
本章学习要点:

- ### 3.1 Shell

3.1.1 Shell 的基本概念

Shell 就是用户与操作系统内核之间的接口，起着协调用户与系统的一致性和在用户与系统之间进行交互的作用。Shell 在 Linux 系统中具有极其重要的地位，如图 3-1 所示



Shell 最重要的功能是命令解释，从这种意义上来说，Shell 是一个命令解释器。Linux 系统中的所有可执行文件都可以作为 Shell 命令来执行。将可执行文件作一个分类，如表 4-1

所示。

表 4-1 可执行文件的分类

类别	说明
Linux 命令	存放在/bin、/sbin 目录下
内置命令	出于效率的考虑，将一些常用命令的解释程序构造在 Shell 内部
实用程序	存放在/usr/bin、/usr/sbin、/usr/local/bin 等目录下的实用程序
用户程序	用户程序经过编译生成可执行文件后，也可作为 Shell 命令运行
Shell 脚本	由 Shell 语言编写的批处理文件

当用户提交了一个命令后，Shell 首先判断它是否为内置命令，如果是就通过 Shell 内部的解释器将其解释为系统功能调用并转交给内核执行；若是外部命令或实用程序就试图在硬盘中查找该命令并将其调入内存，再将其解释为系统功能调用并转交给内核执行。在查找该命令时分为两种情况：

用户给出了命令路径，Shell 就沿着用户给出的路径查找，若找到则调入内存，若没有则输出提示信息；

用户没有给出命令的路径，Shell 就在环境变量 PATH 所制定的路径中依次进行查找，若找到则调入内存，若没找到则输出提示信息。

图 3-2 描述了 Shell 是如何完成命令解释的。

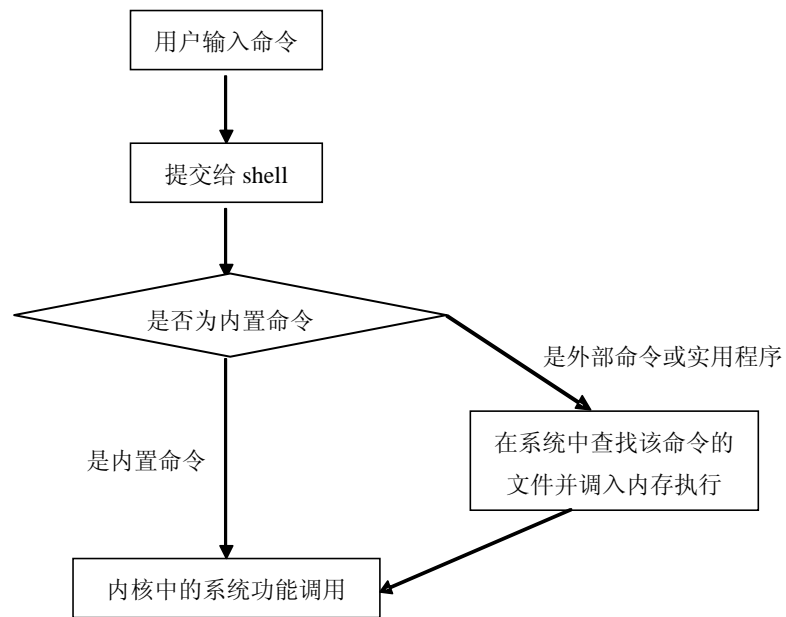


图 3-2 Shell 执行命令解释的过程

此外，Shell 还具有如下的一些功能：

- SHELL 环境变量
- 正则表达式
- 输入输出重定向与管道

3. Shell 的主要版本

表 4-2 列出了几种常见的 Shell 版本。

表 4-2 Shell 的不同版本

版本	说明
Bourne Again Shell (Bash. bsh 的扩展)	Bash 是大多数 Linux 系统的默认 Shell。Bash 与 bsh 完全向后兼容，并且在 bsh 的基础上增加和增强了很多特性。Bash 也包含了很多 C Shell 和 Korn Shell 中的优点。Bash 有很灵活和强大的编程接口，同时又有友好的用户界面。
Korn Shell (ksh)	Korn Shell (ksh)由 Dave Korn 所写。它是 UNIX 系统上的标准 Shell。另外，在 Linux 环境下有一个专门为 Linux 系统编写的 Korn Shell 的扩展版本，即 Public Domain.Korn Shell (pdksh)。
tcsh (csh 的扩展)	tcsh 是 C. Shell 的扩展。tcsh 与 csh 完全向后兼容，但它包含了更多的使用户感觉方便的新特性，其最大的提高是在命令行编辑和历史浏览方面。

3.1.2 Shell 环境变量

Shell 支持具有字符串值的变量。Shell 变量不需要专门的说明语句，通过赋值语句完成变量说明并予以赋值。在命令行或 Shell 脚本文件中使用\$name 的形式引用变量 name 的值。

