Shell Scripting

Operating System Lab
Spring 2015

Roya Razmnoush
CE & IT Department, Amirkabir University of Technology

Contents

- **What is scripting language?**
- **What is the difference between scripting and programming language?**
- **Shell script**
- **Shell variables**
- **Positional parameters**
- **Variables and environment**
- **Control statement**
- **Exercise**

Scripting Concept

What is scripting language?

The definition of scripting language

- A high-level programming language that is interpreted by another program at runtime rather than compiled by the computer's processor as other programming languages (such as C and C++) are. Scripting languages, which can be embedded within HTML, commonly are used to add functionality to a Web page, such as different menu styles or graphic displays or to serve dynamic advertisements. These types of languages are client-side scripting languages, affecting the data that the end user sees in a browser window. Other scripting languages are server-side scripting languages that manipulate the data, usually in a database, on the server.

Examples of scripting language

- PHP
- Perl
- JavaScript
- UNIX shell script
- Phyton
- Tcl
- JSP
- ASP
- Erlang

- A **scripting language** or **script language** is a programming language that supports **scripts**, programs written for a special run-time environment that can interpret (rather than compile) and automate the execution of tasks that could alternatively be executed one-by-one by a human operator

- Environments that can be automated through scripting include software applications, web pages within a web browser, the shells of operating systems (OS), and embedded systems

- A scripting language can be viewed as a domain-specific language for a particular environment; in the case of scripting an application, this is also known as an **extension language**

- Scripting languages are also sometimes referred to as very high-level programming languages, as they operate at a high level of abstraction, or as **control languages**, particularly for job control languages on mainframes.

- The term "scripting language" is also used loosely to refer to dynamic high-level general-purpose language, such as Perl, Tcl, and Python, with the term "script" often used for small programs (up to a few thousand lines of code) in such languages, or in domain-specific languages such as the text-processing languages sed and AWK.

- The spectrum of scripting languages ranges from very small and highly domain-specific languages to general-purpose programming languages used for scripting. Standard examples of scripting languages for specific environments include: Bash, for the Unix or Unix-like operating systems; ECMAScript (JavaScript), for web browsers; and Visual Basic for Applications, for Microsoft Office applications. Python is a general-purpose language that is also commonly used as an extension language.

Types of scripting language

- **Glue languages**
- **Job control languages and shells**
- **GUI scripting**
- **Application-specific languages**
- **Extension/embeddable languages**

Scripting Concept

What is the difference between scripting and programming language?

- C, C++, Pascal
- Scripting languages run inside another program.
- Scripting languages are easy to use and easy to write.
- Scripting languages are not compiled to machine code by the user (python, perl, shell, etc.). Rather, another program (called the interpreter, runs the program and simulates its behavior)

Important files for bash!!!!

- interactive login shell. This is used when logging in to a machine, invoking Bash with the --login option or when logging in to a remote machine with SSH.
- “ordinary” interactive shell. This is normally the case when starting xterm, konsole, gnome-terminal or similar tools.
- non-interactive shell. This is used when invoking a shell script in the command line.

startup files

- These files contain the aliases and environmental variables made available to Bash running as a user shell and to all Bash scripts invoked after system initialization.

- /etc/profile Systemwide defaults, mostly setting the environment (all Bourne-type shells, not just Bash)

- /etc/bashrc systemwide functions and aliases for Bash

- $HOME/.bash_profile user-specific Bash environmental default settings, found in each user's home directory (the local counter part to /etc/profile)

- $HOME/.bashrc user-specific Bash init file, found in each user's home directory (the local counterpart to /etc/bashrc). Only interactive shells and user scripts read this file.

Shell Script

A simple start!

What is shell script?

- A **shell script** is a computer program designed to be run by the Unix shell, a command line interpreter. The various dialects of shell scripts are considered to be scripting languages
- Typical operations performed by shell scripts include file manipulation, program execution, and printing text

What is shell script?

