UNIX 操作系统及应用
第七章 — Perl脚本编程语言

李亦农　唐晓晟

hoplee@bupt.edu.cn　txs@bupt.edu.cn

School of Information Communication Engineering, BUPT

内容简介 I

内容简介 II

7. 格式

8. 示例

概述 I

- Perl的全称是：**Practical Extraction and Report Language**—即“实用摘录和报告语言”。
- 也有人称之为：**Pathological Eclectic Rubbish Lister**—即“反常、折衷的垃圾陈列器”。:)
- 其开发者和唯一的维护者是Larry Wall.
- Perl的设计目标是帮助 UNIX 用户完成一些常见的任务，而这些任务对于 Shell 来说过于复杂。
- Perl语言中包含了 C, C++, shell, sed,awk这几个语言的语法，它最初的目的就是用来取代 UNIX 中sed/awk与脚本语言的组合，用来汇整信息，产生报表。因此Perl语言要远远比前面讲的bash复杂强大。
- Perl简洁的结构允许你开发一些非常漂亮的、一步到位的方案或通用的工具。

概述 II

- 由于Perl的高度可移植性，你也可以将这些工具用于其他的任务。
- Perl是免费的，并且各种常见的操作系统上都存在相应版本的发行。
- Perl脚本的第一行必须是：

```
1 #!/usr/bin/perl
```

- Perl的注释和 Shell 一样，以#开始
- Perl更象一个编译器和解释器的组合。Perl程序在运行之前将进行扫描和分析，但是又不产生庞大的目标代码。未来的版本将能够缓存已编译的代码。
- Perl的所有简单语句均以分号结束。

数据类型

- Perl的数据类型主要有三种：标量、列表数组和关联数组。

标量 I

- 所谓标量就是非矢量、非数组的数据。
- Perl中的标量变量以美元符号“$”和一个字母开始，后面可以跟字母、数字和下划线，Perl的变量区分大小写，因此$a和$A是代表不同的变量。和bash中不同的是Perl语言中的变量即使是在最初赋值的时候也必须在变量前面加上“$”符号，而且Perl不要求“=”左右必须没有空格。
- Perl所处理的标量数据包括数字和字符串两大类。
- Perl的数值型数据只有一种类型：浮点数，所有的整数都将当作等效的浮点数来处理。浮点型常量的表示方式和 C 语言里一样。
- Perl的字符串类型数据的取值可以是整个 ASCII 表，并且其长度遵循“无内置限制”的原则 —从 0 直到填满内存。
- 字符串常量有两种形式：单引号字符串和双引号字符串。

标量 II

- 单引号字符串：单引号用于表示字符串的边界，其中可以包含任意字符并且取消所有元字符的特殊含义（唯一的例外是前后相连的\\或\'）；
- 双引号字符串：类似于 C 语言里的字符串，其中的反斜线\用于表示特定的控制字符的开始；并且其中的变量将被置换

- 运算符
  - 算术运算符 `+, -, *, /, **, %`
  - 算术逻辑运算符 `<, <=, ==, !=, >=, >, < = >`
  - 字符串运算符

`.`	串接运算符
`x`	复制运算符
`eq, ne,`	
`lt, gt,`	
`le, ge, cmp`	

- 数值和字符串之间的转换

标量 III

- 如果某个字符串的值被用于数值运算符的运算域，则Perl将自动将其转换为一个十进制浮点数，并且去掉开头的和末尾的非数字元素。
- 同样的转换发生在某个数值作为字符串使用时。

- 赋值运算符：

  =, +=, -=, *=, /=, %=, **=, .=, x=,
  ++(左、右), --(左、右)

- 特殊运算符：

$w?$x:$y	如果$w为真，则返回$x；如果$w为假，则返回$y
($x..$y)	返回从$x到$y之间的值
chop($x)	将$x的字符串值的最后一个字符去掉

标量 IV

- 例：

```
1  #!/usr/bin/perl
2  $folks="100";
3  print "\$folks = $folks \n";
4  print '\$folks = $folks \n';
5  print "\n\n BEEP! \a \LSOME BLANK \
6         \ELINES HERE \n\n";
7  $date = `date +%D`;
8  print "Today is [$date] \n";
9  chop $date;
10 print "Date after chopping off carriage\
11        return: [".$date."]\n";
```

