

BIF0001 - Fundamentos de Bioinform tica

Shell Script

— Prof. Dr. Tetsu Sakamoto —

Instituto Metr pole Digital - UFRN
Sala A224, ramal 182
Email: tetsu@imd.ufrn.br

Conteúdo da aula

1. Introdução

- a. O que é Shell?
- b. O que é Script?
- c. Por que aprender Shell script?

2. Bash Script

- a. Criar e executar;
- b. Variáveis;
- c. Inputs;
- d. Operadores aritméticos;
- e. If statements;
- f. Loops;
- g. Funções.

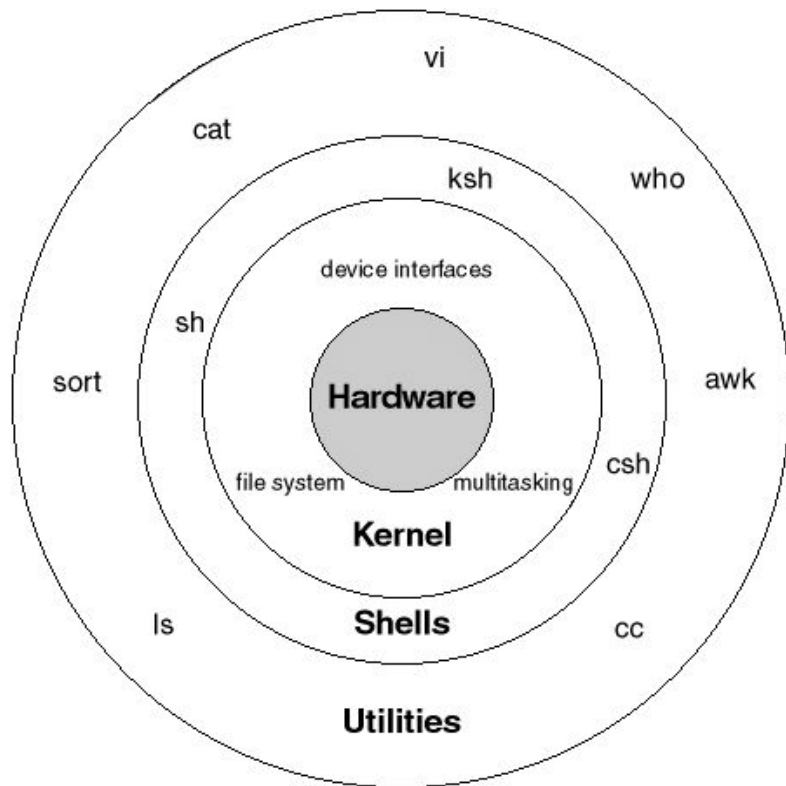
O que é Shell?

É um programa que permite a interação do usuário com o **Kernel**.

Foi desenvolvido por Stephen Bourne em 1979.

Existem vários tipos de Shell: C shell (csh), Z shell (zsh), Korn shell (ksh), Bourne shell (sh), Debian's Alchemist shell (dash).

O mais utilizado hoje é o **Bash** (Bourne Again shell).



O que é Script?

Como se fosse um roteiro de um filme:

O roteiro informa os atores as coisas que eles devem fazer e dizer.

O script para um computador informa ele o que ele deve fazer e dizer.



O que é Script?

Os scripts são escritos na forma de um **texto simples**.

Shell scripts contém uma série de comandos que podem ser executados em um terminal.

“Tudo que está em um shell script, pode ser executado em um terminal; e tudo que é executado em um terminal pode ser colocado em um shell script.”

Convenção: arquivos shell scripts com extensão **.sh**.

```
#!/bin/bash

function gpio()
{
    local verb=$1
    local pin=$2
    local value=$3

    local pins=($GPIO_PINS)
    if [[ "$pin" -lt ${#pins[@]} ]]; then
        local pin=${pins[$pin]}
    fi

    local gpio_path=/sys/class/gpio
    local pin_path=$gpio_path/gpio$pin
```

Por que aprender Shell Script?

- Combinar múltiplos comandos em um;
 - Você pode reunir vários comandos em um Shell Script e executar apenas o script.
- Automação de tarefas frequentes;
 - Tarefas frequentes podem ser reunidos em um Shell Script para ser executado com um comando de linha.
- Relativamente fácil de desenvolver;
 - Muitas ações feitas em linguagens de programação convencionais (como C/C++) podem ser realizadas por um Shell Script;
- Transparência;
 - Como o Shell Script é escrito em um texto simples, você consegue facilmente as suas ações;
- Portável;
 - Um Shell Script pode ser executado por outros sistemas baseados em UNIX.

