Reativa Tecnologia

Linux: Apostila Completa de Bash

Vamos mergulhar nos comandos do Linux!

> ***Dica**: Dê uma olhada em **learnyoubash** — são exercícios interativos baseados nesse documento!*

Instalando o guia através do Node

Você pode instalar esse documento usando `npm`. Execute:

```
$ npm install -g bash-handbook
```

Você será capaz de executar `bash-handbook` na sua linha de comando.

Índice

- Introdução
- Estilos do shell
  - Interativo
  - Não-interativo

    - Códigos de saída
- Comentários
- Variáveis
    - Variáveis locais
    - Variáveis de ambiente
    - Parâmetros de posição
- Expansões do shell
    - Expansões de suporte
    - Substituição de comandos
    - Expansões aritiméticas
    - Aspas simples e duplas
- Arrays
    - Declarando array
    - Expansões de Array
    - Separando Array
    - Adicionando elementos no Array
    - Deletando elementos de um Array
- Streams, pipes e listas
    - Streams
    - Pipes
    - Lista de comandos
- Operadores condicionais
    - Expressões primárias e combinação de expressões
    - Usando a condicional `if`
    - Usando a condicional `case`
- Loops
    - `for` loop

    - `while` loop
    - `until` loop
    - `select` loop
    - Controlando o loop
- Funções
- Depurando
- Posfácio
- Licença

Introdução

Se você é um desenvolvedor, então você sabe o valor que o tempo tem. Otimizar seu processo de trabalho é um dos mais importantes aspectos do seu dia-a-dia.

E, se entrarmos no caminho em direção à eficiência e produtividade, sempre esbarramos em ações que serão repetidas uma vez ou outra, como:

- tirar um *screenshot* e fazer o upload para um servidor
- processar texto em vários formatos
- converter arquivos entre diferentes formatos
- analisar o resultado da execução de um programa

Entra em cena, o **Bash**, nosso salvador.

Bash é um shell Unix escrito por Brian Fox no formato de software livre para o projeto GNU, com a intenção de substituir o Bourne shell. Ele foi lançado em 1989 e tem sido distribuído como shell padrão no Linux e macOS a um longo tempo.

E porque nós precisamos aprender algo que foi escrito a mais de 30 anos? A resposta é simples: essa *coisa*, hoje em dia, é uma das mais poderosas e portáveis ferramentas para escrever scripts para todos os sitemas baseados em Unix. E isso é a razão pela qual você deve aprender bash. Ponto.

Nesse manual, eu vou descrever os conceitos mais importantes do bash através de exemplos. Eu espero que seja útil para você e que você possa aprender algo através deles.

Estilos do shell

O usuário do shell bash pode trabalhar em dois modos - interativo e não-interativo.

Modo Interativo

Se você estiver trabalhando no Ubuntu, você tem sete terminais virtuais disponíveis para você. O ambiente de trabalho se posiciona no sétimo terminal virtual. Você pode voltar para uma GUI mais amigável usando o atalho `Ctrl-Alt-F7`.

Você pode abrir o shell usando o atalho `Ctrl-Alt-F1`. Depois disso, a GUI que você acostuma utilizar irá desaparecer e um dos terminais virtuais será mostrado.

Se você ver algo parecido com isso, então, você está trabalhando no modo interativo:

```
user@host:~$
```

Aqui você pode digitar uma variedade de comandos Unix, como `ls`, `grep`, `cd`, `mkdir`, `rm` e ver o resultado das suas execuções.

Chamamos isso de shell interativo porque ele interage diretamente com o usuário.

Usar um terminal virtual, nem sempre é conveniente. Por exemplo, se você quiser editar um documento e executar um comando ao mesmo tempo, é melhor você usar um emulador de terminais virtuais, como:

- GNOME Terminal
- Terminator
- iTerm2
- ConEmu

Modo não-interativo

No modo não-interativo, o shell recebe comandos de um arquivo ou um *pipe* e executa eles. Quando o interpretador chega no final do arquivo, a sessão de processamento do shell é terminada e o processo anterior é retornado.

Use os seguintes comandos para executar o shell em modo não-interativo:

```
. /path/to/script.sh
bash /path/to/script.sh
```

No exemplo acima, `script.sh` é apenas um arquivo de texto comum, contendo comandos, que o interpretador shell pode executar. `sh` ou `bash` são interpretadores utilizados pelo shell. Você pode criar um `script.sh` usando seu editor de texto preferido (e.g. vim, nano, Sublime Text, Atom, etc).

