Perl 5 教程

by flamephoenix

第一部分 Perl 语言

第一章 概述

一、Perl 是什么?
二、Perl 在哪里?
三、运行
四、注释

第二章 简单变量

一、整型
二、浮点数
三、字符串

第三章 操作符

一、算术操作符
二、整数比较操作符
三、字符串比较操作符
四、逻辑操作符
五、位操作符
六、赋值操作符
七、自增自减操作符
八、字符串联结和重复操作符
九、逗号操作符
十、条件操作符
十一、操作符的次序

第四章 列表和数组变量

第一章 概述

一、Perl 是什么?

Perl 是 Practical Extraction and Report Language 的缩写，它是由 Larry Wall 设计的，并由他不断更新和维护，用于在 UNIX 环境下编程。

.Perl 具有高级语言（如 C）的强大能力和灵活性。事实上，你将看到，它的许多特性是从 C 语言中借用来的。

.与脚本语言一样，Perl 不需要编译器和链接器来运行代码，你要做的只是写出程序并告诉 Perl 来运行而已。这意味着 Perl 对于小的编程问题的快速解决方案和为大型事件创建原型来测试潜在的解决方案是十分理想的。

.Perl 提供脚本语言（如 sed 和 awk）的所有功能，还具有它们所不具备的很多功能。Perl 还支持 sed 到 Perl 及 awd 到 Perl 的翻译器。

简而言之，Perl 象 C 一样强大，象 awk、sed 等脚本描述语言一样方便。

二、Perl 在哪里?

Perl 通常位于/usr/local/bin/perl 或/usr/bin/perl 中。你可以在 Internet 用匿名 FTP 免费得到它，如 ftp://prep.ai.mit.edu/pub/gnu/perl-5.004.tar.gz。

安装过程为：

(1)解压：

```
$gunzip perl-5.004.tar.gz
$tar xvf - <perl-5.004.tar.gz
```

(2)编译：

```
$make makefile
```

(3)放置：

将编译生成的可执行文件拷贝到可执行文件通常所在目录，如：

$copy <compiled excutable file> /usr/local/bin/perl

注：这需要系统管理员权限。

三、运行

用文本编辑器编辑好你的 Perl 程序，加上可执行属性：$chmod +x <program>就可以执行了：$./<program>。如果系统提示："/usr/local/bin/perl not found"，则说明你没有安装成功，请重新安装。

注：你的程序的第一行必须为#!/usr/local/bin/perl（perl 所在位置）。

四、注释：

注释的方法为在语句的开头用字符#，如：

this line is a comment

注：建议经常使用注释使你的程序易读，这是好的编程习惯。

第二章 简单变量

基本上，简单变量就是一个数据单元，这个单元可以是数字或字符串。

一、整型

1、整型

PERL 最常用的简单变量，由于其与其它语言基本相同，不再赘述。

例：

$x = 12345;

```
if (1217 + 116 == 1333) {
# statement block goes here
}
```

整型的限制：

PERL 实际上把整数存在你的计算机中的浮点寄存器中，所以实际上被当作浮点数看待。在多数计算机中，浮点寄存器可以存贮约 16 位数字，长于此的被丢弃。整数实为浮点数的特例。

2、8 进制和 16 进制数

8 进制以 0 打头，16 进制以 0x 打头。

例：$var1 = 047; (等于十进制的 39)

$var2 = 0x1f; (等于十进制的 31)

二、浮点数

如 11.4 、 -0.3 、.3 、 3. 、 54.1e+02 、 5.41e03

浮点寄存器通常不能精确地存贮浮点数，从而产生误差，在运算和比较中要特别注意。指数的范围通常为-309 到+308。

例：

```
#!/usr/local/bin/perl
$value = 9.01e+21 + 0.01 - 9.01e+21;
print ("first value is ", $value, "\n");
$value = 9.01e+21 - 9.01e+21 + 0.01;
print ("second value is ", $value, "\n");

-------------------------------------------------------

$ program3_3
first value is 0
second value is 0.01
```

三、字符串

惯用 C 的程序员要注意，在 PERL 中，字符串的末尾并不含有隐含的 NULL 字符，NULL 字符可以出现在串的任何位置。

. 双引号内的字符串中支持简单变量替换，例如：

```
$number = 11;
$text = "This text contains the number $number.";
```

则$text 的内容为："This text contains the number 11."

.双引号内的字符串中支持转义字符

Table 3.1. Escape sequences in strings.

Escape Sequence	Description
\a	Bell (beep)
\b	Backspace
\cn	The Ctrl+n character
\e	Escape
\E	Ends the effect of \L, \U or \Q
\f	Form feed
\l	Forces the next letter into lowercase
\L	All following letters are lowercase
\n	Newline
\r	Carriage return
\Q	Do not look for special pattern characters
\t	Tab
\u	Force next letter into uppercase
\U	All following letters are uppercase
\v	Vertical tab

\L、\U、\Q 功能可以由\E 关闭掉，如：

```
$a = "T\LHIS IS A \ESTRING"; # same as "This is a STRING"
```

.要在字符串中包含双引号或反斜线，则在其前加一个反斜线，反斜线还可以取消变量替换，如：

```
$res = "A quote \" and A backslash \\";
$result = 14;
print ("The value of \$result is $result.\n")
```

的结果为：

```
The value of $result is 14.
```

.可用\nnn(8 进制)或\xnn(16 进制)来表示 ASCII 字符，如：

```
$result = "\377"; # this is the character 255,or EOF
$result = "\xff"; # this is also 255
```

.单引号字符串

单引号字符串与双引号字符串有两个区别，一是没有变量替换功能，二是反斜线不支持转义字符，而只在包含单引号和反斜线时起作用。单引号另一个特性是可以跨多行，如：

```
$text = 'This is two
lines of text
';
```

与下句等效：

```
$text = "This is two\nlines of text\n";
```

.字符串和数值的互相转换

例 1：

```
$string = "43";
$number = 28;
$result = $string + $number; # $result = 71
```

若字符串中含有非数字的字符，则从左起至第一个非数字的字符，如：

```
$result = "hello" * 5; # $result = 0
$result = "12a34" +1; # $result = 13
```

.变量初始值

在 PERL 中，所有的简单变量都有缺省初始值："",即空字符。

但是建议给所有变量赋初值，否则当程序变得大而复杂后，很容易出现不可预料且很难调试的错误。

第三章 操作符

一、算术操作符 ：+(加)、-(减)、*(乘)、/(除)、**(乘幂)、%(取余)、-(单目负)

(1)乘幂的基数不能为负，如 (-5) ** 2.5 # error;