Como criar e executar um Shell Script

1. Crie um arquivo chamado *hello.sh*

```
$ vi hello.sh
```

Como criar e executar um Shell Script

2. Dentro do arquivo *hello.sh*, inclua as seguintes linhas:

```
#!/bin/bash  
  
# My first bash script  
  
echo Hello World!
```

Para escrever texto no vi:

1. Entre no modo "INSERT" pressionando a tecla "i";
2. Comece a escrever;
3. Para sair do modo "INSERT", pressione a tecla "Esc".

Como criar e executar um Shell Script

3. Salve o arquivo *hello.sh*, volte ao terminal e dê a permissão de execução utilizando o seguinte comando:

```
$ chmod +x hello.sh
```

Para **salvar** e sair do vi:

1. Pressione "Esc" para voltar ao modo original;
2. Pressione a tecla ":" e depois digite "wq!";
3. Pressione "Enter"

Para **descartar** as alterações e sair do vi:

1. Pressione "Esc" para voltar ao modo original;
2. Pressione a tecla ":" e depois digite "q!";
3. Pressione "Enter"

Como criar e executar um Shell Script

4. Execute o seu primeiro shell script usando o seguinte comando:

```
$ ./hello.sh
```

```
Hello World!
```





Algumas observações sobre o seu primeiro script

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

```
# My first bash script
```

```
echo Hello World!
```

Algumas observações sobre o seu primeiro script

<code>#!/bin/bash</code>		Shebang → indica o caminho do interpretador desse script.
<code># My first bash script</code>		Comentário, não é executado pelo interpretador.
<code>echo Hello World!</code>		Comando a ser executado. Pode ser executado no próprio terminal.
<code>-----</code>		
<code>\$./hello.sh</code>		Execução do script.

Variáveis

Informação que é armazenada temporariamente na memória do computador.

Cada variável possui um nome declarado pelo próprio usuário. *Convenção: utilizar **caixa alta** nos nomes das variáveis;*

Duas principais ações realizadas em uma variável:

- Atribuir um valor a uma variável;
- Ler o valor de uma variável.

Variáveis - atribuindo um valor e lendo

```
$ ./var_ex1.sh
```

var_ex1.sh

```
#!/bin/bash  
  
# exemplo do uso de variável  
  
MYVAR=Hello  
  
NAME=Tetsu  
  
echo $MYVAR $NAME
```

Variáveis - atribuindo um valor e lendo

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

```
# exemplo do uso de variável
```

```
AT=~
```

```
ls $AT
```

Variáveis especiais

- **\$0** - Nome do script.
- **\$1 - \$9** - Os primeiros 9 argumentos para o script.
- **\$#** - Número de argumentos passados no script.
- **\$@** - Todos os argumentos passados para o script.
- **\$?** - O status de saída do processo mais recente executado.
- **\$\$** - ID do processo do script atual.
- **\$SECONDS** - Número de segundos passados desde a execução do script.
- **\$RANDOM** - Retorna um número randômico.
- **\$LINENO** - Retorna o número da linha atual no script.

Aspas

```
$ ./arg.sh arg.sh arg2.sh
```

aspas.sh

```
#!/bin/bash
myvar='Hello World'
echo $myvar                # Hello World

newvar="More $myvar"
echo $newvar                # More Hello World

newvar='More $myvar'
echo $newvar                # More $myvar
```

Exportando variáveis

As variáveis declaradas em um script são restritas ao processo que está executando o script.

Para tornar a variável disponível para outros processos, deve-se exportar a variável.

Exportando variáveis

script1.sh

```
#!/bin/bash
# demonstrate variable scope 1.
var1=blah
var2=foo

# Let's verify their current value
echo $0 :: var1 : $var1, var2 : $var2
export var1
./script2.sh

# Let's see what they are now
echo $0 :: var1 : $var1, var2 : $var2
```

script2.sh

```
#!/bin/bash
# demonstrate variable scope 2

# Let's verify their current value
echo $0 :: var1 : $var1, var2 : $var2

# Let's change their values
var1=flop
var2=bleh
```

Variáveis - recebendo argumentos

```
$ ./arg.sh arg.sh arg2.sh
```

arg_ex1.sh

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

```
# copiando os arquivos
```

```
cp $1 $2
```

Inputs

Uma outra forma do programa interagir com o usuário é através do input.

