第6章 shell编程扩展

本章内容

- 文本编辑器
- shell
- bash编程
- 其它shell编程

1. 文本编辑器

- Vi编辑器
- Vim编辑器
- Emacs编辑器
- Gedit编辑器

Vi编辑器

- Vi可以分为三种状态，可通过Esc键进行切换
  - 命令模式（Command mode)
  - 插入模式（Insert mode)
  - 末行模式（Last line mode)
- 它的按键更简洁，通常单个字符按键，即可实现某种操作。
- 但要不停地按Esc在从它的插入模式返回正常模式（Normal mode）。

Vim编辑器

- Vim：Vi最受欢迎的变种之一。
- 除了继承Vi迅捷的编辑方式，Vim的功能更强大。
- Vim的前身Vi和Emacs的设计采用了不同的哲学，Vi更符合Unix传统，它通过管道机制和系统内各种工具打交道，注重和系统内的工具程序协作来完成用户的任务。
- 和Emacs相比，它的定位很明确，就是要做一个强大的编辑器。因此Vim的绝大部分扩展，都是为了更好地完成编辑文本的任务。

Emacs编辑器

- Emacs在1970年代诞生于MIT人工智能实验室。
- 在Unix文化里，Emacs是黑客们关于编辑器优劣之争的两大主角之一，它的对手是Vi、Vim。
- 在Ubuntu系统中，Emacs通常不会默认安装。
- Emacs的安装：

