reiniciar dez vezes por dia e ainda por cima não tem domínio nenhum sobre o que esta acontecendo no seu computador. Mas deixa isso prá lá, pois vou te explicar o que é Shell e os componentes de sua família e ao final da nossa conversa você dirá: "Meu Deus do Shell! Porque eu não optei pelo Linux antes?".

O ambiente Linux

Para você entender o que é e como funciona o Shell, primeiro vou te mostrar como funciona o ambiente em camadas do Linux. Dê uma olhada no gráfico mostrado na Figura 1.

Neste gráfico podemos ver que a camada de hardware é a mais profunda e é formada pelos componentes físicos do seu computador. Em torno dela, vem a camada do kernel que é o cerne do Linux, seu núcleo, e é quem põe o hardware para funcionar, fazendo seu gerenciamento e controle. Os programas e comandos que envolvem o kernel, dele se utilizam para realizar as tarefas para que foram desenvolvidos. Fechando tudo isso vem o Shell, que leva este nome porque, em inglês, Shell significa concha, carapaça, isto é, fica entre o usuário e o sistema operacional, de forma que tudo que interage com o sistema operacional, tem que passar pelo seu crivo.

O ambiente Shell

Bom já que para chegar ao núcleo do Linux, no seu kernel que é o que interessa a todo aplicativo, é necessária a filtragem do Shell, vamos entender como ele funciona de forma a tirar o máximo proveito das inúmeras facilidades que ele nos oferece.

O Linux, por definição, é um sistema multiusuário – não podemos nunca nos esquecer disto – e para permitir o acesso de determinados usuários e barrar a entrada de outros, existe um arquivo chamado */etc/passwd*, que além de fornecer dados para esta função de "leão-de-chácara" do Linux, também provê informações para o início de uma sessão (ou "login", para os íntimos) daqueles que passaram por esta primeira barreira. O último campo de seus registros informa ao sistema qual é o Shell que a pessoa vai receber ao iniciar sua sessão.

Lembra que eu te falei de Shell, família, irmão? Pois é, vamos começar a entender isto: o Shell é a conceituação de concha envolvendo o sistema operacional propriamente dito, é o nome genérico para tratar os filhos desta idéia que, ao longo dos muitos anos de exis-

Quadro 1: Uma rapidinha nos principais sabores de Shell

Bourne Shell (sh): Desenvolvido por Stephen Bourne do Bell Labs (da AT&T, onde também foi desenvolvido o Unix), foi durante muitos anos o Shell padrão do sistema operacional Unix. É também chamado de Standard Shell por ter sido durante vários anos o único, e é até hoje o mais utilizado. Foi portado para praticamente todos os ambientes Unix e distribuições Linux.

Korn Shell (ksh): Desenvolvido por David Korn, também do Bell Labs, é um supercon-junto do sh, isto é, possui todas as facilidades do sh e a elas agregou muitas outras. A compatibilidade total com o sh vem trazendo muitos usuários e programadores de Shell para este ambiente.

Boune Again Shell (bash): Desenvolvido inicialmente por Brian Fox e Chet Ramey, este é o Shell do projeto GNU. O número de seus adeptos é o que mais cresce em todo o mundo, seja por que ele é o Shell padrão do Linux, seja por sua grande diversidade de comandos, que incorpora inclusive diversos comandos característicos do C Shell.

C Shell (csh): Desenvolvido por Bill Joy, da Universidade de Berkley, é o Shell mais utilizado em ambientes BSD. Foi ele quem introduziu o histórico de comandos. A estruturação de seus comandos é bem similar à da linguagem C. Seu grande pecado foi ignorar a compatibilidade com o sh, partindo por um caminho próprio.

Além destes Shells existem outros, mas irei falar somente sobre os três primeiros, tratando-os genericamente por Shell e assinalando as especificidades de cada um.

tência do sistema operacional Unix, foram aparecendo. Atualmente existem diversos sabores de Shell (veja Quadro 1 na página anterior).

Como funciona o Shell

O Shell é o primeiro programa que você ganha ao iniciar sua sessão (se quisermos assassinar a língua portuguesa podemos também dizer "ao se logar") no Linux. É ele quem vai resolver um monte de coisas de forma a não onerar o kernel com tarefas repetitivas, poupando-o para tratar assuntos mais nobres. Como cada usuário possui o seu próprio Shell interpondo-se entre ele e o Linux, é o Shell quem interpreta os comandos digitados e examina as suas sintaxes, passando-os esmiuçados para execução.

