Programação shell script

Shell script: trabalhando com pipes

Ferramentas especiais do shell auxiliam na combinação de comandos para criar aplicativos de maior complexidade.

por Martin Streicher

A linha de comando do Linux (conhecida simplesmente por *shell*, Bash ou terminal) oferece centenas de pequenos utilitários para ler, escrever e analisar dados. Com alguma digitação extra é possível combinar esses utilitários em numerosos aplicativos improvisados e de maior complexidade, para fins diversos. Por exemplo, imagine que é preciso extrair as falas de um ator. Com o texto exibido na **listagem 1**, é preciso produzir *That's what they call a sanity clause* do Groucho. O comando `grep` encontra substrings, strings e padrões em um arquivo de texto. Usamos o comando `grep` para encontrar todas as falas que começam com a string `GROUCHO`. Depois, usamos o parâmetro `cut` (cortar) para dividir as linhas correspondentes em pedaços e combinar os dois comandos com o pipe (`|`).

O comando `grep` faz uma busca no arquivo `marx.txt` por todas as ocorrências da string que aparecem no início de uma linha (`-E '^Groucho'`), ignorando letras em maiúsculas ou minúsculas (`-i`). O `cut` separa a linha em campos delimitados por dois pontos (`-d ':'`) e seleciona o segundo campo (`-f 2`). O operador pipe transforma a saída de `grep` na entrada para o parâmetro `cut`.

O pipe conecta dois comandos quaisquer, possibilitando a construção de uma longa cadeia de comandos com vários pipes. Por exemplo, se quiser contar o número de palavras ditas por Groucho, adicione `| wc -w` ao comando anterior.

O pipe é apenas uma das formas de redirecionamento. Ferramentas de redirecionamento podem alterar a fonte ou o destino, ou ambos, dos dados processados. O shell oferece formas de redirecionamento também, e aprender a lidar com essas ferramentas é a chave para dominar o shell.

Dados que entram, dados que saem

Caso o `grep` seja executado sozinho, ele irá ler dados do *standard input device* (`stdin` – dispositivo de entrada padrão) e emitir os resultados para o *standard output device* (`stdout` – dispositivo de saída padrão). Os erros são enviados para um terceiro canal chamado *standard error device* (`stderr` – dispositivo de erro padrão).

Normalmente, os dados para o `stdin` são fornecidos pelo usuário através do teclado e, por padrão, o `stdout` e o `stderr` são enviados ao terminal conectado ao shell. No entanto, tudo isso pode ser redirecionado. Por exemplo, é possível redirecionar o `stdin` para que este leia dados de um arquivo ao invés de ler a entrada do teclado. Também é possível redirecionar o `stdout` e o `stderr` (separadamente) para que os dados sejam escritos em outro lugar que não seja a janela do terminal.

A sintaxe do redirecionamento depende do terminal utilizado, mas quase todos eles suportam as seguintes operações:

- `< input_file` redireciona o `stdin` para ler dados de um determinado arquivo (neste caso, `output_file`).
- `> output_file` redireciona o `stdout`, enviando os resultados de um comando ou de um pipe (mas não os erros) para o arquivo especificado (neste caso `output_file`). Se o arquivo não existe, ele é criado; caso exista, seu conteúdo é substituído pelo resultado.
- `>> output_file` é similar a `>` mas adiciona o `stdout` ao conteúdo doarquivo determinado. Caso o arquivo não exista, será criado; no entanto, se ele existir, seu conteúdo é preservado e os resultados são adicionados a ele.
- `> & output_file` funciona como o `>` mas captura `stdout` e `stderr` no arquivo especificado, criando o arquivo caso necessário, e sobrescrevendo seu conteúdo no caso dele já existir.

Alguns exemplos são mostrados na **listagem 2**. O primeiro comando deve ser conhecido. O adicional `>groucho.txt` salva a saída da linha de comando para um arquivo chamado `groucho.txt`. O segundo comando anexa a string `I started work on Nov 2 at 9 am` ao arquivo `timecard.txt`. O terceiro comando executa o script Ruby `muapp.rb`. A entrada é tirada de um arquivo chamado `data` e o `stdout` e o `stderr` do comando são capturados no arquivo log.

Uso avançado do pipe

Caso seja necessário buscar um arquivo específico em seu diretório `home` ou em todo o sistema, o utilitário `find` é imprescindível. Por exemplo, para encontrar todos os arquivos do seu diretório `home` que possuem a palavra "time" no nome, digite (lembre-se que `~` é o substituto de `$HOME`): `$ find ~ -name '*time*'`.

