目录堆栈 read 命令

Bash 脚本入门

网道（WangDoc.com），互联网文档计划

脚本（script）就是包含一系列命令的一个文本文件。Shell 读取这个文件，依次执行里面的所有命令，就好像这些命令直接输入到命令行一样。所有能够在命令行完成的任务，都能够用脚本完成。

脚本的好处是可以重复使用，也可以指定在特定场合自动调用，比如系统启动或关闭时自动执行脚本。

目录 [隐藏]

1. Shebang 行

脚本的第一行通常是指定解释器，即这个脚本必须通过什么解释器执行。这一行以 `#!` 字符开头，这个字符称为 Shebang，所以这一行就叫做 Shebang 行。

`#!` 后面就是脚本解释器的位置，Bash 脚本的解释器一般是 `/bin/sh` 或 `/bin/bash` 。

```
#!/bin/sh
# 或者
#!/bin/bash
```

`#!` 与脚本解释器之间有没有空格，都是可以的。

如果 Bash 解释器不放在目录 `/bin` ，脚本就无法执行了。为了保险，可以写成下面这样。

```
#!/usr/bin/env bash
```

上面命令使用 `env` 命令（这个命令总是在 `/usr/bin` 目录），返回 Bash 可执行文件的位置。 `env` 命令的详细介绍，请看后文。

Shebang 行不是必需的，但是建议加上这行。如果缺少该行，就需要手动将脚本传给解释器。举例来说，脚本是 `script.sh` ，有 Shebang 行的时候，可以直接调用执行。

```
$ ./script.sh
```

上面例子中， `script.sh` 是脚本文件名。脚本通常使用 `.sh` 后缀名，不过这不是必需的。

如果没有 Shebang 行，就只能手动将脚本传给解释器来执行。

```
$ /bin/sh ./script.sh
# 或者
$ bash ./script.sh
```


链接

本文源码

代码仓库

反馈

2. 执行权限和路径

前面说过，只要指定了 Shebang 行的脚本，可以直接执行。这有一个前提条件，就是脚本需要有执行权限。可以使用下面的命令，赋予脚本执行权限。

```
# 给所有用户执行权限
$ chmod +x script.sh

# 给所有用户读权限和执行权限
$ chmod +rx script.sh
# 或者
$ chmod 755 script.sh

# 只给脚本拥有者读权限和执行权限
$ chmod u+rx script.sh
```

脚本的权限通常设为 `755` （拥有者有所有权限，其他人有读和执行权限）或者 `700` （只有拥有者可以执行）。

除了执行权限，脚本调用时，一般需要指定脚本的路径（比如 `path/script.sh` ）。如果将脚本放在环境变量 `$PATH` 指定的目录中，就不需要指定路径了。因为 Bash 会自动到这些目录中，寻找是否存在同名的可执行文件。

建议在主目录新建一个 `~/bin` 子目录，专门存放可执行脚本，然后把 `~/bin` 加入 `$PATH` 。

```
export PATH=$PATH:~/bin
```

上面命令改变环境变量 `$PATH` ，将 `~/bin` 添加到 `$PATH` 的末尾。可以将这一行加到 `~/.bashrc` 文件里面，然后重新加载一次 `.bashrc` ，这个配置就可以生效了。

```
$ source ~/.bashrc
```

以后不管在什么目录，直接输入脚本文件名，脚本就会执行。

```
$ script.sh
```

上面命令没有指定脚本路径，因为 `script.sh` 在 `$PATH` 指定的目录中。

3. env 命令

`env` 命令总是指向 `/usr/bin/env` 文件，或者说，这个二进制文件总是在目录 `/usr/bin` 。

`#!/usr/bin/env NAME` 这个语法的意思是，让 Shell 查找 `$PATH` 环境变量里面第一个匹配的 `NAME` 。如果你不知道某个命令的具体路径，或者希望兼容其他用户的机器，这样的写法就很有用。

`/usr/bin/env bash` 的意思就是，返回 `bash` 可执行文件的位置，前提是 `bash` 的路径是在 `$PATH` 里面。其他脚本文件也可以使用这个命令。比如 Node.js 脚本的 Shebang 行，可以写成下面这样。

```
#!/usr/bin/env node
```

`env` 命令的参数如下。

- `-i` ， `--ignore-environment` ： 不带环境变量启动。
- `-u` ， `--unset=NAME` ： 从环境变量中删除一个变量。
- `--help` ： 显示帮助。
- `--version` ： 输出版本信息。

