read 命令 循环

条件判断

网道（WangDoc.com），互联网文档计划

本章介绍 Bash 脚本的条件判断语法。

目录 [隐藏]

1. if 结构
2. test 命令
3. 判断表达式
   - 3.1 文件判断
   - 3.2 字符串判断
   - 3.3 整数判断
   - 3.4 正则判断
   - 3.5 test 判断的逻辑运算
   - 3.6 算术判断
   - 3.7 普通命令的逻辑运算
4. case 结构
5. 参考链接

1. if 结构

`if` 是最常用的条件判断结构，只有符合给定条件时，才会执行指定的命令。它的语法如下。

Bash 脚本教程

1. 简介
2. 基本语法
3. 模式扩展
4. 引号和转义
5. 变量
6. 字符串操作
7. 算术运算
8. 操作历史
9. 行操作
10. 目录堆栈
11. 脚本入门
12. read 命令
13. 条件判断
14. 循环
15. 函数
16. 数组
17. set 命令，shopt 命令

```
if commands; then
  commands
[elif commands; then
  commands...]
[else
  commands]
fi
```

这个命令分成三个部分： `if` 、 `elif` 和 `else` 。其中，后两个部分是可选的。

`if` 关键字后面是主要的判断条件， `elif` 用来添加在主条件不成立时的其他判断条件， `else` 则是所有条件都不成立时要执行的部分。

```
if test $USER = "foo"; then
  echo "Hello foo."
else
  echo "You are not foo."
fi
```

上面的例子中，判断条件是环境变量 `$USER` 是否等于 `foo` ，如果等于就输出 `Hello foo.` ，否则输出其他内容。

`if` 和 `then` 写在同一行时，需要分号分隔。分号是 Bash 的命令分隔符。它们也可以写成两行，这时不需要分号。

```
if true
then
  echo 'hello world'
fi

if false
then
  echo 'it is false' # 本行不会执行
fi
```

上面的例子中， `true` 和 `false` 是两个特殊命令，前者代表操作成功，后者代表操作失败。 `if true` 意味着命令部分总是会执行， `if false` 意味着命令部分永远不会执行。

18. 脚本除错

19. mktemp 命令，trap 命令

20. 启动环境

21. 命令提示符

链接

本文源码

代码仓库

反馈

除了多行的写法， `if` 结构也可以写成单行。

```
$ if true; then echo 'hello world'; fi
hello world

$ if false; then echo "It's true."; fi
```

注意， `if` 关键字后面也可以是一条命令，该条命令执行成功（返回值 `0` ），就意味着判断条件成立。

```
$ if echo 'hi'; then echo 'hello world'; fi
hi
hello world
```

上面命令中， `if` 后面是一条命令 `echo 'hi'` 。该命令会执行，如果返回值是 `0` ，则执行 `then` 的部分。

`if` 后面可以跟任意数量的命令。这时，所有命令都会执行，但是判断真伪只看最后一个命令，即使前面所有命令都失败，只要最后一个命令返回 `0` ，就会执行 `then` 的部分。

```
$ if false; true; then echo 'hello world'; fi
hello world
```

上面例子中， `if` 后面有两条命令（ `false;true;` ），第二条命令（ `true` ）决定了 `then` 的部分是否会执行。

`elif` 部分可以有多个。

```
#!/bin/bash

echo -n "输入一个1到3之间的数字（包含两端）> "
read character
if [ "$character" = "1" ]; then
    echo 1
elif [ "$character" = "2" ]; then
    echo 2
elif [ "$character" = "3" ]; then
    echo 3
else
```

```
    echo 输入不符合要求
fi
```

上面例子中，如果用户输入 `3` ，就会连续判断3次。

2. test 命令

`if` 结构的判断条件，一般使用 `test` 命令，有三种形式。

```
# 写法一
test expression

# 写法二
[ expression ]

# 写法三
[[ expression ]]
```

上面三种形式是等价的，但是第三种形式还支持正则判断，前两种不支持。

上面的 `expression` 是一个表达式。这个表达式为真， `test` 命令执行成功（返回值为 `0` ）；表达式为伪， `test` 命令执行失败（返回值为 `1` ）。注意，第二种和第三种写法，`[` 和 `]` 与内部的表达式之间必须有空格。

```
$ test -f /etc/hosts
$ echo $?
0

$ [ -f /etc/hosts ]
$  echo $?
0
```

上面的例子中， `test` 命令采用两种写法，判断 `/etc/hosts` 文件是否存在，这两种写法是等价的。命令执行后，返回值为 `0` ，表示该文件确实存在。

