

# THE ONE PAGE FOR... BASH!

Autores: Rudson Ribeiro Alves e Maycon Maia Vitali

Última atualização: 16/09/2008

## Bash (GNU Bourne-Again Shell)

Bash é um interpretador de comandos compatível com o conhecido *sh*, atualmente mais utilizados nos *GNU/Linux*'s. Este texto se concentra apenas nos comandos internos do *Bash* (*builtin commands*).

## Algumas Opções

Estas opções podem ser aplicadas ao inicializar o interpretador, ou através do comando interno *set*.

- i põem o *shell* em modo interativo;
- x aciona o modo *debug*;
- l modo *login*. O *bash* carrega os *scripts* de inicialização (*.bashrc*, *.bash\_login*, ...)
- norc  
não carrega os *script* de inicialização geral (*/etc/profile*) ou qualquer outro *script* local (*.bashrc*, *.bash\_login*, ...)

## Palavras Reservadas

O *bash* possui um pequeno conjunto de palavras reservadas:

```
! case do done elif else fi for
function if in select until while { }
time [[ ]]
```

## Comandos e Status de saída

Todo comando em *bash* envia um código de saída que pode ser acessado consultando a variável "\$?". Este código de saída é 0 (zero) quando o comando é bem sucedido e outro valor quando falha. Por exemplo:

```
$ A=5; B=0; echo $?
0
$ C=$(( A/B ))
bash: A/B : division by 0 (error tok...)
$ echo $?
1
```

No exemplo acima, o dólar (\$) representa o *prompt* padrão do *bash*. Na primeira linha, é associado às variáveis A e B os valores 5 e 0, e em seguida imprimido o status da operação, 0 de bem sucedida. Quanto se faz uma divisão por zero, o *bash* retorna o erro correspondente e o *status* passa para 1.

## Pipelines

*Pipelines* é uma seqüência de um ou mais comandos separados por uma caracter |. O formato para um pipeline é:

```
command | command_1 [ | command_2 ... ]
```

O pipe redireciona a saída padrão do comando\_1 para a entrada padrão do comando\_2, e assim sucessivamente. Se algum comando falhar, retorna o *status* do último comando a falhar, interrompendo a execução da *pipeline*.

## Listas de comandos

Listas são seqüências de um ou mais pipelines separados por operadores ;, &, && ou || e, opcionalmente, terminadas por

um ;, & ou *<newline>*.

```
command_1; command_2 & command_3 && ...
```

&& e || possuem a mesma precedência, seguidos por ; e &.

Cada delimitador realiza uma ação específica:

; (ponto e vírgula) – este delimitador separa comandos sem promover qualquer interação entre eles, executando-os em *foreground*. É o mesmo que o *<newline>*.

& (e comercial) – este delimitador faz o mesmo que o ponto e vírgula, acima, porem envia o comando para *background*, liberando o console para a execução do próximo comando.

&& – executa o comando seguinte, somente se o comando anterior for bem sucedido (\$? retornar 0).

|| – executa o comando seguinte, somente se o comando anterior for mal sucedido (\$? retornar diferente de 0).

## Comandos em Blocos

Existem quatro formas de criar blocos de comandos em *bash*:

( *list* ) – a lista de comandos *list* é executada em uma *sub-shell*. Isto significa que qualquer mudança em variáveis, diretório de trabalho, ... não interferem no restante do *shell*. Estes blocos são muito úteis quando se quer isolar a execução de um bloco de comandos.

{ *list* ; } – a lista de comandos *list* é executada na *shell*-corrente, ou seja, todas as mudanças em variáveis, diretório de trabalho, ... irão interferir no restante do *shell*. Blocos entre chaves devem terminar com um *<newline>* ou um ponto e vírgula. Blocos em chaves geralmente são usadas em declarações de funções;

(( *expressão* )) – faz uma avaliação aritmética de uma expressão;

[[ *expressão* ]] – faz uma avaliação lógica de uma expressão.

## [[ Expressões ]]

Expressões podem ser combinadas com os operadores listados abaixo, em ordem de precedência:

( *expressão* ) – retorna o valor da expressão.

! *expressão* – retorna *True* se a expressão for falso.

*expressão1* && *expressão2* – retorna *True* se as duas expressões forem verdadeiras.

*expressão1* || *expressão2* – retorna *True* se uma das expressões forem verdadeiras.

## Controle de Fluxo

### Comando for:

```
for name in words ; do [ list ] ; done
```

o *for* faz com que a variável *name* varie sobre todos os valores em *words*, executando a lista de comandos *list*, a cada novo valor.

```
$ for i in 1 2 3 4 5; do echo $i; done
$ for i in $( seq 5 ); do echo $i; done
```

As duas linhas acima imprimem de 1 a 5 em linhas diferentes.

```
for (( exp1; exp2; exp3 )); do list; done
```

Esta é uma forma C like de se fazer loops:

```
$ for (( i = 1; i < 6; i++ )); do \
```

```
echo $i; done
```

Esta linha também imprime os números de 1 a 5.

