Bash Shell 脚本编程实践

Hank
电子技术博客 UinIO.com

5 人赞同了该文章

Shell 既是一套**命令行工具**（交互式地解释和执行用户输入的命令）也是一种**脚本设计语言**（定义有变量与参数，并提供了控制、循环、分支结构）。**Bash Shell** 是由 GUN 官方项目提供的 Shell 解释器，名称源自于 **B**ourne **A**gain **SH**ell 的英文缩写，整合了传统 **Korn Shell** 以及 **C Shell** 的有效特性，并且尽量遵循 `IEEE POSIX P1003.2/ISO 9945.2` 规范，同时在编程与交互使用方面提供了大量的功能改进，因而在提供丰富功能的基础之上，展现出了良好的兼容特性，大多数 `.sh` 脚本可以无需移植修改即可交由 Bash Shell 来执行。

当用户登入任意一款 Linux 操作系统时，初始化程序 `init` 都将会为用户启动一个**Bash Shell**命令解析器，其即可以用于解析命令行输入并与内核进行交互，也可以作为高效的脚本编程语言，运用其提供的变量、参数、循环、分支等编程语法特性，完成一些批量的自动化的任务处理工作，本文将会围绕 Bash Shell 的脚本编程特性，加以进行详细的分析、说明与示例。

阅读全文完整带书签的版本，可以进入点击下面链接查阅**笔者 GitHub 博客：**

Bash Shell 脚本编程实践
www.uinio.com/Linux/Shell/

基础概念

Shell 脚本的解释过程就是从文件读入字符流，然后进行处理，最后将结果传送至某个**文件**，所以交互式 Shell 命令与 Shell 脚本在本质上并没有区别，只是 Shell 命令的输入是**标准输入**，输出是**标准输出**。

Shell 脚本的注释以 `#` 符号开始，一直到行末结束，例如可以在 Shell 命令行中输入以 `#` 开头的命令，则该命令将会被作为注释而忽略处理。

```
pi@raspberrypi:~ $ # 这是一条注释
```

> 按下【Ctrl + D】组合键将会在**标准输入**上产生一个文件结尾，因此在 Shell 命令当中可以使用该组合键直接**退出** Shell 命令行。

Sha-Bang

Sha-Bang 是 Shell 脚本开头字符 `#!` 连在一起的读音（Sharp Bang），当 Shell 文件被 Linux 系统读取时，内核会通过 `#!` 表示的值（`0x23, 0x21`）识别出随后的解释器路径并调用，最后再将整个脚本内容传递给解释器。由于 Shell 当中 `#` 字符同时表示注释，因此 Shell 解释脚本文件时会自动忽略该行。本文讨论的 Shell 脚本通常以 `#!/bin/sh` 或者 `#!/bin/bash` 开头，表示当前使用的解释器为 **Dash Shell** 或者 **Bash Shell**。本文所涉及的代码都基于 `#!/bin/bash` 路径下的**Bash Shell**。

```
pi@raspberrypi:~ $ echo $SHELL
/bin/bash

pi@raspberrypi:~ $ bash --version
GNU bash，版本 5.0.3(1)-release (arm-unknown-linux-gnueabihf)
```

接下来，编写一个 Bash Shell 版本的 Hello World 程序：

```
#!/bin/bash
echo "Hello Bash Shell !"    # 注释
```

执行方式

Shell 脚本的执行主要存在如下 5 种方式：

1. 将拥有执行权限的脚本作为命令调用，例如：`./hello.sh`；
2. 显式使用 Shell 程序，将脚本文件作为参数来执行，例如：`sh hello.sh`；
3. 将脚本文件重定向至 Shell 程序的标准输入，例如：`sh < hello.sh`；
4. 通过管道符将脚本内容输出至 Shell 程序的标准输入，例如：`cat hello.sh | sh`；
5. 使用 `source` 命令执行，例如：`source hello.sh`；

字符串与引号

Shell 解释器采用了字符流过滤器模型，简而言之，就是一条命令将结果送到标准输出，这个标准输出被连接到下一条命令的标准输入，每条命令的输出结果都是自己处理之后的字符流，接受的输入都是需要进行处理的字符流，所以字符串是 Shell 当中非常重要的组成部分。