- Êpa! Esse negócio de interpretar comando não tem nada a ver com interpretador não, né?
- Tem sim: na verdade o Shell é um interpretador que traz consigo uma poderosa linguagem com comandos de alto nível, que permite construção de loops, de tomadas de decisão e de armazenamento de valores em variáveis, como vou te mostrar.
- Vou explicar as principais tarefas que o Shell cumpre, na sua ordem de execução. Preste atenção, porque esta ordem é fundamental para o entendimento do resto do nosso bate papo.

Análise da linha de comando

Neste exame o Shell identifica os caracteres especiais (reservados) que têm significado para a interpretação da linha e logo em seguida verifica se a linha passada é um comando ou uma atribuição de valores, que são os ítens que vou descrever a seguir.

Comando

Quando um comando é digitado no "prompt" (ou linha de comando) do Linux, ele é dividido em partes, separadas por espaços em branco: a primeira parte é o nome do programa, cuja existência será verificada; em seguida, nesta ordem, vêm as opções/parâmetros, redirecionamentos e variáveis.

Quando o programa identificado existe, o Shell verifica as permissões dos arquivos envolvidos (inclusive o próprio programa), e retorna um erro caso o usuário que chamou o programa não esteja autorizado a executar esta tarefa.

Com que Shell eu vou?

Quando digo que o último campo do arquivo */etc/passwd* informa ao sistema qual é o Shell que o usuário vai usar ao se "logar", isto deve ser interpretado ao pé-da-letra. Se este campo do seu registro contém o termo *prog*, ao acessar o sistema o usuário executará o programa *prog*. Ao término da execução, a sessão do usuário se encerra automaticamente. Imagine quanto se pode incrementar a segurança com este simples artifício.

```
$ ls linux
linux
```

Neste exemplo o Shell identificou o ls como um programa e o linux como um parâmetro passado para o programa ls.

Atribuição

Se o Shell encontra dois campos separados por um sinal de igual (=) sem espaços em branco entre eles, ele identifica esta seqüência como uma atribuição.

```
$ valor=1000
```

Neste caso, por não haver espaços em branco (que é um dos caracteres reservados), o Shell identificou uma atribuição e colocou 1000 na variável valor.

Resolução de Redirecionamentos

Após identificar os componentes da linha que você digitou, o Shell parte para a resolução de redirecionamentos.

O Shell tem incorporado ao seu elenco de habilidades o que chamamos de redirecionamento, que pode ser de entrada (*stdin*), de saída (*stdout*) ou dos erros (*stderr*), conforme vou explicar a seguir. Mas antes precisamos falar de...

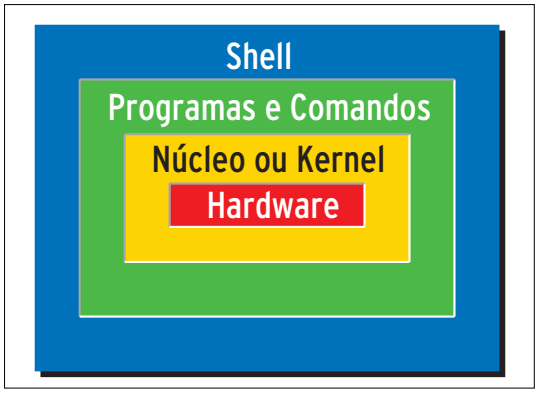

Figura 1: Ambiente em camadas de um sistema Linux

Substituição de Variáveis

Neste ponto, o Shell verifica se as eventuais variáveis (parâmetros começados por $), encontradas no escopo do comando, estão definidas e as substitui por seus valores atuais.

Substituição de Meta-Caracteres

Se algum meta-caracter (ou "coringa", como *, ? ou []*) for encontrado na linha de comando, ele será substituído por seus possíveis valores.

Supondo que o único item no seu diretório corrente cujo nome começa com a letra *n* seja um diretório chamado *nomegrandeprachuchu*, se você fizer:

```
$ cd n*
```

como até aqui quem está manipulando a linha de comando ainda é o Shell e o programa *cd* ainda não foi executado, o Shell expande o *n** para *nomegrandeprachuchu* (a única possibilidade válida) e executa o comando *cd* com sucesso.

