

Linux 基础



第 11 讲 shell 脚本基础

简单 shell 脚本和变量

shell 脚本介绍

- shell 可以从一个文件读取命令并逐条执行，这个文件一般被称为 shell 脚本。
- shell 脚本最常用来做系统管理之类的工作，它通过组合现有的程序完成特定的任务。

shell 脚本和其他脚本语言

- shell 脚本看起来像是一种语言，并且也提供了逻辑和循环等关键字，也支持函数功能。
- 但是严格来说，它只是逐条执行语句，和常见的脚本语言如 PHP、JS、Python 等还是有很大区别的。

为何要使用 shell 脚本

- 没有任何理由要求必须要强制使用 shell 脚本。
- 但是以下几点是需要考虑的因素：
 - 不需要安装其他软件即可完成自动化或批量任务处理
 - 可以通过少量的代码完成复杂的功能
 - 系统服务管理都使用了 shell 脚本
 - 大量软件都使用了 shell 脚本作为启动配置或编译配置预处理
 - 公司要求 …

认识 shell 脚本

- 以下是一段 shell 脚本代码，它显示系统详细信息，并以树形结构显示设备总线。

```
1  #!/bin/bash
2
3  uname -a
4  lspci -vt
5  █
```

位于第一行的 `#!/bin/bash`

- 这不是 shell 所独有的，因为经常会见到：
`#!/usr/bin/python`
`#!/usr/bin/node`
- 这仅仅是告诉系统，如果遇到这类文件，应该使用哪个程序去执行。 `#!` 后面可以是任何可执行的程序路径。

第一行是 `#!/bin/ls` 就会显示当前目录的内容。

系统如何对待脚本类的可执行文件

- **Linux** 使用标志位来识别可执行文件。但是当系统去执行程序时，发现不是支持的可执行文件格式^[1]，那么就会认为是脚本。
- 此时会去扫描第一行 **#!** 的标记，根据路径查找是否存在此程序。存在则交给此程序去执行。

[1] **Linux/Unix** 采用 **ELF** 格式的文件，而 **Windows** 使用了 **PE** 格式，并使用扩展名 **exe** 表示可执行文件。

一个简单的脚本所具备的条件

- 创建一个文本文件，扩展名可以有 `.sh` 也可以没有。
- 把要执行的命令按顺序编写。
- 最开头的 `#!` 如果没有则会使用默认的 `shell` 执行。
- 给文件加上可执行权限（但这不是必需的）。
- `#` 开头表示注释。

执行 shell 脚本的方式

- 添加可执行权限，并使用 `#!` 声明：

```
#!/bin/bash
```

- 指定执行的 shell，这时候 `#!` 的声明会忽略：

```
bash a.sh
```

`#!/bin/sh`

- 脚本中经常出现 `#!/bin/sh` , 而不是:
`#!/bin/bash`
- `sh` 是一个符号链接指向 `/bin/dash` 。
- `dash` 被设计用来快速执行脚本, 功能不如 `bash` 强, 但是执行速度快。

测试脚本： bash 和 dash 的不同

```
1 #!/bin/sh
2
3 for ((i=0; i<5; i++)) ; do
4     echo $i
5 done
```

指定使用 bash 运行：
bash b.sh

0
1
2
3
4

使用默认的 sh 执行，不支持
for 循环扩展计算。

```
./b.sh: 3: ./b.sh: Syntax error: Bad for loop variable
```

变量

- 变量对于正规的程序很重要，它可以保存有用的数据，并且可以用于管理程序状态。
- 在 `shell` 中，变量很简单，就是保存字符串。
- 由于算术运算也是十分必要的，所以 `bash` 也有用运算的机制。

设置变量

- 在 shell 中设置变量使用以下方式:

`a=123` # 变量名称可以是字母数字下划线, 但不能是数字开头

`b=123+234` # 这仅仅是保存了文本, 不会进行计算

`c="go js html css"` # 空格使用双引号或单引号

`d='Linux Unix'` # 使用单引号不会进行解析, 双引号会解析其中的变量

- `=` 左右不能有空格, 否则会认为是命令去执行。

获取变量的值

- 获取变量的值使用 \$ 加变量的名称。
- 输出变量：

```
echo $a
```

计算

- 类似 $a=12+23$ 这样的操作，并不会进行计算。

- 要使用算术运算，需要一些特殊的语法：

$a=\$(12+23)$

$a=234$; $b=345$; $c=\$(a+b)$

- $((\cdots))$ 会扩展其中的表达式进行计算， $\$$ 是取值操作。

使用内建命令 `let`

- `let` 可以对变量进行运算操作。
- 在编程语言中常见的算术操作都被支持。

```
let a++ ; let b=b+a ; let c=b*a+b
```