1. 变量的定义和引用

在 Shell 中，变量的赋值有下列句法格式：

```
name=string
```

其中，name 是变量名，它的值就是 string，“=” 是赋值符号。变量名是以字母或下划线开头的字母、数字和下划线字符序列。

通过在变量名（name）前加\$字符（如\$name）引用变量的值，引用的结果就是用字符串 string 代替\$name。此过程也称为变量替换。

在定义变量时，若 string 中包含空格、制表符和换行符，则 string 必须用'string'或者"sting"的形式，即用单（双）引号将其括起来。双引导内允许变量替换，而单引导内则不可以。

下面给出一个定义和使用 Shell 变量的例子。

使用和定义 Shell 变量举例

//显示字符常量

```
$ echo who are you
```

```
who are you
```

```
$ echo 'who are you'
```

```
who are you
```

```
$ echo "who are you"
```

```
who are you
```

```
$
```

//由于要输出的字符串中没有特殊字符，所以'和"的效果是一样的，不用""相当于使用""

```
$ echo Je t'aime
```

```
>
```

//由于要使用特殊字符（'），

//由于'不匹配，Shell 认为命令行没有结束，回车后会出现系统第二提示符，

//让用户继续输入命令行，按 ctrl+c 结束

```
$
```

//为了解决这个问题，可以使用下面的两种方法

```
$ echo "Je t'aime"
```

```
Je t'aime
```

```
$ echo Je t\`aime
```

```
Je t'aime
```

2. Shell 变量的作用域

与程序设计语言中的变量一样，Shell 变量有其规定的作用范围。Shell 变量分为局部变量和全局变量。

- 局部变量的作用范围仅仅限制在其命令行所在的 Shell 或 Shell 脚本文件中。
- 全局变量的作用范围则包括本 Shell 进程及其所有子进程。
- 可以使用 `export` 内置命令将局部变量设置为全局变量。

下面给出一个 Shell 变量作用域的例子。

Shell 变量作用域的举例

//在当前 Shell 中定义变量 var1

```
$ var1=Linux
```

//在当前 Shell 中定义变量 var2 并将其输出

```
$ var2=unix
```

```
$ export var2
```

//引用变量的值

```
$ echo $var1
```

```
Linux
```

```
$ echo var2
```

```
unix
```

//显示当前 Shell 的 PID

```
$echo $$
```

```
2670
```

```
$
```

//调用子 Shell

```
$ Bash
```

//显示当前 Shell 的 PID

```
$ echo $$
```

```
2709
```

//由于 var1 没有被 export，所以在子 Shell 中已无值

```
$ echo $var1
```

//由于 var2 被 export，所以在子 Shell 中仍有值

```
$ echo $var2
```

```
unix
```

//返回主 Shell，并显示变量的值

```
$ exit
```

```
$echo $$
```

```
2670
```

```
$ echo $var1
```

```
Linux
```

```
$ echo var2

unix

$
```

3. 环境变量

环境变量是指由 Shell 定义和赋初值的 Shell 变量。Shell 用环境变量来确定查找路径、注册目录、终端类型、终端名称、用户名等。所有环境变量都是全局变量，并可以由用户重新设置。表 4-3 列出了一些系统中常用的环境变量。

表 4-3 Shell 中的环境变量

环境变量名	说明	环境变量名	说明
EDITOR、FCEDIT	Bash fc 命令的默认编辑器	PATH	Bash 寻找可执行文件的搜索路径
HISTFILE	用于储存历史命令的文件	PS1	命令行的一级提示符
HISTSIZE	历史命令列表的大小	PS2	命令行的二级提示符
HOME	当前用户的用户目录	PWD	当前工作目录
OLDPWD	前一个工作目录	SECONDS	当前 Shell 开始后所流逝的秒数

不同类型的 Shell 的环境变量有不同的设置方法。在 Bash 中，设置环境变量用 set 命令，命令的格式是：

```
set 环境变量=变量的值
```

例如，设置用户的主目录为/home/johe，可以用以下命令：

```
$ set HOME=/home/john
```

不加任何参数地直接使用 set 命令可以显示出用户当前所有环境变量的设置，如下所示：


```
$ set
```

```
BASH=/bin/Bash
```

```
BASH_ENV=/root/.bashrc
```

```
(略)
```

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/bin/X11
```

```
PS1='[\u@\h \W]\$'
```

```
PS2='>'
```

```
SHELL=/bin/Bash
```

可以看到其中路径 PATH 的设置为：

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/bin/X11
```

总共有 7 个目录，Bash 会在这些目录中依次搜索用户输入的命令的可执行文件。
在环境变量前面加上\$符号，表示引用环境变量的值，例如：

```
# cd $HOME
```

将把目录切换到用户的主目录。

当修改 PATH 变量时，如：将一个路径/tmp 加到 PATH 变量前，应设置为：

```
# PATH=/tmp:$PATH
```

此时，在保存原有 PATH 路径的基础上进行了添加。Shell 在执行命令前，会先查找这个目录。

要将环境变量重新设置为系统默认值，可以使用 unset 命令。例如，下面的命令用于将当前的语言环境重新设置为默认的英文状态。

```
# unset LANG
```

