Shell Script Programming

Glue stuffs together ^O^

ymli

Outline

- Variables and expansion
- Arithmetic and Logic expressions
- Control Structures: if-else, switch-case, for/while loops
- Input/output
- Functions
- Error Handling
- A Shell Script Sample: Failure Detection on Servers

- Appendix: Regular Expression
- Appendix B: sed and awk

Which shell?

We use Bourne Shell

```
% echo $SHELL
/usr/local/bin/tcsh
% sh
$
```

Scripting?

- shebang
  - `#!/bin/sh`
- execution
  - `chmod +x test.sh`
  - `./test.sh`

Shell variables (1)

- Assignment

	Bourne Shell	C Shell
Local variable	`my=test`	`set my=test`
Environment variable	`export my=test`	`setenv my test`

No spece in the both sides of the assignment

- Example:

```
$ export PAGER=/usr/bin/less
% setenv PAGER /usr/bin/less
```

```
$ current_month=`date +%m`
% set current_month =`date +%m`
```

Shell variables (2)

There are two ways to use a variable…

- Usage
  - % $VAR
  - % ${VAR}
  - { } to avoid ambiguity
    - % temp_name="haha"
    - % temp="hehe"
    - % echo $temp
      - hehe
    - % echo $temp_name
      - haha
    - % echo ${temp_name}
      - haha
    - % echo ${temp}_name
      - hehe_name

More clear… but don't do that

Shell variable operator (1)

value assignment

※ BadCond == !GoodCond

BadCond : var is not set or the value is null
GoodCond : var is set and is not null

operator	description
`${var:=value}`	If BadCond, use the value and assign to var
`${var:+value}`	If GoodCond, use value instead else null value is used but not assign to var
`${var:-value}`	If BadCond, use the value but not assign to var
`${var:?value}`	If BadCond, **print** value and shell exits

Print → stderr

The command stops immediately

"Parameter Expansion" in sh(1)

Shell variable operator (2)

```
#!/bin/sh

var1="haha"
echo "01" ${var1:+"hehe"}
echo "02" ${var1}
echo "03" ${var2:+"hehe"}
echo "04" ${var2}
echo "05" ${var1:="hehehe"}
echo "06" ${var1}
echo "07" ${var2:="hehehe"}
echo "08" ${var2}
echo "09" ${var1:-"he"}
echo "10" ${var1}
echo "11" ${var3:-"he"}
echo "12" ${var3}
echo "13" ${var1:?"hoho"}
echo "14" ${var1}
echo "15" ${var3:?"hoho"}
echo "16" ${var3}
```

```
01 hehe
02 haha
03
04
05 haha
06 haha
07 hehehe
08 hehehe
09 haha
10 haha
11 he
12
13 haha
14 haha
hoho
16
```

Shell variable operator (3)

operator	description
`${#var}`	String length
`${var#pattern}`	Remove the smallest prefix
`${var##pattern}`	Remove the largest prefix
`${var%pattern}`	Remove the smallest suffix
`${var%%pattern}`	Remove the largest suffix

```
#!/bin/sh

var="Nothing happened end closing end"


echo ${#var}
echo ${var#*ing}
echo ${var##*ing}
echo ${var%end*}
echo ${var%%end*}
```

These operators do not change the variable itserlf

Results:

```
32
happened end closing end
end
Nothing happened end closing
Nothing happened
```

Predefined shell variables

C program's "int main(argc, args)" – **arguments of program**

- Environment Variables: `env`
- Other useful variables:

variable	description
`$#`	Number of positional arguments
`$0`	The name of the shell script
`$1, $2, ...`	Positional arguments
`$*`	List of positional arguments (useful in for loop)
`$?`	Return code from last command
`$$`	Process number of current command (pid)
`$!`	Process number of last background command

Usage of $@

```
for i in "$@" ;  do
    echo ">> $i"
done



$ test.sh 1 2 3 4
>> 1
>> 2
>> 3
>> 4
```

test command

Checking stuffs

- test(1)
  - test, [
  - `test expression`
  - `[ expression ]` → Spaces are needed
- Test and return 0 (true) or 1 (false) in $?

```
$ test 1 -eq 1 ; echo $?
0
$ test 1 -eq 2 ; echo $?
1

$ [ 1 -eq 1 ] ; echo $?
0
$ [ 1 -eq 2 ] ; echo $?
1
```

→ $? To get the return code

Details on the capability of test command – File test

man test

Details on the capability of test command – String test

- -z string
  - True if the length of string is zero
- -n string
  - True if the length of string is nonzero
- s1 = s2
  - True if the strings s1 and s2 are identical
- s1 != s2
  - True if the strings s1 and s2 are not identical

