103.1 Trabalhar na linha de comando

Referência ao LPI objectivo

LPIC-1 version 5.0, Exam 101, Objective 103.1

Peso

4

Áreas chave de conhecimento

- Usar comandos simples de shell e sequências de comandos de apenas uma linha para executar tarefas básicas na linha de comando.
- Usar e modificar o ambiente de shell incluindo definir, fazer referência e exportar variáveis de ambiente.
- Usar e editar o histórico de comandos.
- Invocar comandos de dentro e de fora do caminho definido.

Segue uma lista parcial dos arquivos, termos e utilitários utilizados

- `bash`
- `echo`
- `env`
- `export`
- `pwd`
- `set`
- `unset`
- `type`

- `which`
- `man`
- `uname`
- `history`
- `.bash_history`
- Quoting

103.1 Lição 1

Certificação:	LPIC-1
Versão:	5.0
Tópico:	103 Comandos GNU e Unix
Objetivo:	103.1 Trabalho na linha de comando
Lição:	1 de 2

Introdução

É comum que os recém-chegados ao mundo da administração do Linux e do shell Bash se sintam um pouco perdidos longe do conforto de uma interface gráfica. Eles estão habituados a acessar, com o botão direito do mouse, as dicas visuais e informações contextuais disponibilizadas pelos utilitários gráficos de gerenciamento de arquivos. Portanto, é importante aprender rapidamente a dominar o conjunto relativamente pequeno de ferramentas de linha de comando que permitem acessar instantaneamente todos os dados oferecidos por sua antiga interface gráfica - e muito mais.

Obtendo informações sobre o sistema

De olhos arregalados diante do tracinho piscando de um prompt de linha de comando, você provavelmente se pergunta “Onde estou?” Ou, mais precisamente, “Onde estou agora no sistema de arquivos do Linux? E, se eu criar um novo arquivo, onde ele vai parar?” O que você está procurando é o *diretório de trabalho atual*, e o comando `pwd` responderá às suas dúvidas:

```
$ pwd
/home/frank
```

Vamos supor que Frank esteja atualmente logado no sistema e em seu diretório pessoal: `/home/frank/`. Se Frank criar um arquivo vazio usando o comando `touch` sem especificar qualquer outro local no sistema de arquivos, o arquivo será criado em `/home/frank/`. Se usarmos `ls` para listar o conteúdo do diretório, veremos esse novo arquivo:

```
$ touch newfile
$ ls
newfile
```

Além de sua localização no sistema de arquivos, você também pode precisar de informações sobre o sistema Linux que está executando, como por exemplo o número exato da versão de sua distribuição ou a versão do kernel do Linux atualmente carregada. A ferramenta `uname` é a resposta. E, em particular, `uname` mais a opção `-a` ("all").

```
$ uname -a
Linux base 4.18.0-18-generic #19~18.04.1-Ubuntu SMP Fri Apr 5 10:22:13 UTC 2019
x86_64 x86_64 x86_64 GNU/Linux
```

Neste caso, `uname` mostra que a máquina de Frank tem o kernel do Linux versão 4.18.0 instalado e está executando o Ubuntu 18.04 em uma CPU de 64 bits (`x86_64`).

Obtendo informações sobre comandos

Freqüentemente, você encontrará documentações falando sobre comandos do Linux com os quais ainda não está familiarizado. A própria linha de comando oferece todo tipo de informações úteis sobre o que os comandos fazem e como usá-los com eficácia. As referências mais úteis provavelmente estarão nos muitos arquivos do sistema `man`.