- A Text File
- With Instructions
- Executable

- Why shell script?
  - Simply and quickly initiate a complex series of tasks or a repetitive procedure

Creating first script

> Type your script in gedit or vi editor

```
 #!/bin/sh
for file in
do
if grep -q POSIX $file
then
echo $file
fi
done
exit 0
```

> Save it with .sh

> Make it executable with chmod

Simple Example

```
$ cat > hello.sh
#!/bin/sh
echo ‘Hello, world’
$ chmod +x hello.sh
$ ./hello.sh  OR  sh hello.sh
Hello, world
```

#!

```
#!/bin/sh
#!/bin/csh
#!/bin/bash
-----------------------------------------
#!/usr/bin/perl
#!/usr/bin/php
#!/bin/false
-----------------------------------------
```

به `parent shell` اطلاع میدهد که چه مفسری برای اجرای این `script` نیاز است.

در همه ی زبان های اسکریپت به معنای کامنت است و یک کامنت توسط مفسر معنی نخواهد شد.

Reminder (Chmod)

4 read (r)
2 write (w)
1 execute (x)

7 = 4+2+1 (read/write/execute)
6 = 4+2 (read/write)
5 = 4+1 (read/execute)
4 = 4 (read)
3 = 2+1 (write/execute)
2 = 2 (write)
1 = 1 (execute)

Reminder (Chmod)

- chmod g+w file_name
- chmod a-w file_name
- chmod ug=rx file_name
- chmod 664 file_name
- chmod +x file_name

Reminder (Chmod)

command	explanation
`chmod a+r publicComments.txt`	read is added for all classes (i.e. User, Group and Others).
`chmod +r publicComments.txt`	omitting the class defaults to all classes, but the resultant permissions are dependent on umask
`chmod a-x publicComments.txt`	execute permission is removed for all classes.
`chmod a+rx viewer.sh`	add read and execute for all classes.
`chmod u=rw,g=r,o= internalPlan.txt`	user(i.e. owner) can read and write, group can read, Others cannot access.
`chmod -R u+w,go-w docs`	add write permissions to the directory *docs* and all its contents (i.e. **R**ecursively) for user and deny write access for everybody else.
`chmod ug=rw groupAgreements.txt`	User and Group members can read and write (update the file).
`chmod 664 global.txt`	sets read and write and no execution access for the user and group, and read, no write, no execute for all others.
`chmod 0744 myCV.txt`	equivalent to `u=rwx (400+200+100),go=r (40+ 4)`. The `0` specifies *no special modes*.
`chmod 1755 findReslts.sh`	the `1000` specifies set sticky bit and the rest is equivalent to `u=rwx (400+200+100),go=rx (40+10 + 4+1)` This suggests that the script be retained in memory.
`chmod 4755 SetCtrls.sh`	the `4` specifies *set user ID* and the rest is equivalent to `u=rwx (400+200+100),go=rx (40+10 + 4+1)`.
`chmod 2755 SetCtrls.sh`	the `2` specifies *set group ID* and the rest is equivalent to `u=rwx (400+200+100),go=rx (40+10 + 4+1)`.
`chmod -R u+rwX,g-rwx,o-rx PersonalStuff`	**R**ecursively set a directory tree to `rwx` for owner directories, `rw` for owner files, `---` (i.e. no access) for group and others.
`chmod -R a-x+X publicDocs`	remove the execute permission on all files in a directory tree (i.e. **R**ecursively), while allowing for directory browsing.

Shell variables & positional parameters & environment variables

Shell Variables

- Environmental variables are used to provide information to the programs you use. You can have both **global environment** and **local shell** variables.

- Global environment variables are set by your login shell and new programs and shells inherit the environment of their parent shell.
- Local shell variables are used only by that shell and are not passed on to other processes. A child process cannot pass a variable back to its parent process.

Some global environment variables are,

HOME	Path to your home directory
HOST	The hostname of your system
LOGNAME	name you login with
PATH	Paths to be searched for commands
SHELL	The login shell you're using
PWD	Present working directory

Useful Environment Variables

HOME	the home directory of the current user
HOST	the current hostname
LANG	when a tool is localized, it uses the language from this environment variable. English can also be set to C
PATH	the search path of the shell, a list of directories separated by colon
PS1	specifies the normal prompt printed before each command
PS2	specifies the secondary prompt printed when you execute a multi-line command
PWD	current working directory
USER	the current user

- To see the value of a variable, insert the name of your variable as an argument: printenv PATH
- A variable, be it global or local, can also be viewed with **echo**: echo $PATH
- To set a local variable, use a variable name followed by the equal sign, followed by the value: PROJECT="SLED“
- To set an environment variable, use **export**: export NAME="tux“
- To remove a variable, use **unset**: unset NAME

Positional Parameters

- The command name and arguments are the positional parameters.
  - Because you can reference them by their position on the command line
  - $0 : Name of the calling program
  - $1 - $9 : Command-line Arguments
    - The first argument is represented by $1
    - The second argument is represented by $2
    - And so on up to $9
    - The rest of arguments have to be shifted to be able to use $1- $9 parameters.

