第 3 章 基本的 bash shell 命令

1. shell 和其它语言一同，shell 的各种功能其实都是由命令实现的，所以 shell 没有函数库。shell 中的函数只不过是方便用户移植代码而已。
2. bash 是弱类型的编程语言，不区分数据类型，把所有数据统统当作字符串处理。
3. bash 中语句后面不需要加分号。

```
#!/bin/sh
var=1
var2=$[var*2]
echo $var2
```

4. 使用 touch 命令来创建空文件。

```
Touch test1
```

5. 硬链接会创建一个独立文件，其中包含了源文件的信息以及位置。引用硬链接文件等同于引用了源文件。
6. stat 命令可以提供文件系统上某个文件的所有状态信息：

```
Stat test10
```

stat 命令的结果显示了几乎所有你想知道的关于被检查文件的信息，甚至连存储该文件的设备的主设备和次设备编号都有，但它不提供文件类型，文件类型使用 file 命令提供。

7. file 命令将文件分成 3 类：

- 文本文件
- 可执行文件
- 数据文件

第 10 章 构建基本脚本

1. 在命令行界面中，如果要两个命令一起运行，可以在同一提示行输入它们，用分号隔开：

```
date;who
```

2. 在创建 shell 脚本文件时，必须在文件的第一行指定要使用的 shell。其格式为

```
#!/bin/bash
```

3. 使用#添加注释。

4. 编写好 shell 文件之后的第一个障碍是让 bash shell 找到脚本文件。要让 shell 找到 test1 脚本，我们只需采取下述做法之一：

- 将 shell 脚本文件所处的目录添加到 PATH 环境变量中；
- 在提示符中用绝对或相对文件路径来引用 shell 脚本文件。

5. echo 命令能显示一个简单的文本字符串。

```
$ echo This is a test
This is a test
```

注意，默认情况下，你不需要使用引号将要显示的文本字符串圈起来。echo 命令可用单引号或双引号来将文本字符串圈起来。如果你在字符串中用到了它们，你需要在文件中使用其中一种引号，而用另外一种来将字符串圈起来：

```
$ echo ‘Rich says “scripting is easy”.’
Rich says “scripting is easy”.
```

如果想在同一行显示一个文本字符串作为命令输出，可以用 echo 语句的-n 参数，只要将第一个 echo 语句改成这样就行：

```
echo -n “The time and date are:”
```

你需要在字符串两侧使用引号来保证在显示的字符串尾部有一个空格。命令输出将会紧接着字符串结束的地方开始。

```
$ cat test1
#!/bin/bash
# This script displays the date and who's logged on
echo  The time and date are:
date
echo "Let's see who's logged into the system: "
who
$
```

输出如下：

```
$ ./test1
The time and date are:
Mon Feb 21 15:41:13 EST 2011
Let's see who's logged into the system:
Christine tty2         2011-02-21 15:26
Samantha tty3         2011-02-21 15:26
Timothy  tty1         2011-02-21 15:26
user     tty7         2011-02-19 14:03 (:0)
user     pts/0        2011-02-21 15:21 (:0.0)
$
```

6. echo 使用重定向符还可以将内容输出到一个文件中：

```
echo "/usr/bin/supervisord -c /etc/supervisord.conf" >> /etc/rc.local
```

7. 星号（*）、美元符等特殊字符需要使用自身含义时，可使用转义字符\，如\$就代表美元符本身。
8. 通过等号给用户变量赋值，但注意，在变量、等号和值之间不能出现空格。用户变量可通过美元符引用。

```
val=10
```

引用一个变量时需要使用美元符，而引用变量来对其进行赋值时不要使用美元符。

```
val=10
val2=${val}
```

变量中的花括号是可选的。

9. 在为变量赋值时，右边的是变量或表达式如果不使用$，则会原样输出：

```
#!/bin/sh
test=5
test2=[5*5]
echo $test2
```

输出为：

```
lan@android:~$ ./test.sh
[5*2]
```

加上$才会计算：

```
#!/bin/sh
test=5
test2=$[5*5]
echo $test2
```

输出为：

```
lan@android:~$ ./test.sh
10
```

10.反引号`允许你将 shell 命令的输出赋值给变量。

```
testing=`date`
```

下面这个例子在脚本中通过反引号获取当前日期并用它来生成唯一文件名：

```
#!/bin/bash
today=`date +%y%m%d`
ls /usr/bin -al > log.$today
```

+%y%m%d 格式告诉 date 命令将日期显示为两位数的年月日的组合。

11.重定向

- 以<改变标准输入：

  program < file 可将 program 的标准输入修改为 file 文件。

- 以>改变标准输出：

  program > file 可将 program 的标准输出修改为 file 文件。

- 以>>附加到文件：

  program >> file 可将 program 的标准输出附加到 file 的结尾处。

- 以|建立管道

  program1 | program2 可将 program1 的标准输出修改为 program2 的标准输入。

12.wc 命令提供了对数据中文本的计数。默认情况下，它会输出 3 个值：

- 文本的行数
- 文本的词数
- 文本的字节数

使用-l 参数统计行数。

13.使用方括号可以执行各种运算，例如数学运算、比较运算等：

```
var=$[$var*2]
```

在将运算结果作为右值赋值给变量时，要使用$[]，否则就是原样输出。

14.<<EOF 和 EOF 之间的代码可以作为子命令或子 Shell 的输入。EOF 有两种形式：
(1) cat<<EOF

```
#cat<<EOF
```

```
>12
>34D
>EOF
12
34D
```

EOF 文本字符串标识了内联重定向数据的开始和结尾。

(2) <<EOF 后接一个重定向文件，表示将标准输出进行重定向，将本应输出到屏幕的内容重定向到文件而已。

```
cat <<EOF > /etc/shadowsocks.json       #将大括号内容重定向到文件中
{
   "server": "0.0.0.0",
   "server_port": 8888,
   "local_port": 1080,
   "local_address": "127.0.0.1",
   "password": "www.qttc.net",
   "method": "aes-256-cfb",
   "timeout": 600
}
EOF
```

15. shell 命令返回值赋给变量：

- 方法一：

```
xxx@xxx-desktop:~/temp$ aaa=`cat 123`
xxx@xxx-desktop:~/temp$ echo $aaa
abc
```

注意 cat 命令外面的那个引号是反引号，键盘上数字 1 旁边那个。

- 方法二：

```
xxx@xxx-desktop:~/temp$ ccc=$(cat 123)
xxx@xxx-desktop:~/temp$ echo $ccc
abc
```

16. linux 提供了$?专属变量来保存上个执行的命令的退出状态码。你必须在你要查看的命令之后马上查看或使用$?变量。

表10-2 Linux退出状态码

状 态 码	描 述
0	命令成功结束
1	通用未知错误
2	误用shell命令
126	命令不可执行
127	没找到命令
128	无效退出参数
128+x	Linux信号x的严重错误
130	命令通过Ctrl+C终止
255	退出状态码越界

17. exit 命令允许你在脚本结束时指定一个退出状态码，退出状态码最大只能是 255：

```
exit 5
或
exit $ var3
```

第 11 章 使用结构化命令

1. 结构化命令中，最基本的类型就是 if-then 语句，而 shell 的 if 语句和其他语言的 if 语句不太一样。

- if-then 语句有如下格式：

```
if command;
then
    commands
fi
```

如果 if 命令的退出状态码是 0（该命令成功运行），位于 then 部分的命令就会被执行。如果该命令的退出状态码是其他什么值，那 then 部分的命令就不会被执行，bash shell 会

继续执行脚本中的下一个命令。在 then 部分，你可以用多个命令。你可以像脚本中其他地方一样列出多条命令。bash shell 会将这部分命令当成一样块。

- if-then-else 语句在语句中提供了另外一组命令：

```
if command;
then
      commands
else
     commands
fi
```

- elif 会通过另一个 if-then 语句来延续 else 部分：

```
if command1;
then
      commands
elif command2;
then
      commands
else
      commands
fi
```

注意：if 语句不能测试跟命令的退出状态码无关的条件。同时，if 后面的 commands 是带分号的：

```
if ! wget --no-check-certificate -O ${shadowsocks_libev_ver}.tar.gz $
{download_link}; then           #获取文件
      echo "Failed to download ${shadowsocks_libev_ver}.tar.gz"
      exit 1
  fi
```

if 语句和 then 语句之间需要一个分号。

2. test 命令提供了在 if-then 语句中测试不同条件的途径。如果 test 命令中列出的条件成立，test 命令就会退出并返回退出状态码 0。如果条件不成立，test 命令退出并返回退出状态码 1。

- test 命令的格式非常简单：

```
test condition
```

condition 是 test 命令要测试的一系列参数和值。

- Bash shell 提供了另一种在 if-then 语句中声明 test 命令的方法：

```
if [ condition ];
then
  commands
fi
```

方括号定义了 test 命令中用到的条件。

```
#!/bin/sh
echo -n "请输入 temp 的值："
read temp
if [ $temp -eq 1 ];then
      echo temp 的值为 1
else
      echo temp 的值不为 1
fi
```

3. Test 命令可以判断 3 类条件：

- 数值比较，数值条件测试可以用在数字和变量上。其中带 e 的表示“等于”，如 eq；带 g 的表示大于，如 gt 和 ge；带 l 的表示小于，如 lt 和 le。
  - int1 -eq int2　　两数相等为真
  - int1 -ne int2　　两数不等为真
  - int1 -gt int2　　int1 大于 int2 为真
  - int1 -ge int2　　int1 大于等于 int2 为真
  - int1 -lt int2　　int1 小于 int2 为真
  - int1 -le int2　　int1 小于等于 int2 为真

  注意：test 命令无法处理变量中存储的浮点值。
- 字符串比较。注意，shell 中的“=”和“==”是一样的，都可以用于 if 的判断中：
  - -z string　　存在指定的变量时，则为真
  - -n string　　不存在指定的变量时，则为真
  - string1 = string2　　如果 string1 与 string2 相同，则为真
  - string1 != string2　如果 string1 与 string2 不同，则为真
  - string1 < string2　　如果 string1 小于 string2，则为真

