sourcemind

Scripting

Artur Davtyan
DevOps Engineer
Webb Fontaine Armenia

sourcemind

Command Substitutions

We're gonna be talking about substitutions, specifically command substitutions. You can use command substitution to populate a variable. For example, we could say TIMEDATE equals quotes dollar parenthesis date plus, and then whatever date syntax we want.

```
adavtyan@artur-lpt:~$ echo $TIMEDATE

adavtyan@artur-lpt:~$ TIMEDATE="$(date +"%x %r %Z")"
adavtyan@artur-lpt:~$ echo $TIMEDATE
07/11/21 09:38:42 +04
adavtyan@artur-lpt:~$ date
Կրկ 11 Հլս 2021 09:38:55  +04
adavtyan@artur-lpt:~$
```

sourcemind

Command Substitutions

Practical Work

Approximate Duration: 5 minutes

Objectives:

- *Working with command Substitutions*

sourcemind

Process Substitutions

So, first of all, everybody is aware of what redirects do. So, if I want to do echo, echo test redirect that to a file named test, everybody understands what that's actually doing.

Process substitution is kind of another method of redirection. If I wanted to diff the contents of the /temp directory and the /bin directory, for example, I could do ls /tmp redirect that to tmpdir, ls /bin redirect that to bindir and then diff tmpdir bindir. And it gives me a lot of output. Wouldn't it be easier, though, to diff ls tmpand ls bin. We get the same output with a lot less work in the terminal.

```
adavtyan@artur-lpt:~$ echo test > test
adavtyan@artur-lpt:~$
adavtyan@artur-lpt:~$ ls /tmp > tmpdir
adavtyan@artur-lpt:~$ ls /bin > bindir
adavtyan@artur-lpt:~$ diff tmpdir bindir
...
adavtyan@artur-lpt:~$ diff <(ls /tmp) <(ls /bin)
adavtyan@artur-lpt:~$
```

sourcemind

For Loop

Bash for loops are relatively simple at their core.

```
#!/bin/bash

for i in $( ls ); do
     echo item; $i
done
```

Example:

```
adavtyan@artur-lpt:~$ for i in $( ls ); do
> echo item: $i
> done
item: Documents
item: Downloads
adavtyan@artur-lpt:~$ for i in $(seq 1 3); do
> echo $i
> done
1
2
3
```

sourcemind

For Loop

Example:

```
adavtyan@artur-lpt:~$ mkdir work
adavtyan@artur-lpt:~/work$  cd work
adavtyan@artur-lpt:~/work$ for i in $(seq 1 10); do  touch file$i; done
adavtyan@artur-lpt:~/work$ ls
file1  file10  file2  file3  file4  file5  file6  file7  file8  file9
adavtyan@artur-lpt:~/work$
```

Practical Work

Approximate Duration: 10 minutes

Objectives:

- *backup files use for loop*

sourcemind

While or Until Loop

The other type of loop, a while loop, operates on a list of unknown size. Its job is to keep running and on each iteration perform a test to see if it should run another time. You can think of it as “while some condition is true, do stuff.”

```
while CONTROL-COMMAND; do
    LOOP COMMANDS
done
```

```
until TEST-COMMAND; do
    LOOP COMMANDS
done
```

```
#!/bin/bash

COUNTER=0

While [ $COUNTER -lt 10 ]; do
     echo The counter is $COUNTER
     let COUNTER=COUNTER+1
done
```

```
#!/bin/bash

COUNTER=20

until [ $COUNTER -lt 10 ]; do
     echo The counter is $COUNTER
     let COUNTER-=1
done
```

sourcemind

While Loop

Example:

```
adavtyan@artur-lpt:~/work$ cat > whiletest.sh
#!/bin/bash
COUNTER=0
while [ $COUNTER -lt 10 ]; do
     touch file$COUNTER
     let COUNTER=COUNTER+1
done
^C
adavtyan@artur-lpt:~/work$ chmod +x whiletest.sh
adavtyan@artur-lpt:~/work$./whiletest.sh
adavtyan@artur-lpt:~/work$ ls
file0  file1  file2  file3  file4  file5  file6  file7  file8  file9  whiletest.sh
adavtyan@artur-lpt:~/work$
```

sourcemind

Until Loop

Example:

```
adavtyan@artur-lpt:~/work$ cat > untiltest.sh
#!/bin/bash
COUNTER=20
until [ $COUNTER -lt 10 ]; do
     touch file$COUNTER
     let COUNTER-=1
done
^C
adavtyan@artur-lpt:~/work$ chmod +x untiltest.sh
adavtyan@artur-lpt:~/work$./untiltest.sh
adavtyan@artur-lpt:~/work$ ls
file0  file1  file10  file11  file12  file13  file14  file15  file16  file17  file18
file19  file2  file20  file3  file4  file5  file6  file7  file8  file9  untiltest.sh
whiletest.sh
adavtyan@artur-lpt:~/work$
```

sourcemind

What are Signals?