4. 工作环境设置文件

Shell 环境依赖于多个文件的设置。用户并不需要每次登录后都对各种环境变量进行手工设置，通过环境设置文件，用户的工作环境的设置可以在登录的时候自动由系统来完成。环境设置文件有两种，一种是系统环境设置文件，另一种是个人环境设置文件。

1) 系统中的用户工作环境设置文件

- 登录环境设置文件：/etc/profile
- 非登录环境设置文件：/etc/bashrc

2) 用户设置的环境设置文件

- 登录环境设置文件：\$HOME/.Bash_profile
- 非登录环境设置文件：\$HOME/.bashrc

注意：只有在特定的情况下才读取 profile 文件，确切地说是在用户登录的时候。当运行 Shell 脚本以后，就无需再读 profile。

系统中的用户环境文件设置对所有用户均生效，而用户设置的环境设置文件对用户自身生效。用户可以修改自己的用户环境设置文件来覆盖在系统环境设置文件中的全局设置。例如：

1) 用户可以将自定义的环境变量存放在\$HOME/.Bash_profile 中；

2) 用户可以将自定义的别名存放在\$HOME/.bashrc 中，以便在每次登陆和调用子 Shell 时生效。

3.1.3 正则表达式

1. grep 命令

在第三章我们已介绍过 grep 命令的用法。grep 命令用来在文本文件中查找内容，它的名字源于“global regular expression print”。指定给 grep 的文本模式叫做“正则表达式”。它可以是普通的字母或者数字，也可以使用特殊字符来匹配不同的文本模式。我们稍后将更详细地讨论正则表达式。grep 命令打印出所有符合指定规则的文本行。例如：

```
$ grep 'match_string' file
```

即从指定文件中找到含有字符串的行。

2. 正则表达式字符

Linux 定义了一个使用正则表达式的模式识别机制。Linux 系统库包含了对正则表达式的支持，鼓励程序中使用这个机制。

遗憾的是 shell 的特殊字符辨认系统没有利用正则表达式，因为它们比 Shell 自己的缩写更加讨厌和难用。Shell 的特殊字符和正则表达式是很相似的，但是为了正确的利用正则表达式，用户必须了解两者之间的区别。

注意：由于正则表达式使用了一些特殊字符，所以所有的正则表达式都必须用单引号括起来。

正则表达式字符可以包含某些特殊的模式匹配字符。句点匹配任意一个字符，相当于

shell 的问号。紧接句号之后的星号匹配零个或多个任意字符，相当于 Shell 的星号。方括号的用法跟 Shell 的一样，只是用^代替了!表示匹配不在指定列表内的字符。

表 4-4 列出了正则表达式的模式匹配字符。

表 4-4 字符表达式

.	匹配单个任意字符
[list]	匹配字符串列表中的其中一个字符
[range]	匹配指定范围中的一个字符
[^]	匹配指定字符串或指定范围以外的一个字符

表 4-5 列出了与上面配合使用的量词。

表 4-5 量词

*	匹配前一个字符零次或多次
\{n\}	匹配前一个字符 n 次
\{n, \}	匹配前一个字符至少 n 次
\{n, m\}	匹配前一个字符 n 次至 m 次

表 4-6 列出了控制字符。

表 4-6 控制字符

^	只在行头匹配正则表达式
\$	只在行末匹配正则表达式
\	引用特殊字符

控制字符是用来标记行头或者行尾的，支持统计字符串的出现次数。
非特殊字符代表它们自己，如果要表示特殊字符需要在前面加上反斜杠。
例如：

<code>help</code>	匹配包含 <code>help</code> 的行..
<code>\.\$</code>	匹配倒数第二个字符是句点的行
<code>^...\$</code>	匹配只有 3 个字符的行
<code>^[0-9]{3}[^0-9]</code>	匹配以 3 个数字开头跟着是一个非数字字符的行
<code>^\([A-Z][A-Z]\)*\$</code>	匹配只包含偶数个大写字母的行

3.1.4 输入输出重定向与管道

1. 重定向

所谓重定向，就是不使用系统的标准输入端口、标准输出端口或标准错误端口，而进行重新的指定，所以重定向分为输入重定向、输出重定向和错误重定向。通常情况下重定向到一个文件。在 `Shell` 中，要实现重定向主要依靠重定向符实现，即 `Shell` 是检查命令行中是否有重定向符来决定是否需要实施重定向。表 4-7 列出了常用的重定向符。

表 4-7 重定向符

重定向符	说明
<	实现输入重定向。输入重定向并不经常使用，因为大多数命令都以参数的形式在命令行上指定输入文件的文件名。尽管如此，当使用一个不接受文件名为输入多数的命令，而需要的输入又是在一个已存在的文件中时，就能用输入重定向解决问题。
>或>>	实现输出重定向。输出重定向比输入重定向更常用。输出重定向使用户能把一个命令的输出重定向到一个文件中，而不是显示在屏幕上。很多情况下都可以使用这种功能。例如，如果某个命令的输出很多，在屏幕上不能完全显示，即可把它重定向到一个文件中，稍后再用文本编辑器来打开这个文件。
2>或 2>>	实现错误重定向。
&>	同时实现输出重定向和错误重定向。