  [root@主机名]# apt-get install emacs
- Emacs的启动：

  [root@主机名]# emacs

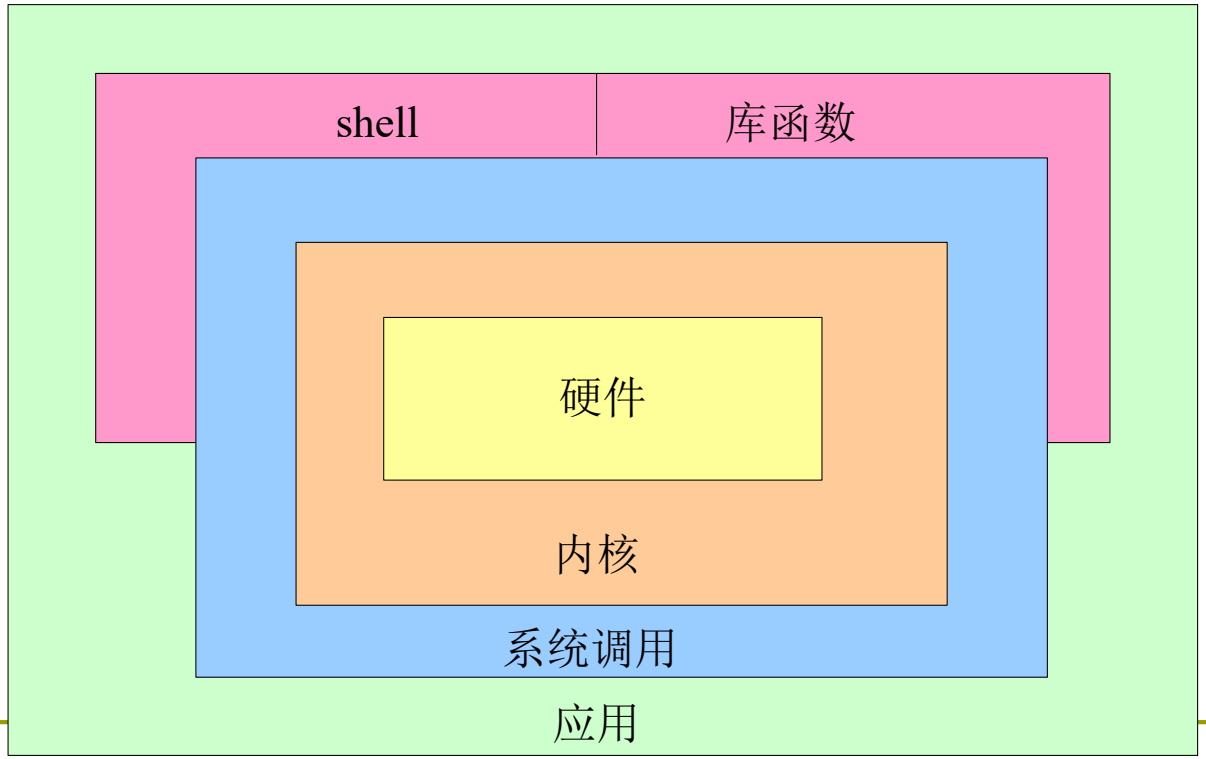
  或者 [root@主机名]# emacs filename

Gedit编辑器

- Gedit是一个GNOME桌面环境下兼容UTF-8的文本编辑器，是一款自由软件，2014年3月24日，其稳定版本号是3.12.0。
- 包含语法高亮和标签编辑多个文件的功能。
- 支持包括多语言拼写检查和一个灵活的插件系统，可以动态地添加新特性。
- 还包括一些小特性，包括行号显示、括号匹配、文本自动换行、自动完成、代码折叠、批量缩进、批量注解、嵌入式终端、当前行高亮，以及自动文件备份。

2. shell

- shell作为命令语言，它交互式解释和执行用户输入的命令或者自动地解释和执行预先设定好的一连串命令。
- shell作为程序设计语言，它定义了各种变量和参数，并提供了许多在高级语言中才具有的控制结构，包括循环和分支。



shell分类

- shell分为两大类：

（1）图形界面shell

  - 应用最为广泛的 Windows Explorer，是微软的Windows系列制作系统，还有广为人知的 Linux shell，其中Linux shell 包括 X window manger (BlackBox和FluxBox），以及功能更强大的CDE、GNOME、KDE、XFCE。

（2）命令行式shell

  - 命令行shell包括：bash、sh、ksh、csh，应用于Unix和Linux系统；cmd.exe命令提示符则应用于WindowsNT 系统。

shell执行

- shell的执行分为两种
  - 交互式模式：等待用户的输入，并且执行用户提交的命令，这种模式也是大多数用户非常熟悉的，如登录、执行一些命令或退出。当用户退出后，shell也终止了。
  - 非交互式模式：shell不与用户进行交互，而是读取存放在文件中的命令，并且执行；当读到文件的结尾时，shell也就终止了。

3. bash编程

- bash是许多Linux平台内定的shell。
- 其名字来源于最初为Unix写bash脚本的作者Steve Bourne。

bash示例

- 程序示例：文件名为hello.sh，目录为/home/yan

```
1 #! /bin/bash
2 # Filename : hello
3 echo "Hello World!“
```

bash程序解释（1）

- 第1行：“#!”符号通常在Linux系统脚本的第一行开头，指明了执行这个脚本文件的解释程序，如果默认的脚本就是bash，那么可以写成：
  #!/bin/sh
  - 如果“#!”之后的解释程序是一个可执行文件，那么执行这个脚本时，它就会把文件名及其参数一起作为参数传给那个解释程序去执行。
  - 如果“#!”指定的解释程序不是一个可执行文件，那么指定的解释程序会被忽略，转而交给当前shell去执行。
  - 如果解释程序没有可执行权限，则会报错；如果“#!”指定的解释程序不存在，那么也会报错。

bash程序解释（2）

- 第2行：注释行用“#”开头。
- 第3行：执行的bash命令。
  - 执行时，首先查看当前用户是否对此文件具有执行权限，如果没有，则需要使用chmod命令来修改，然后执行。

bash程序执行

- 修改权限shell命令：

  [yan@主机名]$ chmod 777 hello.sh
- 执行命令：[yan@主机名]$ ./hello.sh
- 结果：在终端显示Hello World！字符串。

变量

（1）自定义变量：用户自定义的变量由字母、数字和下划线组成，并且变量名的第一个字符不能为数字，且变量名大小写敏感。

例如，定义变量name的值为Jane：
name=Jane

- **注意：bash不能在等号两侧留空格。**

	示例	执行结果	解释
单引号'	$ echo 'a b $USER'	a b $USER	屏蔽掉所有特殊字符的意义
双引号"	$ echo "a b $USER"	a b root	只屏蔽空格的意义
反引号`	$ echo which ls $ echo `which ls`	which ls /bin/ls	预先执行反引号里面的命令