实际上， `[` 这个字符是 `test` 命令的一种简写形式，可以看作是一个独立的命令，这解释了为什么它后面必须有空格。

下面把 `test` 命令的三种形式，用在 `if` 结构中，判断一个文件是否存在。

```
# 写法一
if test -e /tmp/foo.txt ; then
  echo "Found foo.txt"
fi

# 写法二
if [ -e /tmp/foo.txt ] ; then
  echo "Found foo.txt"
fi

# 写法三
if [[ -e /tmp/foo.txt ]] ; then
  echo "Found foo.txt"
fi
```

3. 判断表达式

`if` 关键字后面，跟的是一个命令。这个命令可以是 `test` 命令，也可以是其他命令。命令的返回值为 `0` 表示判断成立，否则表示不成立。因为这些命令主要是为了得到返回值，所以可以视为表达式。

常用的判断表达式有下面这些。

3.1 文件判断

以下表达式用来判断文件状态。

- `[ -a file ]`：如果 file 存在，则为 `true` 。
- `[ -b file ]`：如果 file 存在并且是一个块（设备）文件，则为 `true` 。
- `[ -c file ]`：如果 file 存在并且是一个字符（设备）文件，则为 `true` 。
- `[ -d file ]`：如果 file 存在并且是一个目录，则为 `true` 。

- `[ -e file ]`：如果 file 存在，则为 `true` 。
- `[ -f file ]`：如果 file 存在并且是一个普通文件，则为 `true` 。
- `[ -g file ]`：如果 file 存在并且设置了组 ID，则为 `true` 。
- `[ -G file ]`：如果 file 存在并且属于有效的组 ID，则为 `true` 。
- `[ -h file ]`：如果 file 存在并且是符号链接，则为 `true` 。
- `[ -k file ]`：如果 file 存在并且设置了它的“sticky bit”，则为 `true` 。
- `[ -L file ]`：如果 file 存在并且是一个符号链接，则为 `true` 。
- `[ -N file ]`：如果 file 存在并且自上次读取后已被修改，则为 `true` 。
- `[ -O file ]`：如果 file 存在并且属于有效的用户 ID，则为 `true` 。
- `[ -p file ]`：如果 file 存在并且是一个命名管道，则为 `true` 。
- `[ -r file ]`：如果 file 存在并且可读（当前用户有可读权限），则为 `true` 。
- `[ -s file ]`：如果 file 存在且其长度大于零，则为 `true` 。
- `[ -S file ]`：如果 file 存在且是一个网络 socket，则为 `true` 。
- `[ -t fd ]`：如果 fd 是一个文件描述符，并且重定向到终端，则为 `true` 。 这可以用来判断是否重定向了标准输入 / 输出 / 错误。
- `[ -u file ]`：如果 file 存在并且设置了 setuid 位，则为 `true` 。
- `[ -w file ]`：如果 file 存在并且可写（当前用户拥有可写权限），则为 `true` 。
- `[ -x file ]`：如果 file 存在并且可执行（有效用户有执行 / 搜索权限），则为 `true` 。
- `[ FILE1 -nt FILE2 ]`：如果 FILE1 比 FILE2 的更新时间更近，或者 FILE1 存在而 FILE2 不存在，则为 `true` 。

- `[ FILE1 -ot FILE2 ]`：如果 FILE1 比 FILE2 的更新时间更旧，或者 FILE2 存在而 FILE1 不存在，则为 `true` 。
- `[ FILE1 -ef FILE2 ]`：如果 FILE1 和 FILE2 引用相同的设备和 inode 编号，则为 `true` 。

下面是一个示例。

```
#!/bin/bash

FILE=~/.bashrc

if [ -e "$FILE" ]; then
  if [ -f "$FILE" ]; then
    echo "$FILE is a regular file."
  fi
  if [ -d "$FILE" ]; then
    echo "$FILE is a directory."
  fi
  if [ -r "$FILE" ]; then
    echo "$FILE is readable."
  fi
  if [ -w "$FILE" ]; then
    echo "$FILE is writable."
  fi
  if [ -x "$FILE" ]; then
    echo "$FILE is executable/searchable."
  fi
else
  echo "$FILE does not exist"
  exit 1
fi
```

上面代码中， `$FILE` 要放在双引号之中，这样可以防止变量 `$FILE` 为空，从而出错。因为 `$FILE` 如果为空，这时 `[ -e $FILE ]` 就变成 `[ -e ]` ，这会被判断为真。而 `$FILE` 放在双引号之中， `[ -e "$FILE" ]` 就变成 `[ -e "" ]` ，这会被判断为伪。