- string1 > string2　　如果 string1 大于 string2，则为真

注意：大于小于符号必须转义，否则 shell 会把它们当做重定向符号而把字符串值当做文件名。而且大于小于顺序和 sort 命令所采用的不同，在 test 命令中大写字母小于小写字母，而 sort 命令小写字母小于大写字母，这是由各个命令使用的排序技术不同造成的。

```
if [ $val1 \> $val2 ]
```

- 文件比较
  - -e file　　文件是否存在
  - -r file　　用户可读为真
  - -w file　　用户可写为真
  - -x file　　用户可执行为真
  - -f file　　文件为正规文件为真
  - -d file　　文件为目录为真
  - -c file　　文件为字符特殊文件为真
  - -b file　　文件为块特殊文件为真
  - -s file　　文件大小非 0 时为真
  - -t file　　当文件描述符(默认为 1)指定的设备为终端时为真

```
if [ -e file]                    #检查 file 文件是否存在
```

4. if 命令共有两种用法：

- 和 test 一起使用，用法和其他语言一样。
- 测试退出状态码是否为 0，为 0 则执行指定语句。

5. 使用 test 命令时，if 的每一个字符之间都要有空格：

```
#!/bin/sh
echo -n "请输入 temp 的值："
read temp
if [ $temp -eq 1 ];then
    echo temp 的值为 1
elif [ $temp -eq 2 ];then
    echo temp 的值为 2
```

```
else
    echo temp 的值为$temp
fi
```

6. if 语句如果是测试其退出状态码，是不需要加 test 的，但如果是进行比较判断，就要加 test。
7. if-then 语句允许你使用布尔逻辑来组合测试。有两种布尔运算符可用：

- [condition1] && [condition2]：只有在&&左边的命令返回真，&&右边的命令才会被执行。
- [condition1] || [condition2]：只有在&&左边的命令返回假（执行失败），&&右边的命令才会被执行。

8. 双小括号命令允许你将高级数学表达式放入比较中。test 命令只允许在比较中进行简单的算术操作。双小括号命令格式如下：

```
(( expression ))
```

术语 expression 可以是任意的数学赋值或比较表达式。你可以在 if 语句中用双小括号命令，也可以在脚本中的普通命令里用来赋值：

```
if (( $val1 ** 2 > 90 ))
then
   (( val2 = $val1 ** 2 ))
   echo "The square of $val1 is $val2"
```

注意：你不需要将双小括号中表达式里的大于号转义。

9. 双中括号提供了针对字符串比较的高级特性。双中括号命令的格式如下：

```
[[ expression ]]
```

双中括号里的 expression 使用了 test 命令中采用的标准字符串来进行比较。但它提供了 test 命令未提供的另一个特性——模式匹配。在模式匹配中，你可以定义一个正则表达式来匹配字符串值。

```
if [[ $USER == r* ]]
then
   echo "Hello $USER"
else
   echo "Sorry, I do not know you"
fi
```

10.case 命令类似 c 语言的 switch case 语句，表达式格式如下：

```
Case variable in
pattern1 | pattern2) command1;;
pattern3) commands2;;
*） default commands;;
esac
```

如果变量和模式是匹配的，那么 shell 会执行该模式指定的命令。你可以通过竖线来分割模式，在一行列出多个模式。星号会捕获所有跟所有列出的模式都不匹配的值。注意下面例子分号的位置：

```
#!/bin/bash
# using the case command

case $USER in
rich | barbara)
   echo "Welcome, $USER"
   echo "Please enjoy your visit";;
testing)
  echo "Special testing account";;
jessica)
   echo "Do not forget to log off when you're done";;
*)
   echo "Sorry, you are not allowed here";;
esac
```

第 12 章 更多的结构化命令

1. 循环语句都带 do...done。
2. for 命令

   (1)bash shell 提供了 for 命令，允许你创建一个遍历一系列值的循环。下面是 bash shell 中 for 命令的基本格式：

```
for var in list
do
  commands
done
```

在 lise 中，你提供一系列迭代中要用的一系列值。在每个迭代中，变量 var 会包含列表中的当前值，每一个迭代会使用列表中的第一个值，第二个迭代使用第二个值，以此类推，直到列表中的所有值都过一遍。

for 命令后面既可以加分号，也可以不加分号。

```
#!/bin/bash
#!/bin/bash
date
for i in `seq 1 5`
do
{
   echo "sleep 5"
   sleep 5
} &
done
wait
```

(2)for 命令最基本的用法就是遍历 for 命令自身中定义的一系列值：

```
$ cat test1
#!/bin/bash
# basic for command

for test in Alabama Alaska Arizona Arkansas California Colorado
do
   echo The next state is $test
done
$ ./test1
The next state is Alabama
The next state is Alaska
The next state is Arizona
The next state is Arkansas
The next state is California
The next state is Colorado
$
```

(3)当列表中有单引号等等符号时，有两种解决办法：

- 使用转义字符（反斜线）来将单引号转义
- 使用双引号来定义用到单引号的值

```
$ cat test2
#!/bin/bash
# another example of how not to use the for command

for test in I don\'t know if "this'll" work
do
   echo "word:$test"
done
$ ./test2
word:I
word:don't
word:know
word:if
word:this'll
word:work
$
```

(4)for 命令用空格来划分列表中的每个值。如果在单独的数据值中有空格，那么你必须用双引号来将这些值圈起来：

```
$ cat test3
#!/bin/bash
# an example of how to properly define values

for test in Nevada "New Hampshire" "New Mexico" "New York"
do
   echo "Now going to $test"
done
$ ./test3
Now going to Nevada
Now going to New Hampshire
Now going to New Mexico
Now going to New York
$
```

- in 后面的字符串是以空格符、水平制表符、换行符进行分隔的，而不是单个字符分隔。

(5)使用反引号来执行任何能产生输出的命令，然后在 for 命令中使用该命令的输出：

```
for state in `cat $state`
```

(6)环境变量 IFS 称为内部字段分隔符。默认情况下，bash shell 会将下列字符当作字段分隔符：

- 空格
- 制表符
- 换行符

如果 bash shell 在数据中看到这些字符中的任意一个，它就会假定你在列表中开始了一个新的数据段。你可以在 shell 脚本中临时更改 IFS 环境变量的值来限制一下被 bash shell 当作字段分隔符的字符。比如，下列代码修改 IFS 的值使其只能识别换行符：

```
IFS=$’\n’
```

这条语句告诉 bash shell 在数据值中忽略空格和制表符。

(7)使用 for 命令可以自动遍历满是文件的目录。进行此操作时，你必须在文件名或路径名中使用通配符。

```
#!/bin/bash
# iterate through all the files in a directory

for file in /home/rich/test/*
do

   if [ -d "$file" ]
   then
      echo "$file is a directory"
   elif [ -f "$file" ]
   then
      echo "$file is a file"
   fi
done
```

3．base 中 C 语言风格的 for 循环的基本风格：

```
For (( variable assignment ; condition ; iteration process ))
```

C 语言风格的 for 命令看起来如下：

```
For (( a=1;a<10;a++ ))
```

注意，有一些事情没有遵循标准 bash shell for 命令：

- 给变量赋值可以有空格；

- 条件中的变量不以美元符开头；
- 迭代过程的算式未用 expr 命令格式。

C 语言风格的 for 命令也允许你为迭代使用多个变量：

```
for (( a=1,b=10;a<=10;a++,b-- ))
```

例如，显示未来 7 天的日期：

```
#!/bin/sh
for((i=-1;i>=-7;i--))
do
    date -d "$i days ago" +%y-%m-%d
done
```

4. while 命令允许你定义一个要测试的命令，然后循环执行一组命令，只要定义的测试命令返回的是退出状态码 0。while 命令的格式是：

```
While test command
do
 other commands
done
```

```
#!/bin/bash
# while command test

var1=10
while [ $var1 -gt 0 ]
do
   echo $var1
   var1=$[ $var1 - 1 ]
done
```

只要测试条件成立，while 命令才会继续遍历执行定义好的命令。

while 命令允许在 while 语句行定义多个测试命令，只有最后一个测试命令的退出状态码会被用来决定什么时候结束循环。

```
While echo $var1
    [ $var1 -ge 0 ]
```

第一个测试会简单地显示 var1 变量的当前值。第二个测试用方括号来决定 var1 变量的值。要注意的是在含有多个命令的 while 语句中，在每次迭代中所有的测试命令都会被执行，

包括最后一个测试不成立的最后那次循环。另一个要小心的是你如何指定多个测试命令，每个测试命令都是在单独的一行上。

5. until 命令和 while 命令工作的方式完全相反，until 命令的格式如下：

```
Until test commands
do
 other commands
done
```

注意，这个和 C 语言的 do……while 语句执行次序刚好相反，until 是先判断再执行语句，do……while 先执行语句再判断。

6. 有几个命令能帮我们控制循环内部的情况：

- break 命令
- continue 命令

7. break 命令和 C 语言的大致相同，不同的是 break 命令可以接受单个命令行参数值：

```
break n
```

默认情况下，n 为 1，表示当前循环，如果设为 2，break 命令就会停止下一级的外部循环，依此类推。

8. continue 命令和 C 语言的大致相同，不同的是 continue 命令可以接受单个命令行参数值：

```
continue n
```

其中 n 定义了要继续循环层级。注意，continue 并不跳出循环，只是终止命令当次循环起的 n 次循环，直接跳到第 n+1 次循环

9. 在 shell 脚本中，你要么管接要么重定向循环的输出。你可以在 done 命令之后添加一个处理命令：

```
For file in /home/rich*
do
 if [ -d “$file” ]
 then
  echo “$file is a directory”
 elif
  echo “$file is a file”
 fi
done > output.txt
```

shell 会将 for 命令的结果重定向到文件 output.txt 中，而不是显示在屏幕上。

第 13 章 处理用户输入

1. 向 shell 脚本传数据的最基本方法是使用命令行参数。命令行参数允许在运行脚本时向命令行添加数据值：

```
$ ./addem 10 30
```

2. 使用${#}来获取字符串长度：

```
string="abcd"
echo ${#string} #输出 4
```

3. 使用下列语法提取子字符串：

```
string="runoob is a great site"
echo ${string:1:4} # 输出 unoo
```