Details on the capability of test command – Number test

- n1 -eq n2 equal
- n1 -ne n2 not equal
- n1 -gt n2 greater than
- n1 -ge n2 greater than or equal
- n1 -lt n2 less than
- n1 -le n2 less than or equal

test command – logic operations

- ! expression
  - True if expression is false.
- expression1 -a expression2
  - True if both expression1 and expression2 are true.
- expression1 -o expression2
  - True if either expression1 or expression2 are true.
  - The -a operator has higher precedence than the -o operator.
- (expression)
  - True if expression is true

test command – in script

- test command -> short format using `[ ]`
  - `$ test "haha" = "hehe" ; echo $?`

```
if [ "haha" = "hehe" ] ; then
        echo "haha equals hehe"
else
        echo "haha doesn't equal hehe"
fi
```

test command – in script

```
# AND - OR - NOT


$ [ 1 -eq 1 ] || [ 1 -eq 2 ] ; echo $?
0

$ [ 1 -eq 1 ] && [ 1 -eq 2 ] ; echo $?
1

$ ! [ 1 -eq 2 ] ; echo $?
0

$ [ 1 -eq 2 ] ; echo $?
1
```

Arithmetic Expansion

```
echo $(( 1 + 2 ))


a=5566
echo $(( $a + 2 ))
echo $(( $a - 2 ))
echo $(( $a * 2 ))
echo $(( $a / 2 ))
```

Result:

```
3
5568
5564
11132
2783
```

if-then-else structure

```
if [ test conditions ] ; then
    command-list
elif [ test conditions ] ; then
    command-list
else
    command-list
fi
```

```
#!/bin/sh

a=10
b=12

if [ $a -ne $b ] ; then
    echo "$a not equal $b"
fi
```

switch-case structure

```
case $var in
    pattern)
        action
        ;;
    value1|value2)
        action
        ;;
    *)
        default-action
        ;;
esac
```

```
case $# in
    0)
        echo "Enter file name:"
        read argument1
        ;;
    1)
        argument1=$1
        ;;
    *)
        echo "[Usage] comm file"
        ;;
esac
```

For loop

```
for var in list ; do
    action
done
```

```
for dir in bin doc src ; do
    cd $dir
    for file in * ; do
        echo $file
    done
    cd ..
done
```

While loop

```
while [ … ] ; do
    action
done



break
continue
```

```
month=1
while [ "$month" -le 12 ] ; do
    echo $month
    month=$(( $month + 1 ))
done
```

Read from stdin

```
#!/bin/sh

echo "Hello!"
echo "The answer to life, the universe and everything?"

read line

if [ "$line" = "42" ] ; then
    echo "Wow! You got it!"
else
    echo "wrong answer, G_G"
fi
```

Read from file

```
#!/bin/sh

while read line ; do
    echo $line
done < /etc/passwd
```

Create tmp file/dir

```
mktemp tmp.XXXXXX
mktemp -d dir.XXXXXX

$ mktemp nctu.XXXXXX
nctu.Gj0Ljr

$ mktemp -d dir.XXXXXX
dir.GHwNa8
```

functions (1) - syntax

```
# define function
func () {
    command-list
}

# function call
func

# remove function
unset func
```

- Function definition is local to the current shell

functions (2) - scoping

```
func () {
    # "global" variable
    echo $a
    a="der di yi"
}

a="5566"

func
echo $a
```

Output:

```
5566
der di yi
```

```
func () {
    # "local" variable
    local a="7788"
    echo $a
    a="der di yi"
}

a="5566"

func
echo $a
```

Output:

```
7788
5566
```

functions (3) - arguments check

```
#!/bin/sh
func () {
    if [ $# -eq 2 ] ; then
        local group=$1
        local desc=$2
        echo "$group is $desc"
    else
        echo "wrong args"
    fi
}


func 5566 "gg"
func 5566 "gg" 123
func 5566
func
```

```
5566 is gg
wrong args
wrong args
wrong args
```

functions (4) - return value

```
#!/bin/sh

func () {
    if [ $1 -eq 1 ] ; then
        return 1
    else
        return 2
    fi
}

func 1
echo $?       # 1

func 2
echo $?       # 2
```

Parsing arguments

- Use getopts (recommended)

```
#!/bin/sh

while getopts abcf: op ; do
    echo "${OPTIND}-th arg"

    case $op in
        a|b|c)
            echo "one of ABC"
            ;;
        f)
            echo  $OPTARG
            ;;
        *)
            echo "Default"
            ;;
    esac
done
```

```
$ ./test.sh -a -b -c -f gg
2-th arg
one of ABC
3-th arg
one of ABC
4-th arg
one of ABC
6-th arg
gg
```