保存程序的执行结果

- 使用 ` 包含命令并赋值给变量，可以保存程序的执行结果。
- 例： `a=`ls /usr/share``
- 输出： `echo $a`
- 变量保存的是一个空格分割的列表。

清除变量

- 使用内建命令 `unset` 可以清除不需要的变量：

```
unset a b
```

shell 脚本的一些特殊变量

- `$0` : 当前 shell 脚本的名称。
- `$N` : 参数, 使用 `$1, $2...`, 10 以上要使用 `${10}` 的形式。
- `$@` : 所有参数, 可以使用 `for` 循环遍历。
- `$#` : 参数个数, 不包括脚本名称 (`$0`)。
- `$?` : 上一个命令的返回值, 通常在 `if` 中使用。
- `$$` : 当前 shell 的 PID。

脚本示例

以下脚本用于计算 `/usr/share` 目录所有 `.json` 文件的行数。

```
1 #!/bin/bash
2
3 JSON_LIST=`find /usr/share -iname *.json 2> /dev/null`
4
5 cat $JSON_LIST | wc -l
6
```

逻辑和循环

test 命令

- `test` 是 `shell` 的内建命令，经常用于各类的检测工作，产生的不是一般形式的输出，而是可用的退出状态。
- `test` 经常和 `if` , `while` 等用于条件的关键字配合使用。

test 命令

- `test` 返回的结果在 `shell` 中运行不会有输出。
- 查看 `test` 帮助文档: `help test`。
- `test` 返回 `true` 或 `false`。

shell 中的 true 和 false

- shell 中通过程序的最终的返回值判断程序的运行状态。
- 但是和通常的编程语言不同的是，在 shell 中 true 是 0，非 0 值为 false_[1]。

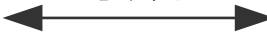
[1] 当初 Ken·Thompson 和 Dennis·Ritchie 设计 Unix 和 C 语言时，把返回值 0 作为程序正确运行的状态，表示 0 错误，其他任何非 0 值都表示出错了。所以在 C 语言中 main 函数最后都会有 return 0;

test 示例

- `test "abc"="abc"` # 检测字符串是否相等
- `test -f c/a.c` # 检测 `c/a.c` 文件是否存在并且为普通文件
- `test -d bin` # 检测 `bin` 是否为目录
- `test -z $A` # 如果字符串为空则返回 `true`
- `test -n $A` # 如果字符串不为空则返回 `true`

test 等效形式

- test 等效于 [...] , 注意括号中要有空格分开。

<pre>if test -f bin/a ; then echo 'bin/a exists' fi</pre>	<p>等效</p> 	<pre>if [-f bin/a] ; then echo 'bin/a exists' fi</pre>
---	--	--

if elif else

```
if [COMMAND]
then
    [COMMAND]
fi
```

```
if [COMMAND]; then
    [COMMAND]
else
    [COMMAND]
fi
```

```
if [COMMAND]; then
    [COMMAND]
elif [COMMAND] ;
then
    [COMMAND]
else
    [COMMAND]
fi
```

写在一行要使用分号分隔：

```
if [COMMAND] ; then [COMMAND] ; fi
```

case

- case 相当于其他编程语言的 switch , case 结构如下:

```
case WORD in
    VALUE1)
        [COMMANDS]
        ;;
    VALUE2)
        [COMMANDS]
        ;;
    *)
        [COMMANDS]
        ;; //esac之前的;;可以省略
esac
```

for 循环

- for 循环用于遍历整个列表。
- `for NAME in WORDS; do COMMANDS; done`
- `for NAME in WORDS ; do
 COMMANDS
done`

while 循环

- while CONDITION ; do
 COMMANDS
done
- while 循环的条件是命令的执行状态。

示例

- 实时显示时间 `lt.sh` :

```
1  #!/bin/bash
2
3  while date ; do
4      sleep 1
5      clear
6  done
```


函数

函数

- shell 中的函数编写很简单。
- 直接写函数结构，并在其中编写命令。
- 函数的名称就像是命令直接调用。

```
3 #定义函数
4 time_say() {
5     while date ; do
6         sleep 1
7         clear
8     done
9 }
10
11 #调用
12 time_say
```

函数的参数

- 函数的参数传递和使用命令传递参数形式相同。
- 在函数中，`$@` 获取的是传递给函数的参数，`shell` 的参数此时被隐藏，或者说被掩盖。

```
3  funca() {  
4      for a in $@ ; do  
5          echo $a  
6      done  
7  }  
8  
9  funca 1 2 3  
10  
11 for a in $@ ; do  
12     echo $a  
13 done
```