Entrega da linha de comando para o kernel

Completadas todas as tarefas anteriores, o Shell monta a linha de comando, já com todas as substituições feitas e chama o kernel para executá-la em um novo Shell (Shell filho), que ganha um número de processo (PID ou Process IDentification) e fica inativo, tirando uma soneca durante a execução do programa. Uma vez encerrado este processo (e o Shell filho), o "Shell pai" recebe novamente o controle e exibe um "prompt", mostrando que está pronto para executar outros comandos.

Cuidado na Atribuição

Jamais faça:

```
$ valor = 1000
bash: valor: not found
```

Neste caso, o Bash achou a palavra valor isolada por espaços e julgou que você estivesse mandando executar um programa chamado *valor*, para o qual estaria passando dois parâmetros: = e *1000*.

Decifrando a Pedra de Roseta

Para tirar aquela sensação que você tem quando vê um script Shell, que mais parece uma sopa de letrinhas ou um conjunto de hieróglifos, vou lhe mostrar os principais caracteres especiais para que você saia por aí como Champollion decifrando a Pedra de Roseta.

Caracteres para remoção do significado.

É isso mesmo, quando não desejamos que o Shell interprete um caractere específico, devemos "escondê-lo" dele. Isso pode ser feito de três maneiras diferentes, cada uma com sua peculiaridade:

- **Apóstrofo (')**: quando o Shell vê uma cadeia de caracteres entre apóstrofos, ele retira os apóstrofos da cadeia e não interpreta seu conteúdo.

```
$ ls linuxm*
linuxmagazine
$ ls 'linuxm*'
bash: linuxm* no such file
or directory
```

No primeiro caso o Shell "expandiu" o asterisco e descobriu o arquivo *linuxmagazine* para listar. No segundo, os apóstrofos inibiram a interpretação do Shell e veio a resposta que não existe o arquivo *linuxm**.

- **Contrabarra ou Barra Invertida (\)**: idêntico aos apóstrofos exceto que a barra invertida inibe a interpretação somente do caractere que a segue.
  Suponha que você, acidentalmente, tenha criado um arquivo chamado * (asterisco) – o que alguns sabores de Unix permitem – e deseja removê-lo. Se você fizesse:

```
$ rm *
```

Você estaria na maior encrenca, pois o *rm* removeria todos os arquivos do diretório corrente. A melhor forma de fazer o serviço é:

```
$ rm \*
```

Desta forma, o Shell não interpreta o asterisco, evitando a sua expansão.
Faça a seguinte experiência científica:

```
$ cd /etc
$ echo '*'
$ echo \*
$ echo *
```

Viu a diferença?

- **Aspas ("):** exatamente iguais ao apóstrofo, exceto que, se a cadeia entre aspas contiver um cifrão ($), uma crase (`), ou uma barra invertida (\), estes caracteres serão interpretados pelo Shell.
  Não precisa se estressar, eu não te dei exemplos do uso das aspas por que você ainda não conhece o cifrão ($) nem a crase (`). Daqui para frente - veremos com muita constância o uso destes caracteres especiais; o mais importante é entender seu significado.

Caracteres de redirecionamento

A maioria dos comandos tem uma entrada, uma saída e pode gerar erros. Esta entrada é chamada Entrada Padrão ou *stdin* e seu dispositivo padrão é o teclado do terminal. Analogamente, a saída do comando é chamada Saída Padrão ou *stdout* e seu dispositivo padrão é a tela do terminal. Para a tela também são enviadas normalmente as mensagens de erro oriundas dos comandos, chamada neste caso de Saída de Erro Padrão ou *stderr*. Veremos agora como alterar este estado de coisas.

Vamos fazer um programa gago. Para isto digite (tecle "Enter" ao final de cada linha – comandos do usuário são ilustrados em negrito):

```
$ cat
E-e-eu sou gago. Vai encarar?
E-e-eu sou gago. Vai encarar?
```

O *cat* é um comando que lista o conteúdo do arquivo especificado para a Saída Padrão (*stdout*). Caso a entrada não seja definida, ele espera os dados da *stdin* (a entrada padrão). Ora como eu não especifiquei a entrada, ele a está esperando pelo teclado (Entrada Padrão) e como também não citei a saída, o que eu teclar irá para a tela (Saída Padrão), criando desta forma – como eu havia proposto – um programa gago. Experimente!

