Shell Scripts

HUAWEI

Foreword

- This course is based on openEuler and includes:
- Basic shell knowledge
- Shell script compilation best practices based on actual cases

Objectives

- Upon completion of this course, you will be familiar with:
  - Basic shell knowledge
  - Shell programming basics
  - Compiling common shell scripts

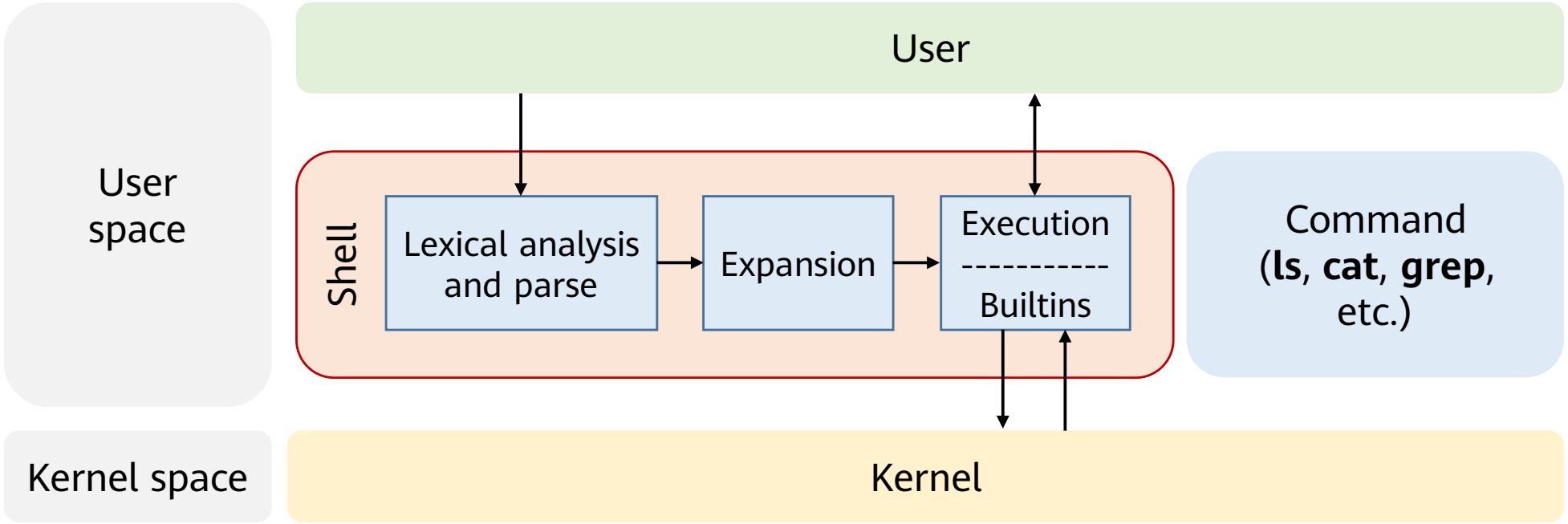
Contents

Overview and Objectives

- Know the role of shell in openEuler.
- Master basic shell script operations.

Shell Overview

- The shell is the user interface (UI) provided by the operating system for users to interact with the kernel.
- The shell sends commands entered by users to the kernel for execution, and returns the execution result.
- The shell is programmable, allowing users to write programs which include shell commands.



Shell Development History

- 1971: Ken Thompson of Bell Labs develops the first Unix shell, the V6 shell. It was an independent user program executed outside the kernel.
- 1977: Steven Bourne creates the Bourne shell for V7 in AT&T Bell Labs.
- 1978: Bill Joy develops the C shell (csh) for the BSD during his postgraduate studies at University of California, Berkeley.
- 1983: Ken Greer develops tcsh at Carnegie Mellon University by introducing some functions of the Tenex system, such as command line editing and automatic file name and command completion, to the C shell.
- 1983: David Korn creates the KornShell (ksh) at AT&T Bell Labs. It provided more powerful functions as well as associative array expression calculation.
- 1989: Brian Fox develops the Bourne again shell (Bash) for the open source GNU project, as a free software replacement for the Bourne shell. Bash has since then become the most popular shell in the world. Compatible with sh, csh, and ksh, it is the default shell of the Linux system.