Positional Parameters

- $1-$9 allows you to access 10 arguments
  - How to access others?
- Promote command-line arguments: shift
  - Built-in command shift promotes each of the command-line arguments.
    - The first argument (which was $1) is discarded
    - The second argument (which was $2) becomes $1
    - The third becomes the second
    - And so on
  - Makes additional arguments available
  - Repeatedly using shift is a convenient way to loop over all the command-line arguments

Positional Parameters

- Example:

$ more demo_shift

#!/bin/tcsh

echo $1 $2 $3

shift

echo $1 $2

shift

echo $1

$ ./demo_shift 1 2 3

1 2 3

2 3

3

Using Argument Variables

- foo.sh "Tux Penguin" 2000
- #!/bin/sh

echo \"$1\" \"$2\" \"$3\" \"$4\„

- "Tux Penguin" "2000" "" ""

Special Parameters

- The number of arguments: $#
  - Return a decimal number
  - Use the test to perform logical test on this number
- Exit status: $?
  - When a process stops executing for any reason, it returns an exit status to its parent process.
  - By convention,
    - Nonzero represents a false value that the command failed.
    - A zero value is true and means that the command was successful
- Value of Command-line arguments: $* and $@
  - $* and $@represent all the command_line arguments (not just the first nine)
  - "$*" : treats the entire list of arguments as a single argument
  - "$@" : produce a list of separate arguments (Only bash/ksh/sh)

Environment Variables

Environment Variable	Description
$HOME	The home directory of the current user.
$PATH	A colon-separated list of directories to search for commands.
$IFS	An input field separator
$0	The name of the shell script.
$#	The number of parameters passed.
$$	The process ID of the shell script, often used inside a script for generating unique temporary filenames; for example /tmp/tmp- file_$$.

Parameter Variables

Parameter Variables	Description
$1, $2, . . .	The parameters given to the script.
$*	A list of all the parameters, in a single variable, separated by the first character in the environment variable IFS

Programming in shell & Control statements

Condition

Boolean test command is `[` or `test`.

```
#!  /bin/sh

if [ $1 = $2 ]
then
 echo "same" else
 echo "different"
fi
```

Test Parameters

String Comparison	Result
string1 = string2	True if the strings are equal.
string1 != string2	True if the strings are not equal.
-n string True	if the string is not null.
-z string True	if the string is null (an empty string).

Test Param, Continue

Arithmetic Comparison	Result
exp1 -eq exp2	True if the expressions are equal.
exp1 -ne exp2	True if the expressions are not equal.
exp1 -gt exp2	True if expression1 is greater than expression2.
exp1 -ge exp2	True if expression1 is greater than or equal to expression2.
exp1 -lt exp2	True if expression1 is less than expression2.
exp1 -le exp2	True if expression1 is less than or equal to expression2.
! exp	True if the expression is false, and vice versa.

Test Param, Continue

File Conditional	Result
-d file	True if the file is a directory.
-e file	True if the file exists. Note that, historically, the -e option has not been portable, so -f is usually used.
-f file	True if the file is a regular file.
-g file	True if set-group-id is set on file.
-r file	True if the file is readable.
-s file	True if the file has nonzero size.
-u file	True if set-user-id is set on file.
-w file	True if the file is writable.
-x file	True if the file is executable.

Control Structure

```
if condition
then
   statements
else
   statements
fi
```

elif equal to `else if`

`for` construct

```
for variable in values
do
 statements
done
```

while construct

```
while condition do
 statements  done
```

Case

```
case variable in
 pattern [ | pattern] ...)  statements;;
 pattern [ | pattern] ...)  statements;;
 ...
esac
```

An example

```
 #!/bin/sh
echo Is it morning?  Please answer yes
or no
 read timeofday
 case "$timeofday" in
  yes) echo "Good Morning";;
  no ) echo "Good Afternoon";;
  y ) echo "Good Morning";;
  n ) echo "Good Afternoon";;
  * ) echo "Sorry, answer not
recognized";;
 esac
exit 0
```