105.2: Scripts shell

Rafael Obelheiro

13/09/2024

Desenvolvimento de scripts

- Scripts raramente são desenvolvidos do zero diretamente como scripts
- Mais comum é agrupar sequências de comandos que realizam tarefas repetidas
  - muitas vezes acrescentando parametrização, tratamento de erros, formatação da saída, etc.

Desenvolvimento incremental de scripts

1. Desenvolva o script (ou partes dele) como um pipeline, um passo de cada vez, inteiramente na linha de comando
   - use Bash para este processo
2. Envie a saída para a saída padrão e verifique se está tudo certo
   - em vez de executar diretamente os comandos, use `echo` para mostrar como eles ficarão
3. A cada passo, use o histórico de comandos do shell para recuperar pipelines anteriores e os mecanismos de edição para refiná-los
4. Enquanto a saída não parecer correta, você não executou realmente nada, e não haverá nada a desfazer caso o comando esteja incorreto
5. Quando a saída estiver correta, execute os comandos e verifique que eles funcionam conforme o desejado
   - remova os `echo`s de depuração
6. Recupere o que você fez (use o histórico), deixe organizado e guarde como um script

Tratamento de erros

```
#!/bin/sh

show_usage() {
    echo "Usage: $0 source_dir dest_dir" 1>&2
    exit 1
}

# Main program starts here

if [ $# -ne 2 ]; then
    show_usage
else # There are two arguments
    if [ -d $1 ]; then
        source_dir=$1
    else
        echo 'Invalid source directory' 1>&2
        show_usage
    fi
    if [ -d $2 ]; then
        dest_dir=$2
    else
        echo 'Invalid destination directory' 1>&2
        show_usage
    fi
fi

printf "Source directory is ${source_dir}\n"
printf "Destination directory is ${dest_dir}\n"
```

Figure 1

Flags comuns de `test` (1)

Elementary sh comparison operators

String	Numeric	True if
`x = y`	`x -eq y`	x is equal to y
`x != y`	`x -ne y`	x is not equal to y
`x <`[a] `y`	`x -lt y`	x is less than y
n/a	`x -le y`	x is less than or equal to y
`x >`[a] `y`	`x -gt y`	x is greater than y
n/a	`x -ge y`	x is greater than or equal to y
`-n x`	*n/a*	x is not null
`-z x`	*n/a*	x is null

a. Must be backslash-escaped or double bracketed to prevent interpretation as an input or output redirection character.

Figure 2

Flags comuns de `test` (2)

sh file evaluation operators

Operator	True if
`-d` *file*	*file* exists and is a directory
`-e` *file*	*file* exists
`-f` *file*	*file* exists and is a regular file
`-r` *file*	User has read permission on *file*
`-s` *file*	*file* exists and is not empty
`-w` *file*	User has write permission on *file*
file1 `-nt` *file2*	*file1* is newer than *file2*
file1 `-ot` *file2*	*file1* is older than *file2*

Figure 3

Argumentos de linha de comando

- O comando `shift [N]` remove o primeiro argumento (ou mais, se N for especificado)
  - o número de argumentos (`$#`) é decrementado de N
- É recomendável usar aspas ao referenciar os argumentos
  - minimiza problemas com argumentos contendo espaços ou não informados
- Exemplo: listar todos os argumentos passados para um script

```
i=0
while [ $# -gt 0 ]
do
    ((i=i+1))
    printf "%2d: %s\n" $i "$1"
    shift
done
```


Shebang e variações de nome de caminho

- A especificação de interpretador de comandos para scripts (`#!`) requer um nome de caminho absoluto
- Especialmente no Linux, o Bash costuma residir em `/bin`
- Em alguns sistemas, porém, o Bash pode ser encontrado em outros diretórios, notadamente `/usr/local/bin`
- Para outros interpretadores de comandos (como Python ou Ruby), a variação é maior
- Uma forma de minimizar tais variações é usar `env`:

```
#! /usr/bin/env bash
```