Shell 当中存在 `'`、`"`、`` ` `` 三种引号类型，其具体使用区别分别如下所示：

- **单引号** `'` 当中的字符串 Shell 不会进行处理，仅在需要保持字符串原样不变的时候使用；`bash pi@raspberrypi:~ $ writer=Hank pi@raspberrypi:~ $ echo '本文作者是：$writer' 本文作者是：$writer`
- **双引号** `"` 当中的字符串 Shell 会进行处理，如果其中包含有可以求值的部分（变量、表达式、命令），则会被 Shell 替换为相应的求值结果；`bash pi@raspberrypi:~ $ writer=Hank pi@raspberrypi:~ $ echo "本文作者是：$writer" 本文作者是：Hank`
- **反引号** `` ` `` 用于引用一条 Linux 命令，其作用是将该命令的执行结果输出，效果类似于 `"$()"`；`bash pi@raspberrypi:~ $ echo 现在时间是 `date` 现在时间是 2020年 07月 12日 星期日 14:56:40 CST`

特殊字符

- `*` 和 `?` 都是通配符，前者匹配任意个字符，后者仅匹配一个字符；
- `:` 表示空命令，其返回值恒为 `0`，循环语句当中，可以与 `true` 命令等价；
- `;` 是分行符，标识一行命令结束，可以通过它将多条命令编写在一行；`bash pi@raspberrypi:~ $ echo "Hello"; echo "World"; echo "at `date`" Hello World at 2020年 07月 12日 星期日 16:00:29 CST`
- `$` 可以用于获取变量或者表达式的值，结合大括号 `${}` 使用，可以在变量出现在字符串当中时，不与字符串内容相混淆；`bash pi@raspberrypi:~ $ writer=Hank pi@raspberrypi:~ $ echo ${writer} is the author of this blog Hank is the author of this blog` 结合小括号 `$()` 可以取一个命令的值作为字符串内容，其效果与反引号 `` ` `` 相同；`bash pi@raspberrypi:~ $ echo Now the time is $(date) Now the time is 2020年 07月 12日 星期日 16:48:33 CST` 通过双小括号 `$(())` 可以取得一个数学表达式的计算结果，例如在使用 `*` 运算符计算一个乘积；`bash pi@raspberrypi:~ $ echo $((2*2)) 4`
- `.` 句点符号，等效于 `source` 命令，可用于在 Shell 进程上调用脚本；
- `\` 反斜线表示转义符，是一种引用单个字符的方法，也可以用于 Shell 命令的换行；

```
pi@raspberrypi:~ $ echo "hello" \ "bash" \ "shell"   hello bash shell
```

- **空格**作为参数命令的做分隔符，例如：`touch a b` 会创建 `a` 和 `b` 两个文件，而 `touch c\ d` 则只会创建一个名为 `'c d'` 文件；

```
pi@raspberrypi:~ $ touch c\ d
pi@raspberrypi:~ $ ls -l   -rw-r--r-- 1 pi pi      0 7月  12 16:55 'c d'
```

内/外部命令

外部命令：Shell 的绝大多数命令如同 `/bin/ls` 一样，是一个独立的外部可执行程序。当外部命令被调用时，本质就是调用了另外一个程序，首先 Shell 会创建子进程，然后在子进程当中运行该程序； **内部命令**：内建在 Shell 程序当中，由 Shell 软件内部进行实现的命令，例如： `cd` 、 `source` 、 `export` 、 `time` 等，它们都运行在 Shell 进程当中。

注意：如果希望脚本能够改变当前 Shell 自身的一些属性，则必须在 Shell 进程内执行调用。例如修改 `/etc/profile` 、 `~/.profile` 、 `~/.bashrc` 环境变量之后，必须使用 `source` 命令执行它们，以使其生效。

```
pi@raspberrypi:~ $ source /etc/profile

pi@raspberrypi:~ $ source ~/.profile

pi@raspberrypi:~ $ source ~/.bashrc
```

重定向

Shell 的设计哲学是**字符流 + 过滤器**，即将一个程序的输出，作为另一个程序的输入，这样就能将各种用途简单的小工具组合起来，完成一些看起来不可思议的功能。

默认情况下，Linux 当中的每一个进程都拥有 3 个特殊的文件描述指针：

- **标准输入**：Standard Input，文件描述指针为 `0` ；
- **标准输出**：Standard Output，文件描述指针为 `1` ；
- **标准错误输出**：Standard Eror，文件描述指针为 `2` ；

IO 重定向就是捕捉命令、程序、脚本甚至代码块的输出，然后将其作为输入传递给另外的文件、命令、程序、脚本。

输出重定向

输出重定向符号 `>` 和 `>>` ，可以将**标准输出**重定向至一个文件当中，如果该文件不存在则创建文件。其中，前者 `>` 会覆盖原文件内容：

```
pi@raspberrypi:~ $ echo "this is line 1">output.txt
pi@raspberrypi:~ $ echo "this is line 2">output.txt
pi@raspberrypi:~ $ echo "this is line 3">output.txt
pi@raspberrypi:~ $ cat output.txt

this is line 3
```