(2)乘幂结果不能超出计算机表示的限制，如 10 ** 999999 # error

(3)取余的操作数如不是整数，四舍五入成整数后运算；运算符右侧不能为零

(4)单目负可用于变量： - $y ; # 等效于 $y * -1

二、整数比较操作符

Table 3.1. 整数比较操作符

操作符	描述
<	小于
>	大于
==	等于

<=	小于等于
>=	大于等于
!=	不等于
<=>	比较，返回 1, 0, or -1

操作符<=>结果为：

0 - 两个值相等

1 - 第一个值大

1 - 第二个值大

三、字符串比较操作符

Table 3.2. 字符串比较操作符

操作符	**描述**	
lt	小于	
gt	大于	
eq	等于	
le	小于等于	
ge	大于等于	
ne	不等于	
cmp	比较，返回 1, 0, or -1	

四、逻辑操作符

逻辑或：$a || $b 或 $a or $b

逻辑与：$a && $b 或 $a and $b

逻辑非：! $a 或 not $a

逻辑异或：$a xor $b

五、位操作符

位与：&

位或：|

位非：~

位异或：^

左移：$x << 1

右移：$x >> 2

注：不要将&用于负整数，因为 PERL 将会把它们转化为无符号数。

六、赋值操作符

Table 3.3. 赋值操作符

操作符	描述
=	Assignment only
+=	Addition and assignment
-=	Subtraction and assignment
*=	Multiplication and assignment
/=	Division and assignment
%=	Remainder and assignment
**=	Exponentiation and assignment
&=	Bitwise AND and assignment
|=	Bitwise OR and assignment
^=	Bitwise XOR and assignment

Table 3.4. 赋值操作符例子

表达式	等效表达式
$a = 1;	none (basic assignment)
$a -= 1;	$a = $a - 1;
$a *= 2;	$a = $a * 2;
$a /= 2;	$a = $a / 2;
$a %= 2;	$a = $a % 2;
$a **= 2;	$a = $a ** 2;
$a &= 2;	$a = $a & 2;
$a |= 2;	$a = $a | 2;
$a ^= 2;	$a = $a ^ 2;

.=可在一个赋值语句中出现多次，如：

```
$value1 = $value2 = "a string";
```

.=作为子表达式

```
($a = $b) += 3;
```

等价于

```
$a = $b;
$a += 3;
```

但建议不要使用这种方式。

七、自增自减操作符 ：++、--(与 C++中的用法相同)

.不要在变量两边都使用此种操作符：++$var-- # error

.不要在变量自增/减后在同一表达式中再次使用：$var2 = $var1 + ++$var1; # error

.在 PERL 中++可用于字符串，但当结尾字符为'z'、'Z'、'9'时进位，如：

```
$stringvar = "abc";
$stringvar++; # $stringvar contains "abd" now

$stringvar = "aBC";
```

```
$stringvar++; # $stringvar contains "aBD" now

$stringvar = "abz";
$stringvar++; # $stringvar now contains "aca"

$stringvar = "AGZZZ";
$stringvar++; # $stringvar now contains "AHAAA"

$stringvar = "ab4";
$stringvar++; # $stringvar now contains "ab5"

$stringvar = "bc999";
$stringvar++; # $stringvar now contains "bd000"
```

.不要使用--，PERL将先将字符串转换为数字再进行自减

```
$stringvar = "abc";
$stringvar--; # $stringvar = -1 now
```

.如果字符串中含有非字母且非数字的字符，或数字位于字母中，则经过++运算前值转换为数字零，因此结果为1，如：

```
$stringvar = "ab*c";
$stringvar++;
$stringvar = "ab5c";
$stringvar++;
```

八、字符串联结和重复操作符

联接：.

重复：x

联接且赋值(类似+=)：　.=

例：

```
$newstring = "potato" . "head";
$newstring = "t" x 5;
$a = "be";
$a .= "witched"; # $a is now "bewitched"
```

九、逗号操作符

其前面的表达式先进行运算，如：

```
$var1 += 1, $var2 = $var1;
```

等价于

```
$var1 += 1;
$var2 = $var1;
```

使用此操作符的唯一理由是提高程序的可读性，将关系密切的两个表达式结合在一起，如：

```
$val = 26;
$result = (++$val, $val + 5); # $result = 32
```

注意如果此处没有括号则意义不同：

```
$val = 26;
$result = ++$val, $val + 5; # $result = 27
```

十、条件操作符

与 C 中类似，条件?值 1:值 2，当条件为真时取值 1，为假时取值 2，如：

```
$result = $var == 0 ? 14 : 7;
$result = 43 + ($divisor == 0 ? 0 : $dividend / $divisor);
```

PERL 5 中，还可以在赋值式左边使用条件操作符来选择被赋值的变量，如：

```
$condvar == 43 ? $var1 : $var2 = 14;
$condvar == 43 ? $var1 = 14 : $var2 = 14;
```

十一、操作符的次序

Table 3.6. 操作符次序

操作符	描述
++, --	自增，自减
-, ~, !	单目
**	乘方
=~, !~	模式匹配
*, /, %, x	乘，除，取余，重复
+, -, .	加，减，联接
<<, >>	移位
-e, -r, etc.	文件状态

<, <=, >, >=, lt, le, gt, ge	不等比较
==, !=, <=>, eq, ne, cmp	相等比较
&	位与
|, ^	位或，位异或
&&	逻辑与
||	逻辑或
..	列表范围
? and :	条件操作符
=, +=, -=, *=,	赋值
and so on	
,	逗号操作符
not	Low-precedence logical NOT
and	Low-precedence logical AND
or, xor	Low-precedence logical OR and XOR

.操作符结合性(associativity):

Table 3.7. 操作符结合性

操作符	结合性
++, --	无
-, ~, !	Right-to-left
**	Right-to-left
=~, !~	Left-to-right
*, /, %, x	Left-to-right

+, -, .	Left-to-right
<<, >>	Left-to-right
-e, -r,	无
<, <=, >, >=, lt, le, gt, ge	Left-to-right
==, !=, <=>, eq, ne, cmp	Left-to-right
&	Left-to-right
|, ^	Left-to-right
&&	Left-to-right
||	Left-to-right
..	Left-to-right
? and :	Right-to-left
=, +=, -=, *=,	Right-to-left
and so on	
,	Left-to-right
not	Left-to-right
and	Left-to-right
or, xor	Left-to-right

建议：

1、当你不确定某操作符是否先执行时，一定要用括号明确之。

2、用多行、空格等方式提高程序的可读性。

第四章 列表和数组变量

一、列表

二、数组--列表的存贮

一、列表

列表是包含在括号里的一序列的值，可以为任何数值，也可为空，如：(1, 5.3 , "hello" , 2)，空列表：()。

注：只含有一个数值的列表(如：(43.2))与该数值本身(即：43.2)是不同的，但它们可以互相转化或赋值。

列表例：

(17, $var, "a string")

(17, 26 << 2)

(17, $var1 + $var2)

($value, "The answer is $value")

二、数组--列表的存贮

列表存贮于数组变量中，与简单变量不同，数组变量以字符"@"打头，如：

@array = (1, 2, 3);

注：

(1)数组变量创建时初始值为空列表：()。

(2)因为 PERL 用@和$来区分数组变量和简单变量，所以同一个名字可以同时用于数组变量和简单变量，如：

$var = 1;

@var = (11, 27.1 , "a string");

但这样很容易混淆，故不推荐。

1、数组的存取

.对数组中的值通过下标存取，第一个元素下标为 0。试图访问不存在的数组元素，则结果为 NULL，但如果给超出数组大小的元素赋值，则数组自动增长，原来没有的元素值为 NULL。如：

@array = (1, 2, 3, 4);

$scalar = $array[0];

$array[3] = 5; # now @array is (1,2,3,5)

$scalar = $array[4]; # now $scalar = null;

$array[6] = 17; # now @array is (1,2,3,5,"","",17)

.数组间拷贝

@result = @original;

.用数组给列表赋值

@list1 = (2, 3, 4);

@list2 = (1, @list1, 5); # @list2 = (1, 2, 3, 4, 5)

.数组对简单变量的赋值

(1) @array = (5, 7, 11);

($var1, $var2) = @array; # $var1 = 5, $var2 = 7, 11 被忽略

(2) @array = (5, 7);

($var1, $var2, $var3) = @array; # $var1 = 5, $var2 = 7, $var3 ="" (null)

.从标准输入(STDIN)给变量赋值

$var = <STDIN>;

@array = <STDIN>; # ^D 为结束输入的符号

2、字符串中的方括号和变量替换

"$var[0]" 为数组@var 的第一个元素。

"$var\[0]" 将字符"["转义，等价于"$var". "[0]"，$var 被变量替换，[0]保持不变。

"${var}[0]" 亦等价于"$var" ."[0]"。

"$\{var}"则取消了大括号的变量替换功能，包含文字：${var}.