Redirecionamentop da Saída Padrão

Para especificarmos a saída de um programa usamos o símbolo " > " ou o " > > ", seguido do nome do arquivo para o qual se deseja mandar a saída.

Vamos transformar o programa anterior em um "editor de textos":

```
$ cat > Arq
```

O *cat* continua sem ter a entrada especificada, portanto está aguardando que os dados sejam teclados, porém a sua saída está sendo desviada para o arquivo *Arq*. Assim sendo, tudo que esta sendo teclado esta indo para dentro de *Arq*, de forma que fizemos o editor de textos mais curto e ruim do planeta.

Se eu fizer novamente:

```
$ cat > Arq
```

Os dados contidos em *Arq* serão perdidos, já que antes do redirecionamento o Shell criará um *Arq* vazio. Para colocar mais informações no final do arquivo eu deveria ter feito:

```
$ cat >> Arq
```

Redirecionamento da Saída de Erro Padrão

Assim como por padrão o Shell recebe os dados do teclado e envia a saída para a tela, os erros também vão para a tela se você não especificar para onde eles devem ser enviados. Para redirecionar os erros, use *2 > SaidaDeErro*. Note que entre o número *2* e o sinal de maior (>) não existe espaço em branco.

Vamos supor que durante a execução de um script você pode, ou não (dependendo do rumo tomado pela execução do programa), ter criado um arquivo chamado */tmp/seraqueexiste$$*. Como não quer ficar com sujeira no disco rígido, ao final do script você coloca a linha a seguir:

```
rm /tmp/seraqueexiste$$
```

Redirecionamento Perigoso

Como já havia dito, o Shell resolve a linha e depois manda o comando para a execução. Assim, se você redirecionar a saída de um arquivo para ele próprio, primeiramente o Shell "esvazia" este arquivo e depois manda o comando para execução! Desta forma, para sua alegria, você acabou de perder o conteúdo de seu querido arquivo.

Dados ou Erros?

Preste atenção! Não confunda >> com 2>. O primeiro anexa dados ao final de um arquivo, e o segundo redireciona a Saída de Erro Padrão (*stderr*) para um arquivo que está sendo designado. Isto é importante!

Caso o arquivo não existisse seria enviado para a tela uma mensagem de erro. Para que isso não aconteça faça:

```
rm /tmp/seraqueexiste$$ 2> ➥
/dev/null
```

Para que você teste a Saída de Erro Padrão direto no prompt do seu Shell, vou dar mais um exemplo. Faça:

```
$ ls naoexiste
bash: naoexiste no such file ➥
or directory
$ ls naoexiste 2> arquivodeerros
$
$ cat arquivodeerros
bash: naoexiste no such file ➥
or directory
```

Neste exemplo, vimos que quando fizemos um *ls* em *naoexiste*, ganhamos uma mensagem de erro. Após redirecionar a Saída de Erro Padrão para *arquivodeerros* e executar o mesmo comando, recebemos somente o "prompt" na tela. Quando listamos o conteúdo do arquivo para o qual foi redirecionada a Saída de Erro Padrão, vimos que a mensagem de erro tinha sido armazenada nele.

É interessante notar que estes caracteres de redirecionamento são cumulativos, isto é, se no exemplo anterior fizéssemos o seguinte:

```
$ ls naoexiste 2>> ➥
arquivodeerros
```

a mensagem de erro oriunda do *ls* seria anexada ao final de *arquivodeerros*.

Redirecionamento da Entrada Padrão

Para fazermos o redirecionamento da Entrada Padrão usamos o < (menor que). "E pra que serve isso?", você vai me perguntar. Deixa eu dar um exemplo, que você vai entender rapidinho.

Suponha que você queira mandar um mail para o seu chefe. Para o chefe nós caprichamos, né? Então ao invés de sair redigindo o mail direto no "prompt", de forma a tornar impossível a correção de uma frase anterior onde, sem querer, você escreveu um "nós vai", você edita um arquivo com o conteúdo da mensagem e após umas quinze verificações sem constatar nenhum erro, decide enviá-lo e para tal faz:

```
$ mail chefe@chefia.com.br < ➥
arquivocommailparaochefe
```

e o chefe receberá uma mensagem com o conteúdo do *arquivocommailparaochefe*.

Outro tipo de redirecionamento "muito louco" que o Shell permite é o chamado "here document". Ele é representado por < < e serve para indicar ao Shell que o escopo de um comando começa na linha seguinte e termina quando encontra uma linha cujo conteúdo seja unicamente o "label" que segue o sinal < <.