In Linux, multiple shell programs are available.
For example, dash, csh, and zsh.
You can view the default shell in **/bin/sh** and modify it in **/etc/passwd**.

Viewing the Shell

- Log in to openEuler and check the default shell installed in the system.

  ```
  cat /etc/shells
  ```

- Check the default shell of the current login user.

  ```
  echo $SHELL
  ```

- View the current shell.

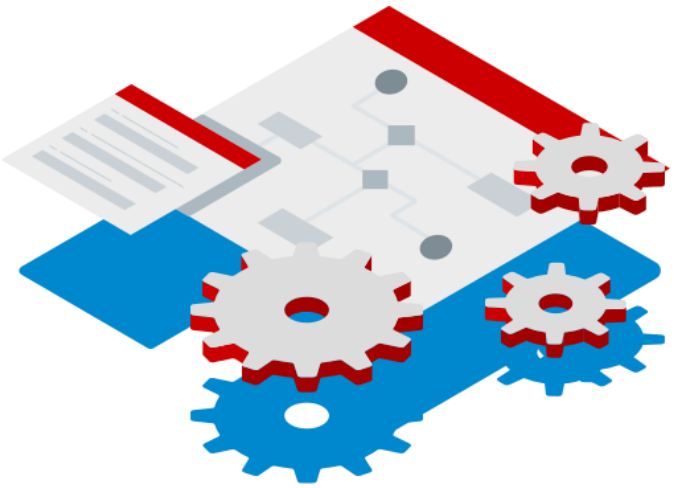
  ```
  echo $0
  ```

Contents

Basic Shell Script Knowledge

- In Unix/Linux, a program/command performs only one operation.
- Combining multiple commands can solve complex problems.
- The most simple shell script is an executable file made up of a series of commands. These commands can be reused in other scripts.

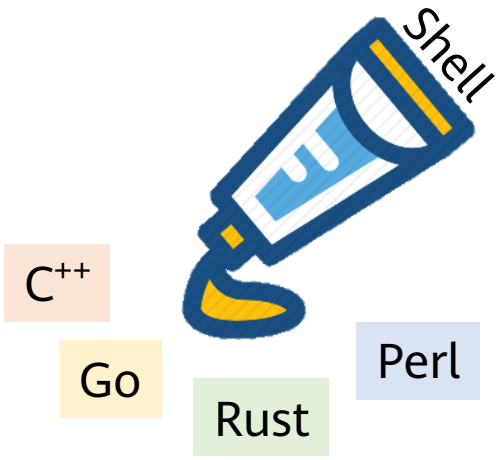
> Writing a shell script with a good style, which is easily readable, improves the automation and accuracy of routine tasks.



Shell Script Constraints

- There is no "silver bullet."
  - Shell scripts can complete many tasks, but they are not applicable to every case.
  - Shell scripts are a good choice for tasks that can be done by invoking other command-line utilities.
  - Shell scripts can be used as a glue language to integrate other programming languages.

> If a shell script solution to a problem is complex and/or inefficient, you can consider using other programming languages.



Shell Script Development Environment

- To create a shell script, you can open a new file in any text editor.
- Advanced editors such as Vim and Emacs can highlight, check, and supplement syntax after identifying .sh files.

```
[root@openEuler ~]# vim demo.sh # Create a script file and write the following content into the file:
#!/bin/bash
echo "Hello World"
[root@openEuler ~]# sh demo.sh
Hello World
```

Specifying an Interpreter for Shell Scripts

- The shell script is static code. To output the result, an interpreter is required.
- Generally, the interpreter that executes a script is specified in the first line of the script.
- If no interpreter is specified, the script can run properly in the default interpreter. For clearer specifications and better security, however, it is recommended to specify an interpreter as in the two examples below:

```
#!/bin/bash

#!/bin/csh
```

➢ What is the difference between these two paths?