后者 `>>` 则会在原文件尾部追加新的内容：

```
pi@raspberrypi:~ $ echo "this is line 1">>output.txt
pi@raspberrypi:~ $ echo "this is line 2">>output.txt
pi@raspberrypi:~ $ echo "this is line 3">>output.txt
pi@raspberrypi:~ $ cat output.txt

this is line 1
this is line 2
this is line 3
```

输入重定向

输出重定向符号 `<` 和 `<<`，用于将**标准输入**重定向至一个文件。如果 `<` 后跟着一个 Shell 脚本文件，则相当于将 `.sh` 脚本中的命令逐条输入至 Shell 程序当中执行：

```
pi@raspberrypi:~ $ cat<hello.sh
#!/bin/bash
echo "Hello Bash Shell !"    # 注释

pi@raspberrypi:~ $ bash < hello.sh
Hello Bash Shell !
```

`<<` 可以用于 **Here Document**，即将文本直接写在 Shell 脚本之中，并以添加终止符 `EOF`（即 Linux 系统读取至文件结尾时所返回的信号值 `-1`），该文本相当于一份独立的文件内容，例如：执行下面的 `hello.sh` 脚本以后：

```
#!/bin/bash
echo "文件 hello 不存在"
ls -l   # 列出当前目录所有文件

# 使用 Here Document 方式产生 hello.c 文件
cat>hello.c<<EOF

#include<stdio.h>
#include<stdlib.h>

int main(void){
    printf("Hello World\n");
    return EXIT_SUCCESS;
}
EOF

# 编译 hello.c
cc -W -Wall -o hello hello.c
ls -l   # 列出当前目录所有文件

# 执行 hello，然后清除新生成的文件
./hello
rm hello hello.c
```

将会动态生成一个 `hello.c` 源文件，然后编译产生二进制文件 `hello`，最后执行并且展示结果，同时删除新生成的 2 个文件。

```
pi@raspberrypi:~ $ ./hello.sh

文件 hello 不存在
总用量 1
-rwxrwxrwx 1 pi pi    234 7月  12 18:51  hello.sh
总用量 3
-rwxr-xr-x 1 pi pi   7980 7月  12 18:51  hello
-rw-r--r-- 1 pi pi    111 7月  12 18:51  hello.c
-rwxrwxrwx 1 pi pi    234 7月  12 18:51  hello.sh
Hello World
```

> **注意**：**Here Document** 通常用于进行复杂的多行文本输入时，从而代替 `echo` 命令繁琐的硬编码操作。

管道

管道符 `|` 用于连接 Linux 命令，前一条命令的标准输出会成为下一条命令的标准输入。管道的最大特点在于是管道符 `|` 两边分别属于不同的进程。例如：从 `dmesg` 输出的内核日志信息中，通过 `grep` 查找 USB 相关的内容。

```
pi@raspberrypi:~ $ dmesg | grep USB

[    0.369739] xhci_hcd 0000:01:00.0: new USB bus registered, assigned bus numb
[    0.372689] usb usb1: New USB device found, idVendor=1d6b, idProduct=0002, b
[    0.372704] usb usb1: New USB device strings: Mfr=3, Product=2, SerialNumber
[    0.373268] hub 1-0:1.0: USB hub found
[    0.373828] xhci_hcd 0000:01:00.0: new USB bus registered, assigned bus numb
[    0.373846] xhci_hcd 0000:01:00.0: Host supports USB 3.0 SuperSpeed
[    0.374254] usb usb2: New USB device found, idVendor=1d6b, idProduct=0003, b
[    0.374267] usb usb2: New USB device strings: Mfr=3, Product=2, SerialNumber
[    0.374779] hub 2-0:1.0: USB hub found
[    0.383583] usbhid: USB HID core driver
[    0.737890] usb 1-1: new high-speed USB device number 2 using xhci_hcd
[    0.920531] usb 1-1: New USB device found, idVendor=2109, idProduct=3431, bc
[    0.920545] usb 1-1: New USB device strings: Mfr=0, Product=1, SerialNumber=
[    0.920558] usb 1-1: Product: USB2.0 Hub
[    0.922299] hub 1-1:1.0: USB hub found
```

