【Linux Programming】Day4

Tags	
Date	@May 18, 2022
Summary	Shell Programming: Quoting, Variable Declaration, Environment/Parameter Variable

【Ch2】Shell Programming

2.4 Shell Syntax

2.4.1 Variables

We don't usually declare variables in the shell before using them. We create them by simply using them(for example, when we assign an initial value to them).

By default, all variables are considered and stored as strings, even when they are assigned numeric values.

Linux is a case-sensitive system, so the shell considers the variable `foo` to be different from `Foo`, and both to be different from `FOO`.

Within the shell we can access the contents of a variable by preceding its name with a `$`.

Whenever we extract the contents of a variable, we must give the variable a preceding $. When we assign a value to a variable, just use the name of the variable.

An easy way to check the contents of a variable is to `echo` it to the terminal, preceding its name with a $.

```
# No space between str and "Arthur"
$ str=Arthur
$ echo $str
Arthur
$ str = "Hello Arthur"
$ echo $str
Hello Arthur
```

Note how a string must be delimited by quote marks if it contains space. In addition, there cannot be any spaces on either side of the equals sign.

We can assign user input to a variable by using the `read` command. This takes one parameter, the name of the variable to be read into, and then waits for the user to enter some text.

The read normally completes when the user presses Enter. When reading a variable from the terminal, we don't usually need the quote marks:

```
$ read salutation
Wie Hi
$ echo $salutation
Wie Hi
```

2.4.2 Quoting

Normally, parameters in scripts are separated by whitespace characters(e.g., a space, a tab, or a newline character). If we want a parameter to contain one or more whitespace characters, we must quote the parameter.

The behavior of variables such as `$foo` inside quotes depends on the type of quotes we use.

If we enclose a $ variable expression in double quotes, then it's replaced with its value when the line is executed.

If we enclose it in single quotes, then no substitution takes place.

We can also remove the special meaning of the `$` symbol by prefacing it with a `\`.

```
#!/bin/sh

myvar="Hi there"

echo $myvar
echo "$myvar"
echo '$myvar'
echo \$myvar

echo Enter some text
read myvar

echo '$myvar' now equals $myvar
exit 0
```

The following figure shows how this script behaves:

```
$ ./variable
Hi there
Hi there
$myvar
$myvar

Enter some text
Hello World
$myvar now equals Hello World
```

2.4.3 Environment Variables

When a shell script starts, some variables are initialized from values in the environment. These are normally in all uppercase form to distinguish them from user-defined(shell) variables in scripts.

Many environment variables are listed in the manual pages, but the principal ones are listed in the following table:

Environment Variable	Description
`$HOME`	The home directory of the current user
`$PATH`	A colon-separated list of directories to search for commands
`$PS1`	A command prompt, frequently `$`, but in bash you can use some more complex values; for example, the string `[\u@\h \W]$` is a popular default that tells you the user, machine name, and current directory, as well as providing a $ prompt.
`$PS2`	A secondary prompt, used when prompting for additional input; usually >.
`$IFS`	An input field separator. This is a list of characters that are used to separate words when the shell is reading input, usually space, tab, and newline characters.
`$0`	The name of the shell script
`$#`	The number of parameters passed
`$$`	The process ID of the shell script, often used inside a script for generating unique temporary filenames; for example `/tmp/tmpfile_$$`

2.4.4 Parameter Variables

If our script is invoked with parameters, some additional variables are created.

If no parameters are passed, the environment variables `$#` still exists but has a value of 0.

The parameter variables are listed in the following table:

Parameter Variable	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`. If `IFS` is modified, then the way `$*` separates the command line into parameters will change.
`$@`	A subtle variation on `$*`; it doesn't use the `IFS` environment variable, so parameters are not run together even if `IFS` is empty.

It's easy to see the differences between $@ and $*:

```
$ IFS=''
$ set foo bar bam
$ echo "$@"
foo bar bam
$ echo "$*"
foobarbam
$ unset IFS
$ echo "$*"
foo bar bam
```