Você também pode simplificar a invocação do script transformando o arquivo em um executável usando o comando `chmod`:

```
chmod +x /path/to/script.sh
```

Além disso, a primeira linha do script deve indicar qual programa deve ser usado para executar o arquivo, como:

```
#!/bin/bash
echo "Hello, world!"
```

Ou, se você preferir usar `sh` ao invés do `bash`, mude `#!/bin/bash` para `#!/bin/sh`. Essa sequência de carácteres `#!`, é conhecida como shebang. Agora você pode executar scripts da seguinte maneira:

```
/path/to/script.sh
```

Um truque útil que usamos acima, é usar o comando `echo` para imprimir o texto na tela do terminal.

Uma outra maneira de usar o *shebang* é:

```
#!/usr/bin/env bash
echo "Hello, world!"
```

A vantagem desse modo de uso do *shebang* é que ele irá utilizar o programa (nesse caso o `bash`) baseado no caminho `PATH` do seu ambiente. Esse modo é, muitas vezes, preferido, ao invés de usar o primeiro método mostrado acima, onde a localização do programa no seu ambiente, pode não ser a mesmo. Isso também é útil se a variável `PATH`, em um sistema, estiver configurada para uma versão diferente do programa. Um exemplo, seria a instalação de uma nova versão do `bash`, enquanto preservamos a versão original e inserimos a localização da nova versão na variável `PATH` do sistema. O uso do `#!/bin/bash` pode resultar no uso da versão original do `bash`, enquanto, `#!/usr/bin/env bash`, fará uso da nova versão.

Códigos de saída

Todo comando retorna um **código de sáida** (**retornando o estado** ou o **estado de saída**). Um comando executado com sucesso, sempre retorna `0` (código-zero), e um comando executado com falha, sempre retorna um valor não-zero (código de erro). Códigos de falhas devem conter um número inteiro positivo entre 1 e 255.

Outro comando útil que nós podemos usar quando escrevemos scripts é o `exit`. Esse omando é usado para finalizar a execução atual e retornar um código de saída para o shell. Executando o `exit`, sem nenhum argumento, irá terminar o script que está em processamento e retornar o código de saída do último comando executado antes do `exit`.

Quando um programa é finalizado, o shell atribui ao seu **código de saída** há variável `$?`. A variável `$?`, é o que normalmente usamos para testar se um script foi executado com sucesso ou não.

Do mesmo modo que podemos usar `exit` para terminar um script, nós podemos usar o comando `return` para sair de uma função e retornar o **código de saída** para quem invocou essa função. Você também pode usar `exit` dentro de uma função, isso irá resultar na saída da função *e* na finalização do programa.

Comentários

Scripts podem conter *comentários*. Comentários são declarações especiais ignoradas pelo interpretador do `shell`. O início de um comentário deve conter o símbolo `#` e continuar até o

final da linha.

Por exemplo:

```
#!/bin/bash
# Esse script irá imprimir seu nome de usuário.
whoami
```

> ***Dica**: Use comentários para explicar o que seu script faz e porque.*

Variáveis

Como na maioria das linguages de programação, você pode criar variáveis no bash.

Bash não conhece nenhum tipo de dados. Variáveis podem conter apenas números ou *strings*. Existem três tipos de variáveis que você pode criar: variáveis locais, variáveis de ambiente e variáveis de *parâmetros posicionados*.

Variáveis locais

Variáveis locais são variáveis que existem apenas no contido script. Elas são inacessíveis para outros programas ou scripts.

Uma variável local pode ser declarada usando o sinal `=` (como regra, **não deve** conter nenhum espaço entre o nome da variável, `=` e o seu valor) e seu valor pode ser acessado usando o sinal `$`. Por exemplo:

```
username="pauloluan"    # declarando a variável
echo $username          # imprimindo seu valor
unset username          # deletando a variável
```

Nós podemos declarar uma variável local para uma única função usando a declaração `local`. Com isso, a variável será automaticamente deletada quando a função terminar de ser executada.

```
local local_var="Sou uma variável local"
```

Variáveis de ambiente

Variáveis de ambiente são variáveis que podem ser acessadas por qualquer programa ou script sendo executado na sessão atual do shell. Elas são criadas como variáveis locais, mas usando a declaração `export` no início delas.

```
export GLOBAL_VAR="Sou uma variável global"
```

Existem *muitas* variáveis globais no bash. Você vai conhecer elas no decorrer do seu dia-a-dia, mas aqui você encontra uma tabela com as mais utilizadas:

Variáveis	Descrição
`$HOME`	O diretório inicial do usuário atual.
`$PATH`	Uma lista separada por dois pontos `[:]` dos diretários que o shell irá procurar por comandos.
`$PWD`	O diretório atual.
`$RANDOM`	Número inteiro randômico entre 0 e 32767.
`$UID`	Versão numérica do ID do usuário atual.
`$PS1`	Sequência primária do seu prompt de comando.
`$PS2`	Sequência secundária do seu prompt de comando.