Running Shell Scripts (1)

- There are two execution modes for script files:
  - **sh script_name.sh**
  - **./script_name.sh**
- In Linux, everything is a file (an object, with permission attributes). A script is a file, as is a command or a program.
- When executing scripts sent by others or downloaded from the Internet, you may encounter permission problems. To solve such problems, you can grant the execute permission.
  - **chmod +x script_name.sh**

Running Shell Scripts (2)

- If a shell script is executable, you can invoke it by entering its name in the command line.
  - To successfully invoke a script, its path must be contained in the **$PATH** variable.
    - Check the **$PATH** variable.
    - Modify the **$PATH** variable.
    - Search for the path of a command or a script.

```
[root@openEuler ~]# echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin
[root@openEuler ~]# PATH=$PATH:/New/path
[root@openEuler ~]# echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin:/New/path
```

Running Shell Scripts in the Background

- The execution of some scripts takes a long time, occupying the CLI. You can run such scripts in the background.
  - **./my_script.sh &**
- Using this way, the script process will stop after you exit the shell. To ensure that the script keeps running, run the following command:
  - **nohup ./my_script.sh &**
  - Using this command, the script's standard output and standard error streams will be redirected to the **nohup.out** file.
- To view processes running in the background, you can run the **jobs** command.

Contents

Overview and Objectives

- Understand the input, output, and redirection of text streams in Linux.
- Master the usage of pipes in Linux.
- Master shell variables and operators.
- Master shell logic judgment, selection structure, and loop structure.

Text Stream in Linux

- The text stream exists in every Linux process.
- When a Linux process is started, three text stream interfaces are enabled: standard input, standard output, and standard error.
- These three interfaces correspond to a program's input, output, and exception throwing.

```
[root@openEuler ~]# date
Wed Jul 29 10:39:17 CST 2020
[root@openEuler ~]# datee
bash: datee: command not found
```

Example:

After a string of characters is entered in bash, the standard input interface in the bash process captures the input in the command line. After being processed, the data is output from the standard output interface and displayed on the screen. If an exception occurs during the processing, the exception is displayed on the screen through the standard error interface.

Output Redirection

- Sometimes, you need to save the output of a program. You can do this by saving it to a file through redirection.
  1. Direct the standard output to a file:
     - **ls > dir_log**
     - Using this command, the standard output of the program will overwrite the file's previous contents.
  2. Append the standard output to a file:
     - **ls >> dir_log**
     - Using this command, the standard output of the program will be appended to the end of the file.
  3. After the command executes, you can run the **cat dir_log** command to view the contents of the saved file.

Input Redirection

- Input redirection, similar to output redirection, redirects a program's standard input.
- Input redirection:
  - Format: *command* < *inputfile*
  - Use the file in the gray box as the standard input, then pass the command as in the format above.
  - Example: **wc -l < /dev/null**

```
[root@openEuler ~]# less << EOF
> item 1
> item 2
> item 3
> EOF
item 1
item 2
item 3
(END)
```

- Inline input redirection:
  - Format: *command* << *maker*
  - Output redirection requires files, but inline input redirection can use instant input text as standard input.
  - *maker* on the right side of the format indicates the standard input's start and end. It is not included in the standard input.
  - In the code snippet in the gray box, **EOF** is the maker.

Pipe

- Sometimes, you need to connect the output of one command to the input of another command. This may be done with redirection, but is complex.
- A pipe (|) can connect the input and output of two commands, or even connect multiple commands in series.
- Format: *command1* | *command2* | *command3*

```
[root@openEuler ~] ls /bin/ | grep python | less
```

- **Pipes are actually a way of inter-process communication (IPC).**

Contents

Characters in the Shell

- Like other programming languages, the shell also has some special characters. When writing scripts, pay attention to these special characters.