4. 拼接字符串：

```
#!/bin/bash
var="google.com"
f="http://www."
domain=${f}${var}
echo ${domain}
```

输出是：

```
http://www.google.com
```

5. Bash shell 会将一些称为位置参数的特殊变量分配给命令行输入的所有参数，包括 shell 执行的程序的名字。位置参数变量是标准的数字：$0 是程序名，$1 是第一个参数，$2 是第二个参数，依次类推，直到第 9 个参数$9。可以在 shell 脚本中像使用其他变量一样使用$1 变量。shell 脚本会自动将命令行参数的值分配给变量。不需要作任何处理。参数值中包含空格时，必须要用引号

```
$ ./test3 ‘Rich Blum’
```

如果参数多于 9 个，在变量数字周围加花括号，比如${10}。

6. 使用$0 参数获取 shell 在命令行启动的程序路径名。注意，$0 的真实字符串是整个脚本的路径，当只需要程序名而不要路径时，可以使用 basename 命令返回程序名。

```
name=`basename $0`
```

7. 特殊变量$#含有脚本运行时就有的命令行参数的个数，你可以在脚本中任何地方使用这个特殊变量，就跟普通变量一样。注意：$0 程序名不计算在内。最后一个变量不能使用${$#}来获取，而应该使用${!#}来获取。
8. $*和$@变量都是命令行参数的列表，在不加双引号的时候，输出是一样的：($*、$@不加双引号、输出一样)

test.sh

```
#/bin/sh
echo 这是$*的输出:
for i in $*
do
    echo $i
done
echo 这是$@的输出:
for i in $@
do
    echo $i
done
```

运行命令“./test.sh a b c”输出都是：

```
lan@android:~$ ./test.sh a b c
这是 a b c 的输出:
a b c
这是 a b c 的输出:
a b c
```

加了双引号之后，输出就不一样了：

- $*：作为一个独立的单词保存。
- $@：保存为一个链表。

test.sh

```
#/bin/sh
echo 这是$*的输出:
for i in "$*"
do
    echo $i
done
```

```
echo 这是$@的输出:
for i in "$@"
do
      echo $i
done
```

输出是:

```
lan@android:~$ ./test.sh a b c
这是 a b c 的输出:
a b c
这是 a b c 的输出:
a
b
c
```

9. linux 使用特殊字符将选项和参数分开，该字符会告诉脚本选项何时结束以及普通参数何时开始。这个特殊的字符是双破折线（--）。shell 使用双破折线来表明选项结束了。看到双破折线，脚本会安全地将剩下的命令行参数当做参数来处理，而不是选项。

```
#!/bin/bash
# extracting options and parameters

while [ -n "$1" ]
do
   case "$1" in
   -a) echo "Found the -a option" ;;
   -b) echo "Found the -b option";;
   -c) echo "Found the -c option" ;;
   --) shift
       break ;;
   *) echo "$1 is not an option";;
   esac
   shift
done

count=1
for param in $@
do
   echo "Parameter #$count: $param"
   count=$[ $count + 1 ]
done
$
```

使用双破折线来将命令行上的选项和参数划分开来：

```
$ ./test16 -c -a -b -- test1 test2 test3
Found the -c option
Found the -a option
Found the -b option
Parameter #1: test1
Parameter #2: test2
Parameter #3: test3
$
```

当脚本遇到双破折线时，它就停止处理选项了，并将剩下的参数都当做命令行参数。

10. 在创建 shell 脚本时，脚本选项含义尽量使用 linux 命令行中通用的含义。

11. read 命令接受标准输入或另一个文件描述符的输入。read 命令的格式如下：

```
read variable
```

从键盘中读取输入，并保存到变量 variable 中。

read 命令包含-p 选项，允许你直接在 read 命令行指定提示符：

```
#!/bin/bash

read -p "Enter your name:" name            #输入的内容保存到 name 中

echo "hello $name, welcome to my program"

exit 0
```

也可以为输入指定多个变量，输入的每个数都会分配给表中的下一个变量。如果变量表在数据之前用完了，剩下的数据就都会分配给最后一个变量：

```
#!/bin/bash

read -p "Enter your name:" first last

echo "hello $name, welcome to my program"

exit 0
```

12. 使用 read 命令时，使用-t 选项来指定一个计时器，指定 read 命令等待输入的秒数，防止脚本一直在等待用户输入。

```
If read -t 5 -p “Please enter your name:” name
```

计时器过期后 read 命令会以非零退出状态码退出。

13. 使用 read 命令的-s 选项阻止将传给 read 命令的数据显示到显示器上（如输入密码）。

14. 也可以用 read 命令来读取 linux 系统上文件里保存的数据。每次调用 read 命令会从文件中读取一行文件，当文件中再没有内容时，read 命令会退出并返回非零退出状态码。

    实现该功能常见方法是将文件运行 cat 命令后的输出再通过管道直接传给含有 read 命令的 while 命令。

```
#!/bin/bash
count=1
cat test | while read line   #cat 命令的输出作为 read 命令的输入,read 读到的值
                    #放在 line 中
do
  echo "Line $count:$line"
  count=$[ $count + 1 ]         //注意中括号中的空格。
done
echo "finish"
exit 0
```

第 14 章 呈现数据

1. linux 使用文件描述符来标识每个文件对象。文件描述符 0 表示标准输入 STDIN，文件描述符 1 表示标准输出 STDOUT，文件描述符 2 表示标准错误 STDERR。

- 使用输入重定义符<来替换标准输入

```
Cat < testfile                    # 将 testfile 文件作为 cat 源
```

- 使用输出重定向符>来替换标准输出

```
ls -l > test2                     # 将 ls 的输出到 test2 文件中
```

- 使用>>将数据追加到文件中

```
who >> test2
```

注意：shell 处理错误消息和跟处理普通输出是分开的。

2. 类似”2>xxx”的形式是将文件描述符 2 重定向到 xxx。例如重定向错误：

- 只重定向错误：

```
ls -al badfile 2> test4
```

- 重定向错误和输出，要使用两个重定向符号：

```
ls -al test test2 test3 badtest 2> test6 1>test7
```

- 特殊的重定向符号&>用于重定向错误和输出：

```
ls -al test test2 test3 badtest &> test7
```

3. 在文件描述符数字之前加一个 and 符（&）实现临时重定向：

```
echo “This is an error message” >&2                    # 打印一行消息到标准错误
```

4. 永久重定向脚本中的所有命令：使用 exec 命令告诉 shell 在脚本执行期间重定向某个特定文件描述符：

```
Exec 1> testout                                        # 重定向标准输出到 testout 文件
```

5. 创建输出文件描述符：

```
Exec 3 >  test13out
echo ” this is one”
echo ” this is two” >&3                                # 这一行输出到 test13out，其余输出到标准输出
echo ” this is three”
```

6. 重定向文件描述符：

```
Exec 3>&1
```

```
exec 1>testout
```

注意，在上面的例子中，第二句虽然把 stdout 重定向到 testout 中，但发送给文件描述符 3 的内容依然会在 stdout 中输出，尽管它已经重定向过。

7．使用类似下列语句来打开单个文件描述符来作为输入和输出：

```
exec 3<>testfile
```

注意，shell 会维护一个内部指针，指明现在在文件中什么位置，任何读写都会从文件指针上次保存的位置开始。

8．"2>&1"的含义：

```
command >out.file  2>&1 &
```

command >out.file 是将 command 的输出重定向到 out.file 文件。2>&1 是将标准出错重定向到标准输出，这里的标准输出已经重定向到了 out.file 文件，即将标准出错也输出到 out.file 文件中。最后一个& ， 是让该命令在后台执行。

9．要关闭文件描述符，将它重定向到特殊符号&-。脚本中看起来如下：

```
Exec 3>&-
```

该语句会关闭文件描述符 3。

10. 类似后台进程等不需要显示脚本输出，这时可以把输出重定向到/dev/null。

11. mktemp 命令用于在/tmp 目录中创建临时文件。创建的文件不用默认的 umask 值，文件属主拥有完整的权限，除 root 之外，其他人无法访问。

- 使用 mktemp 命令创建临时文件时只要指定一个文件名模板，模板可以包含任意文本文件名，在文件名末尾加上 6 个 X 就行了：

```
$ mktemp testing.XXXXXX
```

mktemp 命令会用 6 个字符码替换这 6 个 X，从而保证文件名在目录中是唯一的。

- 使用 mktemp 的-t 选项会强制 mktemp 命令在系统的临时目录来创建该文件：

```
$ mktemp -t test.XXXXXX
/tmp/test.xG3374
```

mktemp 命令会返回用来创建临时文件的全路径，而不是只有文件名

12. mktemp 命令的-d 选项用来创建一个临时目录而不是临时文件。

13. tee 命令可以指定两个输出地，一个是 stdout，另一个是 tee 命令行所指定的文件名：

```
tee filename
```

tee 命令重定向从 stdin 过来的数据，所以可以用它和管道命令一起将任何命令的输出重定向。

```
$ date | tee testfile
```

第 15 章 控制脚本

1. linux 通过信号来在运行在系统上的进程之间通信。默认情况下，bash shell 会忽略收到的任何 SIGQUIT 和 SIGTERM 信号。
2. Bash shell 允许用键盘上的键组合生成两种基本的 linux 信号。Ctrl+C 组合键会生成 SIGINT 信号，并将其发送给 shell 中当前运行的所有进程。Ctrl+Z 组合键停止 shell 中运行的任何进程。停止进程不同于终止进程，停止进程会让程序继续保留在内存中，并能从上次停止的位置继续运行。
3. trap 命令允许指定 shell 脚本要观察哪些 linux 信号并从 shell 中拦截。如果脚本收到了 trap 命令中列出的信号，它会阻止它被 shell 处理，而在本地处理它。trap 命令的格式是：

```
Trap commands signals
```