- $OPTARG: content of arguments
- $OPTIND: the index of the arguments

Handling Error Conditions

- Internal error ← program crash
  - Caused by some command's failing to perform
    - User-error
      - Invalid input
      - Unmatched shell-script usage
    - Command failure
- External error ← signals from OS
  - The system tells you that some system-level event has occurred by sending signal

Handling Error Conditions – Internal Error

- Ex:

```
#!/bin/sh

help () {
    echo "Usage: $0 -c [ -f flag ]"
    exit 1
}

has_c=""
flag=""
invalid=""

while getopts cf: op ; do
    case $op in
        c)
            has_c="1"
            ;;
        f)
            flag=$OPTARG
            ;;
        *)
            invalid="1"
            ;;
    esac
don

if [ -z $has_c ] ; then
    echo "No c!"
    help
fi

if [ ! -z $flag ] && [ $flag != "correct" ]; then
    echo "Error flag!"
    help
fi
```

Handling Error Conditions – External Error (1)

- Using trap in Bourne shell
  - trap [command-list] [signal-list]
    - Perform command-list when receiving any signal in signal-list

```
trap “rm tmp*; exit0” 1 2 3 14 15
```

```
trap "" 1 2 3
```

Handling Error Conditions – External Error (2)

signal(3)

#	Name	Description	Default	Catch	Block	Dump core
1	SIGHUP	Hangup	Terminate	☑	☑	⊘
2	SIGINT	Interrupt (^C)	Terminate	☑	☑	⊘
3	SIGQUIT	Quit	Terminate	☑	☑	☑
9	SIGKILL	Kill	Terminate	⊘	⊘	⊘
10	SIGBUS	Bus error	Terminate	☑	☑	☑
11	SIGSEGV	Segmentation fault	Terminate	☑	☑	☑
15	SIGTERM	Soft. termination	Terminate	☑	☑	⊘
17	SIGSTOP	Stop	Stop	⊘	⊘	⊘
18	SIGTSTP	Stop from tty (^Z)	Stop	☑	☑	⊘
19	SIGCONT	Continue after stop	Ignore	☑	⊘	⊘

Debugging Shell Script

Debug msgs.
print out the
expansion results

❑ Ex:

Debug mode

```
#!/bin/sh -x

var1="haha"
echo "01" ${var1:+"hehe"}
echo "02" ${var1}
echo "03" ${var2:+"hehe"}
echo "04" ${var2}
echo "05" ${var1:="hehehe"}
echo "06" ${var1}
echo "07" ${var2:="hehehe"}
echo "08" ${var2}
echo "09" ${var1:-"he"}
echo "10" ${var1}
```

❑ Result:

```
+ var1=haha
+ echo 01 hehe
01 hehe
+ echo 02 haha
02 haha
+ echo 03
03
+ echo 04
04
+ echo 05 haha
05 haha
+ echo 06 haha
06 haha
+ echo 07 hehehe
07 hehehe
+ echo 08 hehehe
08 hehehe
+ echo 09 haha
09 haha
+ echo 10 haha
10 haha
```

Usefull tools

- ps (1)
- xargs (1)
- tail (1)
- sort (1)
- tail (1)
- head (1)
- tr (1)
- cut (1)

Reference

- Classic Shell Scripting
  - http://shop.oreilly.com/product/9780596005955.do
- Linux Command Line and Shell Scripting Bible
  - http://www.amazon.com/Linux-Command-Scripting-Second-Edition/dp/1118004426
- Learn the bash shell
  - http://shop.oreilly.com/product/9780596009656.do
- Bash Guide for Begginners
  - http://www.tldp.org/LDP/Bash-Beginners-Guide/Bash-Beginners-Guide.pdf
- Advanced Bash Scripting Guide
  - http://www.tldp.org/LDP/abs/abs-guide.pdf

Shell Script Examples

ping check (1)

