Shell Script

Renan Fricks
Marcelo Pedro

O que é Shell?

Shell é a linha de comando do Linux (e UNIX). É o shell quem interpreta a linha de comandos digitada pelo usuário no terminal, em termos técnicos, é ele o responsável pela interface conversacional com o usuário.

Existem várias implementações do shell, dentre elas o *csh, tcsh, sh, bash, ksh, zsh*. Cada um pode executar comandos gerais do sistema de maneira semelhante, porém possuem estruturas e comandos próprios que os diferenciam.

Assim, o Shell Script é Uma linguagem que utiliza o shell para realizar ações automatizadas através de seus scripts, códigos e comandos.

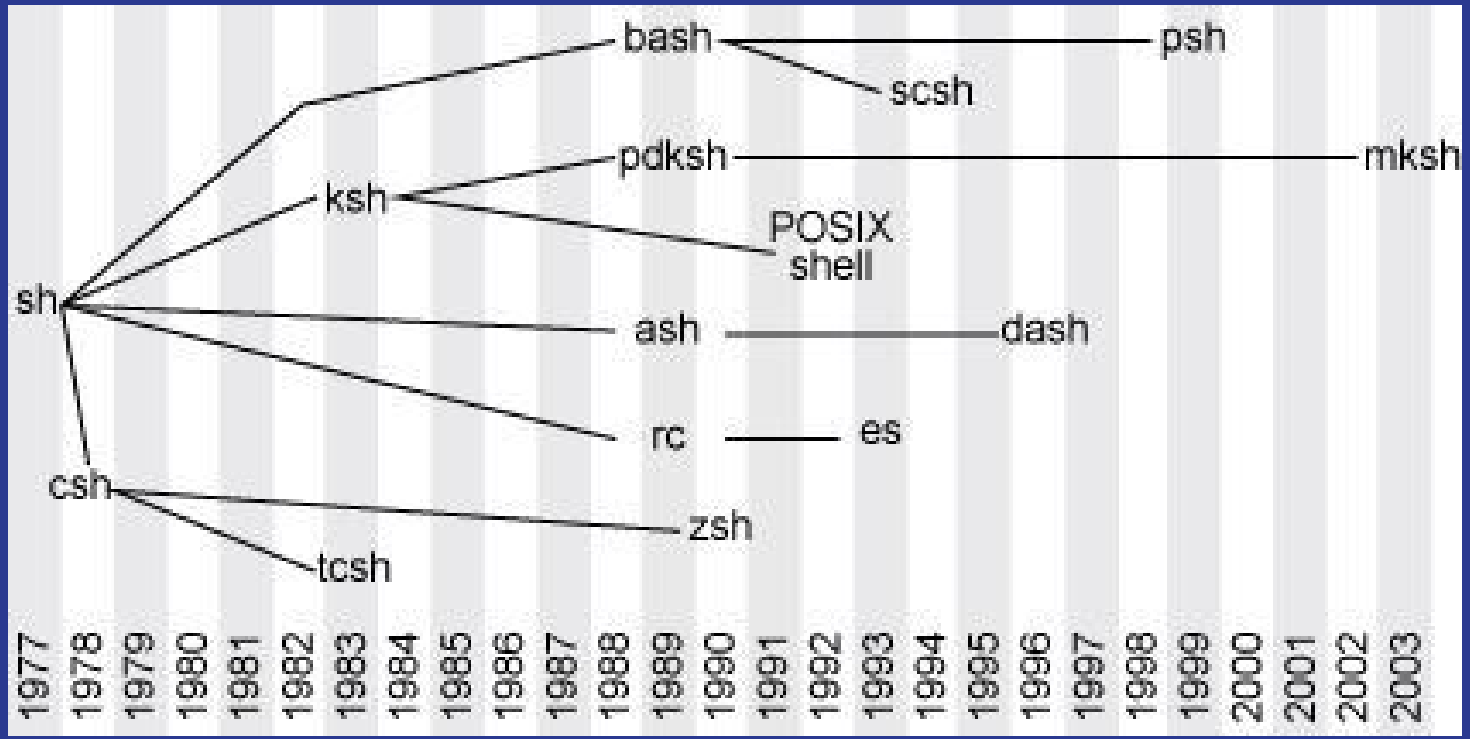
Histórico

A primeira shell para Unix chamada shell V6 foi criada em 1971 por Ken Thompson, distribuído entre as versões de 1 a 6 do Linux durante 1971 a 1975. Em seguida em 1977, foi introduzida a Bourne shell, criada por Stephen Bourne do AT&T Bell Labs para UNIX V7.