如果拦截多个信号，用空格隔开。

```
Trap “echo ‘sorry! I have trapped Ctrl-C’” SIGINT SIGTERM
```

拦截到 SIGINT 或 SIGTERM 信号后显示一行简单文本。

4. 在 trap 命令后加上 EXIT 信号捕捉 shell 脚本的退出。

```
Trap “echo byebye” EXIT
```

当脚本退出时，捕捉被触发，输出 byebye。

5. 用单破折线作为命令，后面跟要移除的信号来移除一组信号捕捉，将其恢复到正常状态。

```
Trap “echo byebye” EXIT
……
trap - EXIT
```

6. 要在命令行界面以后台模式运行 shell 脚本，只要在命令后加个&符就行了。

```
$ ./test1 &
```

可以在命令行提示符下同时启动多个后台作业。linux 系统每次启动一个新作业时，都会分配一个新的作业号和 PID。

7. nohup 命令让脚本一直以后台模式运行，直到其完成，即使退出了终端会话。
   nohup 命令的格式如下：

```
$ nohup ./test1 &
```

终端关闭后，为了保存命令产生的输出，nohup 自动将 stdin 和 stdout 重定向到 nohup.out 文件中。

8. 作业控制中的关键命令是 jobs 命令。jobs 命令允许查看 shell 当前正在处理的作业。

```
$ jobs
[1]+  Stopped                 ./test4
[2]-  Running                 ./test4 >test4out &
$
```

带加号的作业被当做默认作业。使用任何控制命令时，如果不指定作业号，则该作业会被当成操作对象。

9. 要以后台模式重启一个作业，可用 bg 命令加上作业号，以前台模式重启作业，可用 fg 命令加上作业号。

10. linux 系统提供了多个在预选时间运行脚本的方法：at 命令和 cron 表。

11. at 命令允许指定 linux 系统何时运行脚本。at 命令的使用方法有两种：

- 使用-f 指定脚本文件。

```
$ at -f test.sh now
```

- 从标准输入获取命令，然后使用 ctrl+D 来结束输入命令：

```
[root@localhost ~]# at 5pm+3 days
at> /bin/ls
at> <EOT>
```

<EOT>就是 ctrl+D。

```
lan@lan-Aspire:~$ at now
warning: commands will be executed using /bin/sh
at> echo "hello world" > my.txt
at> <EOT>
job 8 at Thu Feb 14 15:27:00 2019
```

12. at 命令支持各种时间格式，常见的时间格式：

(1) 绝对时间+日期

- at 5:30pm
- at 17:30 today
- at 17:30 24.2.99
- at 17:30 2/24/99
- at 17:30 Feb 24

(2) 相对时间

- at now + 5 hours
- at now + 300 minutes

13. atq 命令可以查看系统中有哪些作业在等待。

14. atrm 命令来删除等待中的作业。

```
$ Atrm 59
```

59 是作业号。

15. linux 系统使用 cron 程序来计划要定期执行的作业。cron 程序会在后台运行并检查特殊的称做 cron 时间表的表，来获得计划要执行的作业。cron 时间表的格式如下：(分、时、日、月、星期、命令)

```
Min hour dayofmonth month dayofweek command
```

如每天 10:15 运行一个命令：

```
15 10 * * * command
```

- minute： 表示分钟，可以是从 0 到 59 之间的任何整数。
- hour：表示小时，可以是从 0 到 23 之间的任何整数。
- day：表示日期，可以是从 1 到 31 之间的任何整数。
- month：表示月份，可以是从 1 到 12 之间的任何整数。
- week：表示星期几，可以是从 0 到 7 之间的任何整数，这里的 0 或 7 代表星期日。
- command：要执行的命令，可以是系统命令，也可以是自己编写的脚本文件。

在以上各个字段中，还可以使用以下特殊字符：

- 星号（*）：代表所有可能的值，例如 month 字段如果是星号，则表示在满足其它字段的制约条件后每月都执行该命令操作。

```
*/5 * * * * root /usr/libexec/atrun
```

- 逗号（,）：可以用逗号隔开的值指定一个列表值，例如，“3,15 * * * * command”表示第个小时的第 3 分钟和第 15 分钟。

```
3,15 8-11 * * * command
```

在上午 8 点到 11 点的第 3 和第 15 分钟执行命令

- 中杠（-）：可以用整数之间的中杠表示一个整数范围，例如“2-6”表示“2,3,4,5,6”

```
0 23-7/2，8 * * * echo "Have a good dream" >> /tmp/test.txt
```

晚上 11 点到早上 8 点之间每两个小时和早上八点执行命令。

- 正斜线（/）：可以用正斜线指定时间的间隔频率，例如“0-23/2”表示每两小时执行一次。同时正斜线可以和星号一起使用，例如*/10，如果用在 minute 字段，表示每十分钟执行一次。

```
0 */2 * * * echo "Have a break now." >> /tmp/test.txt
```

每两个小时执行一次命令。

16. 每个系统用户都可以用他自己的 cron 时间表（包括 root 用户）来运行计划好的任务。linux 提供了 crontab 命令来处理 cron 时间表。要列出已有的 cron 时间表，可以用-l 参数:

```
$ crontab -l
0,15,30,45 18-06 * * * /bin/echo `date` > dev/tty1
```

可以使用这种方法在$HOME 目录中对 crontab 文件做一备份:

```
$ crontab -l > $HOME/mycron
```

要为 cron 时间表添加条目，可以用-e 参数。可以像使用 vi 编辑其他任何文件那样修改 crontab 文件并退出。在你操作时，crontab 命令会用已有的 cron 时间表（不存在则为空文件）启动一个文本编辑器。

当创建的脚本不要求有精确的执行时间时，用预配置的 cron 脚本目录会更方便。有 4 个基本目录：hourly、daily、monthly 和 weekly。目录名为/etc/cron.*ly，如/etc/cron.hourly。cron 程序要求系统 7*24 小时运行。

17.大多数 linux 发行版提供了一个本地开机文件专门让系统管理员添加开机时运行的脚本。

表15-5　Linux本地开机文件位置

发　行　版	文件位置
debian	/etc/init.d/rc.local
Fedora	/etc/rc.d/rc.local
Mandriva	/etc/rc.local
openSuse	/etc/init.d/boot.local
Ubuntu	/etc/rc.local

第 16 章 创建函数

1. 有两种格式可以用来在 bash shell 脚本中创建函数：

- 第一种采用关键字 function，后跟分配给该代码块的函数名：

```
Function name{
  commands
}
```

name 属性定义了赋予函数的唯一名称。你必须给脚本中定义的每个赋一个唯一的名称。

- 第二种格式跟在其他编程语言中定义函数很像：

```
Name() {
  commands
}
```

函数名后的圆括号为空，表明正在定义的是一个函数。

2. 在脚本中使用函数，在行上指定函数名就行了，跟使用其他 shell 命令一样。

3. 在 shell 中，有 3 种不同的方法为函数生成退出状态码：

- 默认退出状态码：默认情况下，函数的退出状态码是函数中最后一条命令返回的退出状态码。
- 使用 return 命令来指定退出状态码。
- 使用函数输出：将函数的输出保存到 shell 变量中，可以用这种技术来获得任何类型的函数输出，并将其保存到变量中。

```
$ cat test5b
#!/bin/bash
# using the echo to return a value

function dbl {
   read -p "Enter a value: " value
   echo $[ $value * 2 ]
}

result=`dbl`
echo "The new value is $result"
$
$ ./test5b
Enter a value: 200
The new value is 400
$
$ ./test5b
Enter a value: 1000
The new value is 2000
$
```

4．在脚本中指定函数参数，必须将参数和函数放在同一行，像这样：

```
Funcl $valuel 10
```

模板

```
# badtest1
Function add{
  使用$#、$1、$2 等等获取参数
}
……
value=`addem 10 15`
```

注意，虽然使用$#、$1、$2 等等获取参数，但参数不能直接通过命令行来获取。下面方式是错误的。

```
$ ./badtest1 10 15
```

5．在函数内部使用的局部变量使用 local 关键字定义：

```
local  temp
```

也可以在给变量赋值时在赋值语句中使用 local 关键字：

```
local temp=$[ $value + 5 ]
```

local 关键字保证了变量只局限在该函数中。

6．把多个值放在括号里，给数组赋值，值与值之间用空格分隔：

```
$ mytest=(one two three four)
```

访问单独数组元素时，需要使用它在数组中的索引值：

```
$ echo ${mytest[2]}
three
```

用星号作为通配符可显示整个数组变量：

```
$ echo ${mytest[*]}
one two three four
```

7．使用星号向函数传递数组变量：

```
Function testit{
……
}

myarray=(1 2 3 4 5 6)
testit ${myarray[*]}
```

8．source 命令会在当前的 shell 上下文中执行命令，而不是创建一个新的 shell 来执行命令。source 命令又叫点操作符，要在 shell 脚本中运行 myfuncs 库文件，只需添加下面这行：

```
Source ./myfuncs
或者
. ./myfuncs                    # 注意，两个点的
```