要注意的是，在实际执行命令之前，命令解释程序会自动打开（如果文件不存在则自动创建）且清空该文件（文中已存在的数据将被删除）。当命令完成时，命令解释程序会正确地关闭该文件，而命令在执行时并不知道它的输出流已被重定向。

下面举几个使用重定向的例子：

- （1）将 ls 命令生成的/tmp 目录的一个清单存到当前目录中的 dir 文件中。

```
$ ls -l /tmp >dir
```

- （2）将 ls 命令生成的/etc 目录的一个清单以追加的方式存到当前目录中的 dir 文件中。

```
$ ls -l /tmp >>dir
```

- （3）passwd 文件的内容作为 wc 命令的输入。

```
$ wc </etc/passwd
```

(4) 将命令 `myprogram` 的错误信息保存在当前目录下的 `err_file` 文件中。

```
$ myprogram 2>err_file
```

(5) 将命令 `myprogram` 的输出信息和错误信息保存在当前目录下的 `output_file` 文件中。

```
$ myprogram &>output_file
```

(6) 将命令 `ls` 的错误信息保存在当前目录下的 `err_file` 文件中。

```
$ ls -l 2>err_file
```

注意：该命令并没有产生错误信息，但 `err_file` 文件中的原文件内容会被清空。

当我们输入重定向符时，命令解释程序会检查目标文件是否存在。如果不存在，命令解释程序将会根据给定的文件名创建一个空文件；如果文件已经存在，命令解释程序则会清除其内容并准备写入命令的输出到结果。这种操作方式表明：当重定向到一个已存在的文件时需要十分小心，数据很容易在用户还没有意识到之前就丢失了。

Bash 输入输出重定向可以通过使用下面选项设置为不覆盖已存在文件：

```
$ set -o noclobber
```

这个选项仅用于对当前命令解释程序输入输出进行重定向，而其他程序仍可能覆盖已存在的文件。

(7) `/dev/null`

空设备的一个典型用法是丢弃从 `find` 或 `grep` 等命令送来的错误信息：

```
$ grep delegate /etc/* 2>/dev/null
```

上面的 `grep` 命令的含义是从 `/etc` 目录下的所有文件中搜索包含字符串 `delegate` 的所有行。由于我们是在普通用户的权限下执行该命令，`grep` 命令是无法打开某些文件的，系统会显示一大堆“未得到允许”的错误提示。通过将错误重定向到空设备，我们可以在屏幕上只得到有用的输出。

2. 管道

许多 Linux 命令具有过滤特性，即一条命令通过标准输入端口接收一个文件中的数据，命令执行后产生的结果数据又通过标准输出端口送给后一条命令，作为该命令的输入数据。后一条命令也是通过标准输入端口接收输入数据。

Shell 提供管道命令 “|” 将这些命令前后衔接在一起，形成一个管道线。格式为：

```
命令 1|命令 2|……|命令 n
```

管道线中的每一条命令都作为一个单独的进程运行，每一条命令的输出作为下一条命令的输入。由于管道线中的命令总是从左到右顺序执行的，因此管道线是单向的。

管道线的实现创建了 Linux 系统管道文件并进行重定向，但是管道不同于 I/O 重定向，输入重定向导致一个程序的标准输入来自某个文件，输出重定向是将一个程序的标准输出写到一个文件中，而管道是直接将一个程序的标准输出与另一个程序的标准输入相连接，不需要经过任何中间文件。

例如：

```
$who >tmpfile
```

我们运行命令 `who` 来找出谁已经登录进入系统。该命令的输出结果是每个用户对应一行数据，其中包含了一些有用的信息，我们将这些信息保存在临时文件中。

现在我们运行下面的命令：

```
$wc -l <tmpfile
```

该命令会统计临时文件的行数，最后的结果是登陆入系统中的用户的人数。

我们可以将以上两个命令组合起来。

```
$who|wc -l
```

管道符号告诉命令解释程序将左边的命令（在本例中为 `who`）的标准输出流连接到右边的命令（在本例中为 `wc -l`）的标准输入流。现在命令 `who` 的输出不经过临时文件就可以直接送到命令 `wc` 中了。

下面再举几个使用管道的例子：

- （1）以长格式递归的方式分屏显示 `/etc` 目录下的文件和目录列表。

```
$ls -RI /etc | more
```

- （2）分屏显示文本文件 `/etc/passwd` 的内容。

```
$cat /etc/passwd | more
```

- （3）统计文本文件 `/etc/passwd` 的行数、字数和字符数。

```
$ cat /etc/passwd | wc
```

(4) 查看是否存在 john 用户帐号。

```
$ cat /etc/passwd | grep john
```

(5) 查看系统是否安装了 apache 软件包。

```
$ rpm -qa | grep apache
```

(6) 显示文本文件中的若干行。

```
$ tail +15 myfile | head -3
```

管道仅能操纵命令的标准输出流。如果标准错误输出未重定向，那么任何写入其中的信息都会在终端显示屏幕上显示。管道可用来连接两个以上的命令。由于使用了一种被称为过滤器的服务程序，多级管道在 Linux 中是很普遍的。过滤器只是一段程序，它从自己的标准输入流读入数据，然后写到自己的标准输出流中，这样就能沿着管道过滤数据。在下例中：

```
$ who|grep tty|wc -l
```

who 命令的输出结果由 grep 命令来处理，而 grep 命令则过滤掉（丢弃掉）所有不包含字符串"tty"的行。这个输出结果经过管道送到命令 wc，而该命令的功能是统计剩余的行数，这些行数与网络用户的人数相对应。