Special Character	Description
#	Remarks
'	String reference
\	Escape
/	Path separator
!	Reverse logic

Variables

- Any language has variables.
- The shell is quite different from other strongly typed programming languages such as C, Java, and C++. Shell variables are typeless.
- A variable can be used to reference the value of a memory area. The variable name is the label attached to the memory area.
- Variable value assignment: *variable*=*value*

```
[root@openEuler ~]# a='Hello World'
[root@openEuler ~]# echo a
a
[root@openEuler ~]# echo $a
Hello World
```

Variable Types

- In the Linux shell, there are two types of variables:
  - Environment variables
  - User-defined variables
- Variables of each type are classified into either global variables or local variables according to their scope.
  - Global variables apply to the entire shell session and its subshells.
  - Local variables act in the process that defines them and its subprocesses.
- Viewing variables
  - Run the **printenv** command to view global variables.
  - Run the **set** command to view all variables in a specific process, including local variables, global variables, and user-defined variables.
- Modifying variables
  - Add the export statement to the **.bash_profile** or **.bashrc** file to permanently modify variables.

Using Variables

- Shell variable naming rules:
  - A variable name consists of digits, letters, and underscores (_).
  - It must start with a letter or underscore (_).
  - Keywords in the shell cannot be used.

```
[root@openEuler ~]# help | grep for
 break [n]                                          printf [-v var] format [arguments]
 for NAME [in WORDS ... ] ; do COMMANDS; done       unset [-f] [-v] [-n] [name ...]
 for (( exp1; exp2; exp3 )); do COMMANDS; done      until COMMANDS; do COMMANDS; done
```

- Format of a variable:
  - variable=value
  - variable='value'
  - variable="value"

➢ What is the difference between a single quotation mark (') and a double quotation mark (") in a shell script?

Expanding Variables

- In the following example, if braces are not used, bash interprets **$FIRST_$LAST** as the variable **$FIRST_** followed by the variable **$LAST** instead of the variables **$FIRST** and **$LAST** separated by an underscore (_).
- To make variable expansion work correctly, you must enclose the variable in braces.

```
[root@openEuler ~]# FIRST_=Jane
[root@openEuler ~]# FIRST=john
[root@openEuler ~]# LAST=Doe
[root@openEuler ~]# echo $FIRST_$LAST
Doe
[root@openEuler ~]# echo ${FIRST}_$LAST
john_Doe
```

Value Assignment and Variable Output

- Define and assign values to the following variables: **name** and **time**.
- Output the **name** and **time** variables.
- Read the input and output the variables.

```
name=Euler
time='2020202'
echo "My name is $name, today is $time"
read name
echo "Hello, $name, welcome!"
```

Arithmetic Extensions of the Shell

- Arithmetic extensions can be used to perform simple arithmetic operations of integers.
- Syntax: $[expression]
- Example:

```
[root@openEuler ~]# echo $[1+1]
2
[root@openEuler ~]# echo $[2*2]
4
[root@openEuler ~]# COUNT=1; echo $[$[$COUNT+1]*2]
4
```

Calculating Time in the Shell

- Define the 24-hour, 60-minute, and 60-second variables.
- Define the number of seconds per day and assign a value to the variable.
- Output the variables.

```
[root@openEuler ~]# SEC_PER_MIN=60
[root@openEuler ~]# MIN_PER_HR=60
[root@openEuler ~]# HR_PER_DAY=24
[root@openEuler ~]# SEC_PER_DAY=$[ $SEC_PER_MIN * $MIN_PER_HR * $HR_PER_DAY
]
[root@openEuler ~]# echo "There are $SEC_PER_DAY seconds in a day"
There are 86400 seconds in a day
```