3、列表范围：

(1..10) = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

(2, 5..7, 11) = (2, 5, 6, 7, 11)

(3..3) = (3)

.用于实数

(2.1..5.3) = (2.1, 3.1 ,4.1, 5.1)

(4.5..1.6) = ()

.用于字符串

("aaa".."aad") = ("aaa","aab", "aac", "aad")

@day_of_month = ("01".."31")

.可包含变量或表达式

($var1..$var2+5)

.小技巧：

$fred = "Fred";

print (("Hello, " . $fred . "!\n") x 2);

其结果为：

Hello, Fred!

Hello, Fred!

4、数组的输出：

(1) @array = (1, 2, 3);

print (@array, "\n");

结果为：

123

(2) @array = (1, 2, 3);

print ("@array\n");

结果为：

1 2 3

5、列表/数组的长度

当数组变量出现在预期简单变量出现的地方，则 PERL 解释器取其长度。

@array = (1, 2, 3);

$scalar = @array; # $scalar = 3,即@array 的长度

($scalar) = @array; # $scalar = 1,即@array 第一个元素的值

注：以数组的长度为循环次数可如下编程：

```
$count = 1;
while ($count <= @array) {
print ("element $count: $array[$count-1]\n");
$count++;
}
```

6、子数组

```
@array = (1, 2, 3, 4, 5);
@subarray = @array[0,1]; # @subarray = (1, 2)
@subarray2 = @array[1..3]; # @subarray2 = (2,3,4)
@array[0,1] = ("string", 46); # @array =("string",46,3,4,5) now
```

@array[0..3] = (11, 22, 33, 44); # @array = (11,22,33,44,5) now

@array[1,2,3] = @array[3,2,4]; # @array = (11,44,33,5,5) now

@array[0..2] = @array[3,4]; # @array = (5,5,"",5,5) now

可以用子数组形式来交换元素：

@array[1,2] = @array[2,1];

7、有关数组的库函数

(1)sort--按字符顺序排序

@array = ("this", "is", "a","test");

@array2 = sort(@array); # @array2 = ("a","is", "test", "this")

@array = (70, 100, 8);

@array = sort(@array); # @array = (100, 70, 8) now

(2)reverse--反转数组

@array2 = reverse(@array);

@array2 = reverse sort (@array);

(3)chop--数组去尾

chop 的意义是去掉 STDIN（键盘）输入字符串时最后一个字符--换行符。而如果它作用到数组上，则将数组中每一个元素都做如此处理。

@list = ("rabbit", "12345","quartz");

chop (@list); # @list = ("rabbi", "1234","quart") now

(4)join/split--连接/拆分

join 的第一个参数是连接所用的中间字符，其余则为待连接的字符数组。

$string = join(" ", "this", "is","a", "string"); # 结果为"this is a string"

@list = ("words","and");

$string = join("::", @list, "colons"); #结果为 "words::and::colons"

@array = split(/::/,$string); # @array = ("words","and", "colons") now

第五章 文件读写

一、打开、关闭文件

语法为 open (filevar, filename)，其中 filevar 为文件句柄，或者说是程序中用来代表某文件的代号，filename 为文件名，其路径可为相对路径，亦可为绝对路径。

```
open(FILE1,"file1");
open(FILE1, "/u/jqpublic/file1");
```

打开文件时必须决定访问模式，在 PERL 中有三种访问模式：读、写和添加。后两种模式的区别在于写模式将原文件覆盖，原有内容丢失，形式为：open(outfile,">outfile");而添加模式则在原文件的末尾处继续添加内容，形式为：open(appendfile, ">>appendfile")。要注意的是，不能对文件同时进行读和写/添加操作。

open 的返回值用来确定打开文件的操作是否成功，当其成功时返回非零值，失败时返回零，因此可以如下判断：

```
if (open(MYFILE, "myfile")) {
# here's what to do if the file opened successfully
}
```

当文件打开失败时结束程序：

```
unless (open (MYFILE, "file1")) {
die ("cannot open input file file1\n");
}
```

亦可用逻辑或操作符表示如下：

```
open (MYFILE, "file1") || die ("Could not open file");
```

当文件操作完毕后，用 close(MYFILE); 关闭文件。

二、读文件

语句$line = <MYFILE>;从文件中读取一行数据存储到简单变量$line 中并把文件指针向后移动一行。<STDIN>为标准输入文

件，通常为键盘输入，不需要打开。

语句@array = <MYFILE>;把文件的全部内容读入数组@array，文件的每一行(含回车符)为@array 的一个元素。

三、写文件

形式为：

```
open(OUTFILE, ">outfile");
print OUTFILE ("Here is an output line.\n");
```

注：STDOUT、STDERR 为标准输出和标准错误文件，通常为屏幕，且不需要打开。

四、判断文件状态

1、文件测试操作符

语法为：-op expr，如：

```
if (-e "/path/file1") {
print STDERR ("File file1 exists.\n");
}
```

文件测试操作符

操作符	*描述*
-b	是否为块设备
-c	是否为字符设备
-d	是否为目录
-e	是否存在
-f	是否为普通文件
-g	是否设置了 setgid 位
-k	是否设置了 sticky 位
-l	是否为符号链接
-o	是否拥有该文件
-p	是否为管道
-r	是否可读
-s	是否非空
-t	是否表示终端

-u	是否设置了 setuid 位
-w	是否可写
-x	是否可执行
-z	是否为空文件
-A	距上次访问多长时间
-B	是否为二进制文件
-C	距上次访问文件的 inode 多长时间
-M	距上次修改多长时间
-O	是否只为“真正的用户”所拥有
-R	是否只有“真正的用户”可读
-S	是否为 socket
-T	是否为文本文件
-W	是否只有"真正的用户"可写
-X	是否只有"真正的用户"可执行
注：“真正的用户”指登录时指定的 userid，与当前进程用户 ID 相对，命令 suid 可以改变有效用户 ID。	

例：

```
unless (open(INFILE, "infile")) {
die ("Input file infile cannot be opened.\n");
}
if (-e "outfile") {
die ("Output file outfile already exists.\n");
}
unless (open(OUTFILE, ">outfile")) {
die ("Output file outfile cannot be opened.\n");
}
```

等价于

```
open(INFILE, "infile") && !(-e "outfile") &&
open(OUTFILE, ">outfile") || die("Cannot open files\n");
```