A Bourne shell tinha dois objetivos primários: servir como um interpretador de comando para executar comandos interativamente para o sistema operacional e para criação de scripts. Além de substituir a shell Thompson, a Bourne shell oferecia várias vantagens em relação a sua antecessora. Bourne introduziu fluxos de controle, loops e variáveis nos scripts, criando uma linguagem mais funcional para interagir com o sistema operacional (interativamente ou não). A shell também permitia usar scripts como filtros, fornecendo suporte integrado para lidar com sinais, mas não tinha a capacidade de definir funções. Por fim, ela incorporava alguns recursos usados hoje, incluindo substituição de comando (usando aspas invertidas).

A Bourne shell foi não apenas um importante passo para frente, mas também a âncora para diversas shells derivadas.

Histórico


bash
psh
scsh
pdksh
mksh
ksh
POSIX
shell
sh
ash
dash
rc
es
csh
zsh
tcsh
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003

Tipos de Shell

- Bourne Shell: É o shell padrão para Unix, ou seja, a matriz dos outros shells, portanto é um dos mais utilizados. É representado por "sh". Foi desenvolvido por Stephen Bourne, por isso Bourne Shell.
- Korn Shell: Este shell é o Bourne Shell evoluído, portando todos os comandos que funcionam no Bourne Shell funcionarão neste com a vantagem de ter mais opções. É representado por "ksh".
- C Shell: É o shell mais utilizado em BSD, e possui uma sintaxe muito parecida com a linguagem C. Este tipo de shell já se distancia mais do Bourne Shell, portanto quem programa para ele terá problemas quanto a portabilidade em outros tipos. É representado por "csh".
- Bourne Again Shell: É o shell desenvolvido para o projeto GNU usado pelo GNU/Linux, é muito usado pois o sistema que o porta evolui e é adotado rapidamente. Possui uma boa portabilidade, pois possui características do Korn Shell e C Shell. É representado por "bash". O nosso estudo estará focado neste.

Implementação

Shell Script é uma linguagem interpretada. Sendo assim, não precisa de um compilador, e sim de um interpretador como o bash ou sh para ler os comandos, interpretá-los e executá-los.

Para interpretar um código em shell script, basta colocar na primeiro linha do arquivo o interpretador que deseja utilizar, como em todos os exemplos utilizados neste trabalho foi o Bourne Again Shell, a primeira linha sempre terá quer o seguinte comando:

```
1   #!/bin/bash
2   echo "Hello World"
```

E para poder interpretar e executar o código é necessário fornecer permissão ao arquivo.

```
1   chmod +x nome_do_arquivo.sh
```

Paradigmas

O Shell Script se encaixa no paradigma imperativo estruturado. Estruturado por seguir uma sequência na execução dos comandos, permite laços de repetição e iterações com o usuário. Consideramos-o como imperativo porque executa comandos que mudam as variáveis, ele passa uma seqüência de comandos para o computador executar.

Identificadores

Shell Script é case sensitive.

Exemplo de código:

```
1 #!/bin/bash
2
3 xy=10
4 xY=20
5
6 echo "$xy"
```

Imprime:

10

```
1 #!/bin/bash
2
3 if=5
4
5 if [ 10 -eq 10 ];
6 then
7     echo "E igual!"
8 fi
9
10 echo "$if"
11
12 echo "$xy"
```

Imprime:

E igual!
5

Escopo

Shell Script utiliza escopo estático. Onde é possível separar funções em blocos aninhados.

```
1  #!/bin/bash
2
3  function hello(){
4      d=1
5      echo $d
6  }
7
8  a=1
9
10 if [ $a == 1  ];
11 then
12     b=1;
13     echo $b;
14 fi
15
16 b=2;
17 echo $b;
18
19 hello
20
21 d=2
22 echo "$d"
```

Imprime::

