Mahasarakham University

มหาวิทยาลัยมหาสารคาม

Heart of
the Northeast

www.msu.ac.th

Mahasarakham University
มหาวิทยาลัยมหาสารคาม

Introduction to Shell Scripting: The Basics

แนะนำการทำงานของ Shell Script: พื้นฐาน

Heart of the Northeast

www.msu.ac.th

โดย อาจารย์โอฬาริก สุรินต๊ะ

Jan 10, 2017

Agenda

- Introduction to Shell Scripts
- Purpose of using Shell Scripts
- Types of Shells / Running Shells
- Creating Simple Shell Scripts
  - Creating Portable Shell Scripts
  - Commenting
  - Running Shell Scripts

Introduction to Shell Scripts

- Unix Shell scripting is a fascinating combination of art and science that gives you access to the incredible flexibility and power of Linux with very simple tools.
- Unix Shell Scripts are simply files that contain Unix commands that can be run or "issued" as a command (program).

Purposes of Using Shell Scripts

- Automate repetitive tasks (instead of manually issuing commands manually).
- Create new commands or utilities (with specified arguments including options) to meet the system administrator's needs.
- Combine lengthy and repetitive sequences of commands into a single and simple command.

Purposes of Using Shell Scripts

- Generalize a sequence of operations on one set of data into a procedure that can be applied to any similar set of data.
- Create new commands using combinations of utilities in ways the original authors never thought of.
- Simple shell scripts might be written as shell aliases, but the script can be made available to all users and all processes. Shell aliases apply only to the current shell.

Purposes of Using Shell Scripts

- Wrap programs over which you have no control inside an environment that you can control.
- Rapid prototyping.

Typical uses

- System boot scripts (/etc/init.d)
- System administrators, for automating many aspects of computer maintenance, user account creation, etc.
- Application package installation tools
- Application startup scripts, especially unattended applications.
- Any user needing to automate the process of setting up and running applications or their own code.

Type of Shells in Linux

- The Linux operating system offers several shells.
- These command-line interfaces provide powerful environments for software development and system maintenance.
- Shells have many commands in common, each type has unique features.
- Individual programmers come to prefer one type of shell over another.

Type of Shells

- sh
  - The Bourne shell, called "sh," is one of the original shells, developed for Unix computers by Stephen Bourne at AT&T's Bell Labs in 1977.
  - Small, simple and very few internal commands, so it called external programs for even the simplest of tasks.
  - It is always available on everything that looks like Unix.

Type of Shells

- csh
  - The "C" shell. (Bill Joy at Berkeley). Many things in common with the Bourne shell, but many enhancements to improve interactive use.
  - The internal commands used only in scripts are very different from "sh", and similar to the "C" language syntax.

Type of Shells

- tcsh
  - The "TC" shell. Freely available and based on "csh".
  - It has many additional features to make interactive use more convenient.

Type of Shells

- ksh
  - The "Korn" shell, written by David Korn of AT&T Bell Labs.
  - Written as a major upgrade to "sh" and backwards compatible with it, but has many internal commands for the most frequently used functions.
  - It also incorporates many of the features from tcsh which enhance interactive use.

Type of Shells

- **POSIX 1003.2**
  - Standards committees worked over the Bourne shell and added many features of the Korn shell and C shell to define a standard set of features which all compliant shells mush have.

Type of Shells

- **bash**
  - The "Bourne again" shell. Written as part of the GNU/Linux Open Source effort and the default shell for Linux and MacOSX.
  - It is a functional clone of sh with additional features to enhance interactive use, add POSIX compliance, and partial ksh compatability.

Type of Shells

- zsh
  - A freeware functional clone of sh, with parts of ksh, bash and full POSIX compliance, and many new interactive command-line editing features.
  - It was installed as the default shell on early MacOSS systems.

Creating Simple Shell Scripts

- On the first line of the shell script, you can choose which shell in which to run the script:

First Line	Shell
#!/bin/bash	Bash
#!/bin/ksh	Korn
#!/bin/csh	C
#!/bin/zsh	Z
#!/bin/sh	Bourne

Warning:

must be located in first column of first line

! must be located in second column of first line immediately followed by correct pathame or shell is run in “default” shell

Creating Simple Shell Scripts

- It is a good idea to provide comments near the top of your shell script and comments throughout your shell script to help explain what the shell script does.

Creating Simple Shell Scripts

- **Commenting Shell Scripts**
  - A comment is a hash symbol **#**
  - Any text after the hash symbol # is ignored by the shell for the remainder of the line.

Example:

 # This shell script does ....

mkdir mydir *# Command to create directory "mydir"*

Version of shells

```
$ echo ${BASH_VERSION}
```

4.3.46(1)-release

Variables

- **Variables Names**
  - The name of a variable can contain only letters (**a to z** or **A to Z**), numbers (**0 to 9**) or the underscore character (**_**).
  - Names must start with a letter.
  - A name must not contain embedded spaces. Use underscores instead.
  - Punctuation marks is not allowed.

Variables

Valid	Invalid
ALI	2_VAR
TOKEN_A	-VAR1
VAR_1	VAR1-VAR2
VAR_2	VAR_A!