Common Expressions for Arithmetic Operations

Operator	Function
<VARIABLE>++	Post-increment
<VARIABLE>--	Post-decrement
++<VARIABLE>	Pre-increment
--<VARIABLE>	Pre-decrement
-	Unary subtraction
+	Unary addition
**	Exponentiation
*	Multiplication
/	Division
%	Remainder
+	Addition
-	Subtraction

Arithmetic Operation Priorities

- The priorities, in descending order, are as follows:

Operator	Function
<VARIABLE>++, <VARIABLE>--	Post-increment and post-decrement
++<VARIABLE>, --<VARIABLE>	Pre-increment and pre-decrement
-, +	Unary subtraction and addition
**	Exponentiation
*, /, %	Multiplication, division, and remainder
+, -	Addition and subtraction

Difference Between Pre-increment and Post-increment Variables

- Is the output of **echo $[++i]** the same as that of **echo $[i++]**?
- What are the precautions for triggering boundary conditions?

```
[root@openEuler ~]# i=3
[root@openEuler ~]# echo $[i]
3
[root@openEuler ~]# echo $[++i]
4
[root@openEuler ~]# echo $[i++]
4
[root@openEuler ~]# echo $i
5
```

Contents

Structured Commands in the Shell

- In addition to sequential execution, shell scripts require some extra logic control processes.
- Similar to other programming languages, structured commands in shell scripts include conditions and loops.

Conditional Statements

- if-then statement
- Syntax:

```
if command
   then
      commands
fi
```

- The bash shell executes the command following **if** first. If the exit status code is **0**, the bash shell continues to execute the commands in **then**. Otherwise, the bash shell executes the next command in the script.

Multi-branch Judgment Statements

- When a multi-condition judgment is required, there may be a complex **if-then-else** statement, **elif**, which frequently checks the value of a variable. In this scenario, the **case** statement may be a more suitable solution.
- Syntax:

```
case variable in
    pattern1 | pattern2) commands1;;
    pattern3) commands2;;
    *) default commands;;
esac
```

Loop Statements

- Shell scripts often contain some repeated tasks, which are the same as cyclically executing a group of commands until a specific condition is met.
- There are three common loop statements: **for**, **while**, and **until**.
- There are two types of loop control characters: **break** and **continue**. They are used to control the direction of the loop process.

For Loop

- Syntax (shell style)

```
for var in list
    do
        commands
    done
```

Example:

```
for i in {1..10}
do
        printf "$i\n"
done
```

- Syntax (C language style)

```
for ((var assignment ; condition ; iteration process))
do
        commands
done
```

Example:

```
for (( i = 1; i < 10 ; i++))
do
    echo "Hello"
done
```

List in For Loop

```
[root@openeuler ~]# for HOST in host1 host2 host3; do echo $HOST; done
host1
host2
host3
[root@openeuler ~]# for HOST in host{1..3}; do echo $HOST; done
host1
host2
host3
[root@openeuler ~]# for EVEN in $(seq 2 2 8); do echo "$EVEN"; done;
2
4
6
8
```

Odd Accumulator in For Loop

- Define the odd accumulator file and execute it.
- Reference script:

```
#!/bin/bash


sum=0
for i in {1..100..2}
do
    let "sum+=i"
done
echo "sum=$sum"
```

File Display in For Loop

- Display all files in the directory.
- Reference script:

```
#!/bin/bash

for file in $(ls)
do
   echo "file: $file"
done
```

While Loop

- The **while** loop is a pre-test loop. It is used to perform some repetitive tasks that may be executed one or more times according to specific conditions.
- To avoid infinite loops, ensure that the loop body contains the exit condition.

```
#!/bin/bash


sum=0;i=1
while(( i <= 100 ))
do
    let "sum+=i"
    let "i += 2"
done
     echo "sum=$sum"
```