五、命令行参数

象C一样，PERL也有存储命令行参数的数组@ARGV，可以用来分别处理各个命令行参数；与C不同的是，$ARGV[0]是第一个参数，而不是程序名本身。

```
$var = $ARGV[0]; # 第一个参数
$numargs = @ARGV; # 参数的个数
```

PERL中，<>操作符实际上是对数组@ARGV的隐含的引用，其工作原理为：

1、当PERL解释器第一次看到<>时，打开以$ARGV[0]为文件名的文件；

2、执行动作shift(@ARGV); 即把数组@ARGV的元素向前移动一个，其元素数量即减少了一个。

3、<>操作符读取在第一步打开的文件中的所有行。

4、读完后，解释器回到第一步重复。

例：

```
@ARGV = ("myfile1", "myfile2"); #实际上由命令行参数赋值
while ($line = <>) {
print ($line);
}
```

将把文件myfile1和myfile2的内容打印出来。

六、打开管道

用程序的形式也可以象命令行一样打开和使用管道(ex:ls > tempfile)。如语句open (MYPIPE, "| cat >hello"); 打开一个管道，发送到MYPIPE的输出成为命令"cat >hello"的输入。由于cat命令将显示输入文件的内容，故该语句等价于open(MYPIPE, ">hello"); 用管道发送邮件如下：

```
open (MESSAGE, "| mail dave");
print MESSAGE ("Hi, Dave! Your Perl program sent this!\n");
close (MESSAGE);
```

第六章 模式匹配

一、简介

模式指在字符串中寻找的特定序列的字符，由反斜线包含：/def/即模式 def。其用法如结合函数 split 将字符串用某模式分成多个单词：@array = split(/ /, $line);

二、匹配操作符 =~、!~

=~检验匹配是否成功：$result = $var =~ /abc/;若在该字符串中找到了该模式，则返回非零值，即 true，不匹配则返回 0，即 false。!~则相反。

这两个操作符适于条件控制中，如：

```
if ($question =~ /please/) {
  print ("Thank you for being polite!\n");
}
else {
  print ("That was not very polite!\n");
}
```

三、模式中的特殊字符

PERL 在模式中支持一些特殊字符，可以起到一些特殊的作用。

1、字符 +

+意味着一个或多个相同的字符，如：/de+f/指 def、deef、deeeeef 等。它尽量匹配尽可能多的相同字符，如/ab+/在字符串 abbc 中匹配的将是 abb，而不是 ab。

当一行中各单词间的空格多于一个时，可以如下分割：

```
@array = split (/ +/, $line);
```

注：split 函数每次遇到分割模式，总是开始一个新单词，因此若$line 以空格打头，则@array 的第一个元素即为空元素。但其可以区分是否真有单词，如若$line 中只有空格，则@array 则为空数组。且上例中 TAB 字符被当作一个单词。注意修正。

2、字符 []和[^]

[]意味着匹配一组字符中的一个，如/a[0123456789]c/将匹配 a 加数字加 c 的字符串。与+联合使用例：/d[eE]+f/匹配 def、dEf、deef、dEdf、dEEEeeeEef 等。^表示除其之外的所有字符，如：/d[^deE]f/匹配 d 加非 e 字符加 f 的字符串。

3、字符 *和?

它们与+类似，区别在于*匹配 0 个、1 个或多个相同字符，?匹配 0 个或 1 个该字符。如/de*f/匹配 df、def、deeeef 等；/de?f/匹配 df 或 def。

4、转义字符

如果你想在模式中包含通常被看作特殊意义的字符，须在其前加斜线"\"。如：/*+/中*即表示字符*，而不是上面提到的一个或多个字符的含义。斜线的表示为/\\/。在 PERL5 中可用字符对\Q 和\E 来转义。

5、匹配任意字母或数字

上面提到模式/a[0123456789]c/匹配字母 a 加任意数字加 c 的字符串，另一种表示方法为：/a[0-9]c/，类似的，[a-z]表示任意小写字母，[A-Z]表示任意大写字母。任意大小写字母、数字的表示方法为：/[0-9a-zA-Z]/。

6、锚模式

锚	描述
^ 或 \A	仅匹配串首
$ 或 \Z	仅匹配串尾
\b	匹配单词边界
\B	单词内部匹配

例 1：/^def/只匹配以 def 打头的字符串，/$def/只匹配以 def 结尾的字符串，结合起来的/^def$/只匹配字符串 def(?)。\A 和\Z 在多行匹配时与^和$不同。

例 2：检验变量名的类型：

```
if ($varname =~ /^\$[A-Za-z][_0-9a-zA-Z]*$/) {
  print ("$varname is a legal scalar variable\n");
} elsif ($varname =~ /^@[A-Za-z][_0-9a-zA-Z]*$/) {
  print ("$varname is a legal array variable\n");
} elsif ($varname =~ /^[A-Za-z][_0-9a-zA-Z]*$/) {
  print ("$varname is a legal file variable\n");
} else {
  print ("I don't understand what $varname is.\n");
}
```

例 3：\b 在单词边界匹配：/\bdef/匹配 def 和 defghi 等以 def 打头的单词，但不匹配 abcdef。/def\b/匹配 def 和 abcdef 等以 def 结尾的单词，但不匹配 defghi，/\bdef\b/只匹配字符串 def。注意：/\bdef/可匹配$defghi，因为$并不被看作是单词的部分。

例 4：\B 在单词内部匹配：/\Bdef/匹配 abcdef 等，但不匹配 def；/def\B/匹配 defghi 等；/\Bdef\B/匹配 cdefg、abcdefghi 等，但不匹配 def,defghi,abcdef。

7、模式中的变量替换

将句子分成单词：

```
$pattern = "[\\t ]+";
@words = split(/$pattern/, $line);
```

8、字符范围转义

E 转义字符	描述	范围
\d	任意数字	[0-9]
\D	除数字外的任意字符	[^0-9]
\w	任意单词字符	[_0-9a-zA-Z]
\W	任意非单词字符	[^_0-9a-zA-Z]
\s	空白	[\r\t\n\f]
\S	非空白	[^ \r\t\n\f]

例：/[\da-z]/匹配任意数字或小写字母。

9、匹配任意字符

字符"."匹配除换行外的所有字符，通常与*合用。

10、匹配指定数目的字符

字符对{}指定所匹配字符的出现次数。如：/de{1,3}f/匹配 def,deef 和 deeef；/de{3}f/匹配 deeef；/de{3,}f/匹配不少于 3 个

e 在 d 和 f 之间；/de{0,3}f/匹配不多于 3 个 e 在 d 和 f 之间。

11、指定选项

字符"|"指定两个或多个选择来匹配模式。如：/def|ghi/匹配 def 或 ghi。

例：检验数字表示合法性

```
if ($number =~ /^-?\d+$|^-?0[xX][\da-fa-F]+$/) {
  print ("$number is a legal integer.\n");
} else {
  print ("$number is not a legal integer.\n");
}
```