变量（续）

（2）环境变量：

- 可以用set命令给变量赋值或查看环境变量值
- 使用unset命令清除变量值
- 使用export导出变量即可使其它进程访问到该环境变量。

变量（续）

（3）位置变量：

- 位置变量对应于命令行参数，其中$0为脚本名称，$1为第一个参数，以此类推，参数超过9个必须使用${}引用变量。
- shell保留这些变量，不允许用户以另外的方式定义它们，传给脚本或函数的位置变量是局部和只读的，而其余变量为全局的。

变量（续）

(4) 其它变量

变量	解释
$?	保存前一个命令的返回码
$-	在 shell 启动或使用 set 命令时提供选项
$$	当前 shell 的进程号
$!	上一个子进程的进程号
$#	传给脚本或函数的参数个数，即位置变量数减 1，不含脚本名称
$*	传给脚本或函数的参数组成的单个字符串，即除脚本名称后从第一个参数开始的字符串，每个参数以$IFS 分隔（一般内部域分隔符$IFS 为 1 空格）
$@	传给脚本或函数的参数列表，这些参数被表示为多个字符串

操作符

（1）字符串操作符（替换操作符）

操作符	解释
${var:-word}	如果 var 存在且不为空，返回它的值，否则返回 word
${var:=word}	如果 var 存在且不为空，返回它的值，否则将 word 赋给 var， 返回它的值
${var:+word}	如果 var 存在且不为空，返回 word，否则返回空
${var:?message}	如果 var 存在且不为空，返回它的值 否则显示“bash2:$var:$message”，然后退出当前命令或脚本
${var:offset[length]}	从 offset 位置开始返回 var 的一个长为 length 的子串，若没有 length，则默认到 var 串末尾

操作符（续）

（2）模式匹配操作符

操作符	解释
${var#pattern}	从 var 头部开始，删除和 pattern 匹配的最短模式串，返回剩余串
${var##pattern}	从 var 头部开始，删除和 pattern 匹配的最长模式串，返回剩余串，basename path＝${path##*/}
${var%pattern}	从 var 尾部开始，删除和 pattern 匹配的最短模式串，返回剩余串，dirname path＝${path%/*}
${var%%pattern}	从 var 尾部开始，删除和 pattern 匹配的最长模式串，返回剩余串
${var/pattern/string}	用 string 替换 var 中和 pattern 匹配的最长模式串

示例

- 程序示例：字符串模式匹配

```
[yan@主机名]$ var=”hello”
[yan@主机名]$ echo $var ${name:-”world”}!
hello world!
[yan@主机名]$ echo $var ${name:+”world”}!
hello !
[yan@主机名]$ echo $var ${name:=”yan”}!
hello yan!
```

条件和test命令

- bash可以使用[…]结构或test命令测试复杂条件。
- 格式为：

  [expression] 或 test expression

- 注意：左括号后和右括号前空格是必须的语法要求。
- 返回一个代码，表明条件为真还是为假，如果返回0则为真，否则为假。

条件和test命令 (续)

（1）文件测试操作符

操作符	示例	解释（返回真的条件）
-d file	test –d file	file 存在并且是一个目录
-e file	test –e file	file 存在
-f file	test –f file	file 存在并且是一个普通文件
-g file	test –g file	file 存在并且是 SGID(设置组 ID)文件
-r file	test –r file	对 file 有读权限
-s file	test –s file	file 存在并且不为空
-u file	test –u file	file 存在并且是 SUID(设置用户 ID)文件
-w file	test –w file	对 file 有写权限
-x file	test –x file	对 file 有执行权限，如果是目录则有查找权限
-O file	test –O file	拥有 file
-G file	test –G file	测试是否是 file 所属组的一个成员
-L file	test –L file	file 为符号链接
file1 –nt file2	test file1 –nt file2	file1 比 file2 新
file1 –ot file2	test file1 –ot file2	file1 比 file2 旧

条件和test命令 (续)