常量与变量

Shell 支持多种进制的整型**常量**，例如以 `0` 开头的八进制，以 `0x` 开头的十六进制。对于非八进制、十进制、十六进制的整数，可以表示为 `进制#数字` 格式，例如：三进制数 $(120)_3$ 可以表示为 `3#120`，转换为十进制值为 `15`。

```
pi@raspberrypi:~ $ echo $((3#120))
15
```

Shell 中的**变量**在使用前不需要声明，赋值时可以直接使用变量名，且赋值的等号 `=` 两边不能有空格。变量定义之后，引用变量时一定需要使用 `$` 符号。

```
pi@raspberrypi:~ $ writer=Hank
pi@raspberrypi:~ $ echo ${writer}
Hank

pi@raspberrypi:~ $ writer=Jack
pi@raspberrypi:~ $ echo $writer
Jack
```

Shell **变量没有类型**，例如 `annum=2020`，既可以作为十进制整数 `2020` 直接参与算术运算，也可以作为字符串来进行处理。

```
# 使用 let 计算一个算术表达式并且赋值给变量
pi@raspberrypi:~ $ annum=2020
pi@raspberrypi:~ $ let "annum+=1"
pi@raspberrypi:~ $ echo $annum
2021

# 将字符串变量中的 202 替换成为 203
```

```
pi@raspberrypi:~ $ b=${annum/202/203}
pi@raspberrypi:~ $ echo $b
2031
```

Shell **变量有作用域**，默认为对整个 Shell 文件有效的**全局变量**。**局部变量**则需要使用 `local` 关键字进行声明，其只在声明所在的块或者函数当中可见。

```
#!/bin/bash
# test.sh
test() {
    variable1=GLOBAL
    local variable2=LOCAL
    echo "函数内部，variable1=$variable1，variable2=$variable2"
}

test

echo "函数外部，variable1=$variable1，variable2=$variable2"
pi@raspberrypi:~ $ ./test.sh

函数内部，variable1=GLOBAL，variable2=LOCAL
函数外部，variable1=GLOBAL，variable2=
```

`?` 问号也是一个变量，通过 `$?` 可以引用上一条命令的返回值，但是该值只能使用一次，使用完以后就会被目前命令的返回值所替换。

```
pi@raspberrypi:~ $ false

# false 命令返的回值恒为 1
pi@raspberrypi:~ $ echo $?
1

# 使用 echo 查看 1 次后，变量 ? 的值会就被 echo 命令的返回值 0 覆盖
pi@raspberrypi:~ $ echo $?
0
```

环境变量

环境变量是可以改变 Shell 行为的变量，每个进程都拥有各自的环境变量，以用于保存进程相关的各种信息。环境变量的定义通常都是约定俗成的，例如：`PATH` 定义了 Shell 进程查找命令程序的路径。

```
pi@raspberrypi:~ $ echo $PATH

/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/local/games:/
```

Shell 当中的任何变量都可以通过 `export` 导出为环境变量，环境变量可以被子进程继承，因此也可以被视为父子进程信息传递的一种方式。

```
pi@raspberrypi:~ $ export PATH="$PATH:/workspace"

pi@raspberrypi:~ $ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/local/games:/
```

位置参数

位置参数是指调用 Shell 脚本时，按照命令行位置进行引用的参数。脚本当中按照 `$0` 、 `$1` 、 `$2` 的顺序逐个进行引用，依此类推。其中 `$0` 就代表命令本身。

```
#!/bin/bash
# parameter.sh
echo $0
echo $1
echo $2
echo $3
pi@raspberrypi:~ $ ./parameter.sh 2019 2020 2021

./parameter.sh
2019
2020
2021
```

命令行参数相关的特殊变量还有 `$#` 、 `$*` 、 `$@` ，其使用方法如下所示：

- `$#` ：代表命令行参数的个数；
- `$*` ：代表**全部**命令行参数，全部参数作为一个字符串；
- `$@` ：代表**所有**命令行参数，每个参数都是一个独立的字符串；

```
#!/bin/bash
# parameter.sh
echo "命令行参数个数：$#"

touch "$*"    # 由于将全部参数作为 1 个字符串，所以使用全部参数为名称创建 1 个文件
ls -l

touch "$@"    # 因为参数为 3 个字符串，所以分别使用每个参数作为名称，共创建 3 个文件
ls -l
pi@raspberrypi:~ $ ./parameter.sh 1 2 3

命令行参数个数：3
总用量 620
-rw-r--r-- 1 pi pi      0 7月  13 04:31 '1 2 3'
总用量 620
-rw-r--r-- 1 pi pi      0 7月  13 04:31 '1 2 3'
-rw-r--r-- 1 pi pi      0 7月  13 04:31  1
-rw-r--r-- 1 pi pi      0 7月  13 04:31  2
-rw-r--r-- 1 pi pi      0 7月  13 04:31  3
```

