学习报告

唐浩
信息与计算科学　3200102118

2022 年 6 月 28 日

1　shell 文件运行

- shell 脚本文件内容:

```
#!/bin/sh

salutation=" Hello"
echo $salutation
echo  "The program $0 is now running"
echo  "The second parameter was $2"
echo  "The first parameter was $1"
echo  "The parameter list was $* "
echo  "The user' s home directory is $HOME"
echo  "Please enter a new greeting"
read salutation
echo $salutation
echo  "The script is now complete"
exit 0
```

- 尝试运行:

```
sh try.sh
```

- 运行结果如下:

```
Hello
The program ./try_var is now running
The second parameter was bar
```

```
The first parameter was foo
The parameter list was foo bar baz
The user' s home directory is /home/rick
Please enter a new greeting
Sire
Sire
The script is now complete
```

- 工作原理:

操纵参数和环境变量;
该脚本创建变量salutation, 显示其内容;
然后显示各种参数变量和环境变量$HOME是如何存在并具有适当值的。

2 个人理解

对于#!/bin/sh,虽然shell编程是以 为注释,但是"#!/bin/sh" 却不是。它是对shell的声明,说是指此脚本使用/bin/sh 来解释执行,#!是特殊的表示符,其后面跟的是解释此脚本的shell的路径)如果没有声明, 则脚本将在默认的 shell 中执行, 默认 shell 是由用户所在的系统定义为执行 shell 脚本的 shell. 如果脚本被编写为在 Kornshell ksh 中运行, 而默认运行 shell 脚本的为 C shell csh, 则脚本在执行过程中很可能失败。所以建议大家就把 "#!/bin/sh" 当成C 语言的main函数一样,写shell必须有,以使shell程序更严密

echo 用于显示字符串, 直接在 echo 后面加上想要显示的内容就好。在 echo 的后面, 不仅可以加字符串, 还可以加变量名:

```
#定义变量str
str = "Hello world"

#在echo后加上str变量, 一样可以显示出来
echr "$str, good morning"

Hello world, good morning
```

2.1 管道和重定向

比如: `ls -l` 是列出当前目录下的全部文件, 但这样会导致一屏幕可能放不下. `more` 命令则是将一个超出一屏幕的输出, 在输出一满屏内容后暂停, 然后可以逐行 (按回车) 输出. 将两个命令用 | 管道 (pipe) 链接在一起: `ls -l | more` 就是将当前目录下所有文件列出, 按屏停顿.

我们之前还有一个例子, `env` 实际上是输出全部环境变量, 而 `grep` 是过滤的意思, 所以 `env | grep PATH` 就是输出全部环境变量中包含 `PATH` 内容的东西.

这些组合都可以灵活搭配使用. UNIX 哲学在所有分支中都是通行的.

大家可能注意到, shell 本身也是一个命令. 事实上, 目前流行的 shell 有多个版本, 我们使用的是 bash, 这也是开源社区最通行的版本. 使用 Mac OS 的同学可能注意到 Mac 的默认 shell 不是 bash. 这个命令本身的位置在 `/bin/bash`. Linux 的命令实际上在 shell 中运行和反馈, shell 则提供 pipe 和环境变量等等这样的运行环境. 除了 pipe, shell 还有一个重要特性是重定向 (redirection).

我们知道, C 语言中输出, 用 `printf` 函数, 本质上是向一个被称为 `stdout` 的设备文件输出字符流. `printf` 和 `stdout` 都在头文件 `stdio.h` 定义. 在正常情况下, `stdout` 是定义在显示器上的. 但我们可以用重定向, 将其重定向为一个文本文件. 比如:

```
ls -l > re.txt
```

将当前目录的内容重定向到 `re.txt` 下. 这里 `>` 每一次都会重新生成一个新的输出结果文件. 而 `>>` 则是在文件结尾新增.

```
ls -l >> re.txt
```

有的时候, 我们写了一个程序, 会有大量的输出, 如果从屏幕走, 记录本身就很麻烦. 这时你没有必要专门做一套文本文件的读写机制, 而只需要重定向就行了. 除了 stdout 可以重定向, stderr 也是可以重定向的. 为了区别, 我们可以分别用 1(可以忽略) 和 2 表示这两个不同的通道. 比如:

```
./count 10 >std.txt 2>err.txt
```

或者等价地

```
./count 10 1>std.txt 2>err.txt
```

注意这里 `1>` 和 `2>` 之间不要有空格, 否则会和命令行参数混淆.

这几件规则可以结合在一起发挥作用. 比如:

```
./count 10 | grep count
```

这里你仔细观察, 就会发现, 管道只接管了 `std`, 而且管道之后的命令的执行和 `count` 是同步的. 两边的输出甚至可能交错. 这里涉及到进程并行的问题. 我们先不讨论. 所以我们可以先清理一下输出, 比如把 `stderr` 先重定向掉.

```
./count 10 2>/dev/null | grep count
```

2.2 shell 编程

我们可以做的更过份一点. 比如我们知道在 `/usr/include` 下放了很多 C 语言的头文件, 比如 stdio.h 就在这里. 现在我们想看一下这些头文件中有多少包含了 `stdout` 这个关键字. 比如我们想看一下这个文件是在哪个文件定义的. 那么我们可以用命令:

```
grep stdout /usr/include/*
```

找到这些文件, 然后记下来, 再一个个文件去读. 这就有点蠢了. 就不能一次搞定? 找到这些包含 `stdout` 的头文件, 找到一个, 读一个, 这样就很快能发现是谁定义, 还能马上看到定义周围的细节. 要做到这件事, 我们可以写一个段小的脚本:

```
$ for file in /usr/include/*
> do
> if grep -l stdio $file
> then
> more $file
> fi
> done
```

这样就可以愉快地做到了. 注意我们可以用上箭头调用上一次运行的命令, 用 TAB 键自动补齐一个最有可能的命令. 我们这里用一下上箭头, 你会看到原来刚才那一长串都算一个命令.

考虑到这个命令输入太麻烦, 我们不如把它做成一个可执行脚本算了. 查看脚本 `first`.