其中 ^-?\d+$ 匹配十进制数字，^-?0[xX][\da-fa-F]+$ 匹配十六进制数字。

12、模式的部分重用

当模式中匹配相同的部分出现多次时，可用括号括起来，用\n 来多次引用，以简化表达式：

/\d{2}([\W])\d{2}\1\d{2}/ 匹配：

12-05-92

26.11.87

07 04 92 等

注意：/\d{2}([\W])\d{2}\1\d{2}/ 不同于 /(\d{2})([\W])\1\2\1/ ，后者只匹配形如 17-17-17 的字符串，而不匹配 17-05-91 等。

13、转义和特定字符的执行次序

象操作符一样，转义和特定字符也有执行次序：

特殊字符	描述
()	模式内存
+ * ? {}	出现次数
^ $ \b \B	锚
|	选项

14、指定模式定界符

缺省的，模式定界符为反斜线/，但其可用字母 m 自行指定，

如：

m!/u/jqpublic/perl/prog1! 等价于/\/u\/jqpublic\/perl\/prog1/

注：当用字母'作为定界符时，不做变量替换；当用特殊字符作为定界符时，其转义功能或特殊功能即不能使用。

15、模式次序变量

在模式匹配后调用重用部分的结果可用变量$n，全部的结果用变量$&。

```
$string = "This string contains the number 25.11.";
$string =~ /-?(\d+)\.?(\d+)/; # 匹配结果为 25.11
$integerpart = $1; # now $integerpart = 25
$decimalpart = $2; # now $decimalpart = 11
$totalpart = $&; # now totalpart = 25.11
```

四、模式匹配选项

选项	描述
g	匹配所有可能的模式
i	忽略大小写
m	将串视为多行
o	只赋值一次
s	将串视为单行
x	忽略模式中的空白

1、匹配所有可能的模式(g选项)

@matches = "balata" =~ /.a/g; # now @matches = ("ba", "la", "ta")

匹配的循环：

```
while ("balata" =~ /.a/g) {
```

```
  $match = $&;
  print ("$match\n");
}
```

结果为：

```
  ba
  la
  ta
```

当使用了选项 g 时，可用函数 pos 来控制下次匹配的偏移：

```
  $offset = pos($string);
  pos($string) = $newoffset;
```

2、忽略大小写(i 选项)例

/de/i 匹配 de,dE,De 和 DE。

3、将字符串看作多行(m 选项)

在此情况下，^符号匹配字符串的起始或新的一行的起始；$符号匹配任意行的末尾。

4、只执行一次变量替换例

```
$var = 1;
$line = <STDIN>;
while ($var < 10) {
  $result = $line =~ /$var/o;
  $line = <STDIN>;
  $var++;
}
```

每次均匹配/1/。

5、将字符串看作单行例

/a.*bc/s 匹配字符串 axxxxx \nxxxxbc，但/a.*bc/则不匹配该字符串。

6、在模式中忽略空格

/\d{2} ([\W]) \d{2} \1 \d{2}/x 等价于 /\d{2}([\W])\d{2}\1\d{2}/。

五、替换操作符

语法为 s/pattern/replacement/，其效果为将字符串中与 pattern 匹配的部分换成 replacement。如：

```
  $string = "abc123def";
```

$string =~ s/123/456/; # now $string = "abc456def";

在替换部分可使用模式次序变量$n，如 s/(\d+)/[$1]/，但在替换部分不支持模式的特殊字符，如{},*,+等，如 s/abc/[def]/将把 abc 替换为[def]。

替换操作符的选项如下表：

选项	描述
g	改变模式中的所有匹配
i	忽略模式中的大小写
e	替换字符串作为表达式
m	将待匹配串视为多行
o	仅赋值一次
s	将待匹配串视为单行
x	忽略模式中的空白

注：e 选项把替换部分的字符串看作表达式，在替换之前先计算其值，如：

$string = "0abc1";

$string =~ s/[a-zA-Z]+/$& x 2/e; # now $string = "0abcabc1"

六、翻译操作符

这是另一种替换方式，语法如：tr/string1/string2/。同样，string2 为替换部分，但其效果是把 string1 中的第一个字符替换为 string2 中的第一个字符，把 string1 中的第二个字符替换为 string2 中的第二个字符，依此类推。如：

$string = "abcdefghicba";

$string =~ tr/abc/def/; # now string = "defdefghifed"

当 string1 比 string2 长时，其多余字符替换为 string2 的最后

一个字符；当 string1 中同一个字符出现多次时，将使用第一个替换字符。

翻译操作符的选项如下：

选项	描述
c	翻译所有未指定字符
d	删除所有指定字符
s	把多个相同的输出字符缩成一个

如$string =~ tr/\d/ /c;把所有非数字字符替换为空格。$string =~ tr/\t //d；删除 tab 和空格；　$string =~ tr/0-9/ /cs；把数字间的其它字符替换为一个空格。

七、扩展模式匹配

PERL 支持 PERL4 和标准 UNIX 模式匹配操作所没有的一些模式匹配能力。其语法为：(?<c>pattern)，其中 c 是一个字符，pattern 是起作用的模式或子模式。

1、不存贮括号内的匹配内容

在 PERL 的模式中，括号内的子模式将存贮在内存中，此功能即取消存贮该括号内的匹配内容，如/(?:a|b|c)(d|e)f\1/中的\1 表示已匹配的 d 或 e，而不是 a 或 b 或 c。

2、内嵌模式选项

通常模式选项置于其后，有四个选项：i、m、s、x 可以内嵌使用，语法为：/(?option)pattern/，等价于/pattern/option。

3、肯定的和否定的预见匹配

肯定的预见匹配语法为/pattern(?=string)/，其意义为匹配后面为 string 的模式，相反的，(?!string)意义为匹配后面非 string 的模式，如：

```
$string = "25abc8";
$string =~ /abc(?=[0-9])/;
$matched = $&; # $&为已匹配的模式，此处为 abc，而不是 abc8
```

4、模式注释

PERL5 中可以在模式中用?#来加注释，如：

```
 if ($string =~ /(?i)[a-z]{2,3}(?# match two or three alphabetic characters)/ {
   …
 }
```

第七章 控制结构

一、条件判断

```
if ( <expression>) {
  <statement_block_1>
}
elsif ( <expression> ) {
  <statement_block_2>
}
…
else{
  <statement_block_3>
}
```

二、循环：

1、while 循环

```
while ( <expression> ) {
  <statement_block>
}
```

2、until 循环

```
until ( <expression> ) {
  <statement_block>
}
```

3、类 C 的 for 循环 ，如

```
for ($count=1; $count <= 5; $count++) {
  # statements inside the loop go here
}
```

下面是在 for 循环中使用逗号操作符的例子：

```
for ($line = <STDIN>, $count = 1; $count <= 3;  $line =
<STDIN>, $count++) {
  print ($line);
}
```

它等价于下列语句：

```
$line = <STDIN>;
$count = 1;
while ($count <= 3) {
  print ($line);
  $line = <STDIN>;
  $count++;
}
```

4、针对列表(数组)每个元素的循环：foreach，语法为：

```
foreach localvar (listexpr) {
  statement_block;
}
```

例：

```
foreach $word (@words) {
  if ($word eq "the") {
    print ("found the word 'the'\n");
  }
```

}

注：

(1)此处的循环变量 localvar 是个局部变量，如果在此之前它已有值，则循环后仍恢复该值。

(2)在循环中改变局部变量，相应的数组变量也会改变，如：

```
@list = (1, 2, 3, 4, 5);
foreach $temp (@list) {
  if ($temp == 2) {
    $temp = 20;
  }
}
```