标量 V

- 其输出结果如下：

```
1 $folks = 100
2 $folks = $folks \n
3 BEEP! some blank LINES HERE
4 Today is [03/29/96]
5 Date after chopping off carriage
6   returned:[03/29/96]
```

- 第 3 行显示$folks的值。$之前必须使用转义字符\，以便Perl显示字符串$folks而不是标量变量$folks的值 100。
- 第 4 行使用的是单引号，结果Perl不解释其中的任何内容，只是原封不动地将字符串显示出来。
- 第 7 行使用的是（`），则date +%D命令的执行结果存储在标量变量$date中。

标量 VI

- 上例中使用了一些有特殊意义的字符，下面列出这些字符的含义：

\n	换行
\r	回车
\t	制表符
\a	蜂鸣声
\b	退格符
\L \E	将\L和\E之间的字符转换成小写
\l	将其后的字符转换成小写
\U \E	将\U和\E之间的字符转换成大写
\u	将其后的字符转换成大写
\cC	插入控制字符C
\x##	十六进制数##
\0ooo	八进制数ooo
\\	反斜杠
\	按原样输出下一个字符，例如：\$输出$

标量 VII

- 简单变量是标量，是Perl处理的最简单的数据类型。标量可以是数字 (如2,3或2.5e6)，也可以是字符串。
- 另外在Perl语言里，我们常会看到my这样的变量定义，如：

```
1 my $a = "local var";
```

表示$a是一个程序块的局部变量。

列表数组 I

- 列表数组常量是位于括号内用逗号分开的一系列值，这些值可以是标量常量或是表达式。
- 列表数组变量名以@号开头。
- 列表数组的元素可以包含“..”运算符，这个运算符以 1 为增量创建一个从左边标量值开始到右边标量值结束的数值列表。如果右边的标量小于左边的标量则产生空表。如果前后两个值的差值不是整数，则表尾的元素是不超出范围的最后一个值。
- 列表数组运算符
  - 赋值：=

列表数组 II

```
1  @fred=(1,2,3);
2  @barney=@fred;
3  @huh=1;
4  @fred=("one","two");
5  @barney=(4,5,@fred,6,7);
6  @barney=(8,@barney);
7  @barney=(@barney,"last");
8  ($a,$b,$c)=(1,2,3);
9  ($a,$b)=($b,$a);
10 ($d,@fred)=($a,$b,$c);
11 ($e,@fred)=@fred;
12 #after that, @fred=($c), $e=$b
13 #Note: (@fred,$e)=@fred; make $e undef
```

- 如果赋值号两边的表中元素数目不等，则等号右边任何多出来的值都被截去；等号左边多出来的变量都被赋为undef

列表数组 III

- 如果把列表数组变量赋给标量，则标量变量的值就是列表数组的长度
- 赋值表达式的值为列表数组变量得到的值：

```
1 @fred=($barney=(2,3,4));
2 @fred=@barney=(2,3,4);
```

- 上述两式结果相同。
- 列表数组元素的下标都是从 0 开始，增量为 1
- 下标运算符为[ ]。

```
1 ($fred[0],$fred[1])=($fred[1],$fred[0]);
```

- 片段（slice）：
- 对同一个列表数组的一部分元素的访问可以使用 slice 表达式：

列表数组 IV

```
1  @fred[0,1]=@fred[1,0];      # exchange
2  @fred[0,1,2]=@fred[1,1,1];
3  # assigned the 1st value to the
4  # first 3 elements
5  @who=("fred","barney","betty")[1,2];
6  #equal @who=("barney","betty");
7  @fred=(7,8,9);
8  @barney=(2,1,0);
9  @backfred=@fred[@barney];#(9,8,7)
10 @fred=(1,2,3);
11 $fred[3]="hi";
12 $fred[6]="ho";#(1,2,3,"hi",undef,undef,"ho")
```

- 可以使用$#fred来得到列表数组@fred的末尾元素的索引值；并通过对他赋值来改变列表数组@fred的长度。

列表数组 V

- push()和pop()运算符：

```
1 push(@myarr, $newvalue);
2 $oldvalue=pop(@myarr);
```

- shift()和unshift()运算符
- 类似与push()和pop()，只不过他们是对列表数组左边的元素操作。
- reverse()运算符：返回列表数组元素反序后的结果，但是不改变参数。
- sort()运算符：将所有参数都当成是 ASCII 字符串，按升序方式排序，返回排序后的结果但是不改变原列表。
- chop()运算符：可以删去列表数组中每个元素的最后一个字符。