1
2
1
2

Palavras reservadas

- read, echo
- if, then, elif, else, fi
- case, esac
- for, while, do, done
- function
- declare
- let
- pwd

```
1 #!/bin/bash
2
3 echo "Digite um valor: "
4 read n
5
6 echo "O valor digitado foi: $n"
```

Palavras reservadas

- read, echo
- if, then, elif, else, fi
- case, esac
- for, while, do, done
- function
- declare
- let
- pwd

```
1  #!/bin/bash
2
3  n1=4
4  n2=4
5
6  if [ $n1 -gt $n2 ];
7  then
8      echo "n1 e maior que n2!"
9
10 elif [ $n1 -lt $n2 ];
11 then
12     echo "n1 e menor que n2!"
13
14 else
15     echo "n1 e n2 sao iguais"
16 fi
```

Imprime::

n1 e n2 sao iguais

Palavras reservadas

- read, echo
- if, then, elif, else, fi
- case, esac
- for, while, do, done
- function
- declare
- let
- pwd

```
1  #!/bin/bash
2
3  echo "Digite o valor:"
4  read cont
5
6  case $cont in
7    "1")
8      echo "A entrada foi 1"
9    ;;
10   "2")
11     echo "A entrada foi 2"
12   ;;
13   "3")
14     echo "A entrada foi 3"
15   ;;
16 esac
```

Palavras reservadas

- read, echo
- if, then, elif, else, fi
- case, esac
- for, while, do, done
- function
- declare
- let
- pwd

```
1 #!/bin/bash
2
3 for i in {0..10};
4 do
5    echo "$i"
6 done
```

Imprime:

0
1
2
3
4
5
6
7
8
9
10

```
1 #!/bin/bash
2
3 i=0
4
5 while [ $i -ne 10 ];
6 do
7      echo "O valor e $i"
8      i=$[ $i + 1 ]
9 done
```

Imprime:

0
1
2
3
4
5
6
7
8
9

Palavras reservadas

- read, echo
- if, then, elif, else, fi
- case, esac
- for, while, do, done
- function
- declare
- let
- pwd

```
#!/bin/bash

function soma {
    echo "Digite o valor 1"
    read a

    echo "Digite o valor 2"
    read b

    echo "A soma: $[ $a + $b ]"
}

function subtrair {
    echo "Digite o valor 1"
    read a

    echo "Digite o valor 2"
    read b

    echo "A subtracao: $[ $a - $b ]"
}

echo "Digite 1 para somar ou 2 para subtrair"
read cont

case $cont in
    "1")
        soma
    ;;
    "2")
        subtrair
    ;;
esac
```

Palavras reservadas

- read, echo
- if, then, elif, else, fi
- case, esac
- for, while, do, done
- function
- declare
- let
- pwd

Serve para declarar as variáveis e atribuir valores. Podemos combinar isso com algumas opções, podendo assim atribuir tipos as variáveis:

-i declarar números inteiro;
-r para variáveis de somente leitura;
-a array;
-f para funções;
-x para variáveis que podem ser 'exportadas' fora do ambiente do script em si.

Palavras reservadas

- read, echo
- if, then, elif, else, fi
- case, esac
- for, while, do, done
- function
- declare
- let
- pwd

```
 1 #!/bin/bash
 2
 3 cont=0
 4
 5 let cont=$cont+1;
 6 let cont=$cont+1;
 7 let cont=$cont+1;
 8 let cont=$cont+1;
 9 let cont=$cont+1;
10
11 echo "$cont"
```

Imprime:

5

Palavras reservadas

- read, echo
- if, then, elif, else, fi
- case, esac
- for, while, do, done
- function
- declare
- let
- pwd

```
1 #!/bin/bash
2
3 caminho=$(pwd)
4
5 echo "$caminho"
```

Tipos

Shell Script possui tipagem dinâmica, o que quer dizer que não é necessário uma declaração prévia do tipo da variável. O tipo é assumido ao inserir um valor.

Shell Script é type-safe, ou seja, assume uma medida que desestimula ou impede erros de tipo.