操作符

Shell 当中的每一条命令同时也是一个逻辑表达式，其返回值为 `0` 表示真，返回值为非 `0` 表示假，该值本质上就是当前命令所对应 `main()` 函数的返回值，可以通过 `$?` 来进行获取。Shell 支持基本的数学运算符号以及各种逻辑操作符。

数学运算符包括 `+` 、 `-` 、 `*` 、 `/` 、 `%` 以及幂运算 `**` ，Bash Shell 本身只支持整数运算，如果需要使用到浮点运算，则可以调用 `bc` 和 `dc` 等外部命令。

```
pi@raspberrypi:~ $ echo $((3.14+2))

-bash: 3.14+2: 语法错误: 无效的算术运算符 (错误符号是 ".14+2")
```

逻辑操作符包括 `&&` 和 `||`，分别代表逻辑**与**和逻辑**或**。对于逻辑与 `&&` 而言，如果左侧表达式为 `false`，则右侧表达式无需执行即可确定整个表达式的结果为 `false`；

```
pi@raspberrypi:~ $ true && date
2020年 07月 13日 星期一 04:58:28 CST

pi@raspberrypi:~ $ false && date
```

对于逻辑或 `||` 而言，如果左侧表达式为 `true`，则右侧表达式无需执行即可确定整个表达式的结果为 `true`；

```
pi@raspberrypi:~ $ true || date

pi@raspberrypi:~ $ false || date
2020年 07月 13日 星期一 05:01:43 CST
```

脚本返回值

通常情况下，Shell 脚本在最后都应该拥有一个返回值，如果未显式的通过 `exit` 指定返回值，则默认使用脚本最后一条命令的返回值；

```
#!/bin/bash
# hello.sh
echo "hello world"
false    # false 的返回值为 1
exit 0   # 显式声明返回值为 0
pi@raspberrypi:~ $ ./hello.sh

hello world
```

函数

Shell 脚本当中的函数有 2 种定义方法，其中一种是通过 `function` 关键字进行定义：

```
function function_name(){
  # command
}
```

另外一种与 C 语言当中函数的定义方式相类似，这种方式可移植性更好，更加推荐使用：

```
function_name(){
  # command
}
```

Shell 当中的函数必须在其被调用之前完整的进行定义，调用函数时直接通过函数名称 `function_name` 直接调用即可；

```
#!/bin/bash
# test.sh
echo "Shell 函数调用示例: "

function test() {
  echo "函数 test 被调用!"
}
test

echo "函数调用结束!"
pi@raspberrypi:~ $ ./test.sh

Shell 函数调用示例:
函数 test 被调用!
函数调用结束!
```

条件测试

Shell 提供了一系列条件测试运算符，用于判断某种条件是否成立，条件测试运算符主要包含如下 3 种：

```
test expression
[ expresssion ]     # 条件和左右括号之间必须带有空格
[[ expression ]]    # 新版本 Bash Shell 提供
```

文件测试

文件测试通常用于判断文件属性，常用的文件测试条件如下所示：

条件	含义	示例
-e 或 -a	文件存在（-a 已弃用）	[-e ~/.bashrc]
-f	普通文件	[-f ~/.profile]
-s	文件长度不为 0	[-s /etc/mtab]
-d	文件是目录	[-d /etc]
-b	文件是块设备文件	[-b /dev/sda]
-c	文件是字符设备	[-c /dev/ttyS0]
-p	文件是管道	[-p /tmp/fifo]
-h/-L	文件是符号链接	[-L /etc/mtab]
-S	文件是 Socket	[-S /tmp/socket]
-t	是否为关联到终端的文件描述符	[-t /dev/stdout]
-r	文件可读	[-r ~/.bashrc]
-w	文件可写	[-w ~/.profile]
-x	文件可执行	[-x /bin/ls]
-g	文件有 SGID 标识	[-g /bin/su]
-u	文件有 SUID 标识	[-u /usr/bin/sudo]
-k	具有粘滞位	[-k /tmp]
-O	测试者是文件拥有者	[-O ~/.bashrc]
-G	文件的组 ID 与测试者相同	[-G ~/.profile]
-N	文件从最后一次查看到现在，是否有被修改过	[-N ~/.profile]
file1 -nt file2	文件 file1 比文件 file2 更新	[~/.bashrc -nt ~/.profile]
file1 -ot file2	文件 file1 比文件 file2 更旧	[~/.bashrc -ot ~/.profile]
file1 -ef file2	file1 和 file2 都是同一个文件的硬链接	[/usr/bin/test -ef /usr/bin/[ ]
!	取反测试结果，如果没有条件则返回 true	[! -d ~/.profile]

