bash-programming-how-to-translate Documentation

Release 0.0.1

Ciro S. Costa

October 18, 2016

Contents

O documento original redigido por Mike G pode ser encontrado na seção de HOW-TOs da The Linux Documentation Project.

Este não é um guia avançado, mas bastante simplificado e direto quanto a conceitos e funções básicas. Não é a pretenção deste abordar a fundo os tópicos, o que pode ser percebido inclusive pelo tamanho destes.

De grande valia este documento pode ser para aqueles que desejem iniciar algo rápido para possuir alguma base para futuros estudos.

Para aprendizado posterior a este recomendo a leitura do Advanced Bash-Scripting Guide de Mendel Cooper.

CHAPTER 1

Este Documento

Este documento busca primariamente ser uma fonte de futura referência para o autor e, quem sabe, para a comunidade. Mais do que apenas traduzir, o autor espera que ao apresentar ao público geral um simples documento praticamente puro de HTML numa forma mais iterativa/visualmente agradável, incentivar (ou tentar não desistimular) um número maior de pessoas.

CHAPTER 2

Guia

2.1 Introduction

2.1.1 Obtendo a última versão

http://www.linuxdoc.org/HOWTO/Bash-Prog-Intro-HOWTO.html

2.1.2 Requisitos

Familiaridade com linhas de comando GNU/Linux e também com conceitos básicos de programação são desejáveis. Ao passo que este não se trata de uma introdução à programação, explica (ao ao menos tenta) muito conceitos básicos.

2.1.3 Usos deste documento

Este documento tenta ser útil para as seguinte situações:

- Você possui alguma ideia sobre programação e deseja começar a programar alguns script para shell.
- Você possui uma vaga ideia sobre programação para shell e deseja alguma referência.
- Você quer ver alguns scripts e comentários para começar a escrever seus próprios.
- Você está migranto do DOS/Windows (ou já fez) e deseja fazer alguns "batch process" (execução de uma série de programas)
- Você é um completo nerd e lê tudo o que há de disponível.

2.2 Scripts muito simples

Este *COMO FAZER* lhe dará algumas dicas sobre como programação por schell script é fortemente baseada em exemplos.

Nesta seção você encontrará alguns pequenos scripts que com sorte lhe ajudarão a entender algumas técnicas.

2.2.1 O tradicional script de Hello World

```
#!/bin/bash
echo Hello World
```

O script possui apenas duas linhas. A primeira indica o sistema que o programa usa para rodar o arquivo.

A segunda linha é a única ação performada pelo script, que printa 'Hello World' no terminal.

Se você receber algo parecido com *./hello.sh: Command not found* provavelmente a primeira linha está errada. Envie `whereis bash` ou busque por `buscando bash` para ver como escrever esta linha.

2.2.2 Um script de backup muito simples

```
#!/bin/bash
tar -cZf /var/my-backup.tgz /home/me/
```

Neste script, ao invés de printar a mensagem no terminal, criamos um arquivo tar-ball com o diretorio completo do usuario.

Isto não é pretendido para ser utilizado. Um script mais útil de backup será apresentado mais à frente neste documento.

2.3 Tudo sobre redirecionamento

2.3.1 Teoria e rápida referência

Existem três descritores de arquivo: **stdin**, **stdout** e **stderror** (sendo std = Standard = Padrão).

Basicamente você pode:

- redirecionar **stdout para** um **arquivo**
- redirecionar **stderr para** um **arquivo**
- redirecionar **stdout para** um **stderr**
- redirecionar **stderr para** uma **stdout**
- redirecionar **stderr e stdout** para **arquivo**
- redirecionar **stderr e stdout ** para **stdout**
- redirecionar **stderr e stdout** para **stderr**

Em ShellScript **1** representará *stdout* e **2** representará *stderr*.

Uma pequena anotação quanto à visualização destes: com o comando *less* você pode ver tanto o stdout (que permanecerá no buffer) quanto o stderr que será printado na tela, mas apagado quando tentar 'navegar' (browse) pelo buffer.

2.3.2 Exemplo: stdout para arquivo

Isto fará com que a saida do programa seja escrita no arquivo.

```
ls -l > ls-l.txt
```

Aqui, o arquivo chamado *ls-l.txt* será criado e então conterá o que poderia ser visto na tela caso o comando *ls -l* fosse executado.