Veja o fragmento de script a seguir, com uma rotina de ftp:

```
ftp -ivn hostremoto << fimftp
user $Usuario $Senha
binary
get arquivoremoto
fimftp
```

neste pedacinho de programa temos um monte de detalhes interessantes:

- As opções usadas para o *ftp* (*-ivn*) servem para ele listar tudo que está acontecendo (opção *-v* de "verbose"), para não ficar perguntando se você tem certeza que deseja transmitir cada arquivo (opção *-i* de "interactive") e finalmente a opção *-n* serve para dizer ao *ftp* para ele não solicitar o usuário e sua senha, pois estes serão informados pela instrução específica (*user*);
- Quando eu usei o < < *fimftp*, estava dizendo o seguinte para o interpretador: "Olha aqui Shell, não se meta em nada a partir deste ponto até encontrar o 'label' *fimftp*. Você não entenderia droga nenhuma, já que são instruções específicas do *ftp*".

  Se fosse só isso seria simples, mas pelo próprio exemplo dá para ver que existem duas variáveis (*$Usuario* e *$Senha*), que o Shell vai resolver antes do redirecionamento. Mas a grande vantagem deste tipo de construção é que ela permite que comandos também sejam interpretados dentro do escopo do "here document", o que, aliás, contraria o que acabei de dizer. Logo a seguir te explico como esse negócio funciona. Agora ainda não dá, estão faltando ferramentas.
- O comando *user* é do repertório de instruções do *ftp* e serve para passar o usuário e a senha que haviam sido lidos em uma rotina anterior a este fragmento de código e colocados respectivamente nas duas variáveis: *$Usuario* e *$Senha*.
- O *binary* é outra instrução do *ftp*, que serve para indicar que a transferência de *arquivoremoto* será feita em modo binário, isto é o conteúdo do arquivo não será inteerpretado para saber se está em ASCII, EBCDIC, …
- O comando *get arquivoremoto* diz ao cliente *ftp* para pegar este arquivo no servidor *hostremoto* e trazê-lo para a nossa máquina local. Se quiséssemos enviar um arquivo, bastaria usar, por exemplo, o comando *put arquivolocal*.

Direito de Posse

O *$$* contém o PID, isto é, o número do seu processo. Como o Linux é multiusuário, é bom anexar sempre o *$$* ao nome dos seus arquivos para não haver problema de propriedade, isto é, caso você batizasse o seu arquivo simplesmente como *seraqueexiste*, a primeira pessoa que o usasse (criando-o então) seria o seu dono e a segunda ganharia um erro quando tentasse gravar algo nele.

Etiquetas Erradas

Um erro comum no uso de labels (como o fimftp do exemplo anterior) é causado pela presença de espaços em branco antes ou após o mesmo. Fique muito atento quanto a isso, por que este tipo de erro costuma dar uma boa surra no programador, até que seja detectado. Lembre-se: um label que se preze tem que ter uma linha inteira só para ele.

Redirecionamento de comandos

Os redirecionamentos de que falamos até agora sempre se referiam a arquivos, isto é, mandavam para arquivo, recebiam de arquivo, simulavam arquivo local, … O que veremos a partir de agora, redireciona a saída de um comando para a entrada de outro. É utilíssimo e, apesar de não ser macaco gordo, sempre quebra os

maiores galhos. Seu nome é "pipe" (que em inglês significa tubo, já que ele canaliza a saída de um comando para a entrada de outro) e sua representação é a | (barra vertical).

```
$ ls | wc -l
21
```

O comando *ls* passou a lista de arquivos para o comando *wc*, que quando está com a opção *-l* conta a quantidade de linhas que recebeu. Desta forma, podemos afirmar categoricamente que no meu diretório existiam 21 arquivos.

```
$ cat /etc/passwd | sort | lp
```

A linha de comandos acima manda a listagem do arquivo */etc/passwd* para a entrada do comando *sort*. Este a classifica e envia para o *lp* que é o gerenciador da fila de impressão.

Caracteres de ambiente

Quando queremos priorizar uma expressão, nós a colocamos entre parênteses, não é? Pois é, por causa da aritmética é normal pensarmos deste jeito. Mas em Shell o que prioriza mesmo são as crases (`) e não os parênteses. Vou dar exemplos para você entender melhor.