Como regra, os desenvolvedores Linux escrevem arquivos `man` e os distribuem junto com os utilitários que criam. Os arquivos `man` são documentos altamente estruturados cujo conteúdo é dividido intuitivamente em cabeçalhos padronizados. Basta digitar `man` seguido do nome de um comando para exibir informações como o nome do comando, uma breve sinopse de seu uso, uma descrição mais detalhada e alguns dados importantes sobre o histórico e as licenças de uso. Eis um exemplo:

```
$ man uname
UNAME(1)            User Commands            UNAME(1)
NAME
   uname - print system information
SYNOPSIS
```

```
   uname [OPTION]...
DESCRIPTION
   Print certain system information.  With no OPTION, same as -s.
   -a, --all
      print  all information, in the following order, except omit -p
      and -i if unknown:
   -s, --kernel-name
      print the kernel name
   -n, --nodename
      print the network node hostname
   -r, --kernel-release
      print the kernel release
   -v, --kernel-version
      print the kernel version
   -m, --machine
      print the machine hardware name
   -p, --processor
      print the processor type (non-portable)
   -i, --hardware-platform
      print the hardware platform (non-portable)
   -o, --operating-system
      print the operating system
   --help display this help and exit
   --version
      output version information and exit
AUTHOR
   Written by David MacKenzie.
REPORTING BUGS
   GNU coreutils online help: <http://www.gnu.org/software/coreutils/>
   Report uname translation bugs to
   <http://translationproject.org/team/>
COPYRIGHT
   Copyright©2017 Free Software Foundation, Inc. License  GPLv3+: GNU
   GPL version 3 or later <http://gnu.org/licenses/gpl.html>.
   This is free software: you are free to change and redistribute it.
   There is NO WARRANTY, to the extent permitted by law.
SEE ALSO
   arch(1), uname(2)
   Full documentation at: <http://www.gnu.org/software/coreutils/uname>
   or available locally via: info '(coreutils) uname invocation'
GNU coreutils 8.28       January 2018             UNAME(1)
```

O comando `man` só funciona quando fornecemos um nome de comando exato. Porém, se não tiver certeza do nome do comando que deseja pesquisar, use o comando `apropos` para explorar os nomes e

descrições das páginas `man`. Se, por exemplo, você não consegue se lembrar de que `uname` informa a versão atual do kernel do Linux, pode passar a palavra `kernel` para `apropos`. Aparecerão muitas linhas de saída, mas dentre elas haverá o seguinte:

```
$ apropos kernel
systemd-udevd-kernel.socket (8) - Device event managing daemon
uname (2)            - get name and information about current kernel
urandom (4)          - kernel random number source devices
```

Caso não precise da documentação completa de um comando, pode obter seus dados básicos rapidamente usando `type`. Neste exemplo, usamos `type` para consultar quatro comandos separados ao mesmo tempo. Os resultados mostram que `cp` ("copy") é um programa que vive em `/bin/cp` e que `kill` (muda o estado de um processo em execução) é um comando interno do shell (*shell builtin*)—o que significa que é, na verdade, parte do próprio shell Bash:

```
$ type uname cp kill which
uname is hashed (/bin/uname)
cp is /bin/cp
kill is a shell builtin
which is /usr/bin/which
```

Note que, além de ser um comando binário regular como `cp`, `uname` também aparece como "hashed". A razão para isso é que Frank recentemente usou `uname` e, para aumentar a eficiência do sistema, o comando foi adicionado a uma tabela de hash para ficar mais acessível na próxima vez que for executado. Se Frank rodasse `type uname` após a inicialização do sistema, ele constataria que `type` voltaria a descrever `uname` como um binário regular.

NOTE Uma maneira mais rápida de limpar a tabela de hash é executar o comando `hash -d`.

Às vezes—principalmente ao trabalhar com scripts automatizados—precisamos de uma fonte mais simples de informações sobre um comando. O comando `which`, que o comando `type` do exemplo anterior rastreou para nós, retorna somente a localização absoluta de um comando. Este exemplo localiza os comandos `uname` e `which`.

```
$ which uname which
/bin/uname
/usr/bin/which
```

NOTE Se quiser exibir informações sobre comandos internos do shell (“builtin”), use o comando `help`.

Usando o histórico de comandos

Pode acontecer de você pesquisar cuidadosamente o uso adequado de um comando e executá-lo com êxito junto com uma seleção complicada de opções e argumentos. Mas o que ocorre algumas semanas depois, quando você precisa executar o mesmo comando com as mesmas opções e argumentos, mas não consegue se lembrar dos detalhes? Ao invés de recomeçar a pesquisa do zero, vale a pena tentar recuperar o comando original usando `history`.

