03_Variable

April 14, 2024

1 What is Vriable In SS?

In Shell Scripting, a variable is a symbol or name that stands for a value. Variables are used for storing data, controlling the flow of the script, and for performing operations.

Here's how you can define and use variables in a shell script:

[ ]:
```
#!/bin/bash
# Defining a variable
greeting="Hello, World!"

# Using the variable
echo $greeting
```

In this script, `greeting` is a variable that is assigned the string "Hello, World!". The `echo` command is then used to print the value of the variable. Note that when using the variable, you need to prefix it with a `$`.

There are two types of variables in shell scripting:

1. **User-defined variables**: These are the variables that are defined in the script by the user, like the `greeting` variable in the example above.
2. **Environment variables**: These are variables that are defined by the shell or operating system, and they provide information about the system environment. Examples include `PATH`, `HOME`, and `USER`. You can view all environment variables by typing `printenv` in the terminal.

Remember, variable names are case-sensitive in shell scripting, so `greeting`, `Greeting`, and `GREETING` would be three different variables.

2 How can I assign a value to a variable in a shell script?

In a shell script, you can assign a value to a variable using the `=` operator, with no spaces around it. Here's an example:

[ ]:
```
#!/bin/bash
# Assigning a value to a variable
my_variable="Hello, World!"
```

```
# Printing the value of the variable
echo $my_variable
```

In this script, `my_variable` is a variable that is assigned the string "Hello, World!". The `echo` command is then used to print the value of the variable. Note that when using the variable, you need to prefix it with a `$`.

Remember, there should be no spaces around the `=` operator. If you add spaces, the shell will interpret the parts as separate arguments and not as a variable assignment. For example, `my_variable = "Hello, World!"` would result in an error.

3 How can I read user input and store it in a variable in a shell script?

In a shell script, you can read user input using the `read` command and store it in a variable. Here's an example:

[ ]:
```
#!/bin/bash
# Ask the user for their name
echo "Please enter your name:"

# Read the user's input and store it in a variable
read name

# Print a greeting to the user
echo "Hello, $name!"
```

In this script, the `echo` command is used to ask the user for their name. The `read` command then waits for the user to enter their name and press `Enter`. The input provided by the user is stored in the `name` variable. The script then prints a greeting to the user using the value of the `name` variable.

4 How can I perform arithmetic operations in a shell script?

In a shell script, you can perform arithmetic operations using several methods. Here's an example using the `let` command, the `((...))` construct, and the `$((...))` construct:

[ ]:
```
#!/bin/bash

# Using the let command
let a=5+5
echo "The result of 5+5 using let is: $a"

# Using the ((...)) construct
(( b = 5 * 5 ))
echo "The result of 5*5 using ((...)) is: $b"

# Using the $((...)) construct
```

```
c=$(( 5 - 5 ))
echo "The result of 5-5 using \$((...)) is: $c"

# Using the expr command
d=$(expr 5 / 5)
echo "The result of 5/5 using expr is: $d"
```

In this script:

- The `let` command is used to perform arithmetic operations and assign the result to a variable.
- The `((...))` construct allows for arithmetic expansion. It's similar to `let` but doesn't require the `$` when referencing variables.
- The `$((...))` construct allows for arithmetic expansion and can be used anywhere in the script, not just in assignments.
- The `expr` command is an older way to do arithmetic in shell scripts. It's not as flexible as the other methods and requires spaces around operators, but it's still used in some older scripts.

Remember, shell arithmetic does integer arithmetic only, so operations like division may not give the result you expect if you're used to floating-point arithmetic. For example, `5 / 2` would result in `2`, not `2.5`. If you need floating-point arithmetic, you might need to use an external command like `bc` or `awk`, or a different scripting language like Python or Perl.

5 Special Variables

In shell scripting, there are several special variables that are set by the shell and have special meanings. Here are some of them:

1. `$0`: The name of the script itself.
2. `$1` to `$9`: The first 9 arguments to the script. For example, in a script run with `./myscript arg1 arg2 arg3`, `$1` would be `arg1`, `$2` would be `arg2`, etc.
3. `$#`: The number of arguments passed to the script.
4. `$@`: All the arguments passed to the script. This is similar to `$*`, but when quoted (`"$@"`), it preserves whitespace in arguments.
5. `$*`: All the arguments passed to the script as a single string. This is similar to `$@`, but when quoted (`"$*"`), it does not preserve whitespace in arguments.
6. `$$`: The process ID of the current shell.
7. `$?`: The exit status of the last command executed.
8. `$!`: The process ID of the last background command.

Here's an example script that uses some of these special variables:

[ ]:
```
#!/bin/bash
echo "The script name: $0"
echo "The first argument: $1"
echo "The second argument: $2"
```

```
echo "The number of arguments: $#"
echo "All arguments: $@"
echo "All arguments as a single string: $*"
echo "The process ID of the current shell: $$"
```

You can run this script with some arguments, like `./myscript arg1 arg2`, and see the output.

6 Special Parameters $* and $@

In shell scripting, `$*` and `$@` are special parameters that hold all the positional parameters of the script. They seem similar, but there's a key difference in how they behave when quoted.

1. `$*`: This special parameter expands to the positional parameters, starting from one. When the expansion occurs within double quotes, it expands to a single word with the value of each parameter separated by the first character of the `IFS` special variable, which is a space by default.
2. `$@`: This special parameter also expands to the positional parameters, starting from one. However, when the expansion occurs within double quotes, each parameter expands to a separate word. That is, `"$@"` is equivalent to `"$1" "$2" ...`.

Here's an example to illustrate the difference:

[ ]:
```
#!/bin/bash
echo "Using \$*"
for arg in "$*"
do
    echo "$arg"
done

echo "Using \$@"
for arg in "$@"
do
    echo "$arg"
done
```

If you run this script with the arguments `arg1 arg2 arg3` (i.e., `./myscript arg1 arg2 arg3`), the `"$*"` version will output a single line `arg1 arg2 arg3`, because it treats the entire set of positional parameters as a single word. The `"$@"` version will output three lines `arg1`, `arg2`, and `arg3`, because it treats each positional parameter as a separate word.

7 readonly variable?

In shell scripting, a `readonly` variable is a variable that is marked as read-only, meaning its value cannot be changed once it has been set. This can be useful when you want to ensure that a variable's value remains constant throughout the script.

Here's how you can define a `readonly` variable:

[ ]:
```
#!/bin/bash
# Define a readonly variable
readonly greeting="Hello, World!"

# Try to change the value of the readonly variable
greeting="Goodbye, World!"
```

In this script, `greeting` is a `readonly` variable that is assigned the string "Hello, World!". The script then tries to change the value of the `readonly` variable to "Goodbye, World!". However, this will result in an error, because `greeting` is a `readonly` variable and its value cannot be changed.

If you run this script, you will see an error message like this:

[ ]:
```
./myscript: line 5: greeting: readonly variable
```

This indicates that the script tried to change the value of a `readonly` variable, which is not allowed.

8 How can I unset a variable in a shell script?

In a shell script, you can unset a variable using the `unset` command. Here's an example:

[ ]:
```
#!/bin/bash
# Define a variable
greeting="Hello, World!"

# Unset the variable
unset greeting

# Try to print the value of the variable
echo $greeting
```

In this script, `greeting` is a variable that is assigned the string "Hello, World!". The `unset` command is then used to unset the `greeting` variable, which removes it from the list of defined variables. The script then tries to print the value of the `greeting` variable, but since it has been unset, nothing is printed.

Note that you cannot unset `readonly` variables. If you try to unset a `readonly` variable, you will get an error.

9 Thank You!