2.3.3 Exemplo: stderr 2 arquivo

Isto fará com que a saída do **stderr** seja escrita para um arquivo.

```
grep da * 2> grep-errors.txt
```

Aqui o arquivo chamado *grep-errors.txt' será criado e então conterá o que poderia ser visto da porção do **stderr** da saída do comando 'grep da* *.

2.3.4 Exemplo: stdout 2 stderr

Isto fara com que a saída de **stderr** do programa seja escrita para o mesmo arquivo do **stdout**.

```
grep da * 1>&2
```

Here, the stdout portion of the command is sent to stderr, you may notice that in differen ways. 3.5 Exemplo: stderr 2 stdout

This will cause the stderr ouput of a program to be written to the same arquivodescriptor than stdout.

```
grep * 2>&1
```

Here, the stderr portion of the command is sent to stdout, if you pipe to less, you'll see that lines that normally 'dissapear' (as they are written to stderr) are being kept now (because they're on stdout). 3.6 Exemplo: stderr and stdout 2 arquivo

This will place every output of a program to a arquivo. This is suitable sometimes for cron entries, if you want a command to pass in absolute silence.

```
rm -f $(find / -name core) &> /dev/null
```

This (thinking on the cron entry) will delete every arquivo called 'core' in any directory. Notice that you should be pretty sure of what a command is doing if you are going to wipe it's output.

2.4 Pipes

Esta seção explicará de uma maneira bem breve e prática como se utilizar pipes (e você deve querê-la).

2.4.1 O que eles são e porque vai usá-los

Pipes deixam você utilizar (de uma maneira muito simples, insisto) a saída de um programa como a entrada de outro.

2.4.2 Pipe simples com 'sed'

Esta é uma maneira muito simples de utilizar pipes:

```
ls -l | sed -e "s/[aeio]/u/g"
```

O que acontece: primeiro o comando *ls -l* é executado e sua saída, ao invés de printada, é enviada (piped) para o programa *sed*, que, no caso, printa sua saída.

2.4.3 Exemplo: uma alternativa para ls -l *.txt

Provavelmente esta seja uma maneira mais difícil do que paneas fazer *ls -l *.txt*, porém deseja-se apenas ilustrar o uso dos pipes, e não resolver tal dilema.

```
ls -l | grep "\.txt$"
```

Aqui a saída do programa *ls l* é enviada para o *grep*, que, no caso, irá printar linhas cujo padrão combinam com a expressão regulas (regex) *\txt$*

2.5 Variáveis

Voc pode usar variáveis assim como em qualquer outra linguagem de programação. Não há tipos de dados. A variável em bash pode conter um numero, um caracter ou uma lista de caracteres.

Não há necessidade que a variável seja declarada, apenas assinalar um valor há sua referência irá criá-la.

2.5.1 Exemplo: Hello World! usando variáveis

```
#!/bin/bash
STR="Hello World"
echo $STR
```

A segunda linha cria uma variável chamada STR e a ela assinala a string "Hello World". Então o *valor* de sua variável é obtido colocando o *$'(*dollar-sign*) antes da mesma. Note (e tente) que se não utilizar o '$* a saída do programa será diferente, e provavelmente não o que quer que seja.

2.5.2 Exemplo: Um script de backup muito simples (um pouco melhor)

```
#!/bin/bash
OF=/var/my-backup-$(data +%Y%m%d).tgz
tar -cZf $OF /home/me
```

Este script introduz outra coisa. Pirmeiro de tudo você deve familiarizar-se com a criação e atribuição de valor à variável na segunda linha. Note a expressão *$(date +%Y%m%d)*. Se você rodar este script irá notar que é chamado o comando de dentro do parêntesis e é então capturado sua saída.

Note que neste script o arquivo de saída será diferente a cada dia, dado que o mesmo é alterado pelo comando de formatação de datas *(date +%Y%m%d)*. (Você pode alterar isto especificando um formato diferente).

Outro exemplo: .. code-block:: bash

    #!/bin/bash echo ls echo $(ls)

2.5.3 Variáveis Locais

Variáveis locais podem ser criadas utilizando a palavra chave *local*.

```
#!/bin/bash
HELLO=Hello
function hello {
        local HELLO=World
        echo $HELLO
```

```
}
echo $HELLO
hello
echo $HELLO
```

Este exemplo deve ser o suficiente para mostrar como se utiliza uma variável local.

2.6 Condicionais

Condicionais dão-lhe o poder de decidir entre perfomar uma ação ou não após ser verificada uma expressão.