Variables

- **Defining Variables**
  - Variables are defined as follows

variable_name=variable_value

For example:

NAME=″Olarik Surinta″

Variables

Valid	Invalid
NAME=″Olarik″	NAME = “Olarik”
NAME=″Olarik KONG″	NAME= “Olarik”
	NAME =“Olarik”

Variables

- **Accessing Values**
  - To access the value stored in a variable, prefix its name with the dollar sign ($)

```
NAME="Olarik"
echo $NAME
```

Variables

- **Read-only Variables**
  - The shell provides a way to mark variables as read-only by using the readonly command.
  - After a variable is marked read-only, its value cannot be changed.

```
NAME="Olarik"
readonly NAME
NAME="Olarik Surinta"
```

Result:

```
bash: NAME: readonly variable
```

Variables

- **Unsetting Variables**
  - Unsetting or deleting a variable tells the shell to remove the variable from the list of variables that it tracks.
  - Once you unset a variable, you would not be able to access stored value in the variable.

```
NAME="Olarik"
unset NAME
Echo $NAME
```

Variables

- **Variable Types**
  - Local Variables
  - Environment Variables
  - Shell Variables

Variables

- **Local Variables**
  - A local variable is a variable that is present within the current instance of the shell.
  - It is not available to programs that are started by the shell.
  - They are set at command prompt.

Variables

- **Environment Variables**
  - An environment variable is a variable that is available to any child process of the shell.
  - Some programs need environment variables in order to function correctly.
  - Usually a shell script defines only those environment variables that are needed by the programs that it runs.

Variables

- **Shell Variables**
  - A shell variable is a special variable that is set by the shell and is required by the shell in order to function correclty.
  - Some of these variables are environment variables whereas others are local variables.

Creating Simple Shell Scripts

```
#!/bin/sh

# Built-in commands and keywords (e.g. echo)
echo "Username $USER"

# show date
echo "Today is "
date

var=$(date)
echo "Today is $var"

# Print working directory
workDir=$(pwd)
echo "We are working on the $workDir"
```

Running Shell Scripts

- There are two methods of running shell scripts.
  - Type the shell type with the name of the shell script as an argument

    eg. **bash script-name**
  - Type the name of the shell script.
  - This requires that that shell script file has at least read and execute permissions.

    eg. **script-name** or **./script-name**

Special Variables in Shell Scripts

Variable	Description
$0	The filename of the current script.
$n	These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on).
$#	The number of arguments supplied to a script.
$*	All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.
$@	All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.
$?	The exit status of the last command executed.
$$	The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
$!	The process number of the last background command.

Command-Line Arguments

- The command-line arguments $1, $2, ..., $9 are positional parameters, with $0 pointing to the acutal command, program, shell script, or function and $1, $2, ..., $9 as the arguments to the command.

Command-Line Arguments

```
#!/bin/sh

echo "File Name: $0"
echo "First Parameter : $1"
echo "Second Parameter : $2"
echo "Quoted Values: $@"
echo "Quoted Values: $*"
echo "Total Number of Parameters : $#"
```

Command-Line Arguments

```
$./test.sh Zara Ali

File Name : ./test.sh
First Parameter : Zara
Second Parameter : Ali
Quoted Values: Zara Ali
Quoted Values: Zara Ali
Total Number of Parameters : 2
```

Special Parameters $* and $@

- $* and $@ both will act the same unless they are enclosed in double quotes, "".
- Both parameter specifies all command-line arguments but the "$*" special parameter takes the entire list as one argument with spaces between and the "$@" special parameter takes the entire list and separates it into separate arguments.

Special Parameters $* and $@

```
#!/bin/sh

for TOKEN in $*
do
    echo $TOKEN
done
```

Special Parameters $* and $@

$./test.sh Zara Ali 10 Years Old

Zara

Ali

10

Years

Old

Shell Arrays

- A shell variable is capable enough to hold a single value. This type of variables are called scalar variables.
- Shell supports a different type of variable called an array variable that can hold multiple values at the same time.
- Arrays provide a method of grouping a set of variables. Instead of creating a new name for each variable that is required, you can use a single array variable that stores all the other variables.

Shell Arrays

- **Defining Array Values**
  - Trying to represent the names of various students as a set of variables. Each of the individual variables is a scalar variable as follows

```
NAME01="Zara"
NAME02="Qadir"
NAME03="Mahnaz"
NAME04="Ayan"
NAME05="Daisy"
```

Shell Arrays

- **Defining Array Values**
  - We can use a single array to store all the above mentioned names.
  - Following is the simplest method of creating an array variable is to assign a value to one of its indices.

array_name[index]=value

Shell Arrays

- Defining Array Values

```
NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
```

- If you are using **bash** shell, here is the syntax of array initialization

```
array_name=(value1 ... valuen)
```

Shell Arrays

- **Accessing Array Values**
  - After you have set any array variable, you access it as follows

${array_name[index]}

Shell Arrays

- **Accessing Array Values**

```
#!/bin/sh

NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
echo "First Index: ${NAME[0]}"
echo "Second Index: ${NAME[1]}"
```