知乎 @Hank

整数比较

条件	含义	示例
-eq	等于	["$m" -eq "$n"]
-ne	不等于	["$m" -ne "$n"]
-gt	大于	["$m" -gt "$n"]
-ge	大于等于	["$m" -ge "$n"]
-lt	小于	["$m" -lt "$n"]
-le	小于等于	["$m" -le "$n"]
<	小于，需要以 (()) 方式测试	(("$m" < "$n"))
<=	小于等于，需要以 (()) 方式测试	(("$m" <= "$n"))
>	大于，需要以 (()) 方式测试	(("$m" > "$n"))
>=	大于等于，需要以 (()) 方式测试	(("$m" >= "$n"))

知乎 @Hank

字符串比较

条件	含义	示例
= 或 ==	相等，== 在 [] 和 [[]] 里的行为可能会表现不同	["$str1" = "$str2"]
!=	不相等	["$str1" != "$str2"]
>	大于，按照 ASCII 顺序进行比较，在 [] 中使用时需要转义为 \>	["$str1" \> "$str2"]
<	小于，按照 ASCII 顺序进行比较，在 [] 中使用时需要转义为 \<	["$str1" \< "$str2"]
-z	长度为 0	[-z "$str"]
-n	长度不为 0，在 [] 当中使用时，需要将字符串放入 "" 里面	[-n "$str"]

知乎 @Hank

混合比较

`test` 方式还支持在多个表达式之间进行逻辑运算，其中 `-a` 表示与运算， `-o` 表示或运算。下面的示例用于测试命令行参数提供的整数是否介于 `0 ~ 100` 之间，若位于该区间范围输出 `yes`，不在则向控制台输出 `no` 。

```
#!/bin/bash
# compare.sh
[ "$1" -ge 0 -a "$1" -le 100 ] && echo yes || echo no
[ "$1" -lt 0 -o "$1" -gt 100 ] && echo no || echo yes
pi@raspberrypi:~ $ ./compare.sh 0
yes
yes

pi@raspberrypi:~ $ ./compare.sh 85
yes
yes

pi@raspberrypi:~ $ ./compare.sh 100
yes
yes

pi@raspberrypi:~ $ ./compare.sh 101
no
no
```

条件判断

if then

根据 `if` 表达式的逻辑值，决定是否执行 `then` 里的内容。通常 `if` 会与条件测试表达式一同使用，但也可以结合其它命令或者函数。最后， `if` 需要通过 `fi` 结束条件流程。

```
if 条件
  then
    代码块
fi
```

如果 `if` 与 `then` 编写在相同的行，则需要额外再添加一个 `;` 分号：

```
if 条件; then
  代码块
fi
```

下面代码当中，仅当 `if` 后面的表达式为 `true` 时， `then` 里的 `echo` 命令才会得到执行。

```
#!/bin/bash
# decide.sh

if true
  then
    echo "true 分支"
fi

if false; then
  echo "false 分支"
fi
pi@raspberrypi:~ $ ./decide.sh

true 分支
```

if then else

条件流程控制语句还可以拥有一个 `else` 分支，用于条件不成立的情况。

```
if 条件; then
    代码块 1
  else
    代码块 2
fi
```

这样 `then` 和 `else` 后面各有一个代码块，根据 `if` 后面表达式的逻辑值来决定具体执行哪个，下面是一个具体的示例：

```
#!/bin/bash
# decide.sh
condition=false

if $condition; then
    echo "true 分支"
  else
    echo "false 分支"
fi
pi@raspberrypi:~ $ ./decide.sh
```

```
false 分支
```

if then elif else

如果存在多个并列并且互斥的条件，则可用采用 `elif` 来依次判断条件：

```
if 条件1; then
    # 代码块 1
  elif 条件2; then
    # 代码块 2
  # ... ...
  elif 条件n; then
    # 代码块 n
  else
    # 代码块 n+1
fi
```