关联数组 I

- 关联数组也是由一系列标量数据组成的集合，它与列表数组的区别是它的索引值不再是非负的整数而是任意的标量，这些表示索引的标量称为关键字（key）。
- 实际上关联数组是数据结构里的散列表。
- 关联数组的元素没有特定的顺序。
- 关联数组常量：
  - 关联数组常量由含有偶数个元素的列表数组表示：

```
1 %fred=("aaa","bbb","234.5",456.7)
```

  将生成一个含有两个“键 —值对”的关联数组。
  - 在展开表中的“键 —值对”的顺序可以是任意的，其在内存中的具体顺序是由Perl建立的，用于提高访问单个元素的效率。
- 关联数组变量
  - 关联数组变量名以%号开头。

关联数组 II

- 在创建和访问关联数组时只需使用对数组元素的引用即可：
- 关联数组%arr的每个元素都可被$arr{$key}引用

- 关联数组操作符
  - keys()操作符：keys(%arr)将生成由关联数组%arr中的所有关键字组成的列表数组。其中的圆括号是可选的
  - 例：

```
1 foreach $key (keys %fred) {
2   print "at $key we get $fred{$key}\n";
3 }
```

  - 在标量环境中，keys()返回关联数组中键 —值对的个数。
  - values()操作符：values(%arr)返回由%arr中的值构成的列表数组。圆括号是可选的。
  - each()操作符：each(%arr)返回%arr中的一个键 —值对列表，对同一关联数组再次使用此操作符将返回下一个键 —值对，当处理到数组的最后一个元素之后，将返回一个空表。

关联数组 III

- 例：

```
1 while(($first,$last)=each(%arr)){
2   print "The last name of $first is $last\n";
3 }
```

- 给整个数组赋新值将使each()操作符重置到数组的起始位置。
- delete()操作符：其操作数是关联数组的引用。用于删除指定的键—值对。

内部变量 I

- 常用内部变量

$_	缺省的输入和模式搜索空间
$n	标记寄存器，用于存储由前面的标记正则表达式匹配的内容。只读
$&	最近一个成功匹配的字符串。只读
$`	最近一个成功匹配的字符串之前的字符串。只读
$'	最近一个成功匹配的字符串之后的字符串。只读
$.	当前记录序号。当文件句柄被显式关闭时此参数将被重置为 0
$/	输入记录分隔符。缺省为NewLine
$|	如果将其值置为非 0，则每当你向当前的输出通道写或打印时就将强制 flush 一下。缺省为 0
$,	输出字段分隔符。缺省为空

内部变量 II

$\	输出记录分隔符。缺省为空
$"	类似于$,，不过应用于数组元素的输出。缺省为空格
$#	缺省的数字输出格式。初始值为%.ng，其中n为系统中float.h文件中宏DBL_DIG的值
$;	模拟多维数组的下标分隔符。缺省为\034
$%	当前输出通道里的当前页号
$=	当前输出通道中每页能包含的行数，缺省为 60
$-	当前输出页中剩余的行数
$~	当前输出报告的格式名称
$^	当前页眉的格式名称
$:	当前断字符。缺省为"\n-"
$^L	输出换页符。缺省为\f
$$	正在运行当前Perl脚本的 pid
$<	上述进程的真实 uid
$>	上述进程的有效 uid

内部变量 III

$(	上述进程的真实 gid
$)	上述进程的有效 gid
$0	当前Perl脚本文件名
$[	列表数组的第一个元素的下标，或者是字符串的第一个字符的下标，缺省为 0
$^O	当前 OS 名
$^T	当前Perl脚本开始运行的时刻（epoch 之后的秒数）
$^X	类似于 C 中的argv[0]
$ARGV	当从<>中读取数据时，表示当前数据文件名
@ARGV	参数列表
@_	函数的实参列表。局部变量
%ENV	环境变量数组
%SIG	信号数组，key表示信号类型，value表示要进行的处理

内部变量 IV

- 例:

```
1 $_ = 'abcdefghi';
2 /def/;
3 print "$`:$&:$'";#prints abc:def:ghi
```

- 例:

```
1 sub handler {
2 # the 1st argument is signal name
3 my($sig) = @_;
4   print "Caught a SIG$sig--shutting down\n";
5   close(LOG);
6   exit(0);
7 }
8 $SIG{'INT'}  = \&handler;
```