Linux 系统的一个最大的优势就是按照这种方式将一些简单的命令连接起来，形成更复杂的、功能更强的命令。那些标准的服务程序仅仅是一些管道应用的单元模块，在管道中它们的作用更加明显。

3.1.5 Shell 脚本

Shell 最强大的功能在于它是一个功能强大的编程语言。用户可以在文件中存放一系列的命令，这被称为 Shell 脚本或 Shell 程序，将命令、变量和流程控制有机地结合起来将会得到一个功能强大的编程工具。Shell 脚本语言非常擅长处理文本类型的数据，由于 Linux 系统中的所有配置文件都是纯文本的，所以 Shell 脚本语言在管理 Linux 系统中发挥了巨大作用。

1. 脚本的内容

Shell 脚本是以行为单位的，在执行脚本的时候会分解成一行一行依次执行。脚本中所包含的成分主要有注释、命令、Shell 变量和结构控制语句。其中：

1) 注释。用于对脚本进行解释和说明，在注释行的前面要加上符号“#”，这样在执行脚本的时候 Shell 就不会对该行进行解释。

2) 命令。在 Shell 脚本中可以出现任何在交互方式下可以使用的命令。

3) 变量。Shell 支持具有字符串值的变量。Shell 变量不需要专门的说明语句，通过赋值语句完成变量说明并予以赋值。在命令行或 Shell 脚本文件中使用\$name 的形式引用变量name 的值。

4) 流程控制。主要为一些用于流程控制的内部命令。

表 4-8 列出了 Shell 用于流程控制的内置命令

表 4-8 Shell 中用于流程控制的内置命令

命令	说明
test expr 或[expr]	用于测试一个表达式 expr 如真假
if expr then command-table fi	用于实现单分支结构
if expr then command-table else command-talbe fi	用于实现双分支结构
case ... case	用于实现多分支结构
for ...do...done	用于实现 for 型循环
while...do...done	用于实现当型循环
until...do...done	用于实现直到型循环
break	用于跳出循环结构
continue	用于重新开始下一轮循环

2. 脚本的建立与执行

用户可以使用任何文本编辑器编辑 Shell 脚本文件，如 Vi、gedit 等。

Shell 对 Shell 脚本文件的调用可以采用 3 种方式：

1) 一种是将文件名作为 Shell 命令的参数，其调用格式为：

```
$ Bash script_file
```

当要被执行脚本文件没有可执行权限时只能使用这种调用方式。

2) 另一种调用方法是先将脚本文件的访问权限改为可执行，以便该文件可以作为执行文件调用。具体方法是：

```
$ chmod +x script_file
```

```
$ PATH=$PATH:$PWD
```

```
$ script_file
```

3) 当执行一个脚本文件时，Shell 就产生一个子 Shell（即一个子进程）去执行文件中的命令。因此，脚本文件中的变量值不能传递到当前 Shell（即父进程）。为了使的脚本文件中的变量值传递到当前 Shell，必须在命令文件名前面加“.”命令。即：

```
$ ./script_file
```

“.”命令的功能是在当前 Shell 中执行脚本文件中的命令，而不是产生一个子 Shell 执行命令文件中的命令。下面给出一个执行 Shell 脚本的例子。

编写脚本 myfile

```
$ cat >myfile
```

```
mydir='pwd'
```

```
export mydir
```

```
^d
```

//显示脚本 myfile 的内容

```
$ cat myfile
```

```
mydir='pwd'
```

```
export mydir
```

//为脚本添加执行权限并执行

```
$ chmod +x myfile
```

```
$ ./myfile
```

//显示变量 mydir 的值

```
$ echo $mydir
```

//由于这种脚本的方式是在子 Shell 中执行

//所以当脚本执行结束返回主 Shell 后，变量已没有值

//用.命令执行脚本

```
$ ./myfile
```

//显示变量 mydir 的值

```
$ echo $mydir
```

```
/home/johe
```

//由于这种执行脚本的方式是在当前 Shell 中执行，

//当脚本执行结束变量依然有值

3.2 Vi 编辑器

Vi 是 Visual interface 的简称，它可以执行输出、删除、查找、替换、块操作等众多文本操作，而且用户可以根据自己的需要对其进行定制，这是其他编辑程序所没有的。Vi 不是

一个排版程序，它不像 WORD 或 WPS 那样可以对字体、格式、段落等其他属性进行编排，它只是一个文本编辑程序。Vi 是全屏幕文本编辑器，它没有菜单，只有命令。