2.6.1 Teoria Pura

Condicionais possuem várias formas. Em sua forma mais básica:

```
se expressao, então
        declaração
```

é apenas executada se a expressão for verificada como Verdade. Por exemplo:

```
# Verdadeiro
2 < 1
#Falso
2 > 1
```

Condicionais possuem outras formas tais como:

```
se condição, então
        declaração1
caso contrário
        declaração2
```

Neste caso *declaração1* é executada apenas se *condição* é verdadeira, caso contrário *declaração2* é executada.

Enquanto outra forma de condicional é:

```
se condição1, então
        declaraçaõ1
senão se condição2, então
        declaração2
caso contrário
        declaração3
```

Neste caso apenas é adicionada a condição da *condição2* para ser avaliada também, que chamará a *declaração2* se a condição for verdadeira. O restante é como pode ser imaginado (veja anteriores).

Sobre a sintaxe:

A base dos *se* e *senão* (**if** e **else**) em bash é:

```
if [expression];
then
code if 'expression' is true.
fi
```

2.6.2 Exemplo: Condicional básica if .. then

```
 #!/bin/bash
if [ "foo" = "foo" ]; then
   echo expressão avaliada como verdadeira
fi
```

O código a ser executado se a expressão entre colchetes é verdadeira pode ser encontrada depois da palavra *then* e antes do *fi*, que indica o fim de um código condicionalmente executado.

2.6.3 Exemplo: Condicional básica if .. then ... else

```
 #!/bin/bash
if [ "foo" = "foo" ]; then
   echo expressão avaliada como verdadeira
else
   echo expressão avaliada como false
fi
```

2.6.4 Exemplo: Condicionais com Variáveis

```
 #!/bin/bash
T1="foo"
T2="bar"
if [ "$T1" = "$T2" ]; then
    echo expressão avaliada como verdadeira
else
    echo expressão avaliada como false
fi
```

2.7 Loops, For, While e Until

Nesta seção você aprenderá sobre os loops *for*, *while* e *until*.

O loop *for* é um pouco diferente daquele conhecido em outras linguagens de programaão. Basicamente ele itera sobre uma série de *palavras* dentro de uma lista.

O *while* executa um pedaço de código se a expressão de controle é avaliada como **verdadeira** e apenas para quando esta for então avaliada como **falsa** (ou então explicitamente parada por um break no código).

O loop *until* é praticamente igual ao *while*, exceto pelo fato deste ser executado enquando a expressão de controle é avaliada como *falsa*.

Se suspeitar que o **while** assemelha-se muito ao **until** você está correto.

2.7.1 Por Exemplo

```
 #!/bin/bash
for i in $( ls ); do
    echo item: $i
done
```

On the second line, we declare i to be the variable that will take the different values contained in $(ls).

The third line could be longer if needed, or there could be more lines before the done (4).

'done' (4) indicates that the code that used the value of $i has finished and $i can take a new value.

This script has very little sense, but a more useful way to use the for loop would be to use it to match only certain files on the previous example

2.7.2 'for' parecido com C

fiesh suggested adding this form of looping. It's a for loop more similar to C/perl... for.

```
 #!/bin/bash
for i in `seq 1 10`;
do
        echo $i
done
```

2.7.3 Exemplo While

```
 #!/bin/bash
COUNTER=0
while [  $COUNTER -lt 10 ]; do
    echo The counter is $COUNTER
    let COUNTER=COUNTER+1
done
```

This script 'emulates' the well known (C, Pascal, perl, etc) 'for' structure

2.7.4 Exemplo Until

```
 #!/bin/bash
COUNTER=20
until [  $COUNTER -lt 10 ]; do
    echo COUNTER $COUNTER
    let COUNTER-=1
done
```

2.8 Functions

As in almost any programming language, you can use functions to group pieces of code in a more logical way or practice the divine art of recursion.

Declaring a function is just a matter of writing function my_func { my_code }.

Calling a function is just like calling another program, you just write its name.

2.8.1 Exemplo de Função

```
 #!/bin/bash
function quit {
    exit
}
function hello {
    echo Hello!
}
hello
quit
echo foo
```

Lines 2-4 contain the 'quit' function. Lines 5-7 contain the 'hello' function If you are not absolutely sure about what this script does, please try it!.

Notice that a functions don't need to be declared in any specific order.

When running the script you'll notice that first: the function 'hello' is called, second the 'quit' function, and the program never reaches line 10.