### Comando while/until:

```
while list1; do list2; done
```

Executa a *list2*, enquanto *list1* for verdadeira.

```
while [ "$a" != 'f' ]; do read a; done
```

O *until* trabalha de forma parecida:

```
until list1; do list2; done
```

executa a *list2*, enquanto *list1* for falsa.

```
until [ "$a" != 'f' ]; do read a; done
```

Nos dois casos, as aspas duplas, são necessárias, para o caso de \$a ser uma *string* vazia.

### Comando if:

```
if list1; then list2; [ elif list3; then list4; ] ... [ else listN; ] fi
```

Se a *list1* for verdadeira, executa a *list2*, caso contrário, verifica de a *list3* é verdadeira e executa a *list4*, ..., caso contrário, executa a *listN*.

```
if [ $A -lt 10 ]; then
```

```
    let A++
```

```
elif [ $A -lt 20 ]; then
```

```
    let A+=2
```

```
else
```

```
    let A=0
```

```
fi
```

### Comando case:

```
case word in pattern [ | pattern ] ... ) list ;; ] ... esac
```

o comando *case* primeiro expande *word* e verifica se ela casa com um dos *pattern*. Se for o caso executa a lista de comandos *list* correspondente.

### Passando argumentos

Argumentos são acessados em um script através das variáveis \$N, onde N é um inteiro representando a ordem do argumento passado.

**\$0** – nome do *script* com o *path* completo;

**\$1** – primeiro parâmetro passado ao *script*;

**\$2** – segundo parâmetro passado ao *script*;

...

**\$\*** – todos os parâmetros passados em uma única string, separados por **IFS** (geralmente espaço);

**\$@** – todos os parâmetros passados em strings separadas, isoladas por aspas dupla;

**\$#** – número de parâmetros passados;

**shift** – incrementa o ponteiro para o próximo argumento;

### Expansão de variáveis

**\${var:-text}** – se *var* não está definida, retorna *text*;

**\${var:=text}** – se *var* não está definida, retorna *text* e a define como texto;

**\${var:?text}** – se *var* não está definida, retorna *text* e emite uma mensagem de erro.

**\${var:+text}** – se *var* está definida retorna *text*, se não retorna vazio;

**\${var}** – o mesmo que *\$var*. Retorna o valor de *var*;

**\${#var}** – retorna o tamanho da string *var*;

**\${var:N[:size]}** – retorna a partir do N-ésimo caracter. Se *size* estiver presente, retorna *size* caracteres;

**\${var#text}** – corta *text* do início da *string*;

**\${var%text}** – corta *text* do final da *string*;

**\${var/text1/text2}** – substitui a primeira ocorrência de *text1* em *var* por *text2*;

**\${var//text1/text2}** – substitui todas as ocorrências de *text1* em *var* por *text2*;

### Redirecionamento

Redirecionamento permite alterar a entrada e saída padrão do sistema para um arquivo ou outro comando.

```
command1 <redirc> command2/file
```

> – redireciona a saída padrão;

< – redireciona a entrada padrão;

2> – redireciona a saída de erro;

>> – redireciona a saída padrão, anexando;

2>> – redireciona a saída de erro, anexando;

| – conecta a saída padrão do comando *cmd1* ao *cmd2*

2>&1 – conecta a saída de erro a saída padrão;

>&2 – conecta a saída padrão à saída de erro;

>&- – fecha a saída padrão

>&2- – fecha a saída de erro

### Teste em variáveis e arquivos

#### Números

**-gt/-lt** – maior que / menor que

**-ge/-le** – maior ou igual a / menor ou igual a

**-eq/-ne** – igual a / diferente de

#### Strings

**= ou ==** – igual a

**!=** – diferente de

**-n** – a *string* não é nula

**-z** – a *string* é nula

**!** – não (nega a sentença)

**-a** – e lógico

**-o** – ou lógico

#### Arquivos

**-d** – é um diretório

**-e** – arquivo existe

**-f** – é um arquivo

**-x** – é um executável

### Funções

Em *bash* é possível definir funções. Sua sintaxe é:

```
[ function ] name () List [ redirection ]
```

A passagem de parâmetros segue as mesmas regras em *Passando Argumentos*.

```
function getip() {  
    /sbin/ifconfig eth0 | sed '/inet addr:/'  
d; s/.*addr:\(.*\) B.*\/1/'  
}
```