Digite `history` para exibir os comandos mais recentes em ordem de execução. É fácil pesquisar nesses comandos usando um pipe para canalizar uma string específica para o comando `grep`. O exemplo abaixo procura por qualquer comando que inclua o texto `bash_history`:

```
$ history | grep bash_history
1605 sudo find /home -name ".bash_history" | xargs grep sudo
```

Aqui, um único comando é retornado junto com seu número na seqüência, 1605.

E por falar em `bash_history`, esse é na verdade o nome de um arquivo oculto que costuma estar no diretório inicial do usuário. Por se tratar de um arquivo oculto (indicado pelo ponto que precede seu nome de arquivo), ele só será visível ao se listar o conteúdo do diretório, usando `ls` com o argumento `-a`:

```
$ ls /home/frank
newfile
$ ls -a /home/frank
.  ..  .bash_history  .bash_logout  .bashrc  .profile  .ssh  newfile
```

O que contém o arquivo `.bash_history`? Dê uma olhada: você encontrará ali centenas e centenas de seus comandos recentes. No entanto, você pode se surpreender ao descobrir que alguns de seus comandos *mais* recentes estão ausentes. Isso porque, embora eles sejam instantaneamente adicionados ao banco de dados dinâmico `history`, as últimas adições ao seu histórico de comandos não são gravadas no arquivo `.bash_history` até o encerramento da sessão.

Para aproveitar o conteúdo de `history` e tornar sua experiência na linha de comando muito mais rápida e eficiente, use as setinhas para cima e para baixo de seu teclado. Pressione a tecla para cima várias vezes para preencher a linha de comando com os comandos recentes. Quando chegar ao que está procurando, basta dar Enter para executá-lo. Assim, é fácil recuperar e, se desejado, modificar

comandos diversas vezes durante uma sessão no shell.

Exercícios Guiados

1. Consultando o sistema `man`, descubra como fazer o `apropos` mostrar apenas uma explicação curta de seu uso e em seguida voltar ao shell.

2. Use o sistema `man` para determinar qual licença de copyright é atribuída ao comando `grep`.

Exercícios Exploratórios

1. Identifique a arquitetura de hardware e a versão do kernel do Linux que estão sendo usadas no seu computador em um formato de saída fácil de ler.

2. Imprima na tela as últimas vinte linhas do banco de dados dinâmico `history` e do arquivo `.bash_history` para compará-los.

3. Use a ferramenta `apropos` para identificar a página `man` na qual se encontra o comando necessário para mostrar o tamanho de um dispositivo de bloco físico conectado em bytes, ao invés de megabytes ou gigabytes.

Resumo

Nesta lição, você aprendeu:

- Como obter informações sobre a localização do sistema de arquivos e a pilha de software do sistema operacional.
- Como encontrar ajuda para o uso de comandos.
- Como identificar a localização do sistema de arquivos e os tipos de binários de comandos.
- Como encontrar e reutilizar comandos executados anteriormente.

Os seguintes comandos foram abordados nesta lição:

`pwd`

Exibe o caminho para o diretório de trabalho atual.

`uname`

Exibe a arquitetura de hardware do sistema, a versão do kernel do Linux, a distribuição e a versão da distribuição.

`man`

Acessa os arquivos de ajuda com a documentação do uso dos comandos.

`type`

Exibe a localização de um ou mais comandos no sistema de arquivos e seu tipo.

`which`

Exibe a localização de um comando no sistema de arquivos.

`history`

Exibe ou reutiliza comandos executados anteriormente.

Respostas aos Exercícios Guiados

1. Consultando o sistema `man`, descubra como fazer o `apropos` mostrar apenas uma explicação curta de seu uso e em seguida voltar ao shell.

   Rode `man apropos` e desça a sessão “Options” até encontrar o parágrafo `--usage`.

2. Use o sistema `man` para determinar qual licença de copyright é atribuída ao comando `grep`.

   Rode `man grep` e desça até a seção “Copyright” do documento. Note que o programa usa um copyright da Free Software Foundation.