Shell Arrays

- **Accessing Array Values**
  - You can access **all the items** in an array in one of the following ways

${array_name[*]}

${array_name[@]}

Shell Arrays

```
#!/bin/bash

NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"

tLen=${#NAME[@]}
echo "Length of array: $tLen"

j=1;
for ((i=0; i<$tLen; i++));
do
  echo "[$j] ${NAME[$i]}"
  j=$((j+1))
done
```

```
Length of array: 5
[1] Zara
[2] Qadir
[3] Mahnaz
[4] Ayan
[5] Daisy
```

Shell Basic Operators

- There are various operators supported by each shell.
- This slide is based on default shell (Bourne).
- There are following operators which we are going to discuss.
  - Arithmetic Operators
  - Relational Operators
  - Boolean Operators
  - String Operators
  - File Test Operators

Arithmetic Operators

- Here is simple example to add two numbers

Ex1:

```
#!/bin/sh
val='expr 2 + 2'
echo "Total value: $val"
```

Ex2:

```
#!/bin/sh
let "val = 2 + 2"
echo "Total value: $val"
```

Arithmetic Operators

Ex3:

```
#!/bin/sh
let "val = 2 + 2"
printf 'Total value: %s\n' "$val"
```

Arithmetic Operators

- Parenthesis in expr arithmetic

```
$ expr 3 * (2 + 1)
bash: syntax error near unexpected token `('
```

How can we solve this problem?

Arithmetic Operators

- Solution 1

```
$ expr 3 '*' '(' 2 '+' 1 ')'
```

- Solution 2

```
$ let 'val=3*(2+1)'
$ echo $val
```

Arithmetic Operators

- Solution 3

```
$ echo "$(( 3 * (2 + 1) ))"
```

- Solution 4

```
$ val=$(( 3*(2+1) ))
$ echo "$val"
```

Arithmetic Operators

Operator	Description
+	Addition - Adds values on either side of the operator
-	Subtraction - Subtracts right hand operand from left hand operand
*	Multiplication - Multiplies values on either side of the operator
/	Division - Divides left hand operand by right hand operand
%	Modulus - Divides left hand operand by right hand operand and returns remainder
=	Assignment - Assign right operand in left operand
==	*Equality - Compares two numbers, if both are same then returns true.*
!=	*Not Equality - Compares two numbers, if both are different then returns true.*

Arithmetic Operators

Ex:

```
#!/bin/sh

a=10
b=20
if [ $a == $b ]
then
    echo "a is equal to b"
fi

if [ $a != $b ]
then
    echo "a is not equal to b"
fi
```

Relational Operators

- Bourne shell supports following relational operators which are specific to numeric values.
- These operators would not work for string values unless their value is numeric.

Relational Operators

Operator	Description
-eq	Checks if the value of two operands are equal or not, if yes then condition becomes true.
-ne	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.
-gt	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
-lt	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
-le	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

Boolean Operators

Operator	Description
!	This is logical negation. This inverts a true condition into false and vice versa.
-o	This is logical OR. If one of the operands is true then condition would be true.
-a	This is logical AND. If both the operands are true then condition would be true otherwise it would be false.

String Operators

Operator	Description
=	Checks if the value of two operands are equal or not, if yes then condition becomes true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.
-z	Checks if the given string operand size is zero. If it is zero length then it returns true.
-n	Checks if the given string operand size is non-zero. If it is non-zero length then it returns true.
str	Check if str is not the empty string. If it is empty then it returns false.

String Operators

```
#!/bin/sh

a="abc"
b="efg"

if [ $a = $b ]
then
   echo "$a = $b : a is equal to b"
else
   echo "$a = $b: a is not equal to b"
fi

if [ $a != $b ]
then
   echo "$a != $b : a is not equal to b"
else
   echo "$a != $b: a is equal to b"
fi
```

```
if [ -z $a ]
then
   echo "-z $a : string length is zero"
else
   echo "-z $a : string length is not zero"
fi

if [ -n $a ]
then
   echo "-n $a : string length is not zero"
else
   echo "-n $a : string length is zero"
fi

if [ $a ]
then
   echo "$a : string is not empty"
else
   echo "$a : string is empty"
fi
```

The Beginner's Guide to Nano

- http://www.howtogeek.com/howto/42980/the-beginners-guide-to-nano-the-linux-command-line-text-editor/

The Beginner's Guide to Nano

- **Shortcuts**
  - Ctrl+G (^G) Help documentation
  - Ctrl+X (^X) Exit
  - ^Y Prev page
  - ^V Next page
  - ^W Search text
  - ^J Justify
    - ^U Unjustify
  - ^K Cut text
    - ^U Uncut text
  - ^D Delete 1 character

The Beginner's Guide to Nano

- Shortcuts
  - ^O WriteOut
  - ^R Read File (Insert file into current working file)
    - ^T Show list of files

- https://www.dartmouth.edu/~rc/classes/ksh/welcome.html
- http://slideplayer.com/slide/8877306/
- https://www.tutorialspoint.com/unix/unix-using-variables.htm
-