Eu quero saber quantos usuários estão "logados" no computador que eu administro. Eu posso fazer:

```
$ who | wc -l
8
```

O comando *who* passa a lista de usuários conectados ao sistema para o comando *wc -l*, que conta quantas linhas recebeu e mostra a resposta na tela. Muito bem, mas ao invés de ter um número oito solto na tela, o que eu quero mesmo é que ele esteja no meio de uma frase. Ora, para mandar frases para a tela eu só preciso usar o comando *echo*; então vamos ver como é que fica:

Buraco Negro

Em Unix existe um arquivo fantasma. Chama-se */dev/null*. Tudo que é enviado para este arquivo some. Assemelha-se a um Buraco Negro. No caso do exemplo, como não me interessava guardar a possível mensagem de erro oriunda do comando *rm*, redirecionei-a para este arquivo.

```
$ echo "Existem who | wc -l ➥
usuarios conectados"
Existem who | wc -l usuarios ➥
conectados
```

Hi! Olha só, não funcionou! É mesmo, não funcionou e não foi por causa das aspas que eu coloquei, mas sim por que eu teria que ter executado o *who | wc -l* antes do echo. Para resolver este problema, tenho que priorizar a segunda parte do comando com o uso de crases:

```
$ echo "Existem `who | wc -l` ➥
usuarios conectados"
Existem      8 usuarios ➥
conectados
```

Para eliminar esse monte de brancos antes do 8 que o *wc -l* produziu, basta retirar as aspas. Assim:

```
$ echo Existem `who | wc -l` ➥
usuarios conectados
Existem 8 usuarios conectados
```

As aspas protegem da interpretação do Shell tudo que está dentro dos seus limites. Como para o Shell basta um espaço em branco como separador, o monte de espaços será trocado por um único após a retirada das aspas.

Outra coisa interessante é o uso do ponto-e-vírgula. Quando estiver no Shell, você deve sempre dar um comando em cada linha. Para agrupar comandos em uma mesma linha, temos que separá-los por ponto-e-vírgula. Então:

```
$ pwd ; cd /etc; pwd ;cd -;pwd
/home/meudir
/etc
/home/meudir
```

Neste exemplo, listei o nome do diretório corrente com o comando *pwd*, mudei para o diretório */etc*, novamente listei o nome do diretório e finalmente voltei para o diretório onde estava anteriormente (*cd -*), listando seu nome. Repare que coloquei o ponto-e-vírgula de todas as formas possíveis, para mostrar que não importa se existem espaços em branco antes ou após este caracter.

Finalmente, vamos ver o caso dos parênteses. No exemplo a seguir, colocamos diversos comandos separados por ponto-e-vírgula entre parênteses:

```
$ (pwd ; cd /etc ; pwd)
/home/meudir
/etc
$ pwd
/home/meudir
```

"Quequeiiisso" minha gente? Eu estava no */home/meudir*, mudei para o */etc*, constatei que estava neste diretório com o *pwd* seguinte e quando o agrupamento de comandos terminou, eu vi que continuava no */etc/meudir*!

Hi! Será que tem coisa do mágico Mandrake por aí? Nada disso. O interessante do uso de parênteses é que eles invocam um novo Shell para executar os comandos que estão em seu interior. Desta forma, fomos realmente para o diretório */etc*, porém após a execução de todos os comandos, o novo Shell que estava no diretório */etc* morreu e retornamos ao Shell anterior que estava em */home/meudir*.

Que tal usar nossos novos conceitos?

```
$ mail suporte@linux.br << FIM
Ola suporte, hoje as `date ➥
"+%hh:mm"` ocorreu novamente ➥
aquele problema que eu havia ➥
reportado por telefone. De ➥
acordo com seu pedido segue a ➥
listagem do diretorio:
`ls -l`
Abracos a todos.
FIM
```

Finalmente agora podemos demonstrar o que conversamos anteriormente sobre "here document". Os comandos entre crases tem prioridade, portanto o Shell os executará antes do redirecionamento do "here document". Quando o suporte receber a mensagem, verá que os comandos *date* e *ls* foram executados antes do comando mail, recebendo então um instantâneo do ambiente no momento de envio do email.

- Garçom, passa a régua! ■

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, sistema operacional, similar ao Unix, da Cobra Computadores. É professor do curso de Mestrado em Software Livre das Faculdades Estácio de Sá, no Rio de Janeiro.