Programs in Linux are managed partially by **signals** from the kernel:

- SIGKILL
- SIGINT [Interapt]
- SIGTERM [Terminate]
- ....

List of all Signals run this command in the terminal

```
adavtyan@artur-lpt:~$ trap -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
adavtyan@artur-lpt:~$
```

sourcemind

What are Signals?

Practical Work

Approximate Duration: 15 minutes

Objectives:

- *backup files use for loop*

```
#!/bin/bash

ctrlc=0

function trap_ctrlc {
    let ctrlc++
    if [[ $ctrlc == 1 ]]; then
        echo "Stop doing that."
    elif [[ $ctrlc == 2 ]]; then
        echo "I warned you..."
    else
        echo "Throwing in the towel."
        exit
    fi
}

trap trap_ctrlc SIGINT

while true
do
    echo Sleeping...
    sleep 10
done
```

sourcemind

If Conditional

It is time to make your script do different functions based on tests, called branching. The if statement is the basic operator to implement branching.

A basic if statement looks like this:

```
#!/bin/bash

if (list of commands)
then
     command1
else
     command2
fi
```

[List of Commands]: The **if** statement will act on the exit status of the list of commands
[Then of Commands]: the **then** statement will run if the **if** statement returns true (in this case, an exit status of zero).
[Else Commands]: The **else** statement will run if the **if** statement returns false (in this case, an exit status of non-zero).

Example: {Testing with Square Brackets}

```
#!/bin/bash

if [ $VAR1 -eq $VAR2 ]
then
     command1
else
     command2
fi
```

[List of Commands]: The if statement will act on the evaluation of the comparison given.
[Then of Commands]: The then statement will run if the if statement returns true (in the case, if the variables are equal)
[Else Commands]: The else statement will run if the if statement returns false (in this case, if the variables are not equal)

sourcemind

Test Commands

The test command gives you easy access to comparison and file test operators. For example:

Command	Description
test -f /dev/ttyS0	0 if the file exists
test ! -f /dev/ttyS0	0 if the file doesn't exists
test -d /tmp	0 if the directory exists
test -x `which ls`	substitute the location of ***ls*** then **test** if the user can execute
test 1 -eq 1	0 if the numeric comparison succeeds
test ! 1 -eq 1	NOT - 0 if comparison fails
test 1 -ne 1	Easier, **test** for numeric inequality
test "a" = "a"	0 if the string comparison succeeds
test "a" != "a"	0 if the string are different
test 1 -eq 1 -o 2 -eq 2	**-o** is OR: either can be the same
test 1 -eq 1 -a 2 -eq 2	**-a** is AND: both must be the same

Note:It is important to note that test looks at integer and string comparisons differently. 01 and 1 are the same by numeric comparison, but not by string comparison. You must always be careful to remember what kind of input you expect.

sourcemind

Test Commands

There are many more tests, such as –gt for greater than, ways to test if one file is newer than the other, and many more. Consult the test man page for more.

test is fairly verbose for a command that gets used so frequently, so there is an alias for it called [(left square bracket). If you enclose your conditions in square brackets, it's the same as running test. So, these statements are identical.

```
if test -f /tmp/foo; then
```

```
if [ -f /tmp/foo ]; then
```

While the latter form is most often used, it is important to understand that the square bracket is a command on its own that operates similarly to test except that it requires the closing square bracket.

The if statement has a final form that lets you do multiple comparisons at one time using elif (short for else if).

```
adavtyan@artur-lpt:~$ cat test.sh
#!/bin/bash
if [ "$1" = "hello" ]; then
  echo "hello yourself"
elif [ "$1" = "goodbye" ]; then
  echo "nice to have met you"
  echo "I hope to see you again"
else
  echo "I didn't understand that"
 fi
```

sourcemind

Builtins

Builtin commands contained within the bash shell itself. How do I list all built-in bash commands on Linux like operating systems without reading large size bash man page?