O comando `find` pode buscar arquivos usando vários critérios, incluindo modificações de tempo e tamanho e também pode ser usado como base para todo tipo de análise de arquivos. Por exemplo, considere a seguinte combinação na linha de comando:

```
$ find /caminhos/dos/arquivos ➥
-type f | xargs grep -H -I ➥
-i -n string
```

Esse comando enumera todos os arquivos simples no caminho indicado, procura em cada um deles ocorrências de uma determinada string e gera uma lista de arquivos que contêm a string, incluindo o número da linha no texto específico correspondente. O comando `find` faz uma busca em toda a hierarquia em `/caminho/dos/arquivos`, procurando por arquivos simples (`-type f`). A saída é uma lista de arquivos.

O parâmetro `xargs` é especial, pois inicia um comando – no exemplo, trata-se do `grep` mais todo o resto até o fim da linha – uma vez para cada arquivo listado pelo `find`. As opções `-H` e `-n` prefaciam cada combinação com o nome do arquivo e o número da linha de cada combinação, respectivamente. A opção `-i` ignora maiúsculas e minúsculas. A opção `-I` não computa arquivos binários.

Assumindo que o diretório `/caminho/da/fonte` contém os arquivos `a`, `b` e `c`, o uso do `find` junto com o `xargs` é o equivalente a:

```
$ find /caminho/da/fonte
a
b
c
$ grep -H -I -i -n string a
$ grep -H -I -i -n string b
$ grep -H -I -i -n string c
```

Na verdade, a busca de vários arquivos é tão comum que o `grep` possui sua própria opção para recuperar a hierarquia de um sistema de arquivos. Use o parâmetro `-d recurse` ou seus sinônimos `-R` ou `-r`. Por exemplo, o comando `grep -H -I -i -n -R string /caminho/para/fonte` funciona tão bem quanto a combinação de `find` e `xargs`. No entanto, caso precise ser seletivo e recuperar apenas tipos específicos de arquivos, use o `find`.

Cemitério dos bits

Como vimos, a maioria dos comandos emite saídas de um tipo ou outro. A maioria dos comandos de linha de comando usa `stdout` e `stderr` para mostrar algum tipo de resultado e mensagens de erro, nessa ordem. Se quiser ignorar esse tipo de produção – o que é útil, pois isso geralmente interfere com o trabalho na linha de comando – redirecione sua saída para o "cemitério dos bits", em `/dev/null`. Os bits entram, mas não saem.

A **listagem 3** mostra um exemplo simples. Se redirecionar a saída padrão do comando `cat` para `/dev/null`, nada é exibido (todos os bits são jogados no arquivo virtual vertical). No entanto, caso um erro ocorra, as mensagens de erro, que são enviadas

Listagem 1: Trecho de um script dos irmãos Marx

```
01 GROUCHO: That's what they call a sanity clause.
02 CHICO: Ah, you fool wit me. There ain't no Sanity Claus!
01 $ grep -i -E '^Groucho' marx.txt | cut -d ':' -f 2
02 That's what they call a sanite clause.
```

Listagem 2: Exemplos de redirecionamento

```
01 $ # Primeiro comando
02 $ grep -i -E '^Groucho' marx.txt | cut -d ':' -f 2  > groucho.txt
03 $ cat groucho.txt
04 That's what they call a sanity clause.
05
06 $ cat timecard.txt
07 I started work on Nov 1 at 8.15 am.
08 I finished work on Nov 1 at 5 pm.
09
10 $ # Segundo comando
11 $echo 'I started work on Nov 2 at 9  am.' >> timecard.txt
12
13 $ cat timecard.txt
14 I started work on Nov 1 at 8.15 am.
15 I finished work on Nov 1 at 5 pm.
16 I started work on Nov 2 at 9 am.
17
18 $ # Terceiro comando
19 $ ruby myapp.rb < data >& log
```

para `stderr`, serão exibidas. Se quiser ignorar todas as saídas, utilize o operador `>&` para enviar `stdout` e `stderr` para o cemitério dos bits.

Também é possível usar `/dev/null` como um arquivo sem tamanho para esvaziar arquivos existentes ou criar novos arquivos vazios (**listagem 4**).