Until Loop

- The until loop is similar to the while loop, but the conditions for exiting the loop are different.

```
#!/bin/bash
# odd accumulator
sum=0;i=1
until(( i > 100 ))
do
    let "sum+=i"
    let "i += 2"
done
    echo "sum=$sum"
```

Printing a Multiplication Table Using a Loop

- Reference script:

```
#!/bin/bash


for (( i = 1; i <=9; i++ ))do
    for (( j=1; j <= i; j++ ))do
    let "temp = i * j"
    echo -n "$i*$j=$temp  "
    done
printf "\n"
done
```

```
[root@openEuler ~]# sh demo.sh
1*1=1
2*1=2  2*2=4
3*1=3  3*2=6  3*3=9
4*1=4  4*2=8  4*3=12  4*4=16
5*1=5  5*2=10  5*3=15  5*4=20  5*5=25
6*1=6  6*2=12  6*3=18  6*4=24  6*5=30  6*6=36
7*1=7  7*2=14  7*3=21  7*4=28  7*5=35  7*6=42  7*7=49
8*1=8  8*2=16  8*3=24  8*4=32  8*5=40  8*6=48  8*7=56  8*8=64
9*1=9  9*2=18  9*3=27  9*4=36  9*5=45  9*6=54  9*7=63  9*8=72  9*9=81
```

Section Summary

- This section described the elements involved in shell script programming, including characters, variables, and logic control structures.
- It demonstrated, with examples, how to compile common scripts.

Contents

Overview and Objectives

- Master bash debugging.
- Master basic shell script debugging methods.
- Understand the importance of a good style when compiling shell scripts.
- Master common script compilation methods.

Troubleshooting Shell Script Errors

- Administrators who write, use, or maintain shell scripts inevitably encounter script errors.
  - Errors are usually caused by input errors, syntax errors, or incorrect script logic.
  - When writing scripts, using a text editor in conjunction with the bash syntax highlighter helps make errors more obvious.
  - The most direct way to find and correct errors in a script is to debug.
  - Another simple way to avoid introducing errors into a script is to follow a good style during script creation.

Debugging Mode (1)

- To activate the debugging mode on a script, add the **-x** option to the command interpreter in the first line of the script.
- Example:

```
#!/bin/bash -x
```

```
[root@euler ~]# cat adder.sh
#!/bin/bash -x
```

Debugging Mode (2)

- The debugging mode of bash prints the commands executed by the script before the script is executed.
- The results of all shell extensions that have been executed are displayed in the print output. The state of all variable data is printed in real time for tracing.

```
[root@openEuler tmp]# bash -x adder.sh
```

Evaluation Exit Code (1)

- Every command returns an exit state, also known as a return state or an exit code.
- The exit code can be used for script debugging.
- The following examples show the execution and exit state retrieval of several common commands.

```
[root@openEuler tmp]# ls /etc/hosts
/etc/hosts
[root@openEuler tmp]# echo $?
0
------------------------------------------------------------
[root@openEuler tmp]# ls /etc/nofile
ls: cannot access /etc/nofile: No such file or directory
[root@openEuler tmp]# echo $?
2
```

Evaluation Exit Code (2)

- Use an exit code in a script.
- A script, once executed, will exit after it has processed everything. It may sometimes exit earlier, for example after encountering an error condition.
- Exiting earlier can also be achieved using the **exit** command. When a script encounters the **exit** command, it exits immediately, skipping the remaining content.

```
[root@openEuler tmp]# cat hello.sh
#!/bin/bash
echo "Hello World"
exit 1
[root@openEuler tmp]# ./hello.sh
Hello World
[root@openEuler tmp]# echo $?
1
```

Contents

Good Style (1)

- The following are some specific practices which can be followed:
  - Divide a long command into multiple lines of smaller code blocks. Shorter code segments are easier for readers to understand.
  - Arrange the beginning and end of multiple statements so you can see where the control structures start and end and whether they are closed correctly.
  - Indent lines containing multiple statement lines to show code logic hierarchy and control structure flow.
  - Use line spacing to separate command blocks, to clarify the start and end of code segments.
  - Use the same format throughout a script.