此时@list 已变成了(1, 20, 3, 4, 5)。

5、do 循环

```
do {
  statement_block
} while_or_until (condexpr);
```

do 循环至少执行一次循环。

6、循环控制

退出循环为 last，与 C 中的 break 作用相同；执行下一个循环为 next，与 C 中的 continue 作用相同；PERL 特有的一个命令是 redo，其含义是重复此次循环，即循环变量不变，回到循环起始点，但要注意，redo 命令在 do 循环中不起作用。

7、传统的 goto label;语句。

三、单行条件

语法为 statement keyword condexpr。其中 keyword 可为 if、unless、while 或 until，如：

```
print ("This is zero.\n") if ($var == 0);
print ("This is zero.\n") unless ($var != 0);
print ("Not zero yet.\n") while ($var-- > 0);
print ("Not zero yet.\n") until ($var-- == 0);
```

虽然条件判断写在后面，但却是先执行的。

第八章 子程序

一、定义

子程序即执行一个特殊任务的一段分离的代码，它可以使减少重复代码且使程序易读。PERL 中，子程序可以出现在程序的任何地方。定义方法为：

```
sub subroutine{
  statements;
}
```

二、调用

调用方法如下：

1、用&调用

```
&subname;
…
sub subname{
  …
}
```

2、先定义后调用 ，可以省略&符号

```
sub subname{
  …
}
…
subname;
```

3、前向引用 ，先定义子程序名，后面再定义子程序体

```
sub subname;
…
subname;
…
sub subname{
  …
}
```

4、用 do 调用

```
do my_sub(1, 2, 3);等价于&my_sub(1, 2, 3);
```

三、返回值

缺省的，子程序中最后一个语句的值将用作返回值。语句 return (retval);也可以推出子程序并返回值 retval，retval 可以为列表。

四、局部变量

子程序中局部变量的定义有两种方法：my 和 local。其区别是：my 定义的变量只在该子程序中存在；而 local 定义的变量不存在于主程序中，但存在于该子程序和该子程序调用的子程序中(在 PERL4 中没有 my)。定义时可以给其赋值，如：

```
my($scalar) = 43;
local(@array) = (1, 2, 3);
```

五、子程序参数传递

1、形式

```
&sub1(&number1, $number2, $nubmer3);
…
sub sub1{
  my($number1, $number2, $number3) = @_;
  …
}
```

2、传送数组

```
&addlist (@mylist);
&addlist ("14", "6", "11");
&addlist ($value1, @sublist, $value2);
...
sub addlist {
  my (@list) = @_;
  ...
}
```

参数为数组时，子程序只将它赋给一个数组变量。如

```
sub twolists {
  my (@list1, @list2) = @_;
}
```

中@list2必然为空。但简单变量和数组变量可以同时传递：

```
&twoargs(47, @mylist); # 47 赋给$scalar，@mylist 赋给@list
&twoargs(@mylist); # @mylist 的第一个元素赋给$scalar，其余的元素赋给@list
...
sub twoargs {
  my ($scalar, @list) = @_;
  ...
}
```

六、递归子程序

PERL中，子程序可以互相调用，其调用方法与上述相同，当调用该子程序本身时，即成了递归子程序。递归子程序有两个条件：1、除了不被子程序改变的变量外，所有的变量必须的局部的；2、该子程序要含有停止调用本身的代码。

七、用别名传递数组参数

1、用前面讲到的调用方法&my_sub(@array)将把数组@array的数据拷贝到子程序中的变量@_中，当数组很大时，将会花费较多的资源和时间，而用别名传递将不做这些工作，而对该数组直接操作。形式如：

```
@myarray = (1, 2, 3, 4, 5);
&my_sub(*myarray);
sub my_sub {
```

```
  my (*subarray) = @_;
}
```

2、此方法类似于C语言中的传递数组的起始地址指针，但并不一样，在定义数组的别名之后，如果有同名的简单变量，则对该变量也是起作用的。如：

```
$foo = 26;
@foo = ("here's", "a", "list");
&testsub (*foo);
...
sub testsub {
  local (*printarray) = @_;
  ...
  $printarray = 61;
}
```

当子程序执行完，主程序中的$foo的值已经成了61，而不再是26了。

3、用别名的方法可以传递多个数组，如：

```
@array1 = (1, 2, 3);
@array2 = (4, 5, 6);
&two_array_sub (*array1, *array2);
sub two_array_sub {
  my (*subarray1, *subarray2) = @_;
}
```

在该子程序中，subarray1是array1的别名，subarray2是array2的别名。

八、预定义的子程序

PERL5预定义了三个子程序，分别在特定的时间执行，它们是：BEGIN子程序在程序启动时被调用；END子程序在程序结束时被调用；AUTOLOAD子程序在找不到某个子程序时被调用。你可以自己定义它们，以在特定时间执行所需要的动作。如：

```
BEGIN {
  print("Hi! Welcome to Perl!\n");
}
AUTOLOAD{
```

```
    print("subroutine $AUTOLOAD not found\n"); # 变量
$AUTOLOAD 即未找到的子程序名
    print("arguments passed: @_\n");
  }
```

若同一个预定义子程序定义了多个，则 BEGIN 顺序执行，END 逆序执行。

第九章 关联数组/哈希表

一、数组变量的限制

在前面讲的数组变量中，可以通过下标访问其中的元素。例如，下列语句访问数组@array 的第三个元素：

```
$scalar = $array[2];
```

虽然数组很有用，但它们有一个显著缺陷，即很难记住哪个元素存贮的什么内容。假如我们来写一个程序计算某文件中首字母大写的单词出现的次数，用数组来实现就比较困难，程序代码如下：

```
1 : #!/usr/local/bin/perl
2 :
3 : while ($inputline = <STDIN>) {
4 :     while ($inputline =~ /\b[A-Z]\S+/g) {
5 :       $word = $&;
6 :       $word =~ s/[;.,:-]$//; # remove punctuation
7 :       for ($count = 1; $count <= @wordlist;
8 :            $count++) {
9 :         $found = 0;
10:         if ($wordlist[$count-1] eq $word) {
11:           $found = 1;
12:           $wordcount[$count-1] += 1;
13:           last;
14:         }
15:       }
16:       if ($found == 0) {
17:         $oldlength = @wordlist;
18:         $wordlist[$oldlength] = $word;
19:         $wordcount[$oldlength] = 1;
20:       }
21:     }
22: }
23: print ("Capitalized words and number of occurrences:\n");
24: for ($count = 1; $count <= @wordlist; $count++) {
25:     print ("$wordlist[$count-1]: $wordcount[$count-1]\n");
26: }
```

运行结果如下：

```
Here is a line of Input.
This Input contains some Capitalized words.
^D
Capitalized words and number of occurrences:
Here: 1
Input: 2
```

This: 1
Capitalized: 1

这个程序每次从标准输入文件读一行文字，第四行起的循环匹配每行中首字母大写的单词，每找到一个循环一次，赋给简单变量$word。在第六行中去掉标点后，查看该单词是否曾出现过，7~15 行中在@wordlist 中挨个元素做此检查，如果某个元素与$word 相等，@wordcount 中相应的元素就增加一个数。如果没有出现过，即@wordlist 中没有元素与$word 相等，16~20 行给@wordlist 和@wordcount 增加一个新元素。