Outros truques

Além do redirecionamento, o shell oferece vários outros truques para poupar tempo e esforço. Utilizar argumentos entre crases (`` `...` ``) expande alguns comandos. Uma frase entre crases é executada primeiro, enquanto o shell interpreta a linha de comando, e sua saída substitui a frase original. É possível usar crases para dar prioridade, por exemplo, a um nome de arquivo ou determinado dado:

Listagem 3: O cemitério dos bits

```
01 $ ls
02 secret.txt
03 $ cat secret.txt
04 I am the Walrus.
05 $ cat secret.txt > /dev/null
06 $ cat socrates.txt > /dev/null
07 cat: socrates.txt: No such file or ➥
   directory
08 $ cat socrates.txt >& /dev/null
09 $ echo Done.
10 Done.
```

Listagem 4: Arquivos vazios

```
01 $ cat secret.txt
02 Anakin Skywalker is Darth Vader.
03 $ cp /dev/null secret.txt
04 $ cat secret.txt
05
06 $ echo "The moon is made of ➥
   cheese!" > secret.txt
07 $ cat secret.txt
08 The moon is made of cheese!
09 $ cat /dev/null > secret.txt
10 $ cat secret.txt
11
12 $ cp /dev/null newsecret.txt
13 $ cat newsecret.txt
14
15 $ echo Done.
16 Done.
```

```
$ ps > state.`date '+%F'`
$ ls state*
state.2009-11-21
$ cat state.2009-11-21
13842 ttys001 0:00.54 -bash
30600 ttys001 1:57.15
ruby ./script/server
$ cat `ls state.*`
13842 ttys001 0:00.54 -bash
30600 ttys001 1:57.15
ruby ./script/server
```

A primeira linha de comando captura a lista de processos em execução em um arquivo chamado algo como `state.AAAA-MM-DD`, onde a parte de data do nome é gerada pelo comando `date '+ F%'`. As aspas simples em torno do argumento evitam que o shell interprete `+` e `%`. O último comando mostra um outro exemplo da crase. A execução de `ls state.*` produz um nome de arquivo.

Falando de resultados de captura, se for preciso capturar a saída de uma série de comandos, combine-os dentro de chaves ({...}): `$ { ps; w } > state.`date '+%F'``. No comando anterior, `ps` é executado, seguido por `w` (que mostra quem está usando a máquina) e a saída de tudo é capturada em um arquivo.

É possível também colocar uma sequência de comandos entre parênteses para alcançar o mesmo resultado, com uma diferença importante: a série de comandos entre parênteses é executada em um subshell, e não afeta o estado do shell corrente.

Por exemplo, espera-se que o comando `{ cd $HOME; ls -1}; pwd` produza a mesma saída de `(cd $HOME; ls); pwd`; no entanto, os comandos entre chaves alteram o diretório de trabalho do shell corrente. A segunda técnica não faz isso.

O uso de uma combinação ou um subshell depende de suas intenções, embora o subshell seja muito mais poderoso. É possível usar um subshell para expandir um comando, assim como com crases. Melhor ainda, um subshell pode conter outro subshell, desse modo uma expansão pode conter outra.

O comando a seguir: `$ {ps; w} > state.$(date '+%F')` é idêntico a `{ ps; w } > state.`date '+%F'``. A notação `$()` executa os comandos dentro dos parênteses que depois são substituídos pela saída. Em outras palavras, `$()` expande o comando, do mesmo modo que a crase; no entanto, diferentemente da crase, utilizar `$()` pode ser bem complexo e pode até incluir outras expansões `$()`:

```
$ (cd $(grep strike /etc/passwd | 
cut -f6 -d':'); ls)xw
```

Esse comando busca o arquivo de senha do sistema para encontrar uma entrada para o usuário *strike*, retém o campo do diretório `home` (campo 6 no arquivo de senhas, contando a partir do zero), vai até esse diretório e lista seu conteúdo. A saída de `grep /etc/passwd strike | cut -f6 -d':'` é expandida antes de qualquer outra operação. O subshell possui muitas utilizações, por isso é preferível usá-lo no lugar dos operadores `{ }` ou das crases.

Conclusão

Consulte a documentação do seu terminal shell para aprender suas características especiais e truques. Se estiver usando um shell mais recente em vez do Bash, é possível encontrar opções adicionais disponíveis. Por exemplo, o Z Shell fornece redirecionamento multi-way para ler vários arquivos de entrada e emitir várias cópias de saída. ■

Gostou do artigo?

Queremos ouvir sua opinião. Fale conosco em cartas@linuxmagazine.com.br

Este artigo no nosso site: http://lnm.com.br/article/4572