Good Style (2)

- Adding comments and spaces can greatly improve script readability.
- These simple methods make it much easier for users to find errors during compilation and improve the readability of scripts for future readers.
- Example of code with poor readability:

```
#!/bin/bash

for PACKAGE in $(rpm -qa | grep kernel); do echo "$PACKAGE was installed on $(date -d
@$(rpm -q --qf "%{INSTALLTIME}\n" $PACKAGE))"; done
```

Good Style (3)

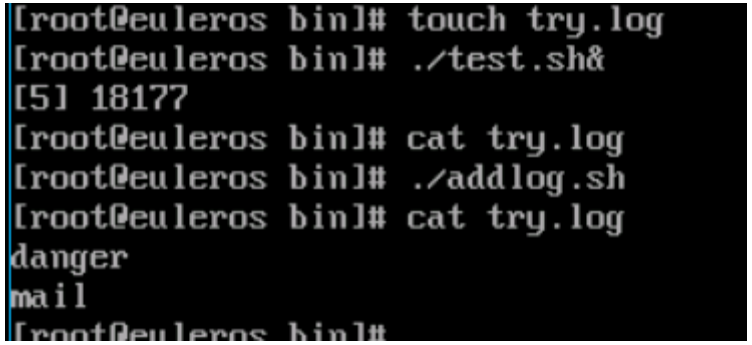
- Script after modification:

```
#!/bin/bash
# This script is used to read kernel-related software package information and query the software package installation time from the RPM database.
PACKAGETYPE=kernel
PACKAGES=$(rpm -qa | grep $PACKAGETYPE)
# Loop information
for PACKAGE in $PACKAGES; do
    # Query the installation time of each software package.
    INSTALLEPOCH=$(rpm -q --qf "%{INSTALLTIME}\n" $PACKAGE)
    # Convert the time to a common date and time.
    INSTALLDATETIME=$(date -d @$INSTALLEPOCH)
    # Print information.
    echo "$PACKAGE was installed on $INSTALLDATETIME"
done
```

Contents

Monitoring Log Alarms and Sending Email Notifications (1)

- Scenario:
  - A carrier requires long-term monitoring of the current environment's **error.log** file, with an alarm email sent if the keyword **danger** is found.
- Requirements:
  - Create a log file and use a script to simulate log file writing.
  - Run the **mail** command to send emails. In a lab environment, however, you can write the **mail** character string to the log file instead.
  - The following figure shows the process.



Run the monitoring script.

The initial log content is empty. Enter the keyword **danger**.

Automatically send an email (replaced by writing **mail**).

Monitoring Log Alarms and Sending Email Notifications (2)

- Run the monitoring script in the background.
  - **./test.sh &**
- Simulate the error reporting.
  - **./addlog.sh**
- View the **error.log** file.
  - **cat /var/log/error.log**

```
#!/bin/bash
# Log detection script test.sh

tail -f /var/log/error.log | while read danger;
do echo 'mail' >> /var/log/error.log;
sleep 1m;
done
```

```
#!/bin/bash
# Error reporting simulation script addlog.sh

echo 'danger' >> /var/log/error.log
```

Summary

This chapter described shell script elements including execution, compilation, and debugging, using actual cases and sample code to help readers easily apply the material in their own work.

- Execution: Common foreground and background running modes as well as corresponding operations were introduced.
- Compilation: Variables and statement structures required for compiling shell scripts were introduced.
- Debugging: Common debugging methods and the importance of a good style when writing were introduced.

Thank you.

把数字世界带入每个人、每个家庭、
每个组织，构建万物互联的智能世界。

Bring digital to every person, home, and organization for a fully connected, intelligent world.

HUAWEI