Se queremos que o programa peça ao usuário que entre com um input, utilizamos o comando **read**.

inputs_ex1.sh

```
#!/bin/bash

# Ask the user for their name

echo Hello, who am I talking to?

read varname

echo It\'s nice to meet you $varname
```

Inputs

Podemos alterar o comportamento do **read**, acrescentando algumas opções.

Veja no script ao lado algumas das opções mais comuns:

-p → Permite que ele mostre uma mensagem;

-s → Oculta o input na tela.

inputs_ex2.sh

```
#!/bin/bash

# Ask the user for login details

read -p 'Username: ' uservar

read -sp 'Password: ' passvar

echo

echo Thank you $uservar we now have
your login details
```

Inputs

Um único comando do read pode receber mais de uma variável de uma vez;

O input é cortado por espaço, e o primeiro elemento do input é atribuído a primeira variável, o segundo elemento do input, a segunda variável e assim por diante.

inputs_ex3.sh

```
#!/bin/bash

# Demonstrate how read actually works

echo What cars do you like?

read car1 car2 car3

echo Your first car was: $car1

echo Your second car was: $car2

echo Your third car was: $car3
```

Lendo do STDIN

É comum no sistema UNIX utilizar **pipe** para executar comandos em cadeia.

O Bash acomoda as entradas e as saídas em arquivos especiais:

- STDIN - /dev/stdin
- STDOUT - /dev/stdout
- STDERR - /dev/stderr

Podemos executar o script ao lado desta forma:

```
$ cat RANBP9.fasta.txt | ./stdin_ex1.sh
```

stdin_ex1.sh

```
#!/bin/bash

# A basic seq count report

echo Here is the number of seqs:

cat /dev/stdin | grep -c '>'
```


Substituição de comando

\$./subs_ex1.sh

subs_ex1.sh

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

```
MYVAR=$( ls /etc | wc -l )
```

```
echo There are $MYVAR entries in directory /etc
```

Operações aritméticas

Operador	Operação
+, -, *, /	adição, subtração, multiplicação, divisão
var++	Incrementa 1 para a variável var
var--	Decrementa 1 para a variável var
%	Módulo (Resto da divisão)

Operações aritméticas

Existem algumas formas de você utilizar realizar operações aritméticas em Bash Script. São elas:

- `let`
 - Permite salvar o resultado em uma variável.
 - Não pode haver espaços.
 - Se quiser espaço, colocar a expressão entre aspas.

let_ex.sh

```
#!/bin/bash
# Basic arithmetic using let
let a=5+4
echo $a # 9

let "a = 5 + 4"
echo $a # 9

let a++
echo $a # 10

let "a = 4 * 5"
echo $a # 20

let "a = $1 + 30"
echo $a # 30 + primeiro argumento
```

Operações aritméticas

Existem algumas formas de você utilizar realizar operações aritméticas em Bash Script. São elas:

- `let`
 - Permite salvar o resultado em uma variável.
- `expr`
 - Imprime o resultado na tela.
 - Necessita do espaço entre os números e o operador.

expr_ex.sh

```
#!/bin/bash
# Basic arithmetic using expr
expr 5 + 4

expr "5 + 4"

expr 5+4

expr 5 \* $1

expr 11 % 2

a=$( expr 10 - 3 )
echo $a # 7
```

Operações aritméticas

Existem algumas formas de você utilizar realizar operações aritméticas em Bash Script. São elas:

- `let`
 - Permite salvar o resultado em uma variável.
- `expr`
 - Imprime o resultado na tela.
- parênteses duplo
 - Forma mais flexível e recomendado

dblpar_ex.sh

```
#!/bin/bash
# Basic arithmetic using double parentheses
a=$(( 4 + 5 ))
echo $a # 9
a=$((3+5))
echo $a # 8
b=$(( a + 3 ))
echo $b # 11
b=$(( $a + 4 ))
echo $b # 12
(( b++ ))
echo $b # 13
(( b += 3 ))
echo $b # 16
a=$(( 4 * 5 ))
echo $a # 20
```

If statements

Permite que o seu script tome decisões baseado em uma condição. Assim, você faz com que uma parte do código seja executada se a condição for satisfeita.

Estrutura básica de um if statements:

```
if [ <condição> ]
```

```
then
```

```
    <commandos>
```

```
fi
```

If statements

O `[]` é uma referência ao comando **test**. Isto significa que todos os operadores utilizados em test pode ser utilizado aqui.

Ao lado um exemplo simples de um “if statement”.