内部变量 V

```
 9 $SIG{'QUIT'} = \&handler;
10 ...
11 $SIG{'INT'} = 'DEFAULT';
12 # restore default action
13 $SIG{'QUIT'} = 'IGNORE';
14 # ignore SIGQUIT
```

流程控制 I

- 语句块：位于一对花括号之间的语句序列。

```
1 {
2     statement;
3     statement;
4     ......
5     statement;
6 }
```

- 控制表达式expression是作为字符串计算的，如果是空串或只包括单个字符 “0”，则表达式为假，否则为真。

流程控制 II

- 例：

```
1 0         # false
2 1-1       # false
3 1         # true
4 ""        # false
5 "1"       # true
6 "00"      # true
7 "0.000"   # true
8 undef     # false
```

流程控制 III

- if语句:

```
1  if(expression 1) {
2      statements block 1
3  }
4  elsif(expression 2) {
5      statements block 2
6  }
7  ...
8  else{
9      statements block n
10 }
```

流程控制 IV

- unless语句:

```
1 unless=if not
```

- while语句:

```
1 while(expressions) {
2     statements block
3 }
```

流程控制 V

- until语句:

```
1 until(expressions) {
2     statement block
3 }
```

- for语句:

```
1 for(expr1; expr2; expr3) {
2     statement block
3 }
```

流程控制 VI

- foreach语句:

```
1 foreach $var (arr_expr) {
2     statement block
3 }
```

- arr_expr可以是任意表达式。
- 如果arr_expr是单个数组变量的引用，那么在循环体中对$var的修改将直接作用于相应的数组元素。即这时是**byref**引用。

流程控制 VII

- 例：

```
1 @a=(3,5,7,9)
2 foreach $one (@a) {
3     $one *= 3;
4 }
5 # now @a is (9,15,21,27)
```

流程控制 VIII

- last语句：终止最近的封闭循环块
- 例：

```
1 while(expr) {
2     statement block
3     last;
4 }
5 # last to here
```

流程控制 IX

- next语句：跳过最近的封闭循环块中的剩余部分，进行下一次循环
- 例：

```
1 while(expr) {
2     statement block 1
3     next;
4     statement block 2
5     # next to here
6 }
```

流程控制 X

- redo语句：跳到当前循环块的最开始位置
- 例：

```
1 while(expr) {
2     # redo to here
3     statement block
4     redo;
5 }
```

- label语句：
  - 使用label给语句块起一个名字，用于last和next语句
  - 使用标号只能跳出，不能跳入

流程控制 XI

- 表达式简写方式 1(倒置)

```
1 exp2 if exp1;
2 exp2 unless exp1;
3 exp2 while exp1;
4 exp2 until exp1;
```

- 先对exp1求值，并据此判断是否执行exp2

流程控制 XII

- 表达式简写方式 2(&&,||,?)

```
1 exp1 && exp2;     # if(exp1) {exp2;}
2 exp1 || exp2;     # unless(exp1) {exp2;}
3 exp1 ?  exp2 : exp3;
4 # if(exp1) {exp2;} else {exp3;}
```

基本 I/O

- STDIN：使用<STDIN>操作符表示从标准输入读取数据，直到遇到$/为止。
- <>：和<STDIN>类似，但是可以从Perl的命令行参数 —文件名指定的输入文件中读取记录。
- STDOUT
  - print()操作符，返回成功 (1) 或失败 (0)
  - printf(fmt_str, value_list)，类似于 C 和awk中的格式。

文件操作 I

- 文件句柄：已经见过的有STDIN, STDOUT, STDERR
- 文件句柄的打开和关闭

```
1 open(FILEHANDLE,"[>[>]]file_name");
```

- 表示以读 [覆盖写 [追加写]] 方式打开文件。

```
1 close(FILEHANDLE);
```

- 关闭文件句柄。
- die()操作符在可选的圆括号里带有一列表，以标准错误的方式输出该列表，然后以非零的 UNIX 退出状态结束该Perl进程。
- 如果die()的列表的最后没有一个\n，则退出时的信息中自动带有Perl程序的名字及行数。

文件操作 II

- 例：

```
1 open(IN,$a)||die "Cannot open $a for reading.";
2 open(OUT,">$b")||die "Cannot create $b.";
3 while(<IN>){
4     print OUT $_;
5 }
6 close(OUT);
7 close(IN);
```