Suporta:
Inteiros, ponto flutuante, caracter, strings e vetores unidimensionais

Não suporta:
Enumerado, produto cartesiano,recursivo, ponteiro e conjuntos potência

Tipos

Shell Script possui um fraca tipagem, o que quer dizer na prática o seguinte:

par=3
par=”Jose da Silva”
par=14.5
par=(“Casa” “Jose” “Bola”)

Constantes

Para se declarar constantes basta escolher um desses três comando:
readonly varName=value;
declare -r varName=value;
typeset – r varName= value;

```
1 #!/bin/bash
2
3 readonly DATA='oi'
4 echo $DATA
5
6 DATA='ola'
7 echo $DATA
```

//Error … readonly variable

Variáveis Globais e Locais

Todas as variáveis são globais por definição:

```
1  #!/bin/bash
2
3  function variavel {
4      val=3
5  }
6
7  val=0
8
9  variavel
10
11 echo "$val"
```

Imprime:

3

Variáveis Globais e Locais

Para que uma variável seja local e visível apenas em seu bloco de programa, basta usar o comando local:

```
1  #!/bin/bash
2
3  function variavel {
4       local val=3
5
6       echo "$val"
7  }
8
9  val=0
10
11 variavel
12
13 echo "$val"
```

Imprime:

3
0

I/O Arquivos

Exemplo de código que ler uma linha de um arquivo e exibe-a no terminal:

```
1 #!/bin/bash
2
3 while read linha;
4 do
5       echo $linha;
6
7 done < exemplo.txt
```

I/O Arquivos

Exemplo de código que lê uma mensagem no terminal e escreve num arquivo:

```
1 #!/bin/bash
2
3 echo "Qual mensagem?";
4 read resp;
5
6 echo $resp >> "/home/Desktop/arquivo.txt";
```

Array

Array pode ser declaradas passando um conjunto de dados juntos para o mesmo. No bash é declarado da seguinte forma.

```
nomes=("Maria" "Joao" "Pedro" "Paulo")
```

Ou então pode ser declarado passando os dados aos poucos dando sua posição:

```
nomes[0]="Maria"
nomes[1]="Joao"
nomes[2]="Pedro"
nomes[3]="Paulo"
```

Array

Para referenciar e acessar o seu valor basta usar a declaração “${}”.
Com o exemplo anterior:

nomes=("Maria" "Joao" "Pedro" "Paulo")

Se quiser acessar o segundo elemento basta fazer:

${nomes[1]}

Retorna Joao

Array

Para referenciar e acessar o seu valor basta usar a declaração “${}”.
Com o exemplo anterior:

nomes=("Maria" "Joao" "Pedro" "Paulo")

Se quiser acessar o segundo elemento basta fazer:

pessoa=${nomes[1]}

pessoa receberá Joao.

Array

```
1 #!/bin/bash
2
3 nomes=("Maria" "Joao" "Pedro" "Paulo")
4
5 for i in {0..3}
6 do
7     echo "${nomes[$i]}"
8 done
```

Imprime:

Maria
Joao
Pedro
Paulo

Array

Mais alguns comandos:

Com o vetor: nomes=("Maria" "Joao" "Pedro" "Paulo" "Raul" "Carlos" "Ana" "Lais")

- *${nomes[@]:n}* -> Retorna todos os elementos a partir da posição *n.*
  Para o vetor nomes, com n=2, o retorno seria: "Pedro" "Paulo" "Raul" "Carlos" "Ana" "Lais".

- *${nomes[@]:n:p}* -> Retorna o elemento *n* mais *p-1* elementos.
  Para o vetor nomes, com n=2 e p=4, o retorno seria: "Pedro" "Paulo" "Raul" "Carlos".

- *${#nomes[@]}* -> Retorna a quantidade de elementos no vetor.
  Para o vetor nomes, o retorno seria: 8.

String

Para declarar uma string basta inserir uma em uma variável
var=”Linguagens de progamacao”

Existem alguns comandos que facilita o tratamento com strings.

${#var} -> Retorna o tamanho de uma string, neste caso: 25.
${var%cao} -> Retorna a string ‘var’ mas sem o “cao” no final, neste caso: “Linguagens de progama”.
${var#Lin} -> Retorna a string ‘var’ mas sem “Lin” no inicio, neste caso: “guagens de progamacao”.
${var:5} -> Retorna a string ‘var’ a partir da posição 5: “agens de progamacao”.
${var:2:5} -> Retorna 5 elementos a partir do elemento de posição 2: “nguag”.
${var#*g} -> Elimina o caracter selecionado e tudo a esquerda: “uagens de programacao”.
${var%g*} -> Elimina o caracter selecionado e tudo a direita: “Linguagens de pro”.