下面是一个例子，新建一个不带任何环境变量的 Shell。

```
$ env -i /bin/sh
```

4. 注释

Bash 脚本中， `#` 表示注释，可以放在行首，也可以放在行尾。

```
# 本行是注释
echo 'Hello World!'

echo 'Hello World!' # 井号后面的部分也是注释
```

建议在脚本开头，使用注释说明当前脚本的作用，这样有利于日后的维护。

5. 脚本参数

调用脚本的时候，脚本文件名后面可以带有参数。

```
$ script.sh word1 word2 word3
```

上面例子中， `script.sh` 是一个脚本文件， `word1` 、 `word2` 和 `word3` 是三个参数。

脚本文件内部，可以使用特殊变量，引用这些参数。

- `$0` ：脚本文件名，即 `script.sh` 。
- `$1` ~ `$9` ：对应脚本的第一个参数到第九个参数。
- `$#` ：参数的总数。
- `$@` ：全部的参数，参数之间使用空格分隔。
- `$*` ：全部的参数，参数之间使用变量 `$IFS` 值的第一个字符分隔，默认为空格，但是可以自定义。

如果脚本的参数多于9个，那么第10个参数可以用 `${10}` 的形式引用，以此类推。

注意，如果命令是 `command -o foo bar` ，那么 `-o` 是 `$1` ， `foo` 是 `$2` ， `bar` 是 `$3` 。

下面是一个脚本内部读取命令行参数的例子。

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

```
echo "全部参数：" $@
echo "命令行参数数量：" $#
echo '$0 = ' $0
echo '$1 = ' $1
echo '$2 = ' $2
echo '$3 = ' $3
```

执行结果如下。

```
$ ./script.sh a b c
全部参数：a b c
命令行参数数量：3
$0 =  script.sh
$1 =  a
$2 =  b
$3 =  c
```

用户可以输入任意数量的参数，利用 `for` 循环，可以读取每一个参数。

```
#!/bin/bash

for i in "$@"; do
  echo $i
done
```

上面例子中， `$@` 返回一个全部参数的列表，然后使用 `for` 循环遍历。

如果多个参数放在双引号里面，视为一个参数。

```
$ ./script.sh "a b"
```

上面例子中，Bash 会认为 `"a b"` 是一个参数， `$1` 会返回 `a b` 。注意，返回时不包括双引号。

6. shift 命令

shift 命令可以改变脚本参数，每次执行都会移除脚本当前的第一个参数（ $1 ），使得后面的参数向前一位，即 $2 变成 $1 、 $3 变成 $2 、 $4 变成 $3 ，以此类推。

while 循环结合 shift 命令，也可以读取每一个参数。

```
#!/bin/bash

echo "一共输入了 $# 个参数"

while [ "$1" != "" ]; do
  echo "剩下 $# 个参数"
  echo "参数: $1"
  shift
done
```

上面例子中， shift 命令每次移除当前第一个参数，从而通过 while 循环遍历所有参数。

shift 命令可以接受一个整数作为参数，指定所要移除的参数个数，默认为 1 。

```
shift 3
```

上面的命令移除前三个参数，原来的 $4 变成 $1 。

7. getopts 命令

getopts 命令用在脚本内部，可以解析复杂的脚本命令行参数，通常与 while 循环一起使用，取出脚本所有的带有前置连词线（ - ）的参数。

```
getopts optstring name
```

它带有两个参数。第一个参数 optstring 是字符串，给出脚本所有的连词线参数。比如，某个脚本可以有三个配置项参数 -l 、 -h 、 -a ，其中只有 -a 可以带有参数值，而 -l 和 -h 是开关参数，那么 getopts 的第一个参数写成 lha: ，顺序不重要。注意， a 后面有一个冒号，表示该参

数带有参数值， `getopts` 规定带有参数值的配置项参数，后面必须带有一个冒号（ `:` ）。 `getopts` 的第二个参数 `name` 是一个变量名，用来保存当前取到的配置项参数，即 `l` 、 `h` 或 `a` 。

下面是一个例子。

```
while getopts 'lha:' OPTION; do
  case "$OPTION" in
    l)
      echo "linuxconfig"
      ;;

    h)
      echo "h stands for h"
      ;;

    a)
      avalue="$OPTARG"
      echo "The value provided is $OPTARG"
      ;;
    ?)
      echo "script usage: $(basename $0) [-l] [-h]
      exit 1
      ;;
  esac
done
shift "$(($OPTIND - 1))"
```