Respostas aos Exercícios Exploratórios

1. Identifique a arquitetura de hardware e a versão do kernel do Linux que estão sendo usadas no seu computador em um formato de saída fácil de ler.

   Rode `man uname`, leia a seção "Description" e identifique os argumentos de comandos que permitem exibir somente os resultados exatos que se deseja. Note que `-v` mostra a versão do kernel e `-i` a plataforma de hardware.

   ```
   $ man uname
   $ uname -v
   $ uname -i
   ```

2. Imprima na tela as últimas vinte linhas do banco de dados dinâmico `history` e do arquivo `.bash_history` para compará-los.

   ```
   $ history 20
   $ tail -n 20 .bash_history
   ```

3. Use a ferramenta `apropos` para identificar a página `man` na qual se encontra o comando necessário para mostrar o tamanho de um dispositivo de bloco físico conectado em bytes, ao invés de megabytes ou gigabytes.

   Uma maneira seria executar `apropos` com o string `block`, ler os resultados, notar que `lsblk` lista os dispositivos de bloco (e assim seria a ferramenta mais provável para dar a resposta que buscamos), rodar `man lsblk`, descer pela seção "Description" e notar que `-b` exibe o tamanho de um dispositivo em bytes. Finalmente, executamos `lsblk -b` para ver o que aparece.

   ```
   $ apropos block
   $ man lsblk
   $ lsblk -b
   ```

103.1 Lição 2

Certificação:	LPIC-1
Versão:	5.0
Tópico:	103 Comandos GNU e Unix
Objetivo:	103.1 Trabalho na linha de comando
Lição:	2 de 2

Introdução

Um ambiente de sistema operacional inclui as ferramentas básicas—como shells de linha de comando e, às vezes, uma interface gráfica—necessárias para trabalhar. Mas seu ambiente também virá com um catálogo de atalhos e valores predefinidos. Nesta lição aprenderemos como listar, invocar e gerenciar esses valores.

Encontrando suas variáveis de ambiente

Então, como identificamos os valores atuais para cada uma de nossas variáveis de ambiente? Uma maneira de fazer isso é por meio do comando `env`:

```
$ env
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
XDG_RUNTIME_DIR=/run/user/1000
XAUTHORITY=/run/user/1000/gdm/Xauthority
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/l
ocal/games:/snap/bin
GJS_DEBUG_TOPICS=JS ERROR;JS LOG
```

```
[...]
```

Aparecerão muitos resultados—bem mais do que os incluídos no trecho acima. Mas, por enquanto, observe a entrada `PATH`, que contém os diretórios nos quais seu shell (e outros aplicativos) procura por outros programas sem a necessidade de especificar um caminho completo. Neste caso, seria possível executar um programa binário que reside, digamos, em `/usr/local/bin` de dentro do seu diretório pessoal, e ele seria executado como se o arquivo fosse local.

Vamos mudar de assunto um pouquinho. O comando `echo` exibe na tela o que você mandar. Acredite ou não, haverá diversas ocasiões em que será muito útil fazer `echo` literalmente repetir algo.

```
$ echo "Hi. How are you?"
Hi. How are you?
```

Mas `echo` tem outras cartas na manga. Quando você o alimenta com o nome de uma variável de ambiente—e informa que se trata de uma variável, com o prefixo `$`—ao invés de apenas exibir o nome da variável, o shell irá expandi-lo, informando o valor. Não tem certeza se o seu diretório favorito está atualmente em `PATH`? Você pode verificar rapidamente com o `echo`:

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/
games:/snap/bin
```

Criando novas variáveis de ambiente

Você pode adicionar suas próprias variáveis personalizadas ao seu ambiente. A maneira mais simples é usar o caractere `=`. A string à esquerda será o nome da nova variável e a string à direita, seu valor. Depois, alimente `echo` com o nome da variável para confirmar se funcionou:

```
$ myvar=hello
$ echo $myvar
hello
```

NOTE Note que não há espaços em torno do sinal de igual durante a atribuição de variáveis.

Mas será que funcionou mesmo? Digite `bash` no terminal para abrir um novo shell. Ele terá exatamente a mesma aparência daquele que acabamos de usar, mas na verdade é um *filho* do original (que chamamos de *pai*). Agora, dentro desse novo shell filho, tente fazer com que o `echo` opere sua

mágica da mesma maneira de antes. Nada. O que aconteceu?