String

Mais alguns comandos:

Remove a partir de um certo valor.

```
1  #!/bin/bash
2
3  co="de"
4  var="Linguagens de programacao"
5
6  var2=${var%%$co*}
7
8  echo "$var2"
9
10 ### Saida: Linguagens
```

Remove até um certo valor:

```
1  #!/bin/bash
2
3  co="de"
4  var="Linguagens de programacao"
5
6  var2=${var##*$co}
7
8  echo "$var2"
9
10 ### Saida:  programacao
```

String

Mais alguns comandos:

Substitui a primeira ocorrência de uma string por outra.

```
1 #!/bin/bash
2
3 var="Linguagens de programacao de"
4
5 var2=${var/de/para}
6
7 echo "$var2"
8
9 ### Saida:  Linguagens para programacao de
```

Substitui todas as ocorrências de uma string por outra.

```
1 #!/bin/bash
2
3 var="Linguagens de programacao de"
4
5 var2=${var//de/para}
6
7 echo "$var2"
8
9 ### Saida:  Linguagens para programacao para
```

Modularização

Assim como C possui a opção de incluir uma biblioteca, o Shell Script possui uma maneira de incluir outros arquivos com script para tentar modularizar melhor o código.

Exemplo:

Se tivéssemos que criar um código que medisse a área e o perímetro de um retângulo, poderíamos definir dois subprogramas e colocá los em arquivos diferentes. O diretório ficaria assim:

|Exemplo

|perimetro.sh

|area.sh

|main.sh

Modularização

No arquivo perimetro.sh teríamos o nosso código:

```
1  #!/bin/bash
2
3  function perimetro() {
4      "Digite a base do retangulo"
5      read b
6
7      "Digite a altura do retangulo"
8      read h
9
10     p=$[ $[ $b * 2 ] + $[ $h * 2 ] ]
11
12     echo "$p"
13 }
```

Modularização

No arquivo area.sh teríamos o nosso código:

```
1  #!/bin/bash
2
3  function area {
4      "Digite a base do retangulo"
5      read b
6
7      "Digite a altura do retangulo"
8      read h
9
10     a=$[ $b * $h ]
11
12     echo "$a"
13 }
```

Modularização

No arquivo main.sh teríamos o nosso código:

```
1  #!/bin/bash
2
3  caminho=$(pwd)
4
5  caminhoPerimetro=$caminho'/perimetro.sh'
6  caminhoArea=$caminho'/area.sh'
7
8  . $caminhoPerimetro
9  . $caminhoArea
10
11 perimetro
12 area
```

Gerência de memória

Toda memória alocada é liberada ao final do programa. E todo gerenciamento da memória é feito pelo sistema operacional Unix.

Não há suporte para serialização

Operadores

id++ id--	pós incrementador e decrementador
++id --id	pré incrementador e decrementador
+ -	adição subtração
* / %	multiplicação, divisão, resto
**	exponenciação
<< >>	left e right shifts bit a bit
<= >= < >	comparação
== !=	igual e diferente
& e	bit a bit
^	ou exclusivo bit a bit

Operadores

| ou bit a bit
&& e lógico
|| ou lógico
expr ? expr : expr operador condicional

Usado para comparar inteiros:

-eq	Equal	Igual
-ne	Not Equal	Diferente
-gt	Greater than	Maior que
-ge	Gretaer or equal	Maior ou igual a
-lt	Less than	Menor que
-le	Less than or equal	Menor ou igual a

Expressões

Literais: 'a', 9 , 10.0

aritméticas: a= expr 1 + 3; let b=2+2; declare -i c =5 - 2;

relacionais:
if [a<=b]; then
x=4;
fi

Expressões

Se quisermos fazer cálculos com números de ponto flutuante ou fazer uma comparação com ponto flutuante precisamos de uma ferramenta chamada bc.

```
r=3.5;
s= echo “$r + 2.2” | bc
echo $s

if [ $( echo “$s < 3.4” | bc ) -eq 1 ]; then
      echo ei;
fi
```

Expressões

Booleanas:

```
if [ x==3 && b==4 ]; then
c=5;
fi
```

Binárias

```
a=3
b=2
echo $(( $a & $b))
echo $(( $a | $b))
```

Comandos

Atribuições:
a= expr $b +3 / $c (atribuição simples)

let a=b=0 (atribuição múltipla)

let a+=3 (atribuição composta)

let a++ (atribuição unária)

Sem ser para atribuição simples, os outros tipos de atribuições precisa do comando let para serem feitas.