第 18 章 初识 sed 和 gawk

1．Sed 是流编辑器，它一次只处理一行信息，也就是说 sed 以行为处理单位，每次从标准输入/文本获取一行信息，存储到其“模式空间 ”（pattern space，实际上是一个临时缓冲区）中，在这个模式空间中，sed 就会将脚本中的处理命令做完，然后就将处理完的数据输出到标准输出（屏幕）。然后 sed 再获取下一行至模式空间处理，这样循环直至文件末尾。Sed 不会修改文本文件的数据。使用 sed 命令格式如下：

```
Sed options script file
```

script 参数指定了将作用在流数据上的单个命令。如果需要多个命令，你必须用-e 选项来在命令行上指定它们，或用-f 选项来在单独的文件中指定。

- s 命令用斜线间指定第二个文本字符串来替换第一个文本字符串

```
$ echo “this is a test” | sed s/test/big test/
this is a big test
```

示例使用第二个 big test 替换第一个 test。注意，sed 最后一个/不能省略。

- sed 的 s 命令默认只替换一个，如果需要替换所有字符串，可以在后面加个“g”：

```
lan@android:~$ echo Hello world hahaha | sed 's/ha/em/g'
```

- 要在 sed 命令行上执行多个命令时，只要用-e 选项就可以了，也可以使用 bash shell 中的次提示符来分隔命令，而不用分号：

```
$ sed -e 's/brown/green/; s/dog/cat/' data1
The quick green fox jumps over the lazy cat.
The quick green fox jumps over the lazy cat.
The quick green fox jumps over the lazy cat.
The quick green fox jumps over the lazy cat.
$
```

```
$ sed -e '
> s/brown/green/
> s/fox/elephant/
> s/dog/cat/' data1
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
$
```

- 通常，处理大量 sed 命令时将它们放一个文件中更方便。可以在 sed 命令中用-f 选项来指定文件：(-f、指定文件)

```
$ cat script1
s/brown/green/
s/fox/elephant/
s/dog/cat/
$
$ sed -f script1 data1
```

注意：data1 是要执行 sed 命令的文件。

- -n：仅显示 script 处理后的结果。

2. 也可以指定在哪几行中查找：

```
sed -i '3,5s/filter/haha/2' test.log
```

在第 3 到第 5 行查找 filter，并替换为 haha。

3. sed 的命令可以有两种使用方式，一种是使斜杠：

```
sed '/hello/d' helloworld
```

另一种使用行号加命令的方式：

```
sed '1,3d' helloworld
```

4. 命令说明：（a 增、c 取、d 删、i 插、p 列、s 取代）

- a：新增， a 的后面可以接字串，而这些字串会在新的一行出现。

```
$ sed -e ‘4a newline’ testfile
```

使用 sed 在第四行后添加新字符串 。

- c：取代， c 的后面可以接字串，这些字串可以取代 n1,n2 之间的行。这个并非是一行一行地取代，而是将所有行取代成一行

```
$ nl /etc/passwd | sed '2,5c No 2-5 number'
```

将第 2-5 行的内容取代成为『No 2-5 number』。

- d：删除，因为是删除啊，所以 d 后面通常不接任何咚咚；

```
$ nl /etc/passwd | sed '2,5d'
```

将 /etc/passwd 的内容列出并且列印行号，同时，请将第 2~5 行删除！

- i： 插入， i 的后面可以接字串，而这些字串会在新的一行出现；

```
Cat /etc/passwd | sed '2i drink tea'
```

注意这个插入指的是行前行后插入，而无法在行中插入，在行中插入只能使用 s 命令：

```
$ echo "hello" | sed 's/e/&11/g'
he11llo
```

- p：打印，亦即将某个选择的数据印出。通常 p 会与参数 sed -n 一起运行。

```
$ nl /etc/passwd | sed '/root/p'
```

搜索 /etc/passwd 有 root 关键字的行。

- s：取代。

5. 正则表达式可以直接使用在 sed 中：

```
sed '/pattern/p' datafile
```

6. Sed 的 g 选项可以替换所有内容，而不是单行内容：

```
[root@www ~]# /sbin/ifconfig eth0 | grep 'inet addr' | sed 's/^.*addr://g'
192.168.1.100 Bcast:192.168.1.255 Mask:255.255.255.0
```

将 addr:前面的内容替换为空。

7. sed 使用-i 参数来直接修改文件的内容：

```
[root@www ~]# sed -i 's/\.$/\!/g' regular_express.txt
```

8. 在 sed 中，使用&来表示已匹配到的字符串：

```
# sed 's/^192.168.0.1/&localhost/' file
192.168.0.1localhost
```

9. 正则表达式 \w\+ 匹配每一个单词：

```
# sed '/\w\+/p'
```

这里的\w\+其实是由\w 和+组成，\w 是匹配一个单词字符，\w\+也就是匹配多个单词字符。

- \W 是匹配非单词字符。注意一个小写，一个大写。

10. \(..\) 用于匹配子串，对于第一个正则表达式匹配到的子串就标记为 \1，依此类推匹配到的第二个正则表达式匹配到的结果就是 \2，例如：

```
# echo aaa BBB | sed 's/\([a-z]*\) \([A-Z]*\)/\2 \1/'
BBB aaa
```

- \1 和\2 这种形式的获取值只能用于获取\(regexp\)这个正则表达式的值。

11. 基本正则表达式：

- *：将*前面的正则表达式匹配的结果重复任意次(含 0 次)。

- \+：与星号(*)相同，只是至少重复 1 次，GNU 的扩展功能。
- \?：与星号(*)相同，只是最多重复 1 次，GNU 的扩展功能。
- \{i\}：与星号(*)相同，只是重复指定的 i 次。
- \{i,j\}：与星号(*)相同，只是重复 i 至 j 次。
- \{i, \}：与星号(*)相同，只是至少重复 i 次。
- \(regexp\)：将 regexp 看作一个整体，用于后向引用，与\digit 配合使用，例如“sed 's/\([a-z]\+\) \([A-Z]\+\)/\2 \1/'”。
- .：匹配任意单个字符。
- ^：匹配模版空间开始处的 NULL 字符串。
- $：匹配的是模版空间结束处的 NULL 字符串。
- [list]：匹配方括号中的字符列表中的任意一个。
- [^list]：否定匹配方括号中的字符列表中的任意一个。
- regexp1\|regexp2：用在相邻的正则表达式之间，表示匹配这些正则表达式中任一个都可以。匹配是从左向右开始的，一旦匹配成功就停止匹配。
- regexp1regexp2：匹配 regexp1 和 regexp2 的连接结果。
- \digit：匹配正则表达式前半部分定义的后向引用的第 digit 个子表达式。digit 为 1 至 9 的数字, 1 为从左开始。
- \n：匹配换行符。
- \meta：将元字符 meta 转换成普通字符，以便匹配该字符本身，有$、 *、 .、 [、 \ 和 ^。

```
# echo aaa BBB | sed 's/\([a-z]\+\) \([A-Z]\+\)/\2 \1/'
BBB aaa
```

- shell 的这些正则表达式和普通的正则表达式在问号和加号的使用上略有不同，特别特别要注意那些带\的。

12. 在 shell 中，相比普通的正则表达式，shell 的正则表达式中的星号*和点号.都不需要转义，而加号+和问号?则需要转义。

```
$ echo "111(222)333" | sed 's/\([0-9]\+\)(\([0-9]\+\))\(.*\)/\3/g'
```

```
333
$ echo "111(222)333" | sed 's/\([0-9]\+\?\)\([0-9]\+\)\(.*\)/\3/g'
333
```

13. 转义字符其实就是改变了它原有的含义的字符。有两种：

- 普通字符转义为特殊字符：\n、\W 等都是这种转义字符。
- 特殊字符转义为普通字符：在 sed 中/就是特殊字符。

14. 使用反斜线\对字符进行转义，例如：

```
sed "/a\/b\/Makefile:106: warning:/d" file
```

删除文件中的“a/b/Makefile:106: warning:”，这里的“\/”就是将特殊字符转义为普通字符。

15. sed 中的括号和大括号、加号和问号要转义。

16. 在一个大文件中，sed 默认只对模式之内的内容进行操作，而模式之外的内容原样输出。所以如果想提取数据，必须先进行 grep 操作提取到符合模式的那行数据，再进行 sed 操作获取数据：

```
$ cat 1.html | grep '<title>\(.*\)<\/title>' | sed 's/<title>\(.*\)<\/title>/\1/g'
```

linux shell 字符串操作详解 （长度，读取，替换，截取，连接，对比，删除，位置 ） - dongwuming 的专栏 - CSDN 博客

要提取一个网页的标题，必须先 grep 出标题那行数据，再对该行数据进行 sed 操作。

17. awk 和 sed 不太一样，sed 更像是一个 linux 命令，而 awk 是一种脚本语言。

18. AWK 是一种处理文本文件的语言，是一个强大的文本分析工具。awk 大括号之内的部分的语法上有点类似 c 语言。

```
awk [选项参数] 'script' var=value file(s)
或
awk [选项参数] -f scriptfile var=value file(s)
```

```
$ awk '$1==2 {print $1,$3}' log.txt    #命令
```

命令要加引号。

选项参数：

- -F：指定输入文件分隔符，如-F:。

```
$  awk -F, '{print $1,$2}'   log.txt
```

注意，print 里面的大括号不可以省略。

- -f scripfile：从脚本文件中读取 awk 命令。

```
$ awk -f cal.awk log.txt
```

- -v var=value：赋值一个用户定义变量。

```
$ awk -va=1 '{print $1,$1+a}' log.txt
```

这里的-va=1 指的是“-v a=1”，这里可以省略空格。

19.awk 内置了几种函数：

- 算数函数
- 字符串函数
- 时间函数
- 位操作函数
- 其它函数

20.awk 的其它函数包括

(1) close(expr)：关闭管道。

(2) delete：从数组中删除元素。

(3) exit：终止脚本执行。

(4) flush：刷新打开文件或管道的缓冲区　。

(5) getline：读入下一行。

```
$ awk '{if ($0 ~/Shyam/) next; print $0}' marks.txt
```

(6) next：停止处理当前记录，并且进入到下一条记录的处理过程。

```
$ awk '{if ($0 ~/Shyam/) next; print $0}' marks.txt
```