3.2.1 Vi 的启动与退出

在系统提示符后输入 Vi 和想要编辑（或建立）的文件名，便可进入 Vi，如：

```
$ vi myfile

$ vi
```

如果只输入 Vi，而不带文件名，也可以进入 Vi。如图 3-3 所示。

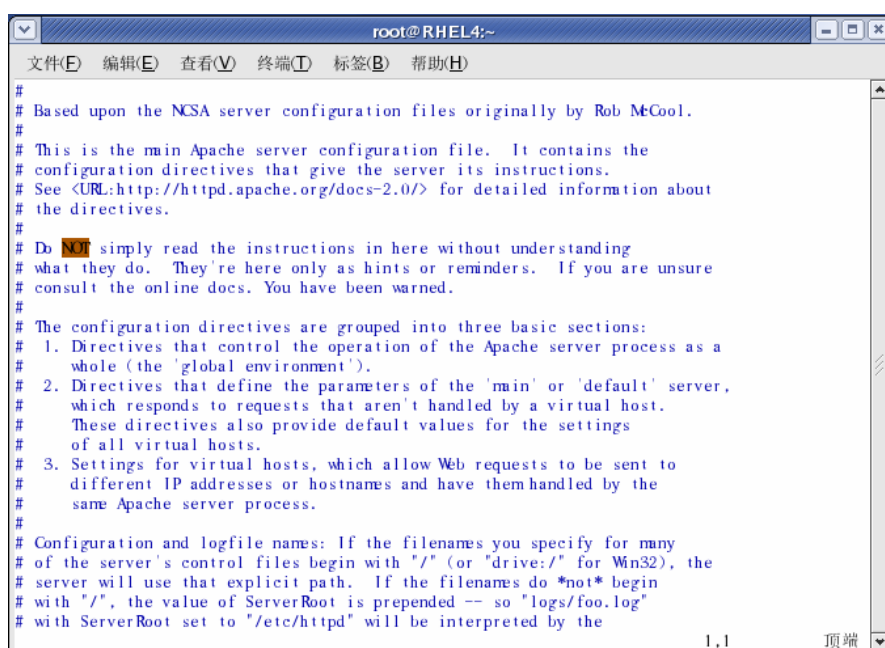


图 3-3 Vi 编辑环境

在命令模式下键入:q, :q!, :wq 或:x（注意:号），就会退出 Vi。其中:wq 和:x 是存盘退出，而:q 是直接退出。如果文件已有新的变化，Vi 会提示你保存文件而:q 命令也会失效，这时你可以用:w 命令保存文件后再用:q 退出，或用:wq 或:x 命令退出，如果你不想保存改变后的文件，你就需要用:q!命令，这个命令将不保存文件而直接退出 Vi，例如：

:w	保存;
:w filename	另存为 filename
:wq!	保存退出
:wq! filename	注: 以 filename 为文件名保存后退出
:q!	不保存退出
:x	应该是保存并退出 , 功能和:wq!相同

3.2.2 Vi 的工作模式

Vi 有 3 种基本工作模式：编辑模式、插入模式和命令模式。考虑到各种用户的需要采用状态切换的方法实现工作模式的转换，切换只是习惯性的问题，一旦你熟练的使用上了 Vi 你就会觉得它其实也很好用。

进入 Vi 之后，首先进入的就是编辑模式，进入编辑模式后 Vi 等待编辑命令输入而不是文本输入，也就是说这时输入的字母都将作为编辑命令来解释。

进入编辑模式后光标停在屏幕第一行首位，用_表示，余各行的行首均有一个“~”符号，表示该行为空行。最后一行是状态行，显示出当前正在编辑的文件名及其状态。如果是[New File]，则表示该文件是一个新建的文件；如果输入 Vi 带文件名后，文件已在系统中存在，则在屏幕上显示出该文件的内容，并且光标停在第一行的首位，在状态行显示出该文件的文件名、行数和字符数。

在编辑模式下输入插入命令 i、附加命令 a、打开命令 o、修改命令 c、取代命令 r 或替换命令 s 都可以进入插入模式。在插入模式下，用户输入的任何字符都被 Vi 当作文件内容保存起来，并将其显示在屏幕上。在文本输入过程中（插入模式下），若想回到命令模式下，按 ESC 键即可。

在编辑模式下，用户按“:”键即可进入命令模式，此时 Vi 会在显示窗口的最后一行（通常也是屏幕的最后一行）显示一个“:”作为命令模式的提示符，等待用户输入命令。多数文件管理命令都是在此模式下执行的。末行命令执行完后，Vi 自动回到编辑模式。