文件操作 III

- 文件测试：

-r	Readable for euid
-w	Writable for euid
-x	Executable for euid
-o	Owner by euid
-R	Readable for ruid
-W	Writable for ruid
-X	Executable for ruid
-O	Owner by ruid
-e	Exist
-z	Exist and size=zero
-s	Exist and size!=zero
-f	Pure file
-d	Directory

文件操作 IV

-l	Link file
-S	Socket
-p	Named pipe
-b	Block device file
-c	Character device file
-uN	uid is N
-gN	gid is N
-k	Has sticky bit
-t	isatty() is true
-T	Text file
-B	Binary file
-M	Up to now in days from LMT, precision in second
-A	from LAT
-C	from LCT

文件操作 V

- stat()和lstat()操作符
- stat()的操作数是一个文件句柄，返回值是有 13 个元素的数组：

```
($dev, $ino, $mode, $nlink, $uid, $gid,
    $rdev, $size, $atime, $mtime, $ctime,
    $blksize, $blocks) = stat();
$dev      device
$ino      inode
$mode     access mode
$nlink    number of hard links
$uid      user id of owner
$gid      group id of owner
$rdev     device type (if inode device)
$size     total size, in bytes
$atime    last access time
```

文件操作 VI

```
13 $mtime    last modification time
14 $ctime    last change time
15 $blksize block size for file system I/O
16 $blocks   number of blocks allocated
```

- 例:

```
1 [Apple]$ ./stat.pl
2   0       dev 773
3   1       ino 95268
4   2      mode 33277
5   3     nlink 1
6   4       uid 500
7   5       gid 500
8   6      rdev 0
9   7      size 14350
```

文件操作 VII

```
10    8     atime 1038893994
11    9     mtime 1038893994
12   10     ctime 1038893994
13   11   blksize 4096
14   12    blocks 32
15 [Apple]$ ls -l code
16 -rwxrwxr-x  1 student  student  14350 Dec.  3 13:39 cod
17 [Apple]$ more stat.pl
18 #!/usr/bin/perl
19 @arr1=("dev", "ino", "mode", "nlink", "uid",\
20    "gid", "rdev", "size", "atime", "mtime",\
21    "ctime", "blksize", "blocks");
22 @arr=stat("/home/student/Examples/Perl/code");
23 for($i = 0; $i < 13; $i = $i + 1) {
24   printf("%3d %8s %s\n",$i,$arr1[$i],$arr[$i]);
25 }
```

文件操作 VIII

- 对于一个符号链接文件来说，stat()返回的是符号链接所指之处的信息，如果你需要关于该符号链接本身的信息，可以使用lstat()
- 这两个操作符默认的操作数是$_。
- 例：

```
1  [Apple]$ ln -s code lcode
2  [Apple]$ ls -l *code
3  -rwxrwxr-x 1 hop hop 350 Dec 3 13:39 code
4  lrwxrwxrwx 1 hop hop   4 Dec 3 13:45 lcode -> code
5  [Apple]$ ./stat.pl
6    0       dev 773
7    1       ino 95268
8    2      mode 33277
9    3     nlink 1
10   4       uid 500
11   5       gid 500
```

文件操作 IX

```
12    6      rdev 0
13    7      size 14350
14    8     atime 1038893994
15    9     mtime 1038893994
16   10     ctime 1038894348
17   11   blksize 4096
18   12    blocks 32
19 [Apple]$ ./lstat.pl
20    0       dev 773
21    1       ino 95269
22    2      mode 41471
23    3     nlink 1
24    4       uid 500
25    5       gid 500
26    6      rdev 0
27    7      size 4
```

文件操作 X

```
28    8     atime 1038894488
29    9     mtime 1038894357
30   10     ctime 1038894357
31   11   blksize 4096
32   12    blocks 0
```

- 每次进行文件测试操作时，Perl都将向系统申请一个该文件的stat缓冲区。在指定的$_文件句柄上进行文件操作就可以让Perl直接使用前一次文件测试的缓冲区。
- 例：

```
1 if(-r $filevar && -w) {
2     print "$filevar is both readable and writable.\n";
3 }
```

格式 I

- Perl提供了简单的报告书写模板的概念，叫做**格式**（**format**）。Format 定义了常量部分 (每列的开头、标签、相应的正文或其它) 以及变量部分 (报告中的数据)。
- 使用格式需要做三件事：
  - 定义格式
  - 提取数据，将其打印到格式的变量部分
  - 申请格式
- 定义格式。格式定义可以出现在程序中的任何位置：

```
1 format fmt_name =
2 FORMLIST
3 .
```