（2）字符串操作符

字符串操作符	解释（返回真的条件）
str1=str2	str1 和 str2 匹配
str1!=str2	str1 和 str2 不匹配
str1<str2	str1 小于 str2
str1>str2	str1 大于 str2
-n str	str 的长度大于 0（不为空）
-z str	str 的长度为 0（空串）

条件和test命令 (续)

(3) 整数操作符

整数操作符	解释（返回真的条件）
var1 –eq var2	var1 等于 var2
var1 –ne var2	var1 不等于 var2
var1 –ge var2	var1 大于等于 var2
var1 –gt var2	var1 大于 var2
var1 –le var2	var1 小于等于 var2
var1 –lt var2	var1 小于 var2

条件和test命令 (续)

(4) 逻辑操作符

逻辑操作符	解释（返回真的条件）
!expr	对 expr 求反
expr1 && expr2 或者 expr1 –a expr2	对 expr1 与 expr2 求逻辑与，当 expr1 为假时不再执行 expr2
expr1 || expr2 或者 expr1 –o expr2	对 expr1 与 expr2 求逻辑或，当 expr1 为真时不再执行 expr2

示例：判断两个字符串之间的关系

```
[yan@主机名]$ test "abc" = "def" ; echo $?
1
[yan@主机名]$ [ "abc" != "def" ]; echo $?
0
[yan@主机名]$ [ "abc" \< "def" ]; echo $?
0
[yan@主机名]$ [ "abc" \> "def" ]; echo $?
1
[yan@主机名]$ [ "abc" \<"abc" ]; echo $?
1
[yan@主机名]$ [ "abc" \> "abc" ]; echo $?
1
```

shell流控制

（1）条件语句：if

语法：if 条件 then 语句

[elif 条件 then 语句]

fi

（2）确定性循环：for

语法：for value in list

do 语句

done

shell流控制

（3）不确定性循环：while和until

语法：while条件 do语句

done

- 提示：如果条件的执行结果为0，或条件判断为真时，执行循环体内的命令。

语法：until 条件 do 语句

done

- 提示：其判断条件和while正好相反，即条件返回非0，或条件为假时，执行循环体内的命令

shell流控制（续）

（4）选择结构：case

case语法：case 表达式 in 表达式和模式

模式1）

语句;; 允许表达式和含有通配符的模式进行匹配

模式2）

语句;;

esac

shell流控制（续）

（5）选择结构：select

select语法：select value [in list]

do 若没有list则默认为位置变量

statements using $value

done

示例

- 程序示例：对输入的不同后缀的文件做不同的处理。

```
#! /bin/bash
for filename in "$@"; do
  case $filename in
    *.jpg ）exit 0 ;;
    *.tga ）echo “$filename is a tga file”;;
    *.xpm ）echo “$filename is a xpm file” ;;
    *.pcx ）echo “$filename is a pcx file”;;
    *.tif ）echo “$filename is a tif file”;;
    *.gif ）echo “$filename is a gif file”;;
    * ）echo "procfile: $filename is an unknown graphics file."
    exit 1 ;;
  esac
done
```

函数定义

- **函数定义:**

```
function fname() {
commands commands
}
```

函数调用

- **函数调用：**

  fname [parm1 parm2 parm3 ...]

- **说明：函数在使用前定义，函数名和调用函数参数成为函数的位置变量。在函数或脚本中，可以使用bash特殊变量来引用参数，可以给这些变量附上$符号的前缀，然后像引用其它 shell 变量那样引用它们。**

函数参数说明

参数	目的
0, 1, 2, …	位置参数从参数 0 开始。参数 0 引用启动 bash 的程序的名称，如果函数在 shell 脚本中运行，则引用 shell 脚本的名称。有关该参数的其他信息，比如 bash 由-c 参数启动，请参阅 bash 手册。由单引号或双引号包围的字符串被作为一个参数进行传递，传递时会去掉引号。如果是双引号，则在调用函数之前将对$HOME 之类的 shell 变量进行扩展。对于包含嵌入空白或其他字符（这些空白或字符可能对 shell 有特殊意义）的参数，需要使用单引号或双引号进行传递。
*	位置参数从参数 1 开始。如果在双引号中进行扩展，则扩展就是一个词，由 IFS 特殊变量的第一个字符将参数分开，如果 IFS 为空，则没有间隔空格。IFS 的默认值是空白、制表符和换行符。如果没有设置 IFS，则使用空白作为分隔符（仅对默认 IFS 而言）。
@	位置参数从参数 1 开始。如果在双引号中进行扩展，则每个参数都会成为一个词，因此 “$@” 与 “$1” “$2” 等效。如果参数有可能包含嵌入空白，那么您将需要使用这种形式。
#	参数数量（不包含参数 0）。