Entre nesse link para ver uma lista extendida de variáveis de ambiente do Bash.

Parâmetros de posição

Parâmetros de posição são variáveis alocadas aos parâmetros de uma função quando ela é executada. A seguinte tabela mostra os parâmetros de posição e outras variáveis especiais e quais os seus significados dentro da função.

Parâmetro	Descrição
`$0`	Nome do script.
`$1 … $9`	Os parâmetros passados de 1 há 9.
`${10} … ${N}`	Os parâmetros passados de 10 há N.
`$*` or `$@`	Todos os parâmetros passados, exceto `$0`.
`$#`	A soma da quantidade de parâmetros foi passada, não contando `$0`.
`$FUNCNAME`	O nome da função (retornada como valor, apenas dentro da função).

No exemplo abaixo, os parâmetros posicinais serão `$0='./script.sh'`, `$1='foo'` e `$2='bar'`:

```
./script.sh foo bar
```

Variáveis também podem ter um valor *padrão*. Nós podemos definir isso usando a sintaxe:

```
 # se a variável estiver vazia, atribua o valor padrão
: ${VAR:='default'}
: ${$1:='first'}
# ou
FOO=${FOO:-'default'}
```

Expansões do shell

Expansões são realizadas na linha de comando após ela ser separada em *símbolos*. Em outras palavras, expansões são mecânismos para calcular operações aritméticas, salvar resultados de execuções de comandos e assim por diante.

Se você estiver interessado, você pode ler mais sobre expansões do shell.

Expansões de suporte

Expansões de suporte nos permite criar *strings* arbitrárias. É parecido com *expansão de nomes de arquivos*. Por exemplo:

```
echo bat{i,a,u}ta # batita batata batuta
```

Expansões também podem ser usadas para criar extensões numéricas, que podem ser iterados em um *loop*.

```
echo {0..5} # 0 1 2 3 4 5
echo {00..8..2} # 00 02 04 06 08
```

Substituição de comandos

Substituição de comandos nos permite avaliar um comando e substituir seus valores em outro comando ou atribuição de variável. Substituição de comandos é realizado quando um comando é anexado por `` ou `$()`. Por exemplo, podemos usar isso da seguinte maneira:

```
now=`date +%T` # horário atual
# ou
now=$(date +%T) # horário atual

echo $now # 19:08:26
```

Expansões aritiméticas

No bash, somos livres para fazer qualquer operação aritmética. Mas, expressões devem ser anexadas por `$(( ))`. O formato da operação aritmética é:

```
result=$(( ((10 + 5*3) - 7) / 2 ))
echo $result # 9
```

Dentro de expressões aritméticas, variáveis geralmente deverm ser usadas sem o prefixo `$`:

```
x=4
y=7
echo $(( x + y ))      # 11
```

```
echo $(( ++x + y++ )) # 12
echo $(( x + y ))     # 13
```

Aspas simples e duplas

Existe uma importante diferença entre aspas simples e duplas. Dentro das aspas duplas, variáveis ou comandos podem ser expandidos. Dentro de aspas simples não. Por exemplo:

```
echo "Seu diretório inicial: $HOME" # Seu diretório inicial: /Users/<username
echo 'Seu diretório inicial: $HOME' # Seu diretório inicial: $HOME
```

Tome cuidado ao expandir variáveis locais ou de ambiente dentro de aspas se eles contiverem espaços em branco. Um exemplo disso, considere o uso do `echo` para imprimir algo:

```
INPUT="Uma frase  com   estranhos    espaços em branco."
echo $INPUT   # Uma frase com estranhos espaços em branco.
echo "$INPUT" # Uma frase  com   estranhos    espaços em branco.
```

O primeiro `echo` será invocado com 7 argumentos separados - $INPUT é separado em cada palavra, `echo` imprimi um único espaço em branco entre cada palavra. No segundo caso, `echo` é invocado com um único argumento (todo o valor do $INPUT, includingo seus espaços em branco).