格式 II

- 第一行包括保留字format，以及该格式的名称（缺省为STDOUT）和一个等号。
- 下面是“模板”本身，可以有任意行文本。每一行文本都应该是下述三种之一：
  - 第一列以#开头的注释
  - 格式字符串给出每个输出行的格式
  - 用于匹配格式字符串的参数列表
- 格式字符串中的普通文本原样输出。
- 以@开头的是字段格式，在字段格式中的字符<,>,|分别表示左、右和中央对齐。字符的个数表明了相应参数的输出值的宽度，如果值的宽度过大，将被截断，过小则填充空格。

格式 III

- 例：

```
1  #!/usr/bin/perl
2  # a report on the /etc/passwd file
3  format PWDLIST_TOP =
4                          Passwd File                          Page @>>
5                                                                  $%
6  No.  Name          Login        Office       Uid   Gid Home
7  ------------------------------------------------------------------
8  .
9  format PWDLIST =
10 @<<< @<<<<<<<<<<< @<<<<<<<<<<< @<<<<<<<<< @>>>> @>>>> @<<<<<<<<<<<<<<<
11 $.,  $name,       $login,      $office,    $uid, $gid, $home
12 .
13 open(PWDLIST, ">passwd_report") || die "Can't create output file.";
14 open(PWDFILE, "/etc/passwd") || die "Can't open data file.";
15 # Change the page height to 10 lines.
16 $old = select(PWDLIST);
17 $= = 18;
18 select($old);
19
20 while(<PWDFILE>) {
21         chop;
22         ($login, $pwd, $uid, $gid, $name, $home, $office)=split(/:/);
23         write PWDLIST;
24 }
25 close(PWDFILE);
26 close(PWDLIST);
```

格式 IV

- 输出文件为：

```
                          Passwd File                          Page  1
No.  Describe     Login_Name    Shell         Uid    Gid Home_Dir
-----------------------------------------------------------------------
1    root,,,      root          /bin/bash       0      0 /root
2    bin          bin                           1      1 /bin
3    daemon       daemon                        2      2 /sbin
4    adm          adm                           3      4 /var/adm
5    lp           lp                            4      7 /var/spool/lpd
...... ......
15   FTP User     ftp                          14     50 /home/ftp
^L                        Passwd File                          Page   5
No.  Describe     Login_Name    Shell         Uid    Gid Home_Dir
-----------------------------------------------------------------------
29   radvd user   radvd         /bin/false     75     75 /
30   PostgreSQL S postgres      /bin/bash      26     26 /var/lib/pgsql
31   Apache       apache        /bin/false     48     48 /var/www
32                squid         /dev/null      23     23 /var/spool/squid
33                pcap          /sbin/nolo     77     77 /var/arpwatch
34   UNIX Student student       /bin/bash     500    500 /home/student
35   Hop Lee      hop           /bin/bash     501    501 /home/hop
```

- @######.##表示数值的格式：六位整数部分，两位小数部分。
- 独占一行的@*表示不截断的多行内容。

格式 V

- 另外一种字段格式符是^，用于表示填充字段。用于在多行中输出过长的内容，并且可以通过~和~~来控制是否自动调整空行。
- 例：

```
format PEOPLE =
Name : @<<<<<<<< Comment : ^<<<<<<<<<<<<<<<<<<<<
       $name,              $comment
                           ^<<<<<<<<<<<<<<<<<<<<
                           $comment
                           ^<<<<<<<<<<<<<<<<<<<<
                           $comment
```

- 如果文本少于三行，将出现空行，可以在前面添加~符号自动去除空行。
- 如果文本多于三行，将被截断，可以在前面添加~~符号自动增长并自动去除空行。

格式 VI

- select()为print()操作指定输出文件句柄，当Perl开始运行时是STDOUT。
- select()一旦选定了新的句柄，它就始终起作用直到下一个select()调用。

示例 I

1. 读入一个字符串和一个数字n，把该字符串重复串接n次后输出；

```
1 print "String: ";
2 $a = <STDIN>;
3 print "Number of times: ";
4 chomp($b=<STDIN>);
5 $c=$a x $b;
6 print "The result is: \n$c";
```