A shell builtin is nothing but command or a function, called from a shell, that is executed directly in the shell itself. The bash shell executes the command directly, without invoking another program. You can view information for Bash built-ins with help command. There are different types of built-in commands.

built-in command types

1. Bourne Shell Builtins: Builtin commands inherited from the Bourne Shell.
2. Bash Builtins: Table of builtins specific to Bash.
3. Modifying Shell Behavior: Builtins to modify shell attributes and optional behavior.
4. Special Builtins: Internal commands classified specially by POSIX.

sourcemind

Builtins

Type the following command to see all bash builtins:

```
adavtyan@artur-lpt:~$ help
adavtyan@artur-lpt:~$ help | less
adavtyan@artur-lpt:~$ help | grep read
```

Another option is to use the NA command:

```
adavtyan@artur-lpt:~$ compgen -b
adavtyan@artur-lpt:~$ compgen -b | more
```

Viewing information for Bash built-ins: To get detailed info run:

```
adavtyan@artur-lpt:~$ help command
adavtyan@artur-lpt:~$ help read
```

To just get a list of all built-ins with a short description, execute:

```
adavtyan@artur-lpt:~$ help -d
```

Note: Even BUILTINS HAVE MAN PAGES: if you're curios, man BUILTIN will bring up the builtin man page and give you more information about any specific command

sourcemind

Bash read command

The read command is just as important as positional parameters and the echo command. How else are you going to catch user input, accept passwords, write functions, loop, and peek into file descriptors? Read on.

What is read?

Read is a bash builtin command that reads the contents of a line into a variable. It allows for word splitting that is tied to the special shell variable IFS. It is primarily used for catching user input but can be used to implement functions taking input from standard input.

Bash read builtin command help

Before we dive into how to use the read command in bash scripts, here is how we get help. There you should see all the options available for the read command along with descriptions that we will try to cover in the example

```
adavtyan@artur-lpt:~$ help read
```

Catching user input

Interactive bash scripts are nothing without catching user input. The read builtin provides methods that user input may be caught within a bash script.

Catching a line of input

To catch a line of input NAMEs and options are not required by read. When NAME is not specified, a variable named REPLY is used to store user input.

sourcemind

Bash read command

Commands

```
{
echo -n "Type something and press enter: ";
read;
echo You typed ${REPLY}
}
```

Output

```
Type something and press enter: something(newline)
You typed something
```

Catching a word of input

To catch a word of input, the -d option is required. In the case of a word we would set -d to a space, read '-d '. That is when the user presses the spacebar read will load REPLY with the word.

Note that when the -d option is set, the backspace does not work as expected. To backspace, while trying to catch a word of input, the -e option may be used, read -e '-d '.

Output

```
{
echo -n "Type something and hit space: ";
read '-d ';
echo "";
echo "You typed ${REPLY}"
}
```

```
Type something and hit space: something(space)
You typed something
```

sourcemind

Bash read command

Prompt user

In interactive bash scripts prompting a user may require a message to tell the user what input is expected. We can always accomplish this using the echo builtin. However, it turns out there is an option using read.

Prompt user for a word

In catching a word of input, we used echo to write Type something and hit space: to standard output before read '-d '. The -p option allows a message to be displayed before reading from standard input.

Commands

```
{
read -p 'Type something and hit space: ' '-d ';
echo "";
echo "You typed ${REPLY}"
}
```

Output

```
Type something and hit space: something(space)
You typed something
```

sourcemind

Bash read command

Prompt user for a secret

When catching user input without it showing up in the terminal, the -s option comes in handy. read -s -p allows you to catch and hide user input as follows.

```
{
read -s -p 'Type something I promise to keep it a
secret: '
echo "";
echo "Your secret is safe with me" ; unset REPLY ;
echo "${REPLY}"
}
```

Output

```
Type something I promise to keep it a secret:
Your secret is safe with me
```

Using read command with -r option to read file lines. example

```
#!/bin/bash
input_file=$1
i=0
while read -r line; do
      let i=i+1
      echo “$line $i”
done < $input_file
```

sourcemind

Thank you for your attention !

Q&A