```
$ bash
$ echo $myvar

$
```

Uma variável criada da maneira como acabamos de fazer só estará disponível localmente—dentro daquela sessão de shell imediata. Se você iniciar um novo shell—ou fechar a sessão usando `exit`—a variável não irá junto. Aqui, se você digitar `exit`, será levado de volta ao shell pai original que, no momento, é onde queremos estar. Execute `echo $myvar` mais uma vez se quiser apenas confirmar que a variável ainda é válida. Depois digite `export myvar` para passar a variável para quaisquer shells filhos que possam ser abertos posteriormente. Experimente: digite `bash` para abrir um novo shell e em seguida use `echo`:

```
$ exit
$ export myvar
$ bash
$ echo $myvar
hello
```

Tudo isso pode parecer bobagem quando estamos criando shells sem um propósito real. Mas é importantíssimo entender como as variáveis do shell são propagadas pelo sistema para quando você começar a escrever scripts sérios.

Removendo as variáveis de ambiente

Quer saber como limpar todas as variáveis efêmeras que você criou? Uma maneira de fazer isso é simplesmente fechar o shell pai—ou reiniciar o computador. Mas existem jeitos mais simples. Como, por exemplo, `unset`. Basta digitar `unset` (sem o `$`) para matar a variável. Comprove com `echo`.

```
$ unset myvar
$ echo $myvar

$
```

Se um comando `unset` existe, pode apostar que também há um comando `set` complementar. A execução de `set` por si só exibirá um monte de saídas, mas nada muito diferente de `env`. Observe a primeira linha da saída obtida ao filtrarmos por `PATH`:

```
$ set | grep PATH
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/l
ocal/games:/snap/bin
[...]
```

Qual a diferença entre `set` e `env`? Para nossos propósitos, a principal delas é que `set` exibe todas as variáveis e funções. Para ilustrar isso, vamos criar uma nova variável chamada `mynewvar` e em seguida confirmar se ela está lá:

```
$ mynewvar=goodbye
$ echo $mynewvar
goodbye
```

Se rodarmos `env` usando `grep` para filtrar pela string `mynewvar`, nenhuma saída será exibida. Mas se rodarmos `set` da mesma maneira, veremos nossa variável local.

```
$ env | grep mynewvar

$ set | grep mynewvar
mynewvar=goodbye
```

Usando aspas para escapar dos caracteres especiais

Este é um bom momento para apresentar o problema dos caracteres especiais. Os caracteres alfanuméricos (a-z e 0-9) normalmente são lidos literalmente pelo Bash. Para criar um novo arquivo chamado `myfile`, bastaria digitar `touch` seguido de `myfile` e o Bash saberia o que fazer. Mas se quisermos incluir um caractere especial no nome do arquivo, teríamos um pouco mais de trabalho.

Para ilustrar esse fato, digitaremos `touch` e em seguida o título: `my big file`. O problema é que existem dois espaços entre as palavras que o Bash interpretará. Embora, tecnicamente, espaços não sejam "caracteres", é assim que eles se comportam, já que o Bash não os lerá literalmente. Se você listar o conteúdo do diretório atual, em vez de um arquivo chamado `my big file`, verá três arquivos chamados, respectivamente, `my`, `big`, e `file`. Isso porque o Bash pensou que você queria criar vários arquivos cujos nomes estavam sendo dados em uma lista:

```
$ touch my big file
$ ls
my big file
```

Os espaços serão interpretados da mesma forma se você excluir (`rm`) os três arquivos com um só comando:

```
$ rm my big file
```

Agora, vamos fazer do jeito certo. Digite `touch` e as três partes do seu nome de arquivo, mas desta vez coloque o nome entre aspas. Desta vez, funciona. Ao listar o conteúdo do diretório, veremos um único arquivo com o nome correto.