二、定义

正如你所看到的，使用数组元素产生了一些问题。首先，@wordlist 中哪个元素对应着哪个单词并不明显；更糟的是，每读进一个新单词，程序必须检查整个列表才能知道该单词是否曾经出现过，当列表变得较大时，这是很耗费时间的。

这些问题产生的原因是数组元素通过数字下标访问，为了解决这类问题，Perl 定义了另一种数组，可以用任意简单变量值来访问其元素，这种数组叫做关联数组，也叫哈希表。

为了区分关联数组变量与普通的数组变量，Perl 使用%作为其首字符，而数组变量以@打头。与其它变量名一样，%后的第一个字符必须为字母，后续字符可以为字母、数字或下划线。

三、访问关联数组的元素

关联数组的下标可以为任何简单/标量值，访问单个元素时以$符号打头，下标用大括号围起来。例如：

```
$fruit{"bananas"}
$number{3.14159}
$integer{-7}
```

简单变量也可作为下标，如：

```
$fruit{$my_fruit}
```

四、增加元素

创建一个关联数组元素最简单的方法是赋值，如语句$fruit{"bananas"} = 1；把 1 赋给关联数组%fruit 下标为 bananas 的元素，如果该元素不存在，则被创建，如果数组%fruit 从未使用过，也被创建。

这一特性使得关联数组很容易用于计数。下面我们用关联数组改写上面的程序，注意实现同样的功能此程序简化了许多。

```
1 : #!/usr/local/bin/perl
2 :
3 : while ($inputline = ) {
4 :     while ($inputline =~ /\b[A-Z]\S+/g) {
5 :       $word = $&;
6 :       $word =~ s/[;.,:-]$//; # remove punctuation
7 :       $wordlist{$word} += 1;
8 :     }
9 : }
10: print ("Capitalized words and number of occurrences:\n");
11: foreach $capword (keys(%wordlist)) {
12:     print ("$capword: $wordlist{$capword}\n");
13: }
```

运行结果如下：

```
Here is a line of Input.
This Input contains some Capitalized words.
^D
Capitalized words and number of occurrences:
This: 1
Input: 2
Here: 1
Capitalized: 1
```

你可以看到，这次程序简单多了，读取输入并存贮各单词数目从 20 行减少到了 7 行。

本程序用关联数组%wordlist 跟踪首字母大写的单词，下标就用单词本身，元素值为该单词出现的次数。第 11 行使用了内嵌函数 keys()。这个函数返回关联数组的下标列表，foreach 语句就用此列表循环。

注：关联数组总是随机存贮的，因此当你用 keys()访问其所有

元素时，不保证元素以任何顺序出现，特别值得一提的是，它们不会以被创建的顺序出现。

要想控制关联数组元素出现的次序，可以用 sort()函数对 keys()返回值进行排列，如：

```
foreach $capword (sort keys(%wordlist)) {
  print ("$capword: $wordlist{$capword}\n");
}
```

五、创建关联数组

可以用单个赋值语句创建关联数组，如：

```
%fruit = ("apples",17,"bananas",9,"oranges","none");
```

此语句创建的关联数组含有下面三个元素：

- 下标为 apples 的元素，值为 17
- 下标为 bananas 的元素，值为 9
- 下标为 oranges 的元素，值为 none

注：用列表给关联数组赋值时，Perl5 允许使用"=>"或","来分隔下标与值，用"=>"可读性更好些，上面语句等效于：

```
%fruit = ("apples"=>17,"bananas"=>9,"oranges"=>"none");
```

六、从数组变量复制到关联数组

与列表一样，也可以通过数组变量创建关联数组，当然，其元素数目应该为偶数，如：

```
@fruit = ("apples",17,"bananas",9,"oranges","none");
%fruit = @fruit;
```

反之，可以把关联数组赋给数组变量，如：

```
%fruit = ("grapes",11,"lemons",27);
@fruit = %fruit;
```

注意，此语句中元素次序未定义，那么数组变量@fruit 可能为("grapes",11,"lemons",27)或("lemons",27,"grapes",11)。

关联数组变量之间可以直接赋值，如：%fruit2 = %fruit1; 还可以把数组变量同时赋给一些简单变量和一个关联数组变量，如：

```
($var1, $var2, %myarray) = @list;
```

此语句把@list 的第一个元素赋给$var1，第二个赋给$var2，其余的赋给%myarray。

最后，关联数组可以通过返回值为列表的内嵌函数或用户定义的子程序来创建，下例中把 split()函数的返回值--一个列表--赋给一个关联数组变量。

```
1: #!/usr/local/bin/perl
2:
3: $inputline = <STDIN>;
4: $inputline =~ s/^\s+|\s+\n$//g;
5: %fruit = split(/\s+/, $inputline);
6: print ("Number of bananas: $fruit{\"bananas\"}\n");
```

运行结果如下：

```
oranges 5 apples 7 bananas 11 cherries 6
Number of bananas: 11
```

七、元素的增删

增加元素已经讲过，可以通过给一个未出现过的元素赋值来向关联数组中增加新元素，如$fruit{"lime"} = 1;创建下标为 lime、值为 1 的新元素。

删除元素的方法是用内嵌函数 delete，如欲删除上述元素，则：

```
delete ($fruit{"lime"});
```

注意：

1、一定要使用 delete 函数来删除关联数组的元素，这是唯一的方法。

2、一定不要对关联数组使用内嵌函数 push、pop、shift 及 splice，因为其元素位置是随机的。

八、列出数组的索引和值

上面已经提到，keys()函数返回关联数组下标的列表，如：

```
%fruit = (″apples″, 9,
          ″bananas″, 23,
          ″cherries″, 11);
@fruitsubs = keys(%fruits);
```

这里，@fruitsubs 被赋给 apples、bananas、cherries 构成的列表，再次提请注意，此列表没有次序，若想按字母顺序排列，可使用 sort()函数。

@fruitindexes = sort keys(%fruits);

这样结果为("apples","bananas","cherries")。类似的，内嵌函数 values()返回关联数组值的列表，如：

```
%fruit = (″apples″, 9,
                        ″bananas″, 23,
                        ″cherries″, 11);
          @fruitvalues = values(%fruits);
```

这里，@fruitvalues 可能的结果为(9,23.11)，次序可能不同。

九、用关联数组循环

前面已经出现过利用 keys()函数的 foreach 循环语句，这种循环效率比较低，因为每返回一个下标，还得再去寻找其值，如：

```
foreach $holder (keys(%records)){
  $record = $records{$holder};
}
```

Perl 提供一种更有效的循环方式，使用内嵌函数 each()，如：

```
%records = ("Maris", 61, "Aaron", 755, "Young", 511);
while (($holder, $record) = each(%records)) {
  # stuff goes here
}
```

each()函数每次返回一个双元素的列表，其第一个元素为下标，第二个元素为相应的值，最后返回一个空列表。

注意：千万不要在 each()循环中添加或删除元素，否则会产生不可预料的后果。

十、用关联数组创建数据结构

用关联数组可以模拟在其它高级语言中常见的多种数据结构，本节讲述如何用之实现：链表、结构和树。

1、(单)链表

链表是一种比较简单的数据结构，可以按一定的次序存贮值。每个元素含有两个域，一个是值，一个是引用（或称指针），指向链表中下一个元素。一个特殊的头指针指向链表的第一个元素。