- Ping

```
ymli@bsd1 [~]$ /sbin/ping -c 4 bsd1.cs.nctu.edu.tw
PING bsd1.cs.nctu.edu.tw (140.113.235.131): 56 data bytes
64 bytes from 140.113.235.131: icmp_seq=0 ttl=64 time=0.027 ms
64 bytes from 140.113.235.131: icmp_seq=1 ttl=64 time=0.029 ms
64 bytes from 140.113.235.131: icmp_seq=2 ttl=64 time=0.025 ms
64 bytes from 140.113.235.131: icmp_seq=3 ttl=64 time=0.027 ms

--- bsd1.cs.nctu.edu.tw ping statistics ---
4 packets transmitted, 4 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 0.025/0.027/0.029/0.001 ms
```

ping check (2)

```
#!/bin/sh
# [Usage] isAlive.sh bsd1

PING=`which ping`
GREP=`which grep`
RM=`which rm`
MAIL=`which mail`

Usage="[Usage] $0 host"
temp="$1.ping"
Admin="liuyh"
count="4"
```

ping check (3)

```
if [ $# != 1 ] ; then
    echo $Usage
else
    $PING -c ${count} $1 | $GREP 'transmitted' > $temp
    Lost=`awk '{print $7}' $temp | awk -F"%" '{print $1}'`

    if [ ${Lost:=0} -ge 50 ] ; then
        MAIL -s "$1 failed" $Admin < $temp
    fi
    $RM $temp
fi
```

Appendix A: Regular Expression

pattern matching

Regular Expression (1)

- Informal definition
  - Basis:
    - A single character "a" is a regex
  - Hypothesis
    - If r and s are regex
  - Inductive
    - Union
    - Concatenation
    - Kleene closure

```
data stream -> regex -> matched
                 |
                 V
              filtered
```

Regular Expression (2)

- Utilities using RE
  - grep
  - awk
  - sed
  - find
  - … etc
- Different tools, different RE
  - BRE (Basic)
  - ERE (Extended)
  - PCRE (Perl Compatible)
- https://en.wikipedia.org/wiki/Regular_expression

Regular Expression (3)

- Union: `A|B`
- Concatenation: `AB`
- Kleene closure: `A*`

Regular Expression (4)

- Pattern-matching
  - special operators

operator	Description
.	A single character (usually except newline)
[]	Any character in [ ]
[^]	Any character not in []
^	Start of a line
$	End of a line
*	Zero or more
?	Zero or once
+	Ocne or more
{m,n}	At least m times and at most n times
{m,}	At least m times.
{m}	Exact m times.
\	Escape

Regular Expression (5)

- Pure text string
  - `asdf apple foobar`
- Anchor
  - `^` and `$`
- Dot (placeholder)
  - `.`
- character class
  - `[abcde], [^xyz]`
- special
  - `[[:alpha:]]`
  - `[[:alnum:]]`
  - `[[:blank:]]`
  - `[[:digit:]]`
  - `[[:lower:]]`
  - `[[:print:]]`
  - `[[:punct:]]`
  - `[[:space:]]`
  - `[[:upper:]]`

Regular Expression (5)

- Repeat
  - `* + ?`
- times
  - `{m}`
  - `{m, n}`
  - `{m,}`
  - `{,n}`
- grouping
  - `()`

Regular Expression (5)

- http://regexcrossword.com/
- Finish at least intermediate level

Appendix B: sed and awk

sed - Stream EDitor

- sed (1)
  - `[address[,address]]function[arguments]`
  - `sed -e 'command' -e 'command' file`
  - `sed -f script_file`
- address
  - line number (range) or $ (last line)
  - RE

- Examples
  - `sed -e '10d'`
  - `sed -e '/man/d'`
  - `sed -e '10,100d'`
  - `sed -e '10,/man/d'`
    - Delete line from line 10 to the line contain "man"

sed - Stream EDitor

- print
  - Write the pattern space to STDOUT
  - `[2addr]p`
  - `sed -ne '1,10p'`
  - `sed -ne '/if/p'`

-n: By default, each line of input is echoed to the standard output after all of the commands have been applied to it. The -n option suppresses this behavior.

sed - Stream EDitor

- delete
  - Delete the pattern space
  - `[2addr]d`
  - `sed -e '1,50d'`
  - `sed -e '/for/d'`
  - `sed -e '10,$d'`
    - `head -n 10`

sed - Stream EDitor

- translate
  - Replace all occurrences of characters in string1 in the pattern space with the corresponding characters from string2
  - `[2addr]y/string1/string2/`
  - `sed –e 'y/abc/def/'`
    - `tr 'abc' 'def'`

sed - Stream EDitor

- substitution
  - Substitute the replacement string for the first instance of the regular expression
  - `[2addr]s/RE/replacement/flags`
  - `sed -e 's/a*c/gg/'`
  - `sed -e 's/\(a*\)\(c*\)/\2\1/'`