3.2 字符串判断

以下表达式用来判断字符串。

- `[ string ]`：如果 `string` 不为空（长度大于0），则判断为真。
- `[ -n string ]`：如果字符串 `string` 的长度大于零，则判断为真。
- `[ -z string ]`：如果字符串 `string` 的长度为零，则判断为真。
- `[ string1 = string2 ]`：如果 `string1` 和 `string2` 相同，则判断为真。
- `[ string1 == string2 ]` 等同于 `[ string1 = string2 ]`。
- `[ string1 != string2 ]`：如果 `string1` 和 `string2` 不相同，则判断为真。
- `[ string1 '>' string2 ]`：如果按照字典顺序 `string1` 排列在 `string2` 之后，则判断为真。
- `[ string1 '<' string2 ]`：如果按照字典顺序 `string1` 排列在 `string2` 之前，则判断为真。

注意，`test` 命令内部的 `>` 和 `<`，必须用引号引起来（或者是用反斜杠转义）。否则，它们会被 shell 解释为重定向操作符。

下面是一个示例。

```
#!/bin/bash

ANSWER=maybe

if [ -z "$ANSWER" ]; then
  echo "There is no answer." >&2
  exit 1
fi
if [ "$ANSWER" = "yes" ]; then
  echo "The answer is YES."
elif [ "$ANSWER" = "no" ]; then
  echo "The answer is NO."
elif [ "$ANSWER" = "maybe" ]; then
  echo "The answer is MAYBE."
else
```

```
  echo "The answer is UNKNOWN."
fi
```

上面代码中，首先确定 `$ANSWER` 字符串是否为空。如果为空，就终止脚本，并把退出状态设为 `1` 。注意，这里的 `echo` 命令把错误信息 `There is no answer.` 重定向到标准错误，这是处理错误信息的常用方法。如果 `$ANSWER` 字符串不为空，就判断它的值是否等于 `yes` 、 `no` 或者 `maybe` 。

注意，字符串判断时，变量要放在双引号之中，比如 `[ -n "$COUNT" ]` ，否则变量替换成字符串以后， `test` 命令可能会报错，提示参数过多。另外，如果不放在双引号之中，变量为空时，命令会变成 `[ -n ]` ，这时会判断为真。如果放在双引号之中， `[ -n "" ]` 就判断为伪。

3.3 整数判断

下面的表达式用于判断整数。

- `[ integer1 -eq integer2 ]` ：如果 `integer1` 等于 `integer2` ，则为 `true` 。
- `[ integer1 -ne integer2 ]` ：如果 `integer1` 不等于 `integer2` ，则为 `true` 。
- `[ integer1 -le integer2 ]` ：如果 `integer1` 小于或等于 `integer2` ，则为 `true` 。
- `[ integer1 -lt integer2 ]` ：如果 `integer1` 小于 `integer2` ，则为 `true` 。
- `[ integer1 -ge integer2 ]` ：如果 `integer1` 大于或等于 `integer2` ，则为 `true` 。
- `[ integer1 -gt integer2 ]` ：如果 `integer1` 大于 `integer2` ，则为 `true` 。

下面是一个用法的例子。

```
#!/bin/bash

INT=-5

if [ -z "$INT" ]; then
  echo "INT is empty." >&2
```

```
  exit 1
fi
if [ $INT -eq 0 ]; then
  echo "INT is zero."
else
  if [ $INT -lt 0 ]; then
    echo "INT is negative."
  else
    echo "INT is positive."
  fi
  if [ $((INT % 2)) -eq 0 ]; then
    echo "INT is even."
  else
    echo "INT is odd."
  fi
fi
```

上面例子中，先判断变量 `$INT` 是否为空，然后判断是否为 `0` ，接着判断正负，最后通过求余数判断奇偶。

3.4 正则判断

`[[ expression ]]` 这种判断形式，支持正则表达式。

```
[[ string1 =~ regex ]]
```

上面的语法中， `regex` 是一个正则表示式， `=~` 是正则比较运算符。

下面是一个例子。

```
#!/bin/bash

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
  echo "INT is an integer."
  exit 0
else
  echo "INT is not an integer." >&2
```

```
  exit 1
fi
```

上面代码中，先判断变量 `INT` 的字符串形式，是否满足 `^-?[0-9]+$` 的正则模式，如果满足就表明它是一个整数。

3.5 test 判断的逻辑运算

通过逻辑运算，可以把多个 `test` 判断表达式结合起来，创造更复杂的判断。三种逻辑运算 `AND` ， `OR` ，和 `NOT` ，都有自己的专用符号。

- `AND` 运算：符号 `&&` ，也可使用参数 `-a` 。
- `OR` 运算：符号 `||` ，也可使用参数 `-o` 。
- `NOT` 运算：符号 `!` 。