Agora, considere um exemplo mais sério:

```
FILE="Minhas coisas favoritas.txt"
cat $FILE   # tentará imprimir 3 arquivos: `Minhas`, `coisas` e `favoritas.tx
cat "$FILE" # imprimi 1 arquivo: `Minhas coisas favoritas.txt`
```

Enquanto o problema desse exemplo pode ser resolvido apenas renomeando FILE para `Minhas-coisas-favoritas.txt`, considere a entrada do nome vindo de uma variável de ambiente, um parâmetro posicional ou o resultado de outro comando (`find`, `cat`, etc). Se a entrada *puder* conter espaços em branco, tome o cuidado de envolver a expansão em aspas.

Arrays

Como em qualquer outra linguagem de programação, um array no bash é uma variável que permite o armazenamento de múltiplos valores. No bash, arrays também são de base zero, ou seja, o primeiro elemento do array tem o íncide 0.

Ao lidar com arrays, nós devemos tomar um cuidado especial com as variáveis de ambiente `IFS`. **IFS**, que significa **Input Field Separator**, em português, algo como, **Separador dos campos de entrada**, são os carácteres que separam os elementos dentro de um array. O valor padrão desses campos é um espaço em braco, `IFS=' '`.

Declarando array

Para criar um array no bash, você pode simplesmente atribuir o valor ao index da variável do array:

```
frutas[0]=Maça
frutas[1]=Pera
frutas[2]=Banana
```

As variáveis de arrays também podem ser criadas a partir de uma atribuição composta, como:

```
frutas=(Maça Pera Banana)
```

Expansões de Array

Elementos individuais do array, são igualmente expansíveis como qualquer outra variável:

```
echo ${frutas[1]} # Pera
```

Todo o array pode ser expansível usando `*` ou `@` no lugar do índice numérico:

```
echo ${frutas[*]} # Maça Pera Banana
echo ${frutas[@]} # Maça Pera Banana
```

Tem uma importante (e súbita) diferença entre as duas linhas acima: considere que um elemento do array tenha espaços em branco:

```
fruta[0]=Maça
fruta[1]="Mamão papaia"
fruta[2]=Banana
```

Nós queremos imprimir cada elemento do array separadamente em uma nova linha, então, vamos tentar usar a função nativa `printf`:

```
printf "+ %s\n" ${frutas[*]}
# + Maça
# + Mamão
# + papaia
# + Banana
```

Porque o `Mamão` e `papaia` foram imprimidos em linhas separadas? Vamos tentar usando aspas:

```
printf "+ %s\n" "${frutas[*]}"
# + Maça Mamão papaia Banana
```

Agora, está tudo em uma linha só - isso não exatamente o que queremos! É aí que `${frutas[@]}` entra no jogo:

```
printf "+ %s\n" "${frutas[@]}"
# + Maça
# + Mamão papaia
# + Banana
```

Dentro das aspas duplas, `${frutas[@]}` é expandido separadamente para cada elemento do array, com seus espaços em branco preservados.

Separando Array

Além disso, você pode extrair um pedaço do array usando os operadores:

```
echo ${frutas[@]:0:2} # Maça Mamão papaia
```

No exemplo acima, `${frutas[@]}` é expandido com todo o conteúdo do seu array, e `:0:2`, extraí o pedaço de tamanho 2, começando no índice 0.

Adicionando elementos no Array

Adicionar elementos no array é bem simples. Atribuições compostas são extremamente úteis nesse caso. Você pode fazer uso dessa maneira:

```
frutas=(Laranja "${frutas[@]}" Melão Ameixa)
echo ${frutas[@]} # Laranja Maça Mamão papaia Banana Melão Ameixa
```

No exemplo acima, `${frutas[@]}` é expandido com todo o conteúdo do seu array e é atribuido ao novo valor dentro do array `frutas`, sendo assim, mutando seu valor original.

Deletando elementos de um Array

Para deletar um elemento de um array, use o comando `unset`:

```
unset frutas[0] # Deleta o item Laranja
echo ${frutas[@]} # Maça Mamão papaia Banana Melão Ameixa
```

Streams, pipes e listas

Bash tem uma poderosa ferramente para trabalhar com outros programas e seus resultados. Usando *streams* nós podemos enviar o resultado de um programa para outro programa ou arquivo, e assim, gravar logs ou fazer qualquer coisa que quisermos.

Pipes te dá a oportunidade de transportar e controlar a execução de comandos.

É fundamental o entendimento de como usar essa poderosa e sofisticada ferramenta do Bash.

Streams

Ao executar qualquer comando no Bash, ele recebe esses dados como parâmetros e envia uma sequência ou *streams* de caracteres. Esses *streams* podem ser redirecionados em arquivos ou em outro *stream*.