(7) nextfile：停止处理当前文件，从下一个文件第一个记录开始处理。

```
$ awk '{ if ($0 ~ /file1:str2/) nextfile; print $0 }' file1.txt file2.txt
```

(8) return：从用户自定义的函数中返回值。

(9) system：执行特定的命令然后返回其退出状态。返回值为 0 表示命令执行成功；非 0 表示命令执行失败。

```
$ awk 'BEGIN { ret = system("date"); print "Return value = " ret }'
2019 年 02 月 04 日 星期一 02:17:27 CST
Return value = 0
```

21.awk 有几个字符串函数：

(1) index(String1, String2)：返回 String2 在 String1 中的位置的索引，如果 String1 中不存在 String2，返回 0。注意，返回的索引值是从 1 开始编号的。

```
$ awk 'BEGIN{info="this is a test2010test!";print index(info,"test");}'
11
```

(2) gsub(r,s,t)：在 t 中使用 s 将所有的 r 替换掉。t 是可选的，如果省略 t 表示在当前的 $0 使用 s 将 r 替换掉。

```
$ awk 'BEGIN{info="this is a test2010test!";gsub(/[0-9]+/,"!",info);print info}'
this is a test!test!
```

(3) length [(String)]：返回字符串 String 的长度。

```
$ awk 'BEGIN{info="this is a test2010test!";print length(info);}'
23
```

还有个类似的 blength [(String)]函数，作用和 length()是差不多的，区别在于 blength()函数如果未给出 String 参数，则返回整个记录的长度（即$0 记录变量）。

(4) substr(s,p,n)：返回字符串 s 中从 p 开始长度为 n 的后缀部分，其中 n 是可选的，如果省略 n，则返回从 p 开始到结尾的部分。p 所代表的索引是从 1 开始编号的。

```
$ awk 'BEGIN{info="this is a test2010test!";print substr(info,6,2);}'
is
```

(5) match(s,r)：测试字符串 s 是否包含匹配子串 r，如果包含，则返回指定的索引值。

```
$ awk 'BEGIN{info="this is a test2010test!";print match(info,"is");}'
3
```

(6) split(s,a,fs)：使用 fs 作为分隔符，将字符串 s 分成一个数组，然后保存到 a 中。

```
$ awk 'BEGIN {split("123#456#789",myarray,"#");for(i=1;i<4;i++)print myarray[i];}'
123
456
789
```

(7) tolower(String)：将 String 更改为小写。

toupper(String)：将 String 更改为大写。

```
$ awk 'BEGIN {print tolower("ABC");}'
abc
```

(8) sprintf(Format, Expr, Expr, . . .)：其作用和 c 语言中的 sprintf 是一样的，都是用于格式化输出。

```
$ awk 'BEGIN{n1=124.113;n2=-1.224;n3=1.2345; printf("%.2f,%.2u,%.2g,%X,%o\
n",n1,n2,n3,n1,n1);}'
124.11,4294967295,1.2,7C,174
```

22. 常用的 awk 内置变量：

(1) $0：当前记录。

(2) $1~$n：当前记录的第 n 个字段，字段间由 FS 分隔。

(3) NF：当前记录中的字段个数，就是有多少列。

(4) NR：已经读出的记录数，就是行号，从 1 开始。

(5) RS：输入的记录分隔符，默认为换行符。

(6) ORS：输出的记录分隔符，默认为换行符。

(7) FS：输入字段分隔符，默认是空格。

(8) OFS：输出字段分隔符，默认也是空格。

(9) ARGC：命令行参数个数。

(10) ARGV 命令行参数数组。

(11) FILENAME：当前输入文件的名字。

(12) ARGIND：当前被处理文件的 ARGV 标志符。

(13) FNR：当前记录数。

23. RS、ORS、FS 和 OFS 的区别：

- RS：记录分隔符，awk 读取文件时，默认将一行作为一条记录。
- ORS：记录输出分隔符。

```
$ cat /etc/passwd | awk -F: 'BEGIN{ORS=","}{print $1}'
root,daemon,bin,sys,sync,games,man,lp,mail,news,uucp,proxy,www-
data,backup,list,irc,gnats,nobody,systemd-network,systemd-
resolve,syslog,messagebus,_apt,uuidd,avahi-
autoipd,usbmux,dnsmasq,rtkit,cups-pk-helper,speech-
dispatcher,whoopsie,kernoops,saned,pulse,avahi,colord,hplip,geoclue,gnome-
initial-setup,gdm,lan,mysql,
```

- FS：其实就是-F 参数指定的那个分隔符。
- OFS：FS 是用来指定输入的，而 OFS 就是用来指定输出的。

```
$ cat /etc/passwd | awk -F: 'BEGIN{OFS=",";}{print $1,$2}'
root,x
daemon,x
bin,x
sys,x
```

24. awk 中也可以使用？：操作符：

```
$ awk 'BEGIN{info="this is a test2010test!";print index(info,"test")?"ok":"no
found";}'
ok
```

25. awk 中对字符串进行拼接的方式和 shell 中是一样的，但变量的使用上和 shell 的变量使用不一样，在 awk 中，引用变量不需要加$：

```
$ cat /etc/passwd | awk -F: '$1=="root"{var=$1$5;print var}'
rootroot
```

26. 通过管道传送给 awk 的内容会显示成一行，而从文件中读取的内容则是一行一行地读取。

27. 在 awk 中，字符串要使用双引号来括起来：

```
awk -F: '$1=="lan" {print $1}' /etc/passwd
```

条件判断在大括号外面。

28. 循环语句:

(1) For 循环

```
$ awk 'BEGIN { for (i = 1; i <= 5; ++i) print i }'
```

(2) While 循环:

```
$ awk 'BEGIN {i = 1; while (i < 6) { print i; ++i } }'
```

(3) Do-While:

```
$ awk 'BEGIN {i = 1; do { print i; ++i } while (i < 6) }'
```

(4) Break

```
$ awk 'BEGIN {
    sum = 0; for (i = 0; i < 20; ++i) {
       sum += i; if (sum > 50) break; else print "Sum =", sum
   }
}'
```

(5) Continue

```
$ awk 'BEGIN {for (i = 1; i <= 20; ++i) {if (i % 2 == 0) print i ; else continue} }'
```

(6) Exit 用于结束脚本程序的执行

```
$ awk 'BEGIN {
   sum = 0; for (i = 0; i < 20; ++i) {
      sum += i; if (sum > 50) exit(10); else print "Sum =", sum}}
'
```

```
Sum = 0
Sum = 1
Sum = 3
Sum = 6
Sum = 10
Sum = 15
Sum = 21
Sum = 28
Sum = 36
Sum = 45
```

29. 除了匹配输出内容，其余的语句都要使用大括号{}括起来。

30. 在 script 中使用比较运算符来选择匹配的输出内容：

```
awk -F: '$1=="lan" {print $1}' /etc/passwd
```

31. 在 awk 中，BEGIN 关键词用于在读取数据进行处理之前执行一些初始化的工作，而 END 关键词是处理完所有的行后需要执行的收尾工作。类似如下：

```
$ cat cal.awk
#!/bin/awk -f
#运行前
BEGIN {
   math = 0
   english = 0
   computer = 0

   printf "NAME    NO.   MATH  ENGLISH  COMPUTER   TOTAL\n"
   printf "---------------------------------------------\n"
}
#运行中
{
   math+=$3
   english+=$4
   computer+=$5
   printf "%-6s %-6s %4d %8d %8d %8d\n", $1, $2, $3,$4,$5, $3+$4+$5
}
#运行后
END {
   printf "---------------------------------------------\n"
```

```
   printf "  TOTAL:%10d %8d %8d \n", math, english, computer
   printf "AVERAGE:%10.2f %8.2f %8.2f\n", math/NR, english/NR, computer/NR
}
```

- 一般如果不需要读取数据，可以直接在 BEGIN 中进行处理。

32. 在 awk 中，$0 表示整行数据，而$1、$2 等表示用字段分隔符分隔之后的数据。

33. awk 定义了几个内建变量，包括$0 和$1 到$9 都是内建变量。

34. Awk 提供了另一个内建变量, 叫做 NR，它会存储当前已经读取了多少行的计数。

35. printf 语句用来打印输入的数据。

```
awk -F: '{print $1}' /etc/passwd
```

36. 在 awk 中只能使用 print 或 printf 来输出内容，不能使用 echo。

37. print 省略内容表示输出整行，而 printf 省略内容会出错。

```
$ awk -F: '$1=="root" {printf}' /etc/passwd
awk: line 1: no arguments in call to printf

$ awk -F: '$1=="root" {print}' /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

38. printf 不会自动产生空格或者新的行，必须是你自己来创建，所以不要忘了 \n，，而 print 会自动产生换行符。

39. 默认情况下，awk 会处理所有行。可以使用选择模式来选择处理某些匹配的行，如果不匹配则不进行处理：

```
$ awk -F: '$1=="root" {print}' /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

打印出总薪资超过 50 美元的员工的薪酬。

40. 可以使用括号和逻辑操作符与 &&， 或 ||， 以及非！对模式进行组合。

41. awk 中的循环语句同样借鉴于 C 语言，支持 while、do/while、for、break、continue，这些关键字的语义和 C 语言中的语义完全相同。：

- if-else 语句：

```
$ awk 'NR!=1{if($6 ~ /TIME|ESTABLISHED/) print > "1.txt";
```

```
else if($6 ~ /LISTEN/) print > "2.txt";
else print > "3.txt" }' netstat.txt
```

- while 语句。
- for 语句：

```
for 变量 in 串行
do
  执行命令
done
```

例如：

```
#!/bin/bash
for k in $( seq 1 10 )
do
  mkdir /home/kuangl/aaa${k}
  cd /home/kuangl/aaa${k}
  for l in $( seq 1 10 )
  do
  mkdir bbb${l}
  cd /home/kuangl/aaa${k}
  done
  cd ..
done
```