若在命令模式下输入命令过程中改变了主意，可用退格键将输入的命令全部删除之后，再按一下退格键，即可使 Vi 回到编辑模式。

3.2.3 Vi 命令

在编辑模式下，输入如表 4-9 所示的命令均可进入插入模式。

表 4-9 进入插入模式的说明

类型	命令	说明
进入插入模式	i	从光标所在位置前开始插入文本
	I	该命令是将光标移到当前行的行首，然后在起前插入文本
	a	用于在光标当前所在位置之后追加新文本
	A	将光标移到所在行的行尾，从哪里开始插入新文本
	o	在光标所在行的下面新开一行，并将光标置于该行行首，等待输入
	O	在光标所在行的上面插入一行，并将光标置于该行行首，等待输入

表 4-10 列出了常用的命令模式下的命令。

表 4-10 常用的命令模式说明

类 型	命 令	说 明
跳 行	:n	直接输入要移动到的行号即可实现跳行
退 出	:q	退出 Vi
	:wq	保存退出 Vi
	:q!	不保存退出 Vi
文 件 相 关	:w	在光标所在行的下面新开一行，并将光标置于该行行首，等待输入
	:w file	在光标所在行的上面插入一行，并将光标置于该行行首，等待输入
	:n1,n2w file	将从 n1 开始到 n2 结束的行写到 file 文件中
	:nw file	将第 n 行写到 file 文件中
	:l,w file	将从第 1 行起到光标当前位置的所有内容写到 file 文件中
	:\$w file	将从光标当前位置起到文件结尾的所有内容写到 file 文件中
文	:r file	打开另一个文件 file

类 型	命令	说明
文件相关	:e file	新建 file 文件
	:f file	把当前文件改名为 file 文件
字符串搜索、替换和删除	:/str/	从当前光标开始往右移动到有 str 的地方
	:?str?	从当前光标开始往左移动到有 str 的地方
	:/str/w file	将包含有 str 的行写到文件 file 中
	:/str1/,/str2/w file	将从 str1 开始到 str2 结束的内容写入 file
	:s/str1/str2/	将第 1 个 str1 替换为 str2
文本的复制、删除和移动	:s/str1/str2/g	将所有的 str1 替换为 str2
	:n1,n2 co n3	将从 n1 开始到 n2 为止的所有内容复制到 n3 后面
	:n1,n2 m n3	将从 n1 开始到 n2 为止的所有内容移动到 n3 后面
	:d	删除当前行

类 型	命令	说明
动	<code>:nd</code>	删除从当前行开始的 n 行
	<code>:n1,n2 d</code>	删除从 n1 开始到 n2 为止的所有内容
	<code>:\$d</code>	删除从当前行到结尾的所有内容
	<code>:/str1,/str2/d</code>	删除从 str1 开始到 str2 为止的所有内容
执 行 Shell 命令	<code>!Cmd</code>	运行 Shell 命令 Cmd
	<code>:n1,n2 w ! Cmd</code>	将 n1 到 n2 行的内容作为 Cmd 命令的输入, 如果不指定 n1 和 n2, 则 将整个文件的内容作为命令 Cmd 的输入
	<code>:r ! Cmd</code>	将命令运行的结果写入当前行位置

这些命令看似复杂, 其实使用时非常简单。例如删除也带有剪切的意思, 当我们删除文字时, 可以把光标移动到某处, 然后按 `shift+p` 键就把内容贴在原处, 然后再移动光标到某处, 然后再按 `p` 或 `shift+p` 又能贴上。

p 在光标之后粘贴

shift+p 在光标之前粘贴

当进行查找和替换时, 我们要输入 `ESC` 键, 进入命令模式; 我们输入/或?就可以进行查找了。比如在一个文件中查找 `swap` 单词, 首先按 `ESC` 键, 进入命令模式, 然后输入:

```
/swap
```

或

```
?swap
```

若把光标所在的行，把所有单词 the，替换成 THE，则需：

```
:s /the/THE/g
```

仅仅是把第 1 行到第 10 行中的 the，替换成 THE：

```
:1,10 s /the/THE/g
```

这些编辑指令非常有弹性，基本上可以说是由指令与范围所构成。而且需要注意的是，我们采用 PC 的键盘来说明 Vi 的操作，但在具体 的环境中还要参考相应的资料。

3.3 练习题

1. Vi 的 3 种运行模式是什么?如何切换?
2. 什么是重定向?什么是管道?什么是命令替换?
3. Shell 变量有哪两种?分别如何定义?
4. 如何建立和执行 Shell 脚本文件?如何使一个 Shell 脚本在当前 Shell 中运行?
5. 如何设置用户自己的工作环境?
6. 关于正则表达式的练习，首先我们要设置好环境，输入以下命令：

```
$cd  
$cd /etc  
$ls -a >~/data  
$cd
```

这样，/etc 目录下的所有文件的列表就会保存在你的主目录下的 data 文件中。

写出可以在 data 文件中查找所有行的正则表达式：

- 1) 以 “P” 开头
- 2) 以 “y” 结尾
- 3) 以 “m” 开头以 “d” 结尾
- 4) 以 “e”、“g” 或 “l” 开头
- 5) 包含 “o”，它后面跟着 “u”
- 6) 包含 “o”，隔一个字母之后是 “u”
- 7) 以小写字母开头
- 8) 包含一个数字