Existem três tipos de saídas de dados, conhecidos como *descritores*:

Código	Descritor	Descrição
`0`	`stdin`	O padrão de entrada de dados.
`1`	`stdout`	O padrão de saída de dados.
`2`	`stderr`	O padrão de saída de erros.

Redirecionamento torna possível o controle de onde a saída do comando vai parar, e, de onde a entrada de dados veem. Para redirecionar *streams*, você pode usar esses operadores:

Operadores	Descrição
`>`	Redireciona a saída de dados
`&>`	Redireciona a saída de dados e de erros
`&>>`	Anexa o redirecionamento de saída e erros
`<`	Redireciona a entrada de dados
`<<`	Sintaxe do comando "Here documents"
`<<<`	Sintaxe do comando "Here strings"

Veja aqui alguns exemplos de redirecionamento:

```
# a saída do comando `ls` será escrita no arquivo lista.txt
ls -l > lista.txt

# adiciona a saída do comando no final do arquivo lista.txt
ls -a >> lista.txt

# todos os erros serão escritos no arquivo erros.txt
grep da * 2> erros.txt

# lê o arquivo erros.txt
less < errors.txt
```

Pipes

Nós podemos redirecionar os *streams* padrões não apenas para arquivos, mas também, para outros programas. **Pipes** nos permite usar a saída de um programa, como entrada de outro.

No exemplo abaixo, `comando1` envia sua saída para `comando2`, que então passa sua saída como entrada para `comando3`:

```
comando1 | comando2 | comando3
```

Construções como essa, são chamadas de **pipelines**.

Na prática, isso pode ser usado para processar dados através de vários programas. Por exemplo, no exemplo a seguir, a saída do `ls -l` é enviada para o comando `grep`, que então imprimi apenas os arquivos que tenham a extensão `.md`, e sua saída, é finalmente enviada para o comando `less`:

```
ls -l | grep .md$ | less
```

Lista de comandos

Uma **lista de comandos** é uma sequência de um ou mais *pipelines* separados pelos operadores `;`, `&`, `&&` ou `||`.

Se um comando termina com um operador `&`, o shell executará o comando asíncronamente através de um *subshell*. Em outras palavras, esse comando será executado em segundo plano (ou *background*).

Comandos separados por `;` serão executados em sequência: um após o outro. O shell esperada a finalização de cada comando para executar o próximo.

```
# comando1 será executado após a finalização do comando1
comando1 ; comando2

# que é o mesmo que
command1
command2
```

Uma lista separada por `&&` e `||` são conhecidos também como listas *AND* e *OR*,

Uma lista *AND* é parecida com isso:

```
# comando2 será executado se, e apenas se, o comando1 finalize seu processo c
comando1 && comando2
```

Uma lista *OR* é parecida com isso:

```
# comando2 será executado se, e apenas se, o comando1 não finalize seu proces
comando1 || comando2
```

O código retornado pelas listas *AND* ou *OR*, são o estado do último comando executado.

Operadores condicionais

Como em qualquer outra linguagem, as condicionais no Bash nos permitem decidir qual ação realizar. O resultado é determinado pela análise da expressão, que deverá ser conter `[[ ]]` em volta dela.

Expressões condicionais podem conter os operadores `&&` e `||`, como vimos, *AND* e *OR*. Além disso, existem várias outras expressões que podem ser utilizadas.

Existem duas condicionais diferentes: a condicional `if`, e a condicional `case`.

Expressões primárias e combinação de expressões

Expressões dentro do `[[ ]]` (ou `[ ]` para `sh`), são chamados de **comandos de teste** ou **primários**. Essas expressões ajudam a indicar o resultado de uma operação condicional. Nas tabelas abaixos, estamos usando `[ ]`, porque ele também funciona para `sh`. Para saber mais, veja aqui a diferença entre aspas simples e aspas duplas dentro dos colchetes no Bash..

Trabalhando com o sistema de arquivos:

Primários	Quer dizer

Primários	Quer dizer
`[ -e FILE ]`	*true* se `FILE` existir, do inglês ***e**xists*.
`[ -f FILE ]`	*true* se `FILE` existir e for um arquivo normal, do inglês ***f**ile*.
`[ -d FILE ]`	*true* se `FILE` existir e for executável, do inglês ***d**irectory*.
`[ -s FILE ]`	*true* se `FILE` existir e não for vazio, seu tamanho é maior que 0, do inglês ***s**ize*.
`[ -r FILE ]`	*true* se `FILE` existir e for possível a leitura, do inglês ***r**eadable*.
`[ -w FILE ]`	*true* se `FILE` existir e for possível a escrita, do inglês ***w**ritable*.
`[ -x FILE ]`	*true* se `FILE` existir e for possíve executá-lo, do inglês *e**x**ecutable*.
`[ -L FILE ]`	*true* se `FILE` existir e for um link simbólico, do inglês *symbolic **l**ink*.
`[ FILE1 -nt FILE2 ]`	FILE1 é mais novo que FILE2, do inglês ***n**ewer **t**han*.
`[ FILE1 -ot FILE2 ]`	FILE1 é mais velho que FILE2, do inglês ***o**lder **t**han*.