2.8.2 Exemplo de função com parâmetro

```
 #!/bin/bash
function quit {
   exit
}
function e {
    echo $1
}
e Hello
e World
quit
echo foo
```

This script is almost identically to the previous one. The main difference is the funcion 'e'. This function, prints the first argument it receives. Arguments, within funtions, are treated in the same manner as arguments given to the script.

2.9 User Interfaces

2.9.1 Usando 'select' para criar menus simples

```
 #!/bin/bash
OPTIONS="Hello Quit"
select opt in $OPTIONS; do
    if [ "$opt" = "Quit" ]; then
     echo done
     exit
    elif [ "$opt" = "Hello" ]; then
     echo Hello World
    else
     clear
     echo bad option
    fi
done
```

If you run this script you'll see that it is a programmer's dream for text based menus. You'll probably notice that it's very similar to the 'for' construction, only rather than looping for each 'word' in $OPTIONS, it prompts the user.

2.9.2 Usando a linha de comando

```
#!/bin/bash
if [ -z "$1" ]; then
  echo usage: $0 directory
  exit
fi
SRCD=$1
TGTD="/var/backups/"
OF=home-$(date +%Y%m%d).tgz
tar -cZf $TGTD$OF $SRCD
```

What this script does should be clear to you. The expression in the first conditional tests if the program has received an argument ($1) and quits if it didn't, showing the user a little usage message. The rest of the script should be clear at this point.

2.10 Misc

2.10.1 10.1 Reading user input with read

In many ocations you may want to prompt the user for some input, and there are several ways to achive this. This is one of those ways:

```
 #!/bin/bash
echo Please, enter your name
read NAME
echo "Hi $NAME!"
```

As a variant, you can get multiple values with read, this example may clarify this.

```
 #!/bin/bash
echo Please, enter your firstname and lastname
read FN LN
echo "Hi! $LN, $FN !"
```

2.10.2 10.2 Arithmetic evaluation

On the command line (or a shell) try this:

echo 1 + 1

If you expected to see '2' you'll be disappointed. What if you want BASH to evaluate some numbers you have? The solution is this:

echo $((1+1))

This will produce a more 'logical' output. This is to evaluate an arithmetic expression. You can achieve this also like this:

echo $[1+1]

If you need to use fractions, or more math or you just want it, you can use bc to evaluate arithmetic expressions.

if i ran *echo $[3/4]* at the command prompt, it would return 0 because bash only uses integers when answering. If you ran *echo 3/4|bc -l*, it would properly return 0.75.

2.10.3 10.3 Finding bash

From a message from mike (see Thanks to)

you always use *#!/bin/bash* .. you might was to give an example of

how to find where bash is located.

locate bash is preferred, but not all machines have locate.

find ./ -name bash from the root dir will work, usually.

Suggested locations to check:

```
ls -l /bin/bash
ls -l /sbin/bash
ls -l /usr/local/bin/bash
ls -l /usr/bin/bash
ls -l /usr/sbin/bash
ls -l /usr/local/sbin/bash
```

(can't think of any other dirs offhand... i've found it in

most of these places before on different system).

You may try also 'which bash'.

2.10.4 10.4 Getting the return value of a program

In bash, the return value of a program is stored in a special variable called $?.

This illustrates how to capture the return value of a program, I assume that the directory dada does not exist. (This was also suggested by mike)

```
 #!/bin/bash
cd /dada &> /dev/null
echo rv: $?
cd $(pwd) &> /dev/null
echo rv: $?
```

2.10.5 10.5 Capturing a commands output

This little scripts show all tables from all databases (assuming you got MySQL installed). Also, consider changing the 'mysql' command to use a valid username and password.

```
 #!/bin/bash
DBS=`mysql -uroot  -e"show databases"`
for b in $DBS ;
do
        mysql -uroot -e"show tables from $b"
done
```

2.10.6 10.6 Multiple source files

You can use multiple files with the command source.

2.11 Tabelas

2.11.1 11.1 String comparison operators

1. s1 = s2
2. s1 != s2
3. s1 < s2
4. s1 > s2
5. -n s1
6. -z s1

1. s1 matches s2
2. s1 does not match s2
3. __TO-**DO**__
4. __TO-**DO**__
5. s1 is not null (contains one or more characters)
6. s1 is null

2.11.2 11.2 String comparison examples

Comparing two strings.

```
    #!/bin/bash
S1='string' S2='String' if [ $S1=$S2 ]; then
    echo "S1('$S1') is not equal to S2('$S2')"
fi if [ $S1=$S1 ]; then
    echo "S1('$S1') is equal to S1('$S1')"
fi
```

I quote here a note from a mail, sent buy Andreas Beck, refering to use if [$1 = $2].