示例

- 程序示例：函数输入参数示例：

```
[yan@主机名]$ testfunc () { echo "$#
  parameters"; echo "$@"; }
[yan@主机名]$ testfunc
0 parameters
[yan@主机名]$ testfunc a b c
3 parameters
a b c
[yan@主机名]$ testfunc a "b c"
2 parameters
a b c
```

示例（续）

- 程序示例：关于参数处理的差异

```
[yan@主机名]$ type testfunc2
testfunc2 is a function
testfunc2 ()
{
    echo "$# parameters";
    echo Using '$*';
    for p in $*;
    do
        echo "[$p]";
    done;
    echo Using "'$*'";
    for p in "$*";
```

```
[yan@主机名]$ IFS="|${IFS}" testfunc2
abc "a bc" "1 2
> 3"
3 parameters
Using $*
[abc]
[a]
[bc]
[1]
[2]
[3]
Using "$*"
```

示例（续）

```
    do
        echo "[$p]";
    done;
    echo Using '$@';
    for p in $@;
    do
        echo "[$p]";
    done;
    echo Using '"$@"';
    for p in "$@";
    do
        echo "[$p]";
    done
}
```

```
[abc|a bc|1 2
3]
Using $@
[abc]
[a]
[bc]
[1]
[2]
[3]
Using "$@"
[abc]
[a bc]
[1 2
3]
```

示例

- 程序示例： 函数输出参数。

```
function myfunc()
{
    local  myresult='some value'
    echo "$myresult"
}
result=$(myfunc)   # or
result=`myfunc` (此处为反引号)
echo $result
```

- 结果：输出“some value”。

4. 其它shell编程

- sed
- Grep
- awk

sed介绍

- sed是一种在线编辑器，它一次处理一行内容。
- 处理时，把当前处理的行存储在临时缓冲区中，称为“模式空间”（pattern space），接着用sed命令处理缓冲区中的内容，处理完成后，把缓冲区的内容送往屏幕。接着处理下一行，这样不断重复，直到文件末尾。
- 主要用来自动编辑一个或多个文件，简化对文件的反复操作，编写转换程序等。

sed命令格式

- **sed命令格式:**

  sed [options] 'command' file(s)

  或者 sed [options] -f scriptfile file(s)

Sed示例

- 程序示例：

（1）[yan@主机名]$ sed ‘2d’ test

- 解释：删除test文件的第二行。

（2）[yan@主机名]$ sed ‘s/34/100/g’ test

- 解释：在整行范围内把34替换为100。如果没有g标记，则只有每行第一个匹配的34被替换成100。

（3）[yan@主机名]$ sed -n '2,/^34/p' test

- 解释：打印从第2行开始到第一个包含以34开始的行之间的所有行。

Sed示例（续）

(4) [yan@主机名]$ sed '/43/i\\new line' test

- 解释：如果43被匹配，则把反斜杠后面的文本插入到匹配行的前面。

(5) [yan@主机名]$ sed -n 'w test1' test

- 解释：将test文件中的内容写入到test1中。

(6) [yan@主机名]$ sed '3q' test

- 解释：打印完第3行后，退出sed。

grep

- grep（Global Regular Expression Print），表示全局正则表达式版本，它的使用权限是所有用户。
- Linux系统中grep命令是一种强大的文本搜索工具，它能使用正则表达式搜索文本，并把匹配的行打印出来。

- grep命令格式：

  grep [options]