程序会依次测试每一个条件，如果条件 `n` 符合，则执行代码块 `n` ，如果所有条件均不符合，则执行最后的 `else` 分支（非必须）。

```
#!/bin/bash
# test.sh

# 判断脚本第 1 个命令行参数是否为 1
if [ "$1" -eq 1 ]; then
  echo "1"
  # 判断脚本第 1 个命令行参数是否为 2 或 3
  elif [ "$1" -eq 2 -o "$1" -eq 3 ]; then
    echo "2 或者 3"
  # 判断脚本第 1 个命令行参数是否为介于 4 ~ 7 之间
  elif [ "$1" -ge 4 -a "$1" -lt 7 ]; then
    echo "[4, 7)"
  # 如果都不是，则输出其它
  else
    echo "其它"
fi
pi@raspberrypi:~ $ ./test.sh 1
1

pi@raspberrypi:~ $ ./test.sh 3
2 或者 3

pi@raspberrypi:~ $ ./test.sh 5
[4, 7)

pi@raspberrypi:~ $ ./test.sh 9
其它
```

循环结构

Bash Shell 支持 `for` 、 `while` 、 `until` 三种不同类型的循环，其循环体当中的内容必须包含在 `do` 和 `done` 语句之间。

for 循环

for 循环的 `列表` 是一个由空格分隔的字符串列表，支持通配符。如果缺省，则会自动使用当前的命令行参数列表 `$@`。

```
for 参数 in [列表]
  do
    命令
  done
```

下面的示例会根据输入的参数，分别循环打印工作日和非工作日：

```
#!/bin/bash
# week.sh

if [ "$1" == "工作日" ]; then
  for wd in Monday Tuesday Wednesday Thursday Friday; do
    echo $wd
  done
elif [ "$1" == "非工作日" ]; then
  for wd in Saturday Sunday
    do
      echo $wd
    done
else
  echo "输入错误！"
fi
pi@raspberrypi:~ $ ./week.sh
输入错误!

pi@raspberrypi:~ $ ./week.sh 工作日
Monday
Tuesday
Wednesday
Thursday
Friday

pi@raspberrypi:~ $ ./week.sh 非工作日
Saturday
Sunday
```

`列表` 当中的通配符会被 Shell 展开，下面示例脚本当中，`*.c` 会被展开为当前目录下所有 `.c` 后缀的非隐藏文件：

```
#!/bin/bash
# file.sh

for filename in *.c
  do
    echo $filename
  done
pi@raspberrypi:~ $ ./file.sh

function.c
main.c
```

Bash Shell 同时也通过双小括号 `(( ))` 支持 C 风格的 `for` 循环。

```
for ((表达式 1; 表达式 2; 表达式 3))
  do
    命令
  done
```

其中，`表达式1` 是循环执行之前的初始化，`表达式2` 是一个代表循环逻辑测试的表达式，`表达式3` 是每次循环体执行完成之后的处理

```
#!/bin/bash
# loop.sh

for ((i=0; i < 7; i++))
  do
    echo $i
  done
pi@raspberrypi:~ $ ./loop.sh

0
1
2
3
4
5
6
```

while 循环

while 循环根据测试条件，反复执行循环体直至条件为**假**，同样拥有 **Shell** 和 **C** 两种风格。

```
# Shell 风格
while [条件]
  do
    命令
  done

# C 风格
while ((表达式))
  do
    命令
  done
```

接下来的示例代码，同时使用 Shell 和 C 两种风格的 `while` 循环，该示例会根据命令行参数的个数来打印它们：

```
#!/bin/bash
# loop.sh

i=0
# Shell 风格循环
while [ "$i" -le "$#" ]
  do
    eval tmp=\$$i     # 以变量 i 的值作为变量名再进行取值
    echo "$tmp"
    i=`expr $i + 1`   # 获取参数个数，并作为 C 风格循环的索引长度
  done
```

```
echo

j=0
# C 风格循环
while ((j++ <= i))
  do
    eval tmp=\$$j
    echo "$tmp"
  done
pi@raspberrypi:~ $ ./loop.sh 1 2 3 4 5

./loop.sh
1
2
3
4
5

1
2
3
4
5
```

until 循环

`until` 循环与 `while` 类似，但是 `util` 循环是在条件为**假**时执行循环体，直至条件为**真**时才结束循环。

```
until [条件]
  do
    命令
  done

# 或者
until ((表达式))
  do
    命令
  done
```