上面例子中， `while` 循环不断执行 `getopts 'lha:' OPTION` 命令，每次执行就会读取一个连词线参数（以及对应的参数值），然后进入循环体。变量 `OPTION` 保存的是，当前处理的那一个连词线参数（即 `l` 、 `h` 或 `a` ）。如果用户输入了没有指定的参数（比如 `-x` ），那么 `OPTION` 等于 `?` 。循环体内使用 `case` 判断，处理这四种不同的情况。

如果某个连词线参数带有参数值，比如 `-a foo` ，那么处理 `a` 参数的时候，环境变量 `$OPTARG` 保存的就是参数值。

注意，只要遇到不带连词线的参数， `getopts` 就会执行失败，从而退出 `while` 循环。比如， `getopts` 可以解析 `command -l foo` ，但不可以解析 `command foo -l` 。另外，

多个连词线参数写在一起的形式，比如 `command -lh` ，`getopts` 也可以正确处理。

变量 `$OPTIND` 在 `getopts` 开始执行前是 `1` ，然后每次执行就会加 `1` 。等到退出 `while` 循环，就意味着连词线参数全部处理完毕。这时，`$OPTIND - 1` 就是已经处理的连词线参数个数，使用 `shift` 命令将这些参数移除，保证后面的代码可以用 `$1` 、`$2` 等处理命令的主参数。

8. 配置项参数终止符 `--`

`-` 和 `--` 开头的参数，会被 Bash 当作配置项解释。但是，有时它们不是配置项，而是实体参数的一部分，比如文件名叫做 `-f` 或 `--file` 。

```
$ cat -f
$ cat --file
```

上面命令的原意是输出文件 `-f` 和 `--file` 的内容，但是会被 Bash 当作配置项解释。

这时就可以使用配置项参数终止符 `--` ，它的作用是告诉 Bash，在它后面的参数开头的 `-` 和 `--` 不是配置项，只能当作实体参数解释。

```
$ cat -- -f
$ cat -- --file
```

上面命令可以正确展示文件 `-f` 和 `--file` 的内容，因为它们放在 `--` 的后面，开头的 `-` 和 `--` 就不再当作配置项解释了。

如果要确保某个变量不会被当作配置项解释，就要在它前面放上参数终止符 `--` 。

```
$ ls -- $myPath
```

上面示例中， `--` 强制变量 `$myPath` 只能当作实体参数（即路径名）解释。如果变量不是路径名，就会报错。

```
$ myPath="-l"
$ ls -- $myPath
ls: 无法访问'-l': 没有那个文件或目录
```

上面例子中，变量 `myPath` 的值为 `-l` ，不是路径。但是， `--` 强制 `$myPath` 只能作为路径解释，导致报错"不存在该路径"。

下面是另一个实际的例子，如果想在文件里面搜索 `--hello` ，这时也要使用参数终止符 `--` 。

```
$ grep -- "--hello" example.txt
```

上面命令在 `example.txt` 文件里面，搜索字符串 `--hello` 。这个字符串是 `--` 开头，如果不用参数终止符， `grep` 命令就会把 `--hello` 当作配置项参数，从而报错。

9. exit 命令

`exit` 命令用于终止当前脚本的执行，并向 Shell 返回一个退出值。

```
$ exit
```

上面命令中止当前脚本，将最后一条命令的退出状态，作为整个脚本的退出状态。

`exit` 命令后面可以跟参数，该参数就是退出状态。

```
# 退出值为0（成功）
$ exit 0

# 退出值为1（失败）
$ exit 1
```

退出时，脚本会返回一个退出值。脚本的退出值， `0` 表示正常， `1` 表示发生错误， `2` 表示用法不对， `126` 表示不是可执行脚本， `127` 表示命令没有发现。如果脚本被信号 `N` 终止，则退出值为 `128 + N` 。简单来说，只要退出值非0，就认为执行出错。