Grep示例

- 程序示例：

（1）[yan@主机名]$ grep ‘test’ d*

- 解释：显示所有以d开头的文件中包含 test的行。

（2）[yan@主机名]$ grep ‘test’ aa bb cc

- 解释：显示在aa、bb、cc文件中匹配test的行。

（3）[yan@主机名]$ grep ‘[a-z]\{5\}’ aa

- 解释：显示所有包含每个字符串至少有5个连续小写字符的字符串的行。

Grep示例（续）

(4) [yan@主机名]$ grep 'w\(es\)t.*\1'aa

- 解释：如果west被匹配，则es就被存储到内存中，并标记为1，然后搜索任意个字符(.*)，这些字符后面紧跟着另外一个es(\1)，找到就显示该行。如果用egrep或grep -E，就不用"\"号进行转义，直接写成'w(es)t.*\1'就可以。

Grep与Find

- find是在磁盘/分区中找到文件，可以按照文件名、文件类型、文件大小、访问时间等来找到指定文件。
- grep是查找文件的利器，可以在一个txt文本中截取到有特定关键字的行，并显示出来；也可以通过关键字，在一个文件夹下查找多个有这些关键字的文件，并生成结果，即根据文件内容递归查找目录。

Grep与Find示例

(1) [yan@主机名]$ find ~ -name "[a-z][0-9]*.txt" –print

- 解释：在主目录中查找以一个小写字母和一个数字开头的txt文件并显示。

(2) [yan@主机名]$ find ~ -mmin +60

- 解释：在主目录中查找60分钟前被改动过的文件。

(3) [yan@主机名]$ grep 'hello world' *

- 解释：在当前目录搜索带有'hello world'行的文件。

(4) [yan@主机名]$ grep –l –r 'hello world' *

- 解释：在当前目录及其子目录下搜索带有'hello world'行的文件，但是不显示匹配的行，只显示匹配的文件。

awk

- Awk 是一种样式扫描与处理工具 。
- awk几乎可以完成 grep和sed所能完成的全部工作，尤其是基于文本的样式扫描和处理，awk所做的工作有些像数据库，但与数据库不同的是，它处理的是文本文件，这些文件没有专门的存储格式，普通的人们就能编辑、阅读、理解和处理它们。
- awk是一种编程语言，用于在Unix/Linux下对文本和数据进行处理，数据可以来自标准输入、一个或多个文件，或其它命令的输出，它支持用户自定义函数和动态正则表达式等先进功能。

awk示例

- 程序示例：

（1）$awk ‘$1>100 {print $1>>”output_file”;print $1} test

- 解释：如果文件第一列的值大于100，则把它输出到文件output_file中，并打印出来。也可以用“>>”来重定向输出，但不清空文件，只做追加操作（test文件中存在两列不同的正整数，下面还会继续用到）。

awk示例（续）

(2) $awk 'BEGIN{ "date" | getline d; print d} END{close("date")}' test

- 解释：执行Linux的date命令，并通过管道输出给getline（使用方法详见附录），然后再把输出赋值给自定义变量d，并打印它。BEGIN模块后紧跟着动作块，这个动作块在awk处理任何输入文件之前执行。END块执行关闭管道操作。

(3) $ awk 'END{print "The number of records is" NR}' test

- 解释：打印所有被处理的记录数。END不匹配任何的输入文件，但是执行动作块中的所有动作，它在整个输入文件处理完成后被执行。

本章小结（1）

- shell是操作系统为使用者提供的软件或命令解析器。
- 通常我们所说的shell指的是命令行式shell，其执行方式分为交互式模式和非交互式模式。
- 常用的文本编辑器包括Vi编辑器、Vim编辑器、Emacs编辑器等，其中，几乎所有Linux版都自带Vi编辑器。
- bash是使用最多的shell之一，也是许多Linux平台内定的shell，它有变量、操作符、条件和流控制，以及函数，可以为系统管理等编写很复杂的程序。

本章小结（2）

- Bash是一个常用脚本。
- sed是一个很好的文件处理工具，本身是一个管道命令，主要是以行为单位进行处理，可以将数据行进行替换、删除、新增、选取等。
- grep命令是一种强大的文本搜索工具，它能使用正则表达式搜索文本，并把匹配的行打印出来。
- awk几乎可以完成 grep和sed所能完成的全部工作，尤其是基于文本的样式扫描和处理。