跳出循环

Shell 循环结构当中，可以使用 `break` 或者 `continue` 跳出循环，它们都可以携带一个用于标识所要跳出**循环层数**的数值，该数值缺省情况下为 `1` ，表示仅跳出当前所在循环。

break

Bash Shell 当中的 `break` 关键字用于中断整个循环，其具体用法如下：

```
break n
```

`n` 表示跳出循环的层数，如果省略 `n` ，则表示仅中断当前循环。 `break` 关键字通常与 `if` 语句联用，即满足条件时中断循环。例如下面代码用于输出一个 `4*4` 的矩阵：

```
#!/bin/bash
# break.sh

i=0
# 外层循环
while ((++i)); do
  j=0;
  # 内层循环
  while ((++j)); do
    if((i>4)); then
      break 2  # 中断内外 2 层循环
    fi
    if((j>4)); then
      break    # 仅中断内层循环
    fi
    printf "%-4d" $((i*j))
  done
  printf "\n"
done
pi@raspberrypi:~ $ ./break.sh

1   2   3   4
2   4   6   8
3   6   9   12
4   8   12  16
```

continue

Bash Shell 当中的 `continue` 关键字用于跳出本次循环，其具体用法如下：

```
continue n
```

其中， `n` 表示循环层数，缺省值为 `1` 。即如果省略，则 `continue` 仅跳出其所在的循环语句，忽略本次循环当中剩余代码的执行，直接进入下一次循环。如果将 `n` 的值设置为 `2` ，那么 `continue` 会对内外两层的循环语句都有效，不但会跳出内层循环，还会跳出外层循环。`continue` 通常与 `if` 配合使用，在满足条件时跳出本次循环。

```
#!/bin/bash
# continue.sh

for((i=1; i<=5; i++)); do
  for((j=1; j<=5; j++)); do
    if((i*j==12)); then
      continue 2     # 跳出内外 2 层循环
    fi
    printf "%d*%d=%-4d" $i $j $((i*j))
  done
  printf "\n"
done
pi@raspberrypi:~ $ ./continue.sh

1*1=1   1*2=2   1*3=3   1*4=4   1*5=5
2*1=2   2*2=4   2*3=6   2*4=8   2*5=10
3*1=3   3*2=6   3*3=9   4*1=4   4*2=8   5*1=5   5*2=10  5*3=15  5*4=20  5*5=25
```

分支结构 case in esac

Shell 通过 `case in esac` 语句实现分支结构，该结构与 C 语言中的 `switch case` 语句非常类似。

```
case "$variable" in
  "$condition1")
    命令
  ;;
  "$condition2")
    命令
  ;;
esac
```

每个条件行都使用 `)` 结尾，每个条件块都以 `;;` 结尾（），关键字 `esac` 用于终止整个分支结构。下面示例脚本当中，会根据第 1 个命令行参数的值，分别打印对应的提示信息，当所有条件都不匹配时，最后会通过通配符 `*` 拦截执行流程，打印一条提示信息：

```
#!/bin/bash
# switch.sh

case "$1" in
  "A")
    echo "当前命令行参数为 A"
  ;;
  "B")
    echo "当前命令行参数为 B"
  ;;
  "C")
    echo "当前命令行参数为 C"
  ;;
  *)
    echo "当前命令行参数 A B C 都不是！"
  ;;
esac
pi@raspberrypi:~ $ ./switch.sh A
当前命令行参数为 A

pi@raspberrypi:~ $ ./switch.sh B
当前命令行参数为 B

pi@raspberrypi:~ $ ./switch.sh C
当前命令行参数为 C

pi@raspberrypi:~ $ ./switch.sh D
当前命令行参数 A B C 都不是!
```

> **更多我所制作的 UINIO 系列开源硬件，可以直接访问下面的链接地址来获取，包括全套原理图与 PCB 布线文件，以及相关的数据手册：**

UINIO 系列开源硬件概览
uinio.com/Project/Overview/


> **答主在成都的 IT 行业工作近十余年，经常会在自己的 电子技术博客 UinIO.com 当中分享一些产业与技术相关的文章，赠人玫瑰，手有余香，大家的【点赞、收藏、加关注】将会是我持续写作的最大动力。**

UinIO.com 电子技术博客
www.uinio.com

编辑于 2023-06-07 18:04 • IP 属地北京

shell 脚本　Shell 编程　Linux

写下你的评论...

1 条评论　默认　最新

M.myl
你们这都太难了 我想写个外挂能教教我吗
2020-09-06　回复　喜欢

文章被以下专栏收录

成都IT圈
聊聊成都 IT 那些事儿

推荐阅读

怎样用 Bash 编程：逻辑操作符和 shell 扩展

Linux... 发表于Linux...

怎样用 Bash 编程：语法和工具

Linux... 发表于Linux