This is not quite a good idea, as if either $S1 or $S2 is empty, you will get a parse error. x$1=x$2 or "$1"="$2" is better.

2.11.3 11.3 Arithmetic operators

-
-
-

/

% (remainder)

2.11.4 11.4 Arithmetic relational operators

-lt (<)

-gt (>)

-le (<=)

-ge (>=)

-eq (==)

-ne (!=)

C programmer's should simple map the operator to its corresponding parenthesis.

2.11.5 11.5 Useful commands

This section was re-written by Kees (see thank to...)

Some of these command's almost contain complete programming languages. From those commands only the basics will be explained. For a more detailed description, have a closer look at the man pages of each command.

sed (stream editor)

Sed is a non-interactive editor. Instead of altering a file by moving the cursor on the screen, you use a script of editing instructions to sed, plus the name of the file to edit. You can also describe sed as a filter. Let's have a look at some examples:

    $sed 's/to_be_replaced/replaced/g' /tmp/dummy

Sed replaces the string 'to_be_replaced' with the string 'replaced' and reads from the /tmp/dummy file. The result will be sent to stdout (normally the console) but you can also add '> capture' to the end of the line above so that sed sends the output to the file 'capture'.

    $sed 12, 18d /tmp/dummy

Sed shows all lines except lines 12 to 18. The original file is not altered by this command.

awk (manipulation of datafiles, text retrieval and processing)

Many implementations of the AWK programming language exist (most known interpreters are GNU's gawk and 'new awk' mawk.) The principle is simple: AWK scans for a pattern, and for every matching pattern a action will be performed.

Again, I've created a dummy file containing the following lines:

"test123

test

tteesstt"

    $awk '/test/ {print}' /tmp/dummy

test123

test

The pattern AWK looks for is ‘test’ and the action it performs when it found a line in the file /tmp/dummy with the string ‘test’ is ‘print’.

$awk ‘/test/ {i=i+1} END {print i}’ /tmp/dummy

3

When you’re searching for many patterns, you should replace the text between the quotes with ‘-f file.awk’ so you can put all patterns and actions in ‘file.awk’.

grep (print lines matching a search pattern)

We’ve already seen quite a few grep commands in the previous chapters, that display the lines matching a pattern. But grep can do more.

$grep “look for this” /var/log/messages -c

12

The string “look for this” has been found 12 times in the file /var/log/messages.

[ok, this example was a fake, the /var/log/messages was tweaked :-)]

wc (counts lines, words and bytes)

In the following example, we see that the output is not what we expected. The dummy file, as used in this example, contains the following text: “bash introduction howto test file”

$wc –words –lines –bytes /tmp/dummy

2 5 34 /tmp/dummy

Wc doesn’t care about the parameter order. Wc always prints them in a standard order, which is, as you can see: .

sort (sort lines of text files)

This time the dummy file contains the following text:

“b

c

a”

$sort /tmp/dummy

This is what the output looks like:

a

b

c

Commands shouldn’t be that easy :-) bc (a calculator programming language)

Bc is accepting calculations from command line (input from file. not from redirector or pipe), but also from a user interface. The following demonstration shows some of the commands. Note that

I start bc using the -q parameter to avoid a welcome message.

$bc -q

1 == 5

0

0.05 == 0.05

1

5 != 5

0

2 ^ 8

256

sqrt(9)

3

while (i != 9) {

i = i + 1;

print i

}

123456789

quit

tput (initialize a terminal or query terminfo database)

A little demonstration of tput's capabilities:

```
$tput cup 10 4
```

The prompt appears at (y10,x4).

```
$tput reset
```

Clears screen and prompt appears at (y1,x1). Note that (y0,x0) is the upper left corner.

```
$tput cols
```

80 Shows the number of characters possible in x direction.

It it higly recommended to be familiarized with these programs (at least). There are tons of little programs that will let you do real magic on the command line.

[some samples are taken from man pages or FAQs]

2.12 Mais Scripts

2.13 Quando algo dá errado

2.14 Sobre o documento

CHAPTER 3

Índices e Tabelas

- genindex
- modindex
- search