42. 因为 awk 中数组的下标可以是数字和字母。一般而言，awk 中的数组用来从记录中收集信息，可以用于计算总和、统计单词以及跟踪模板被匹配的次数等等。

```
awk -F ':' 'BEGIN {count=0;} {name[count] = $1;count++;}; END{for (i = 0; i < NR; i++) print i, name[i]}' /etc/passwd
0 root
1 daemon
2 bin
3 sys
4 sync
5 games
```

BEGIN 和 END 都拥有自己的大括号。

43. 利用 ls 命令和 awk 可以计算指定格式的文件的总大小：

```
$ ls -l *.txt | awk '{sum+=$6} END {print sum}'
```

计算当前目录下的 txt 文件的总大小。

第 19 章 个人总结

1．export 设置环境变量的方法：

```
#export PATH=$PATH:/opt/au1200_rm/build_tools/bin
```

上述命令将/opt/au1200_rm/build_tools/bin 目录追加到 PATH 中，注意$PATH，有$PATH 则是追加，没有则是重新赋值。

2．shell 内建变量：

(1) BASH：显示 bash 的完整路径名。

(2) BASH_VERSION：bash 的版本。

(3) HISTCMD：在历史指令中的排列编号。

(4) HISTCONTROL：控制指令是否存入历史脚本文件中。

(5) HISTFILE：设定历史脚本文件的路径文件名。

(6) HISTFILESIZE：设定历史脚本文件存储指令的最大行数。

(7) HISTIGNORE：不存入历史脚本文件的指令样式。

(8) HOME：用户的家目录位置。

(9) HOSTNAME：显示主机名。

(10) HOSTTYPE：显示主机形态。

(11) MACHTYPE：描述主机形态的 GNU 格式

(12) MAIL：显示当前用户邮件目录

(13) MAILCHECK：每隔多久就检查一次邮件。

(14) PATH：命令的搜寻路径。

(15) PPID：父进程的进程编号。

(16) RANDOM：随机函数。

(17) UID：用户 ID。

(18) USER：用户名 。

(19) LOGNAME：登录用户的用户名。

(20) PWD：当前目录。

3. 反撇号（`）:主要用于命令替换，允许将执行某个命令的屏幕输出结果赋值给变量。注意：反撇号难以在一行命令中实现嵌套命令替换操作，可以使用 $()来代替反撇号操作，已解决嵌套问题。

```
rpm –qc $(rpm –qf $(which  useradd))
```

4. 一般而言，一个 shell 安装脚本如果可用，那么在本地化安装时只需要修改目录、软件版本地址和配置文件即可。

5. date 命令可以获取当前的日期和时间。格式为：

```
date [参数] [格式]
```

其中格式前要加上“+”：

```
date -d "-2 days ago" +%y-%m-%d
```

■ 重要参数：

- -d：显示指定字符串所描述的时间，而非当前时间。
- -s：指定字符串来分开时间。

例如，显示两天后的日期：

```
date -d "－2 days ago" +%y-%m-%d
```

■ 重要格式：

- %a：显示星期几。
- %c：以系统默认方式显示日期、星期几和当前具体时间。
- %d：显示当前是本月的第几天。
- %y、%m、%d：当前的年、月、日。
- %T：当前用 24 小时制显示的具体时间。(%T、24 小时制、显示时间)

- -d 选项是非常有用的选项，它可以指定时间，其中带 ago 字样的就是 xxx 天之前，不带 ago 字样的就是 xxx 天之后：

- date -d "nov 22"：今年的 11 月 22 日是星期三
- date -d '2 weeks'： 2 周后的日期
- date -d 'next monday'：下周一的日期
- date -d yesterday +%Y%m%d：昨天的日期
- date -d last-month +%Y%m ：上个月是几月
- date -d next-month +%Y%m：下个月是几月

使用 ago 指令，您可以得到过去的日期：

- date -d '30 days ago'： 30 天前的日期

使用负数以得到相反的日期：

- date -d '-100 days'：100 天后的日期
- date -d '50 days'：50 天后的日期

6. date 的日期的格式在“+”后面：

```
date -d "－2 days ago" +%y-%m-%d
```

7. grep 是一种强大的行文本搜索工具，它能使用正则表达式搜索匹配的行文本。

```
grep '搜寻的字符串' filename
```

选项与参数：

- -a： 将 binary 文件以 text 文件的方式搜寻数据
- -c： 计算找到 '搜寻字符串' 的次数
- -i： 忽略大小写的不同，所以大小写视为相同
- -n： 顺便输出行号
- -v： 反向选择，亦即显示出没有 '搜寻字符串' 内容的那一行！
- --color=auto： 可以将找到的关键词部分加上颜色的显示喔！

例如：

```
# grep -n root /etc/passwd
```

```
1:root:x:0:0:root:/root:/bin/bash
30:operator:x:11:0:operator:/root:/sbin/nologin
```

8. grep 可以使用正则表达式：

```
[root@www ~]# grep -n 't[ae]st' regular_express.txt
8:I can't finish the test.
9:Oh! The soup taste good.
```

9. grep 显示的是匹配行的整行内容，而不是只显示指定关键词的内容。例如：

```
$ cat /etc/passwd | grep -n root
1:root:x:0:0:root:/root:/bin/bash
```

10. cut 命令可以从一个文本文件或者文本流中提取指定的文本列。语法格式：

```
cut  [-bn] [file]
cut [-c] [file]
cut [-df] [file]
```

参数：

- -b： 以字节为单位进行分割。这些字节位置将忽略多字节字符边界，除非也指定了 -n 标志。
- -c： 以字符为单位进行分割。
- -d： 将字符分隔，自定义分隔符，默认为制表符。
- -f： 与-d 一起使用，后接数字，指定显示第几个区域。
- -n： 取消分割多字节字符。仅和 -b 标志一起使用。如果字符的最后一个字节落在由 -b 标志的 List 参数指示的范围之内，该字符将被写出；否则，该字符将被排除。

```
lan@android:~$ echo Long,long ago,ddddddd,hhhhhhhhhhh | cut -f 2 -d ,
long ago
```

注意，在 2 后面加个负号表示第 2 个区域到最后的所有文本：

```
lan@android:~$ echo Long,long ago,ddddddd,hhhhhhhhhhh | cut -f 2- -d ,
long ago,ddddddd,hhhhhhhhhhh
```

11. uniq 命令用于检查及删除文本文件中重复出现的行列，一般与 sort 命令结合使用。

- -c：count，在每列旁边显示该行重复出现的次数；
- -d：仅显示重复出现的行列；
- -u：unique，仅显示出一次的行列；

12. sort 用于排序，将文件的每一行作为一个单位，相互比较，比较原则是从首字符向后，依次按 ASCII 码值进行比较，最后将他们按升序输出。

- -r：sort 默认是升序排序的，-r 选项用于降序排序。
- -n：以数值来进行排序，避免出现 10<2 这种非数值排序的情况出现。
- -t：指定间隔符。
- -k：指定排序的列数，通常和-t 结合使用。类似“apple:10:2.5”这种可以使用“sort -n -k 2 -t : facebook.txt”来对第二列进行排序。
- -c：会检查文件是否已排好序，如果乱序，则输出第一个乱序的行的相关信息，最后返回 1。

```
[rocrocket@rocrocket programming]$ sort -n -k 2 -t : facebook.txt
apple:10:2.5
orange:20:3.4
banana:30:5.5
pear:90:2.3
```

13. tr 命令可以对来自标准输入的字符进行替换、压缩和删除。

- -c：反选设定字符。也就是符合第 1 部份的不做处理，不符合的剩余部份才进行转换；
- -d：删除所有指定字符集的字符，如果有多个字符集，只删除第一个字符集中的元素；
- -s：把连续重复的字符删除，只留下一个字符，后面可以接一个列表，表示删除指定的重复的字母；
- -t：先删除第一字符集较第二字符集多出的字符。

```
echo "hello 123 world 456" | tr -d '0-9'    #使用 tr 删除字符

cat testfile |tr a-z A-Z        #将文件 testfile 中的小写字母全部转换成大写字母
```

14. tr 命令中类似下面这条不带参数的语句实际上就是用第二个字符集中的值替换第一个字符集中的相同索引的值。

```
$ echo acb | tr "abc" "xyz"
xzy
```

15.SQL 和 shell 控制语句的异同：

(1) if 语句：

- shell：

```
if command1;
then
      commands
elif command2;
then
      commands
else
      commands
fi
```

- SQL：

```
if 布尔表达式
      then 语句或复合语句
elseif 布尔表达式
      then 语句或复合语句
else
      语句或复合语句
end if
```

  - shell 的 if 语句和 case 是以反写来结束的，也就是 fi 和 esac，而 SQL 语句是以 end if 和 end case 来结果。

(2) for 命令：

- shell：

```
for var in list
do
      commands
done
```

注意 for 后面没有分号。

- SQL：

```
declare n integer default 0;
for r as
      select budget from department
      where dept_name='Music'
do
      set n=n-r.budget
end for
```

- shell 循环中间都是 do..done，而 SQL 往往是 do...end。

(3) while 语句：

- shell：

```
While test command
do
      other commands
done
```

- SQL：

```
while 布尔表达式 do
      语句序列;
end while
```

(4) until：

- shell：

```
Until test commands
do
      other commands
done
```

- SQL：

```
repeat
....
until (后接条件) end repeat;
```

16. expr index 命令可以查找出 substring 中字符在 string 第一次出现的位置，格式为：

```
expr index $string substring
```