sed - Stream EDitor

- substitution
  - Flags
    - `N: Make the substitution only for the N'th occurrence of the RE`
    - `g: replace all matches`
    - `p: print the matched and replaced line`
    - `w: write the matched and replaced line to a file`
    - `I: Match the regular expression in a case-insensitive way`

sed - Stream EDitor

- Ex:
  - `sed -e 's/liuyh/LIUYH/2' file`
  - `sed -e 's/liuyh/LIUYH/g' file`
  - `sed -e 's/liuyh/LIUYH/p' file`
  - `sed -n -e 's/liuyh/LIUYH/p' file`
  - `sed -e 's/liuyh/LIUYH/w wfile' file`

file
I am jon
I am john
I am liuyh
I am liuyh
I am nothing

sed – fun

https://github.com/bolknote/SedChess

awk

- pattern-directed scanning and processing language
- A. Aho, B. W. Kernighan, P. J. Weinberger, The AWK Programming Language, Addison-Wesley, 1988. ISBN 0-201-07981-X

awk

- awk (1)
  - `awk [ -F fs ] [ -v var=value ] [ 'prog' | -f progfile ] [ file ... ]`
- Awk scans each input file for lines that match any of a set of patterns specified literally in prog or in one or more files specified as -f progfile.
- With each pattern there can be an associated action that will be performed when a line of a file matches the pattern.

awk

- An input line is normally made up of fields separated by whitespace, or by regular expression FS.
- The fields are denoted $1, $2, ..., while $0 refers to the entire line. If FS is null, the input line is split into one field per character.

awk

- Print first two fields (exchange)
  - `{ print $2, $1 }`
- Program structure
  - `pattern { action }`
  - A missing `{ action }` means print the line
  - a missing pattern always matches

awk

- pattern formats
  - regex

```
awk '/[0-9]+/ { print "This is an integer" }'
awk '/[A-Za-z]+/ { print "This is a string" }'
awk '/^$/ { print "this is a blank line." }'
```

  - BEGIN
    - before the first input line is read

```
awk 'BEGIN { print "Nice to meet you" }'
```

  - END
    - after the last line is read

```
awk 'END { print "Bye Bye" }'
```

awk

- action formats
- An action is a sequence of statements

```
if( expression ) statement [ else statement ]
while( expression ) statement
for( expression ; expression ; expression ) statement
for( var in array ) statement
do statement while( expression )
break
continue
{ [ statement ... ] }
expression                          # commonly var = expression
print [ expression-list ] [ > expression ]
printf format [ , expression-list ] [ > expression ]
return [ expression ]
next                                # skip remaining patterns on this input line
nextfile                            # skip rest of this file, open next, start at top
delete array[ expression ]          # delete an array element
delete array                        # delete all elements of array
exit [ expression ]                 # exit immediately; status is expression
```

awk

```
BEGIN {
    name = "Doraemon"                          # typeless variables
    height = 129.3
    weight = 129.3

    print "Hello, I'm " name
    print "H: " height
    print "W: " weight
    print "BMI: " weight / (height*0.01)**2  # floating point

    for (i = 1; i < 100; i++)
        if ( i ~ 3 )                           # the 'match' operator
            print i " Threeeee!!!!!"

    while ( i --> 0 ) {                        # secret goes-to operator!
        if ( i !~ 2 && i !~ 3 )
            c[i] = i * 10
    }

    for (r in c)                               # associate array
        print r " " c[r]

}
```

awk

- `$0, $1, $2` (column variables)
- `NF` (number of fields)
- `NR` (line number)
- `FILENAME` (name of the input file)
- `FS` (field separator, set by `-F`)
- `OFS` (Output field separator)

awk

- Functions may be defined (at the position of a pattern-action statement) thus:
- `function foo(a, b, c) { ...; return x }`
- Parameters are passed by value if scalar and by reference if array name; functions may be called recursively.
- Parameters are local to the function; all other variables are global.
- Thus local variables may be created by providing excess parameters in the function definition.

Reference

- awk (1)
- sed (1)

- http://shop.oreilly.com/product/9781565922259.do
- http://www.amazon.com/Programming-Pearls-2nd-Edition-Bentley/dp/0201657880
- http://www.amazon.com/More-Programming-Pearls-Confessions-Coder/dp/0201118890

- http://www.grymoire.com/Unix/Awk.html
- http://www.grymoire.com/Unix/Sed.html
- http://www.vectorsite.net/tsawk.html
- http://www.staff.science.uu.nl/~oostr102/docs/nawk/nawk_toc.html
- https://www.gnu.org/software/sed/manual/sed.html
- https://www.gnu.org/software/gawk/manual/gawk.html