下面是一个例子。

```
if [ $(id -u) != "0" ]; then
  echo "根用户才能执行当前脚本"
  exit 1
fi
```

上面的例子中， `id -u` 命令返回用户的 ID，一旦用户的 ID 不等于 `0` （根用户的 ID），脚本就会退出，并且退出码为 `1` ，表示运行失败。

`exit` 与 `return` 命令的差别是， `return` 命令是函数的退出，并返回一个值给调用者，脚本依然执行。 `exit` 是整个脚本的退出，如果在函数之中调用 `exit` ，则退出函数，并终止脚本执行。

10. 命令执行结果

命令执行结束后，会有一个返回值。 `0` 表示执行成功，非 `0` （通常是 `1` ）表示执行失败。环境变量 `$?` 可以读取前一个命令的返回值。

利用这一点，可以在脚本中对命令执行结果进行判断。

```
cd /path/to/somewhere
if [ "$?" = "0" ]; then
  rm *
else
  echo "无法切换目录！" 1>&2
  exit 1
fi
```

上面例子中， `cd /path/to/somewhere` 这个命令如果执行成功（返回值等于 `0` ），就删除该目录里面的文件，否则退出

脚本，整个脚本的返回值变为 `1` ，表示执行失败。

由于 `if` 可以直接判断命令的执行结果，执行相应的操作，上面的脚本可以改写成下面的样子。

```
if cd /path/to/somewhere; then
  rm *
else
  echo "Could not change directory! Aborting." 1>&
  exit 1
fi
```

更简洁的写法是利用两个逻辑运算符 `&&` （且）和 `||` （或）。

```
# 第一步执行成功，才会执行第二步
cd /path/to/somewhere && rm *

# 第一步执行失败，才会执行第二步
cd /path/to/somewhere || exit 1
```

11. source 命令

`source` 命令用于执行一个脚本，通常用于重新加载一个配置文件。

```
$ source .bashrc
```

`source` 命令最大的特点是在当前 Shell 执行脚本，不像直接执行脚本时，会新建一个子 Shell。所以， `source` 命令执行脚本时，不需要 `export` 变量。

```
#!/bin/bash
# test.sh
echo $foo
```

上面脚本输出 `$foo` 变量的值。

```
# 当前 Shell 新建一个变量 foo
$ foo=1

# 打印输出 1
$ source test.sh
1

# 打印输出空字符串
$ bash test.sh
```

上面例子中，当前 Shell 的变量 `foo` 并没有 `export` ，所以直接执行无法读取，但是 `source` 执行可以读取。

`source` 命令的另一个用途，是在脚本内部加载外部库。

```
#!/bin/bash

source ./lib.sh

function_from_lib
```

上面脚本在内部使用 `source` 命令加载了一个外部库，然后就可以在脚本里面，使用这个外部库定义的函数。

`source` 有一个简写形式，可以使用一个点（ `.` ）来表示。

```
$ . .bashrc
```

12. 别名，alias 命令

`alias` 命令用来为一个命令指定别名，这样更便于记忆。下面是 `alias` 的格式。

```
alias NAME=DEFINITION
```

上面命令中， `NAME` 是别名的名称， `DEFINITION` 是别名对应的原始命令。注意，等号两侧不能有空格，否则会报错。

一个常见的例子是为 `grep` 命令起一个 `search` 的别名。

```
alias search=grep
```

`alias` 也可以用来为长命令指定一个更短的别名。下面是通过别名定义一个 `today` 的命令。

```
$ alias today='date +"%A, %B %-d, %Y"'
$ today
星期一, 一月 6, 2020
```

有时为了防止误删除文件，可以指定 `rm` 命令的别名。

```
$ alias rm='rm -i'
```

上面命令指定 `rm` 命令是 `rm -i` ，每次删除文件之前，都会让用户确认。

`alias` 定义的别名也可以接受参数，参数会直接传入原始命令。

```
$ alias echo='echo It says: '
$ echo hello world
It says: hello world
```

上面例子中，别名定义了 `echo` 命令的前两个参数，等同于修改了 `echo` 命令的默认行为。

指定别名以后，就可以像使用其他命令一样使用别名。一般来说，都会把常用的别名写在 `~/.bashrc` 的末尾。另外，只能为命令定义别名，为其他部分（比如很长的路径）定义别名是无效的。

直接调用 `alias` 命令，可以显示所有别名。

```
$ alias
```

`unalias` 命令可以解除别名。

```
$ unalias lt
```

13. 参考链接

- How to use getopts to parse a script options, Egidio Docile

目录堆栈 read 命令

分享本文 

联系：contact@wangdoc.com