示例 II

2. 读入一连串的字符串，然后按相反的顺序打印出来；

```
1 print "Enter the list of strings:\n";
2 @list = <STDIN>;
3 @reverselist = reverse @list;
4 print @reverselist;
5 #
6 # Or
7 print "Enter the list of strings:\n";
8 print reverse <STDIN>;
```

示例 III

3. 读入一连串的字符串，然后按相反的顺序打印出来，不允许使用reverse函数；

```
1 print "Enter the list of strings:\n";
2 @strings = <STDIN>;
3 while (@strings) {
4   print pop @strings;
5 }
```

示例 IV

4. 读入一连串的字符串，用随机数选取其中某一行打印出来。

```
1 srand;
2 print "List of strings: ";
3 @b = <STDIN>;
4 print "Answer: $b[rand(@b)]";
```

示例 V

5. 打印出 0 到 32 的数值及其平方的数值；

```
1  for($n = 0; $n <= 32; $n++) {
2     $sqr = $n * $n;
3     printf "%5g %8g\n", $n, $sqr;
4  }
5  #
6  # Or
7  foreach $n (0..32) {
8     $sqr = $n * $n;
9     printf "%5g %8g\n", $n, $sqr;
10 }
```

示例 VI

6. 读入英文单词（一行一个），统计每个单词出现的频率，把结果按照单词的字典顺序打印出来；

```
1 chomp(@words=<STDIN>);
2 foreach $word (@words) {
3   $count{$word}++;
4 }
5 foreach $word (sort key %count) {
6   print "$word was seen $count{$word} times\n";
7 }
```

示例 VII

7. 读入一连串单词，寻找同时出现 5 个元音字母的行输出；

```
1 while(<STDIN>) {
2   if(/a/i && /e/i && /i/i && /o/i && /u/i) {
3     print;
4   }
5 }
```

示例 VIII

8. 更改上述程序，要求 5 个元音字母按顺序出现；

```
1 while(<STDIN>) {
2   if(/a.*e.*i.*o.*u/i) {
3     print;
4   }
5 }
```

示例 IX

9. 更改上述程序，要求不允许有任何反序存在；

```
1 while(<STDIN>) {
2   if(/^[^eiou]*a[^iou]*e[^aou]*i[^aeu]*o[^aei]*u[^aeio]*/i) {
3     print;
4   }
5 }
```

示例 X

⑩ 写一个函数，以 1 至 9 的数字为参数，并传回其英文（one, two...）。若传入的数字超过范围，则传回原数字；

```
1  sub card {
2    my %card_map;
3    @card_map{1..9}=qw{
4      one two three four five six seven eight nine
5    };
6    my ($num) = @_;
7    $card_map{$num} || $num;
8  }
9  # Driver routine:
10 while (<>) {
11   chomp;
12   print "card of $_ is ", &card($_), "\n";
13 }
```

示例 XI

11. 利用上例的函数，另外编写一个函数执行两数相加，并打印 “Two plus two equals four.” 这样的结果；

```
1 print "Enter first number: ";
2 chomp ($first = <STDIN>);
3 print "Enter second number: ";
4 chomp ($second = <STDIN>);
5 $msg = card($first) . "plus " . card($second) .
6    "equals " . card($first + $second) . ".\n";
7 print "\u$msg";
```

示例 XII

⑫ 改写上面两个程序，使其接受的参数范围为-9～+9；

```
1  sub card {
2    my %card_map;
3    @card_map{0..9}=qw{
4      zero one two three four five six\
5      seven eight nine
6    };
7    my ($num) = @_;
8    my ($negative);
9    if($num < 0) {
10     $negative = "negative";
11     $num = -$num;
12   }
13   $negative . ($card_map{$num} || $num);
14 }
```

示例 XIII

13. 读入文件名，在每行之前加上文件名后一次输出每行的内容；

```
1 print "What file? ";
2 chomp($filename=<STDIN>);
3 open(THEFILE, "$filename") ||
4   die "Cannot open $filename: $!";
5 while(<THEFILE>) {
6   print "$filename: $_";
7 }
8 close(THEFILE);
```

示例 XIV

14. 读入一串文件名，测试每个文件是否可读、可写、可执行以及是否存在；

```
1 while(<>) {
2   chomp;
3   print "$_ is readable\n" if -r;
4   print "$_ is writable\n" if -w;
5   print "$_ is executable\n" if -x;
6   print "$_ does not exist\n" unless -e;
7 }
```

The End

The End of Chapter VII.