Indentação: Não é obrigatória, mas é uma boa prática para facilitar a leitura do código.

if_ex.sh

```
#!/bin/bash
# Basic if statement

if [ $1 -gt 100 ]
then
    echo Hey that\'s a large number.
    pwd
fi

date
```

Alguns operadores

Operador	Descrição
! EXPRESSION	A EXPRESSION é falsa.
-n STRING	O comprimento da STRING é maior que zero.
-z STRING	O comprimento da STRING é zero (vazio).
STRING1 = STRING2	STRING1 é igual a STRING2
STRING1 != STRING2	STRING1 não é igual a STRING2
INTEGER1 -eq INTEGER2	INTEGER1 é numericamente igual a INTEGER2
INTEGER1 -gt INTEGER2	INTEGER1 é numericamente maior que INTEGER2
INTEGER1 -lt INTEGER2	INTEGER1 é numericamente menor que INTEGER2

Alguns operadores

Operator	Description
-d FILE	FILE existe e é um diretório
-e FILE	FILE existe.
-r FILE	FILE existe e possui permissão de read.
-w FILE	FILE existe e possui permissão de write.
-x FILE	FILE existe e possui permissão de execute.
-s FILE	FILE existe e seu tamanho é maior que zero.

Nested If statements

Você pode ter blocos de if statements dentro de um outro bloco.

Perceba como foi a **Indentação** no código ao lado.

nested_if_ex.sh

```
#!/bin/bash
# Nested if statements

if [ $1 -gt 100 ]
then
    echo Hey that\'s a large number.
    if (( $1 % 2 == 0 ))
    then
        echo And is also an even number.
    fi
fi
```

If else

Às vezes queremos executar um bloco de código caso uma condição seja satisfeita, e se não, executar um segundo bloco de código.

```
if [ <condição> ]  
then  
    <comando>  
else  
    <outro comando>  
fi
```

else_ex.sh

```
#!/bin/bash  
  
# else example  
if [ $# -eq 1 ]  
then  
    nl $1  
else  
    nl /dev/stdin  
fi
```

If elif else

Às vezes queremos executar uma série de condições onde cada um conduz para diferentes caminhos.

```
if [ <condição 1> ]
then
    <comandos 1>
elif [ <condição 2> ]
then
    <comandos 2>
else
    <comandos 3>
fi
```

elif_ex.sh

```
#!/bin/bash
# elif statements
if [ $1 -ge 18 ]
then
    echo You may go to the party.
elif [ $2 == 'yes' ]
then
    echo You may go to the party but be
back before midnight.
else
    echo You may not go to the party.
fi
```

Operadores booleanas

Podemos testar múltiplas condições em um mesmo if statements (regra de “e” e do “ou”) utilizando **operadores booleanas**.

- Regra do e → &&
- Regra do ou → ||

and_ex.sh

```
#!/bin/bash

# and example

if [ -r $1 ] && [ -s $1 ]

then

    echo This file is useful.

fi
```

Operadores booleanas

Podemos testar mais de uma condição em um mesmo if statements (regra de “e” e do “ou”) utilizando **operadores booleanas**.

- Regra do e → &&
- Regra do ou → ||

or_ex.sh

```
#!/bin/bash
# or example
if [ $# -gt 2 ] || [ $# -lt 2 ]
then
    echo ERROR: two args required.
else
    echo correct number of argument.
fi
```

Loops

Os loops nos permitem definir um bloco de comando que é executado repetidas vezes até que ele alcance uma situação em particular.

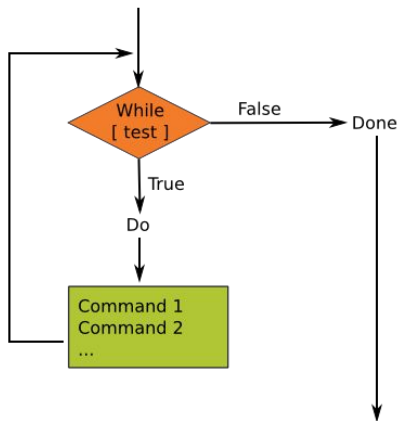
Existem três formas básicas de loops em Bash script:

- While
- Until
- For

While loop

“Enquanto a condição é verdadeira, execute o bloco de código”.

```
while [ <condição> ]  
do  
    <comandos>  
done
```



while_ex.sh

```
#!/bin/bash  
# Basic while loop  
counter=1  
while [ $counter -le 10 ]  
do  
    echo $counter  
    ((counter++))  
done  
echo All done
```


Until loop

“Execute o bloco de código até que a condição seja satisfeita”.

```
until [ <condição> ]  
do  
    <comandos>  
done
```

until_ex.sh

```
#!/bin/bash  
# Basic until loop  
counter=1  
until [ $counter -gt 10 ]  
do  
    echo $counter  
    ((counter++))  
done  
echo All done
```