- 9) 以“s”开头，包含一个“n”
- 10) 只含有 4 个字母
- 11) 只含有 4 个字母，但不包含“f”

实训一 Shell 的使用

一、实训目的

熟悉 shell 的各项功能。

二、实训内容

练习使用 shell 的各项功能。

三、实训练习

(1) 命令补齐功能

- 用 date 命令查看系统当前时间，在输入 da 后，按 tab 键，让 shell 自动补齐命令的后半部分。
- 用 mkdir 命令创建新的目录。首先输入第一个字母 m，然后按 tab 键，由于以 m 开头的命令太多，shell 会提示是否显示全部的可能命令，输入 n。
- 再多输入一个字母 k，按 tab 键，让 shell 列出以 mk 开头的所有命令的列表。
- 在列表中查找 mkdir 命令，看看还需要多输入几个字母才能确定 mkdir 这个命令，然后输入需要的字母，再按 tab 键，让 shell 补齐剩下的命令。
- 最后输入要创建的目录名，按回车键执行命令。
- 多试几个命令利用 tab 键补齐。

(2) 命令别名功能

- 输入 alias 命令，显示目前已经设置好的命令的别名。
- 设置别名 ls 为 ls -l，以长格形式显示文件列表。
- 显示别名 ls 代表的命令，确认设置生效。
- 使用别名 ls 显示当前目录中的文件列表。
- 在使定义的别名不失效的情况下，使用系统的 ls 命令显示当前目录中的命令列表。
- 删除别名。
- 显示别名 ls，确认删除别名已经生效。
- 最后再用命令 ls 显示当前目录中的文件列表。

(3) 输出重定向

- 用 ls 命令显示当前目录中的文件列表。
- 使用输出重定向，把 ls 命令在终端上显示的当前目录中的文件列表重定向到文件 list 中。
- 查看文件 list 中的内容，注意在列表中会多出一个文件 list，其长度为 0。这说明 shell 是首先创建了一个空文件，然后再运行 ls 命令。
- 再次使用输出重定向，把 ls 命令在终端上显示的当前目录中的文件列表重定向到文件 list 中。这次使用管道符号>>进行重定向。
- 查看文件 list 的内容，可以看到用>>进行重定向是把新的输出内容附加在文件的末尾，注意其中两行 list 文件的信息中文件大小的区别。

(4) 输入重定向

- 使用输入重定向，把上面生成的文件 list 用 mail 命令发送给自己。
- 查看新邮件，看看收到的新邮件中其内容是否为 list 文件中的内容。

(5) 管道

- 利用管道和 grep 命令，在上面建立的文件 list 中查找字符串 list。
- 利用管道和 wc 命令，计算文件 list 中的行数、单词数和字符数。

(6) 查看和修改 Shell 变量

- 用 echo 命令查看环境变量 PATH 的值。
- 设置环境变量 PATH 的值，把当前目录加入到命令搜索路径中去。
- 用 echo 命令查看环境变量 PATH 的值。
- 比较前后两次的变化。

四、实训报告

按要求完成实训报告。

实训二 Vi 编辑器的使用

一、实训目的

通过练习两个 C 程序学习 vi 的启动、存盘、文本输入、现有文件的打开、光标移动、复制/剪贴、查找/替换等命令。

二、实训内容

熟练掌握 vi 编辑器的使用。

三、实训练习

(1) 在 vi 中编写一个 abc.c 程序，对程序进行编译、连接、运行。具体如下：

```
[student@enjoy student]$ cd abc
[student@enjoy abc]$ vi abc.c
main()
{
    int i,sum=0;
    for(i=0;i<=100;i++)
    {
        sum=sum+i;
    }
    printf("\n1+2+3+...+99+100=%d\n",sum);
}
[student@enjoy abc]$ gcc -o abc abc.c
[student@enjoy abc]$ ls
abc abc.c
[student@enjoy abc]$ ./abc
1+2+3+...+99+100=5050
[student@enjoy abc]$
[student@enjoy abc]$
```

从如上内容的基础上总结 vi 的启动、存盘、文本输入、现有文件的打开、光标移动、复制/剪贴、查找/替换等命令。

(2) 编写一个程序解决“鸡兔同笼”问题。

参考程序：

```
#include<stdio.h>
main()
{
    int h,f;
    int x,y;
    printf("请输入头数和脚数:");
    scanf("%d,%d",&h,&f);
    x=(4*h-f)/2;
    y=(f-2*h)/2;

    printf("鸡=%d 兔子=%d",x,y);
}
```

运行结果：

```
请输入头数和脚数:18, 48
鸡=12 兔子=6
```

注：

```
鸡+兔子=头
2 鸡+4 兔子=脚
x+y=h
2x+4y=f
```

四、实训思考题

- (1) 输出重定向>和>>的区别是什么？
- (2) 什么是 shell？ shell 分为哪些种类？
- (3) 某用户登陆 Linux 系统后得到的 shell 命令提示符为：[root@long ~]#，请根据此提示符给出登陆的用户名、主机名、当前目录分别是什么？

五、实训报告

按要求完成实训报告。