Trabalhando com *strings*:

Primários	Quer dizer
`[ -z STR ]`	`STR` é vazio, seu tamanho é zero, do inglês ***z**ero*.
`[ -n STR ]`	`STR` não é vazio, seu tamanho não é zero, do inglês ***n**on-zero*.
`[ STR1 == STR2 ]`	`STR1` e `STR2` são iguais.
`[ STR1 != STR2 ]`	`STR1` e `STR2` não são iguais.

Operadores aritiméticos binários:

Primários	Quer dizer
`[ ARG1 -eq ARG2 ]`	`ARG1` é igual ao `ARG2`, do inglês ***eq****ual*.
`[ ARG1 -ne ARG2 ]`	`ARG1` não é igual ao `ARG2`, do inglês ***n****ot* ***e****qual*.
`[ ARG1 -lt ARG2 ]`	`ARG1` é menor que `ARG2`, do inglês ***l****ess* ***t****han*.
`[ ARG1 -le ARG2 ]`	`ARG1` é menor ou igual que `ARG2`, do inglês ***l****ess than or* ***e****qual*.
`[ ARG1 -gt ARG2 ]`	`ARG1` é maior que `ARG2`, do inglês ***g****reater* ***t****han*.
`[ ARG1 -ge ARG2 ]`	`ARG1` é maior ou igual que `ARG2` ***g****reater than or* ***e****qual*.

Condicionais podem ser combinadas usando as **expressões de combinação:**

Expressão	Efeito
`[ ! EXPR ]`	*true* se `EXPR` é falso.
`[ (EXPR) ]`	Retorna o valor da `EXPR`.
`[ EXPR1 -a EXPR2 ]`	Operador lógico *AND*. *true* se `EXPR1` e `EXPR2` são verdadeiros, do inglês ***a****nd*.
`[ EXPR1 -o EXPR2 ]`	Operador lógico *OR*. *true* se `EXPR1` ou `EXPR2` são verdadeiros, do inglês ***o****r*.

Com certeza existem muitos outros comandos e expressões úteis para seu caso, você fácilmente encontra-los na página de manual do Bash.

Usando a condicional `if`

Declarações `if` funcionam da mesma maneira como em outras linguagens de programação. Se a expressão dentro dos colchetes for verdadeira, o codógio dentro do bloco `then` e até o `fi` será executado. `fi` indica o final de uma condicional a ser executada.

```
# única linha
if [[ 1 -eq 1 ]]; then echo "true"; fi

# múltipla linha
if [[ 1 -eq 1 ]]; then
  echo "true"
fi
```

Da mesma forma, podemos usar uma declaração `if..else`, como:

```
# única linha
if [[ 2 -ne 1 ]]; then echo "true"; else echo "false"; fi

# múltipla linha
if [[ 2 -ne 1 ]]; then
  echo "true"
else
  echo "false"
fi
```

As vezes, condicionais `if..else` não são suficientes para o que queremos fazer. Nesse caso, não devemos esquecer da existência da condicional `if..elif..else`, que sempre vêm a calhar.

Veja o exemplo abaixo:

```
if [[ `uname` == "Adão" ]]; then
  echo "Não coma a maça!"
elif [[ `uname` == "Eva" ]]; then
  echo "Não pegue a maça!"
else
  echo "Maças são deliciosas!"
fi
```

Usando a condicional `case`

Se você estiver analisando várias possibilidades diferentes para ter ações diferentes, usar a condicional `case` pode ser mais útil do que várias condicionais `if` aninhadas. Veja abaixo um exemplo complexo de usando a condicional `case`:

```
case "$ext" in
  "jpg"|"jpeg")
    echo "É uma imagem com extensão jpg"
  ;;
  "png")
    echo "É uma imagem com extensão png"
  ;;
  "gif")
    echo "É uma imagem com extensão gif"
  ;;
  *)
    echo "Oops! Não é uma imagem!"
  ;;
esac
```

A condicional `case` verifica a expressão que corresponde a um padrão. O sinal `|` é usado para separar múltiplos padrões e o operador `)` finaliza a lista de padrões. A expressão `*` é o padrão para todo o restante que não corresponder a nenhum item das suas listas. Cada bloco de comandos deve ser separado pelo operador `;;`.