  - isso invoca o primeiro `bash` no `PATH`
- Possíveis problemas
  - suscetível a incompatibilidade de versões de interpretadores
  - não é garantido que `env` resida em `/usr/bin`

if/elif/else

- Além de `if/else`, existe `if/elif/else`
- Exemplo simples:

```
if [ "$1" -lt 0 ] 2>/dev/null ; then
    echo "negativo"
elif [ "$1" -gt 0 ] 2>/dev/null ; then
    echo "positivo"
elif [ "$1" -eq 0 ] 2>/dev/null ; then
    echo "zero"
else
    echo "erro no argumento" >&2
    exit 1
fi
```


case

- Os padrões usados no comando `case` podem usar metacaracteres
  - é usado *globbing* para o casamento de padrões
- Exemplo simples:

```
read -n1 -p "Pressione uma letra ou digito: " opcao
echo
case $opcao in
  [0-9])
    echo "Voce pressionou o digito $opcao" ;;
  [a-z]|[A-Z])
    echo "Voce pressionou a letra $opcao" ;;
  *)
    echo "Voce nao sabe o que sao letras e digitos" ;;
esac
```


Gerando sequências para `for` (1)

- O comando `for` itera sobre uma lista de itens, que não precisa ser uma sequência numérica

```
for f in *.c *.h ; do
    ...
done
```

- Existem diferentes formas de gerar sequências numéricas

	1 a 10	10 a 1	2 a 10, passo 2
específica do Bash	`{1..10}`	`{10..1}`	`{2..10..2}`
`seq`	`seq 10`	`seq 10 -1 1`	`seq 2 2 10`
`jot`	`jot 10`	`jot 10 10 1`	`jot 5 2 10`

  - `seq -s " "` para mudar o separador (default é `\n`)
  - `jot`: primeiro argumento é a quantidade de números gerados

Gerando sequências para `for` (2)

- O Bash faz a expansão de chaves (`{1..10}`) antes de outras expansões e substituições
  - não é possível parametrizar o laço com uma variável (`{1..$n}`)
  - se a expansão gerar muitos elementos, usar chaves pode ser lento e consumir muita memória
- Outra opção específica do Bash é usar o chamado laço aritmético:

```
for ((i = 1; i <= n; i++)) ; do
    echo $i
done
```

  - considerada melhor do que expansão de chaves

Obtendo entrada: `read`

- O comando interno `read` pode ser usado para ler uma entrada

```
echo -n "Digite seu nome: "
read nome
read -p "Digite sua idade: " idade
read -p "Digite sua cidade natal: "
echo ${nome} nasceu em ${REPLY}, e tem ${idade} anos.
```

  - `$REPLY` é específico do Bash
- Outras opções úteis

opção	significado
`-s`	não mostra caracteres
`-n 1`	retorna após ler 1 caracter
`-t 5`	retorna erro se usuário não responder em 5 seg

Processando linhas com `read`

```
i=1
cat "$1" | while read linha
do
  printf "%5d %s\n" $i "$linha"
  ((i = i+1))
done
```


Depurando scripts

- A opção `-n` faz com que o shell apenas analise a sintaxe do script, sem executar os comandos
- A opção `-x` faz com que o shell mostre os comandos que são executados
- O mesmo efeito pode ser obtido usando `set -x`
  - válido apenas no shell que executa o script
  - se necessário, `set +x` restabelece o comportamento padrão

Aritmética inteira no shell

- Forma mais portável:

```
expr 3 + 4
i=$(expr $i + 1)
x=`expr 3 \* $y + $z / 4`
```

- Bash oferece notações mais simples:

```
$[ 3+4 ]      i=$[ i+1 ]     x=$[ 3*y + z/4 ]
$(( 3+4 ))    (( i=i+1 ))    (( x=3*y + z/4 ))
```