下面是一个 `AND` 的例子，判断整数是否在某个范围之内。

```
#!/bin/bash

MIN_VAL=1
MAX_VAL=100

INT=50

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
  if [[ $INT -ge $MIN_VAL && $INT -le $MAX_VAL ]];
    echo "$INT is within $MIN_VAL to $MAX_VAL."
  else
    echo "$INT is out of range."
  fi
else
  echo "INT is not an integer." >&2
  exit 1
fi
```

上面例子中， `&&` 用来连接两个判断条件：大于等于 `$MIN_VAL` ，并且小于等于 `$MAX_VAL` 。

使用否定操作符 `!` 时，最好用圆括号确定转义的范围。

```
if [ ! \( $INT -ge $MIN_VAL -a $INT -le $MAX_VAL \
    echo "$INT is outside $MIN_VAL to $MAX_VAL."
else
    echo "$INT is in range."
fi
```

上面例子中， `test` 命令内部使用的圆括号，必须使用引号或者转义，否则会被 Bash 解释。

使用 `-a` 连接两个判断条件不太直观，一般推荐使用 `&&` 代替，上面的脚本可以改写成下面这样。

```
if !([ $INT -ge $MIN_VAL ] && [ $INT -le $MAX_VAL
  echo "$INT is outside $MIN_VAL to $MAX_VAL."
else
  echo "$INT is in range."
fi
```

3.6 算术判断

Bash 还提供了 `((...))` 作为算术条件，进行算术运算的判断。

```
if ((3 > 2)); then
  echo "true"
fi
```

上面代码执行后，会打印出 `true` 。

注意，算术判断不需要使用 `test` 命令，而是直接使用 `((...))` 结构。这个结构的返回值，决定了判断的真伪。

如果算术计算的结果是非零值，则表示判断成立。这一点跟命令的返回值正好相反，需要小心。

```
$ if ((1)); then echo "It is true."; fi
It is true.
```

```
$ if ((0)); then echo "It is true."; else echo "it
It is false.
```

上面例子中，`((1))` 表示判断成立，`((0))` 表示判断不成立。

算术条件 `((...))` 也可以用于变量赋值。

```
$ if (( foo = 5 ));then echo "foo is $foo"; fi
foo is 5
```

上面例子中，`(( foo = 5 ))` 完成了两件事情。首先把 `5` 赋值给变量 `foo`，然后根据返回值 `5`，判断条件为真。

注意，赋值语句返回等号右边的值，如果返回的是 `0`，则判断为假。

```
$ if (( foo = 0 ));then echo "It is true.";else ec
It is false.
```

下面是用算术条件改写的数值判断脚本。

```
#!/bin/bash

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
  if ((INT == 0)); then
    echo "INT is zero."
  else
    if ((INT < 0)); then
      echo "INT is negative."
    else
      echo "INT is positive."
    fi
    if (( ((INT % 2)) == 0)); then
      echo "INT is even."
    else
      echo "INT is odd."
    fi
```

```
  fi
else
  echo "INT is not an integer." >&2
  exit 1
fi
```

只要是算术表达式，都能用于 `((...))` 语法，详见《Bash 的算术运算》一章。

3.7 普通命令的逻辑运算

如果 `if` 结构使用的不是 `test` 命令，而是普通命令，比如上一节的 `((...))` 算术运算，或者 `test` 命令与普通命令混用，那么可以使用 Bash 的命令控制操作符 `&&` （AND）和 `||` （OR），进行多个命令的逻辑运算。

```
$ command1 && command2
$ command1 || command2
```

对于 `&&` 操作符，先执行 `command1` ，只有 `command1` 执行成功后， 才会执行 `command2` 。对于 `||` 操作符，先执行 `command1` ，只有 `command1` 执行失败后， 才会执行 `command2` 。

```
$ mkdir temp && cd temp
```

上面的命令会创建一个名为 `temp` 的目录，执行成功后，才会执行第二个命令，进入这个目录。

```
$ [ -d temp ] || mkdir temp
```

上面的命令会测试目录 `temp` 是否存在，如果不存在，就会执行第二个命令，创建这个目录。这种写法非常有助于在脚本中处理错误。

```
[ ! -d temp ] && exit 1
```

上面的命令中，如果 `temp` 子目录不存在，脚本会终止，并且返回值为 `1` 。

下面就是 `if` 与 `&&` 结合使用的写法。

```
if [ condition ] && [ condition ]; then
  command
fi
```