Loops

Aqui não teremos nenhuma surpresa. Assim como qualquer linguagem de programação, um loop no bash é um bloco de código que se repete enquanto a condição em controle for verdadeira.

Existem quatro tipos de loops no Bash: `for`, `while`, `until` e `select`.

`for` loop

O `for` é bem similar ao seu irmão em C. Ele se parece com:

```
for arg in elem1 elem2 ... elemN
do
  # código
done
```

Durante cada etapa do loop, `arg` assume os valores de `elem1` até `elemN`. Valores também podem ser espaços reservados ou expansões de suporte.

E também podemos escrever o loop `for` em apenas uma linha, mas nesse caso, é preciso colocar um ponto e vírgula antes do `do`, como no exemplo:

```
for i in {1..5}; do echo $i; done
```

A propósito, se `for..in..do` parece um pouco estranho para você, você também pode escrever o `for` em estilo C, como a seguir:

```
for (( i = 0; i < 10; i++ )); do
  echo $i
done
```

`for` é útil quando nós queremos fazer a mesma operação em cada arquivo em um diretório. Por exemplo, se precisamos mover todos os arquivos `.bash` dentro da pasta `script` e dar aos arquivos permissões de execução, nosso script será parecido com isso:

```
#!/bin/bash

for FILE in $HOME/*.bash; do
  mv "$FILE" "${HOME}/scripts"
  chmod +x "${HOME}/scripts/${FILE}"
done
```

`while` loop

O loop `while` testa uma condição e executa a sequência de comandos desde que a condição seja verdadeira. A condição não é nada mais que uma expressão primária usada também em `if..then`. Então, um loop `while` se parece com:

```
while [[ condition ]]
do
  # código
done
```

Tal como no caso do loop `for`, se quisermos escrever uma condição `do` na mesma linha, temos que usar um ponto e vírgula antes.

Um exemplo prático seria:

```
#!/bin/bash

  # Retorna o quadrado dos números de 0 à 9
x=0
while [[ $x -lt 10 ]]; do # valor de x é menor que 10
  echo $(( x * x ))
  x=$(( x + 1 )) # aumenta o x
done
```

`until` loop

O loop `until` é exatamente o oposto do loop `while`. Assim como o `while`, ele recebe uma condição teste, mas, só continua executando enquanto a condição for falsa:

```
until [[ cond ]]; do
  # código
done
```

`select` loop

O loop `select` nos ajuda a organizar um menu para o usuário. Ele tem quase a mesma sintaxe que o loop `for`:

```
select respostas in elem1 elem2 ... elemN
do
  # código
done
```

O `select` imprimi todos os `elem1..elemN` na tela, junto de suas sequências numéricas, e depois disso, pergunta ao usuário. Normalmente, isso se parece com `$?` (a variável `PS3`). A resposta será salva em `respotas`. Se `respostas` for um número entre `1..N`, então o código será executado e `select` vai para a próxima iteração - isso porquê nós devemos usar a declaração `break`.

Um exemplo prático se parece com esse:

```
#!/bin/bash

PS3="Escolha uma gerenciador de pacotes: "
select ITEM in bower npm gem pip
do
  echo -n "Digite o nome de um de pacote: " && read PACKAGE
  case $ITEM in
    bower) bower install $PACKAGE ;;
    npm)   npm   install $PACKAGE ;;
    gem)   gem   install $PACKAGE ;;
    pip)   pip   install $PACKAGE ;;
  esac
  break # evita loops infinitos
done
```

Esse example pergunta ao usuário qual gerenciador de pacote ele deseja usar. E em seguida, quais pacotes gostaríamos de instalar e finalmente, executa o processo de instalação.

Se rodarmos isso, teremos:

```
$ ./my_script
1) bower
2) npm
3) gem
4) pip
Escolha uma gerenciador de pacotes: 2
Digite o nome de um de pacote: bash-handbook
<installing bash-handbook>
```

Controlando o loop

Existem situações onde precisamos parar o loop antes da sua finalização normal ou pular uma iteração. Nesses casos, nós podemos usar as declarações `break` e `continue`, que são nativas do shell. Ambos funcionam com qualquer tipo de loop. There are situations when we need to stop a loop before its normal ending or step over an iteration. In these cases, we can use the shell built-in `break` and `continue` statements. Both of these work with every kind of loop.

A declaração `break` é usada para sair do loop atual antes da sua finalização. Nós já o conhecemos. The `break` statement is used to exit the current loop before its ending. We have already met with it.