```
$ touch "my big file"
$ ls
'my big file'
```

Existem outras maneiras de obter o mesmo efeito. As aspas simples, por exemplo, funcionam da mesma forma que as aspas duplas (note que as aspas simples preservam o valor literal de todos os caracteres, ao passo que as aspas duplas preservam todos os caracteres *exceto* `$`, `` ` ``, `\` e, em certos casos, `!`.)

```
$ rm 'my big file'
```

Podemos incluir uma barra invertida antes de cada caractere especial para "escapar" dessa característica e fazer com que o Bash o leia literalmente.

```
$ touch my\ big\ file
```

Exercícios Guiados

1. Use o comando `export` para adicionar um novo diretório ao seu caminho (`PATH`) — ele não sobreviverá a uma reinicialização.

2. Use o comando `unset` para remover a variável `PATH`. Tente executar um comando (como `sudo cat /etc/shadow`) usando `sudo`. O que aconteceu? Por quê? (saia do shell para retornar ao estado original.)

Exercícios Exploratórios

1. Procure na internet a lista completa de caracteres especiais e explore-os.
2. Tente executar comandos usando strings compostas de caracteres especiais e diversos métodos para escapar deles. Existem diferenças de comportamento entre esses métodos?

Resumo

Nesta lição, você aprendeu:

- Como identificar as variáveis de ambiente de seu sistema.
- Como criar suas próprias variáveis de ambiente e exportá-la para outros shells.
- Como remover variáveis de ambiente e usar os comandos `env` e `set`.
- Como escapar dos caracteres especiais para que o Bash os leia literalmente.

Os seguintes comandos foram abordados nesta lição:

`echo`
: Exibe as strings e variáveis informadas.

`env`
: Entende e modifica suas variáveis de ambiente.

`export`
: Passa uma variável de ambiente para os shells filhos.

`unset`
: Desconfigura os valores e atributos das variáveis e funções do shell.

Respostas aos Exercícios Guiados

1. Use o comando `export` para adicionar um novo diretório ao seu caminho (`PATH`) — ele não sobreviverá a uma reinicialização.

   Podemos adicionar temporariamente um novo diretório (por exemplo, um chamado `myfiles` dentro do diretório inicial) ao caminho usando `export PATH="/home/yourname/myfiles:$PATH"`. Crie um script simples no diretório `myfiles/`, torne-o executável e tente executá-lo a partir de um diretório diferente. Estes comandos pressupõem que você esteja no diretório inicial, que contém um diretório chamado `myfiles`.

   ```
   $ touch myfiles/myscript.sh
   $ echo '#!/bin/bash' >> myfiles/myscript.sh
   $ echo 'echo Hello' >> myfiles/myscript.sh
   $ chmod +x myfiles/myscript.sh
   $ myscript.sh
   Hello
   ```

2. Use o comando `unset` para remover a variável `PATH`. Tente executar um comando (como `sudo cat /etc/shadow`) usando `sudo`. O que aconteceu? Por quê? (saia do shell para retornar ao estado original.)

   O comando `unset PATH` apaga as configurações de caminho atuais. Não será possível invocar um binário sem seu endereço absoluto. Por essa razão, se tentarmos rodar um comando usando sudo (que por sua vez é um programa binário localizado em `/usr/bin/sudo`) a operação falhará — a menos que especifiquemos a localização absoluta, como em: `/usr/bin/sudo /bin/cat /etc/shadow`. Podemos redefinir o `PATH` usando `export` ou simplesmente saindo do shell.

Respostas aos Exercícios Exploratórios

1. Procure na internet a lista completa de caracteres especiais e explore-os.

   Eis uma lista: `& ; | * ? " ' [ ] ( ) $ < > { } # / \ ! ~`.

2. Tente executar comandos usando strings compostas de caracteres especiais e diversos métodos para escapar deles. Existem diferenças de comportamento entre esses métodos?

   O escape com `"` preserva os valores especiais do cifrão, da crase e da barra invertida. Já o escape com o `'` faz com que *todos* os caracteres sejam interpretados literalmente.

```
$ echo "$mynewvar"
goodbye
$ echo '$mynewvar'
$mynewvar
```