下面是一个示例。

```
#! /bin/bash

filename=$1
word1=$2
word2=$3

if grep $word1 $filename && grep $word2 $filename
then
  echo "$word1 and $word2 are both in $filename."
fi
```

上面的例子只有在指定文件里面，同时存在搜索词 `word1` 和 `word2` ，就会执行 `if` 的命令部分。

下面的示例演示如何将一个 `&&` 判断表达式，改写成对应的 `if` 结构。

```
[[ -d "$dir_name" ]] && cd "$dir_name" && rm *

# 等同于

if [[ ! -d "$dir_name" ]]; then
  echo "No such directory: '$dir_name'" >&2
  exit 1
fi
if ! cd "$dir_name"; then
  echo "Cannot cd to '$dir_name'" >&2
  exit 1
fi
if ! rm *; then
```

```
  echo "File deletion failed. Check results" >&2
  exit 1
fi
```

4. case 结构

`case` 结构用于多值判断，可以为每个值指定对应的命令，跟包含多个 `elif` 的 `if` 结构等价，但是语义更好。它的语法如下。

```
case expression in
  pattern )
    commands ;;
  pattern )
    commands ;;
  ...
esac
```

上面代码中， `expression` 是一个表达式， `pattern` 是表达式的值或者一个模式，可以有多条，用来匹配多个值，每条以两个分号（ `;` ）结尾。

```
#!/bin/bash

echo -n "输入一个1到3之间的数字（包含两端）> "
read character
case $character in
  1 ) echo 1
    ;;
  2 ) echo 2
    ;;
  3 ) echo 3
    ;;
  * ) echo 输入不符合要求
esac
```

上面例子中，最后一条匹配语句的模式是 `*` ，这个通配符可以匹配其他字符和没有输入字符的情况，类似 `if` 的 `else`

部分。

下面是另一个例子。

```
#!/bin/bash

OS=$(uname -s)

case "$OS" in
  FreeBSD) echo "This is FreeBSD" ;;
  Darwin) echo "This is Mac OSX" ;;
  AIX) echo "This is AIX" ;;
  Minix) echo "This is Minix" ;;
  Linux) echo "This is Linux" ;;
  *) echo "Failed to identify this OS" ;;
esac
```

上面的例子判断当前是什么操作系统。

`case` 的匹配模式可以使用各种通配符，下面是一些例子。

- `a)` ：匹配 `a` 。
- `a|b)` ：匹配 `a` 或 `b` 。
- `[[:alpha:]])` ：匹配单个字母。
- `???)` ：匹配3个字符的单词。
- `*.txt)` ：匹配 `.txt` 结尾。
- `*)` ：匹配任意输入，通过作为 `case` 结构的最后一个模式。

```
#!/bin/bash

echo -n "输入一个字母或数字 > "
read character
case $character in
  [[:lower:]] | [[:upper:]] ) echo "输入了字母 $char
                              ;;
  [0-9] )                     echo "输入了数字 $char
                              ;;
  * )                         echo "输入不符合要求"
esac
```

上面例子中，使用通配符 `[[:lower:]] | [[:upper:]]` 匹配字母，`[0-9]` 匹配数字。

Bash 4.0之前，`case` 结构只能匹配一个条件，然后就会退出 `case` 结构。Bash 4.0之后，允许匹配多个条件，这时可以用 `;;&` 终止每个条件块。

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

read -n 1 -p "Type a character > "
echo
case $REPLY in
  [[:upper:]])    echo "'$REPLY' is upper case." ;
  [[:lower:]])    echo "'$REPLY' is lower case." ;
  [[:alpha:]])    echo "'$REPLY' is alphabetic." ;
  [[:digit:]])    echo "'$REPLY' is a digit." ;;&
  [[:graph:]])    echo "'$REPLY' is a visible char
  [[:punct:]])    echo "'$REPLY' is a punctuation
  [[:space:]])    echo "'$REPLY' is a whitespace c
  [[:xdigit:]])   echo "'$REPLY' is a hexadecimal
esac
```

执行上面的脚本，会得到下面的结果。

```
$ test.sh
Type a character > a
'a' is lower case.
'a' is alphabetic.
'a' is a visible character.
'a' is a hexadecimal digit.
```

可以看到条件语句结尾添加了 `;;&` 以后，在匹配一个条件之后，并没有退出 `case` 结构，而是继续判断下一个条件。

5. 参考链接

- The Linux Command Line, William Shotts

分享本文 

联系：contact@wangdoc.com