A delcaração `continue` pula uma iteração. Podemos usa-la desse modo: The `continue` statement steps over one iteration. We can use it as such:

```
for (( i = 0; i < 10; i++ )); do
  if [[ $(( i % 2 )) -eq 0 ]]; then continue; fi
  echo $i
done
```

Se você rodar o exemplo acima, ele vai imprimir os números ímpares de 0 à 9. If we run the example above, it will print all odd numbers from 0 through 9.

Funções

Em scripts, nós temos a habilidade de definir e chamar funções. Assim como em qualquer linguagem de programação, funções no bash são pedaços de códigos, mas elas são tratadas um pouquinho diferentes.

No bash, funções são sequências de comandos agrupados sob um mesmo nome, e esse nome, é o nome da função. Chamar uma função é o mesmo que chamar qualquer outro programa, você escreve o nome da função e ela será invocada.

Podemos declarar funções dessa maneira:

```
my_func () {
  # código
}

my_func # invoca função
```

Devemos declarar a função antes de invoca-la.

Funções podem receber argumentos e retornar um resultado - o código de saída. Argumentos, em funções, são tratados da mesma maneira que os argumentos dados ao script no modo não-interativo - usando os parâmetros de posição. O resultado pode ser retornado usando o comando `return`.

Abaixo é uma função que recebe um nome e retorna `0`, indicando que foi executado com sucesso.

```
# function with params
bemVindo () {
  if [[ -n $1 ]]; then
    echo "Bem-vindo, $1!"
  else
    echo "Bem-vindo, desconhecido!"
  fi
  return 0
}

bemVindo Eduardo  # Hello, Eduardo!
bemVindo          # Hello, desconhecido!
```

Nós já falamos sobre códigos de saída. O comando `return` sem argumentos retorna o código de saída do último comando executado. Acima, `return 0` vai retornar o código bem sucedido, `0`.

Depurando

O shell nós dá ferramentas para depurar nossos scripts. Se você quer rodar um script em modo de depuração, nós usamos um modo especial em nosso *shebang*:

```
#!/bin/bash options
```

Esse `options` é a configuração que muda o comportamento do shell. A tabela abaixo mostra uma lista de opções que podem ser úteis para você:

Atalho	Nome	Descrição
`-f`	noglob	Desativa expanção de nome de arquivos, em inglês, *globbing*.
`-i`	interactive	Script roda no modo *interativo*.
`-n`	noexec	Lê comandos, mas não os executa (verifica a sintaxe).
`-t`	—	Saí da execução depois do primeiro comando.
`-v`	verbose	Imprimi cada comando no `stderr` antes de executa-los.

Atalho	Nome	Descrição
`-x`	xtrace	Imprimi cada comando e expande seus argumentos e envia para o `stderr` antes de executa-los.

Por exemplo, podemos ter scripts com `-x` como opção, assim como:

```
#!/bin/bash -x

for (( i = 0; i < 3; i++ )); do
  echo $i
done
```

Isso vai imprimir o valor das variáveis para o `stdout` junto de outras informações úteis:

```
$ ./my_script
+ (( i = 0 ))
+ (( i < 3 ))
+ echo 0
0
+ (( i++  ))
+ (( i < 3 ))
+ echo 1
1
+ (( i++  ))
+ (( i < 3 ))
+ echo 2
2
+ (( i++  ))
+ (( i < 3 ))
```

As vezes nós precisamos depurar uma parte do script. Nesse caso, usar o comando `set` é mais conveniente. Esse comando habilita e desabilita opções. Opções são desabilitadas usando `-` e habilitadas usando `+`:

```
#!/bin/bash

echo "xtrace está desabilitado"
set -x
echo "xtrace está habilitado"
```

```
set +x
echo "xtrace foi desabilitado novamente"
```

Posfácio

Eu espero que esse pequeno guia tenha sido interação e tenha te ajudado a entender um pouco mais sobre o Bash. Para ser honesto, eu escrevi esse guia para mim mesmo, para assim, não esquecer o básico do bash. Eu tentei escrever de uma maneira concisa, mas significativamente útil e eu espero que você tenha gostado.

Esse guia narra minha própria experiência com o Bash. Ele não tem foco de abranger toda as funcionalidades, e, se você quiser saber mais, pode começar através do `man bash`.

Contribuições são absolutamente bem-vindas, e eu ficarei grato por qualquer correção ou perguntas que você vier a ter e me enviar. Para isso, crie uma nova issue no repositorio original.

Obrigado por ler esse guia de bolso!

Licença