Comandos

Condicionais:

Seleção de caminho condicionado:

```
if [ $x == $a ]; then
      b=3;
fi
```

Seleção de caminho duplo:

```
if [ $x == $a ]; then
      b=3;
else
      b=4;
fi
```

Comandos

Caminhos múltiplos com ifs aninhados:

```
if [ $a < 0 ]; then
        b=0;
elif [ $a > 0 ]; then
        b=1;
else
        b=2;
fi
```

Comandos

Seleção de caminhos múltiplos:

```
case $NUM in
    1) echo "one" ;;
    2) echo "two" ;;
    3) echo "three" ;;
    4) echo "four" ;;
    5) echo "five" ;;
    *) echo "INVALID NUMBER!" ;
esac
```

Comandos

Iterativos:

```
x=1
while [ $x -le 5 ]
do
	echo "Welcome $x times"
	let x++;
done
```

Não tem comando iterativo pós-teste(comando similar ao do-while em C).

Comandos

```
for (( c=1; c<=5; c++ ))
do
   echo "Welcome $c times"
done

for i in {1..5}
do
   echo "Welcome $i times"
done

for i in {0..10..2}
  do
     echo "Welcome $i times"
 done
```

Comandos

Escapes:

```
while [ $a -le 5 ]
do
    let a++
    if [ $a==3 ]
    then
        break
    else
        continue
    fi
done
```

Não há presença de goto

Parâmetros

Possui varargs:

```
function imprimealgo(){
    echo imprime $1 $2
}
a=1
b=2
imprimealgo $a $b

//Saída: Imprime 1 2
```

Parâmetros

Shell Script usa passagem unidirecional de entrada variável, mecanismo de passagem por cópia e momento de passagem é normal.

Polimorfismo

Não existe !!!

Exceções

Só existem as que são lançadas pela linguagem. Lançando apenas mensagem de error e continuando a execução do código a partir da próximo linha em que se gerou a exceção.

Concorrência

Não existe !!!

Comparação entre as linguagens

Critérios Gerais	C	Java	Shell Script
Aplicabilidade	Sim	Parcial	Parcial
Confiabilidade	Não	Sim	Não
Aprendizagem	Não	Não	Sim
Eficiência	Sim	Parcial	Parcial
Portabilidade	Não	Sim	Parcial
Evolutibilidade	Não	Sim	Parcial
Reusabilidade	Parcial	Sim	Parcial
Integração	Sim	Parcial	Sim
Custo	Depende da operação	Depende da operação	Depende da operação

Conclusão

Programar em Shell Script envolve a manipulação de textos e o gerenciamento de processos e de arquivos. As tarefas mais complexas ficam com a responsabilidade das ferramentas do sistema como sed, grep e find.

Além de possuir as funcionalidades básicas de uma linguagem estruturada e a integração com o sistema operacional e suas ferramentas, há as facilidades de redirecionamento, em que é possível combinar vários programas entre si , multiplicando seus poderes e evitando reescrita de código.

Referências

- http://aurelio.net/shell/
- https://www.ibm.com/developerworks/br/library/l-linux-shells/
- https://fit.faccat.br/~guto/artigos/Artigo_ShellScript.pdf
- https://books.google.com.br/books?id=snmJepzoNfgC&pg=PA26&lpg=PA26&dq=shell+script+controla+hardware&source=bl&ots=PMXs1B3LWy&sig=j7MwrQHdGUm50zkI4nFts2ificI&hl=pt-BR&sa=X&ved=0ahUKEwjP4pTVtanQAhWJIpAKHY_TB38Q6AEIQzAG#v=onepage&q&f=true
- https://tiswww.case.edu/php/chet/bash/bashref.html
- https://www.inf.ufes.br/~vitorsouza/lp-20162/