For loop

“Para cada item dentro de uma lista, execute o bloco de código”.

```
for var in <list>
do
    <comandos>
done
```

for_ex1.sh

```
#!/bin/bash
# Basic for loop
names='Stan Kyle Cartman'
for name in $names
do
    echo $name
done
echo All done
```

For loop

“Para cada item dentro de uma lista, execute o bloco de código.

```
for var in <list>
do
    <comandos>
done
```

for_ex2.sh

```
#!/bin/bash

# Basic range in for loop

for value in {1..5}
do

    echo $value

done

echo All done
```

For loop

“Para cada item dentro de uma lista, execute o bloco de código.

```
for var in <list>
do
    <comandos>
done
```

for_ex3.sh

```
#!/bin/bash

# Basic range with steps for loop
for value in {10..0..2}
do

    echo $value

done

echo All done
```

For loop

“Para cada item dentro de uma lista, execute o bloco de código.

```
for var in <list>
do
    <comandos>
done
```

for_ex4.sh

```
#!/bin/bash

# Make a seq count on any fasta file

for value in $1/*.fasta
do

    COUNT=$( grep -c '>' $value )

    echo $value count: $COUNT

done
```

Controlando loops

- break
 - Força a saída no loop.

loop_break.sh

```
#!/bin/bash
# Make a backup set of files
for value in $1/*
do
    used=$( df $1 | tail -1 | awk '{ print $5 }' | sed 's/%//' )
    if [ $used -gt 90 ]
    then
        echo Low disk space
        break
    fi
    cp $value $1/backup/
done
```

Controlando loops

- `continue`
 - Força a parada de uma iteração do loop e segue para a próxima iteração.

loop_continue.sh

```
#!/bin/bash
# Make a backup set of files
for value in $1/*
do
    if [ ! -s $value ]
    then
        echo $value empty
        continue
    fi
    cp $value $1/backup/
done
```

Funções

Funções são formas facilmente executar partes do código que são executadas repetidas vezes.

Estrutura de uma função em Bash script:

```
function function_name {  
  
    <commands>  
  
}
```

A declaração da função deve aparecer antes da sua chamada no código.

function_ex1.sh

```
#!/bin/bash  
  
# Basic function  
  
print_something () {  
  
    echo Hello I am a function  
  
}  
  
print_something  
  
print_something
```


Funções - passando argumentos

Para passar algum argumento para uma função, fazemos de forma similar ao que fazemos para passar argumentos a um Bash script:

function_ex2.sh

```
#!/bin/bash

# Passing arguments to a function

print_something () {

    echo Hello $1

}

print_something Mars

print_something Jupiter
```

Funções - retornando valores

A maioria das linguagens de programação possui o conceito da função retornar valores para o local original onde a função foi chamada.

Em Bash script, não temos isso. Mas ele permite retornar o status do retorno.

function_ex3.sh

```
#!/bin/bash
# Setting a return status for a function
print_something () {
    echo Hello $1
    return 5
}

print_something Mars
print_something Jupiter
echo The function has a return value of $?
```

Funções - retornando valores

Uma forma de contornar isso seria utilizar a substituição do comando e fazer com que a função imprima um resultado, como no código ao lado:

function_ex4.sh

```
#!/bin/bash

# Setting a return value to a function
lines_in_file () {
    cat $1 | wc -l
}

num_lines=$( lines_in_file $1 )
echo The file $1 has $num_lines lines in
it.
```

Funções - escopo das variáveis

Por padrão, quando se declara uma variável, ela é considerada **global** (todo o script consegue ler a variável).

Podemos declarar uma variável de forma que ela seja acessível apenas dentro de um bloco de código (**escopo**), por exemplo dentro de uma função. Para isso, declara-se a variável com **local** na frente.

```
local var1='local 1'
```

scope.sh

```
#!/bin/bash
# Experimenting with variable scope
var_change () {
    local var1='local 1'
    echo Inside function: var1 is $var1 : var2 is $var2
    var1='changed again'
    var2='2 changed again'
}
var1='global 1'
var2='global 2'
echo Before function call: var1 is $var1 : var2 is $var2
var_change
echo After function call: var1 is $var1 : var2 is $var2
```

Referência

<https://ryanstutorials.net/bash-scripting-tutorial/bash-script.php>

<https://guide.bash.academy>

http://linuxcommand.org/lc3_writing_shell_scripts.php