例如：

```
str="hello,world";expr index $str 'o'
```

17.iconv 命令用于转换文件的编码方式。参数如下：

- -f：原始文本编码。
- -t：输出编码。

输出控制：

- -c：从输出中忽略无效的字符
- -o：输出文件。

```
#iconv -f GB2312 -t UTF-8 gb1.txt >gb2.txt
```

将 gb1 里的编码从 GB2312 转化成 UTF-8 并重定向到 gb2.txt

18.字符串操作：

(1) ${#string}：获取字符串 string 的长度。

(2) ${string:position}：从位置 position 开始获取 string 的子串。

```
#!/bin/bash
MyString=123456789
echo ${MyString:3}
```

输出 456789。

(3) ${string:position:length}：在$string 中, 从位置$position 开始提取长度为 $length 的子串。

```
#!/bin/bash
MyString=123456789
echo ${MyString:1:5}
```

输出 23456。

(4) ${string#substring}：匹配最短前缀，然后删除。

```
#!/bin/bash
MyString=123456789
echo ${MyString#1234}
```

输出是 56789。注意是从开头开始匹配，例如${MyString#234}就无法匹配。

- #为前缀匹配，%为后缀匹配。

(5) ${string%substring}：匹配最短后缀，然后删除。

```
#!/bin/bash
MyString=123456789
echo ${MyString%789}
```

输出是 123456。

(6) ${string/substring/replacement}：使用$replacement 替换第一个匹配的 $substring。

```
#!/bin/bash
MyString=123456789
echo ${MyString/789/777}
```

输出是 123456777。

(7) ${string//substring/replacement}：使用$replacement 替换所有匹配的 $substring。

```
#!/bin/bash
MyString=127896789
echo ${MyString//789/777}
```

输出是 127776777。

(8) ${string/#substring/replacement}：匹配前缀匹配，并使用$replacement 来替换匹配到的前缀。

```
#!/bin/bash
MyString=127896789
echo ${MyString/#127/777}
```

输出 777456789。

(9) ${string/%substring/replacement}：匹配后缀，并使用$replacement 来替换匹配到的后缀。

```
#!/bin/bash
MyString=127896789
echo ${MyString/%789/777}
```

输出 123456777。

- 注意：使用下面的命令执行这些字符串会出错：

```
Sh test.sh
```

可以通过 bash 来执行：

```
Bash test.sh
```

19.${}相关字符串处理：

(1) ${var-default}：如果 var 没有被声明，那么就以$default 作为其值。

```
#!/bin/bash
echo ${string-123}
```

输出 123。

(2) ${var:-default}：如果 var 没有被声明，或者其值为空，那么就以$default 作为其值。

```
#!/bin/bash
echo ${string:-123}
```

输出 123。

(3) ${var=default}：如果 var 没有被声明，那么就以$default 作为其值。

(4) ${var:=default}：如果 var 没有被声明， 或者其值为空，那么就以$default 作为其值。

- -和=表示如果 var 没有被声明，就定义默认值。而+表示如果声明了 var，则定义变量的值。

(5) ${var+other}：如果 var 声明了，那么其值就是$other，否则就为 null 字符串。

```
#!/bin/bash
MyString=127896789
echo ${MyString+123}
```

输出 123。

(6) ${var:+other}：如果 var 被设置了，那么其值就是$other，否则就为 null 字符串。

```
#!/bin/bash
MyString=127896789
echo ${MyString+123}
```

(7) ${var?err_msg}：如果 var 没被声明，那么就打印$err_msg。

```
#!/bin/bash
echo ${MyString1?字符串未定义}
```

输出：

```
test.sh: 行 4: MyString1: 字符串未定义
```

注意这个并非是只打印 err_msg，err_msg 只是错误信息。

(8) ${!varprefix*}：匹配之前所有以 varprefix 开头进行声明的变量。

```
#!/bin/bash
MyString=127896789
Mys=2
echo ${!My*}
```

输出：

```
MyString Mys
```

(9) ${!varprefix@}：匹配之前所有以 varprefix 开头进行声明的变量。

```
#!/bin/bash
MyString=127896789
Mys=2
echo ${!My@}
```

输出：

```
MyString Mys
```

20.字符串处理：

(1) 字符串拼接：通过直接连接来实现拼接。

```
#!/bin/bash
s1=123
s2=456
s3=$s1$s2
echo $s3
```

(2) 查找包含关系：

- 利用 grep：

```
#!/bin/bash
strA="long string"
strB="string"
```

```
result=$(echo $strA | grep "${strB}")
if [[ "$result" != "" ]]
then
    echo "包含"
else
    echo "不包含"
fi
```

- 利用字符串运算符=~:

```
#!/bin/bash
strA="helloworld"
strB="low"
if [[ $strA =~ $strB ]]
then
    echo "包含"
else
    echo "不包含"
fi
```

字符串运算符 =~ 直接判断 strA 是否包含 strB。

- 利用通配符:

```
#!/bin/bash
A="helloworld"
B="low"
if [[ $A == *$B* ]]
then
    echo "包含"
else
    echo "不包含"
fi
```

- 利用 case in 语句:

```
thisString="1 2 3 4 5" # 源字符串
searchString="1 2" # 搜索字符串
case $thisString in
    *"$searchString"*) echo Enemy Spot ;;
    *) echo nope ;;
esac
```

(3) 查找指定子字符的索引：

- 通过 expr index 命令：

```
str="hello,world";expr index $str 'o'
```

查找字符 o 第一次出现的索引。

- 利用 awk 的 match 方法来获取索引值：

```
str="hello,world";echo $str | awk '{printf("%d\n",match($0,"lo"))}'
```

查找 lo 第一次出现的索引。

- 索引值从 1 开始的，而且只能查找第一次出现的索引。

(4) 通过 sed 的 s 命令实现字符串的插入操作：

```
$ echo "hello" | sed 's/e/&11/g'
he11llo
```

(5) 使用类似“${string/#substring/replacement}”来实现子串替换。

(6) 字符串比较使用的是 test 命令。

```
#!/bin/bash
str1="hello";
str2="hello1";
if [ $str1 = $str2 ];
then
    echo "字符串相同"
else
    echo "字符串不同"
fi
```

输出“字符串不同”

(7) 字符串排序使用的是 sort 命令：

```
#!/bin/bash
var="get the length of me"
echo $var | tr ' ' '\n' | sort   #正序排
```

输出：

```
get
```

```
length
me
of
the
```

(8) 使用 iconv 进行字符串编码转换。

(9) 转化大小写：

- 利用 typeset 进行转换大小写，其中-l 表示小写，-u 表示大写：

```
typeset -u name
name='asdasdas'
echo $name

typeset -l name
name='asdasdas'
echo $name
```

- 利用 tr：

```
$ echo 'HELLO' | tr 'A-Z' 'a-z'
hello
```

21. 正则表达式的符号分几种：

- 可打印字符：平时打印的字符都是可打印字符，例如 a、b 和 c 等。注意，\w、\d 这类字符应该也是可打印字符。
- 非可打印字符：就是空格符\s、回车符\r 和换行符\n 等这类无法直接打印的符号。
- 限定符：用来指定正则表达式的一个给定组件必须要出现多少次才能满足匹配，例如星号*、加号+和问号?都是限定符，还有{n,m}之类的也是限定符。限定符其实就是限定出现次数的。
- 定位符：用于将正则表达式固定到行首或行尾。定位符只有四个，常用的只有^和$，例如^表示匹配输入字符串开始的位置，$表示匹配输入字符串结束的位置。

22. 正则表达式事实上就是一个或多个可打印字符或非可打印字符，后面接零个或一个限定符来组成一个匹配模式：

```
$echo "<H1>Chapter 1 - 介绍正则表达式</H1>" | grep -P '(<.*?>)'
```

其中的.表示匹配任意字符，*?表示非贪婪匹配。

23.正则表达式分为贪婪匹配和非贪婪匹配两种，使用?来指代非贪婪匹配：

- 贪婪匹配：表示尽可能多地匹配字符。

```
$ echo "<H1>Chapter 1 - 介绍正则表达式</H1>" | grep -P '(<.*>)'
```

匹配到“<H1>Chapter 1 - 介绍正则表达式</H1>”

- 非贪婪匹配：尽可能少地匹配字符。

```
$echo "<H1>Chapter 1 - 介绍正则表达式</H1>" | grep -P '(<.*?>)'
```

注意后面多个?号。

24.正则表达式分查找和提取两项操作。

- 查找：就是查找带指定关键词的行。
- 提取：其实就是获取点号.、星号*等等特殊符号对应的值。

25.grep 只能查找到相关的行，但无法提取正则表达式的匹配内容。shell 中使用 sed 来提取正则表达式的内容：

```
$ echo "111(222)333" | sed 's/\([0-9]*\)(\([0-9]*\))\([0-9]*\)/\2 \3 \1/g'
222 333 111
```

26.在 sed 中使用括号时只有两种情况：

- 正则表达式本身带括号。
- 使用\(regexp\)方式对正则表达式进行匹配。

除此之外，无法像 python 这类语言一样，使用括号将其概括为一个整体。例如：

```
$ echo "111(222)333" | sed 's/\([0-9]*\)(\([0-9]*\))(.*)//g'
111(222)333
```

这条语句后面的 333 使用了括号，但 333 本身没有括号，所以这条语句是错的，其输出实际上是 echo 命令的输出。将其改为下面这样就是正确的：

```
$ echo "111(222)333" | sed 's/\([0-9]*\)(\([0-9]*\))\(.*\)/\3/g'
333
```

27.wait 命令可以使当前 shell 进程挂起，等待所指定的由当前 shell 产生的子进程退出后，wait 命令才返回。wait 命令的参数可以是进程 ID 或是 job specification。

28. 内部字段分隔符 IFS 默认为空格、制表和换行符，但要注意的是内部字段分隔符并非只能包含一个空格、制表或换行符，也可以包含多个空格等。例如 ls 命令打出来的字符串每列是对齐的，其分隔符中空格的数量是不确定，也可以使用字段分隔符来操作：

```
$ ls -l | awk '{print $3}'
```

显示第 3 列。

29.