在 Perl 中，链表很容易用关联数组实现，因为一个元素的值

可以作为下一个元素的索引。下例为按字母顺序排列的单词链表：

```
%words = ("abel", "baker",
          "baker", "charlie",
          "charlie", "delta",
          "delta", "");
$header = "abel";
```

上例中，简单变量$header含有链表中第一个单词，它同时也是关联数组第一个元素的下标，其值baker又是下一个元素的下标，依此类推。

下标为delta的最后一个元素的值为空串，表示链表的结束。

在将要处理的数据个数未知或其随程序运行而增长的情况下，链表十分有用。下例用链表按字母次序输出一个文件中的单词。

```
1 : #!/usr/local/bin/perl
2 :
3 : # initialize list to empty
4 : $header = "";
5 : while ($line = <STDIN>) {
6 :     # remove leading and trailing spaces
7 :     $line =~ s/^\s+|\s+$//g;
8 :     @words = split(/\s+/, $line);
9 :     foreach $word (@words) {
10:         # remove closing punctuation, if any
11:         $word =~ s/[.,;:-]$//;
12:         # convert all words to lower case
13:         $word =~ tr/A-Z/a-z/;
14:         &add_word_to_list($word);
15:     }
16: }
17: &print_list;
18:
19: sub add_word_to_list {
20:     local($word) = @_;
```

```
21:     local($pointer);
22:
23:     # if list is empty, add first item
24:     if ($header eq "") {
25:       $header = $word;
26:       $wordlist{$word} = "";
27:       return;
28:     }
29:     # if word identical to first element in list,
30:     # do nothing
31:     return if ($header eq $word);
32:     # see whether word should be the new
33:     # first word in the list
34:     if ($header gt $word) {
35:       $wordlist{$word} = $header;
36:       $header = $word;
37:       return;
38:     }
39:     # find place where word belongs
40:     $pointer = $header;
41:     while ($wordlist{$pointer} ne "" &&
42:       $wordlist{$pointer} lt $word) {
43:       $pointer = $wordlist{$pointer};
44:     }
45:     # if word already seen, do nothing
46:     return if ($word eq $wordlist{$pointer});
47:     $wordlist{$word} = $wordlist{$pointer};
48:     $wordlist{$pointer} = $word;
49: }
50:
51: sub print_list {
52:     local ($pointer);
53:     print ("Words in this file:\n");
54:     $pointer = $header;
```

```
55:     while ($pointer ne "") {
56:       print ("$pointer\n");
57:       $pointer = $wordlist{$pointer};
58:     }
59: }
```

运行结果如下：

```
Here are some words.
Here are more words.
Here are still more words.
^D
Words in this file:
are
here
more
some
still
words
```

此程序分为三个部分：

- 主程序：读取输入并转换到相应的格式。
- 子程序：add_word_to_list，建立排序单词链表。
- 子程序：print_list，输出单词链表

第 3~17 行为主程序，第 4 行初始化链表，将表头变量$header 设为空串，第 5 行起的循环每次读取一行输入，第 7 行去掉头、尾的空格，第 8 行将句子分割成单词。9~15 行的内循环每次处理一个单词，如果该单词的最后一个字符是标点符号，就去掉。第 13 行把单词转换成全小写形式，第 14 行传递给子程序 add_word_to_list。

子程序 add_word_to_list 先在第 24 行处检查链表是否为空。如果是，第 25 行将单词赋给$header，26 行创建链表第一个元素，存贮在关联数组%wordlist 中。如果链表非空，37 行检查第一个元素是否与该单词相同，如果相同，就立刻返回。下一步检查这一新单词是否应该为链表第一个元素，即其按字母顺序先于 $header。如果是这样，则：

1、创建一个新元素，下标为该新单词，其值为原第一个单词。

2、该新单词赋给$header。

如果该新单词不该为第一个元素，则 40~44 行利用局域变量$pointer 寻找其合适的有效位置，41~44 行循环到$wordlist{$pointer}大于或等于$word 为止。接下来 46 行查看该单词是否已在链表中，如果在就返回，否则 47~48 行将其添加到链表中。首先 47 行创建新元素$wordlist{$word}，其值为$wordlist{$pointer}，这时$wordlist{$word}和$wordlist{$pointer}指向同一个单词。然后，48 行将$wordlist{$pointer}的值赋为$word，即将$wordlist{$pointer}指向刚创建的新元素$wordlist{$word}。

最后当处理完毕后，子程序 print_list()依次输出链表，局域变量$pointer 含有正在输出的值，$wordlist{$pointer}为下一个要输出的值。

注：一般不需要用链表来做这些工作，用 sort()和 keys()在关联数组中循环就足够了，如：

```
foreach $word (sort keys(%wordlist)) {
  # print the sorted list, or whatever }
```

但是，这里涉及的指针的概念在其它数据结构中很有意义。

2、结构

许多编程语言可以定义结构(structure)，即一组数据的集合。结构中的每个元素有其自己的名字，并通过该名字来访问。

Perl 不直接提供结构这种数据结构，但可以用关联数组来模拟。例如模拟 C 语言中如下的结构：

```
          struce{
  int field1;
  int field2;
  int field3; }mystructvar;
```

我们要做的是定义一个含有三个元素的关联数组，下标分别为 field1、field2、field3，如：

```
%mystructvar = ("field1" , "" ,
      "field2" , "" ,
      "field3" , "" ,);
```

像上面 C 语言的定义一样，这个关联数组%mystrctvar 有三个元素，下标分别为 field1、field2、field3，各元素初始值均为空串。

对各元素的访问和赋值通过指定下标来进行，如：

```
$mystructvar{"field1"} = 17;
```

3、树

另一个经常使用的数据结构是树。树与链表类似，但每个节点指向的元素多于一个。最简单的树是二叉树，每个节点指向另外两个元素，称为左子节点和右子节点（或称孩子），每个子节点又指向两个孙子节点，依此类推。

注：此处所说的树像上述链表一样是单向的，每个节点指向其子节点，但子节点并不指向父节点。

树的概念可以如下描述：

- 因为每个子节点均为一个树，所以左/右子节点也称为左/右子树。（有时称左/右分支）
- 第一个节点（不是任何节点的子节点的节点）称为树的根。
- 没有孩子（子节点）的节点称为叶节点。

有多种使用关联数组实现树结构的方法，最好的一种应该是：给子节点分别加上 left 和 right 以访问之。例如，alphaleft 和 alpharight 指向 alpha 的左右子节点。下面是用此方法创建二叉树并遍历的例程：

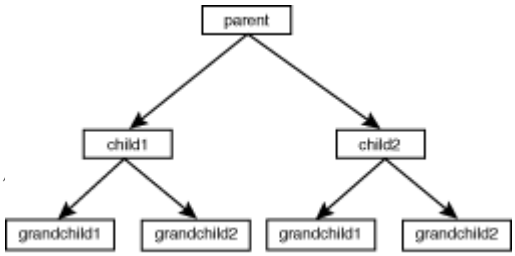
```
1 : #!/usr/local/bin/perl
2 :
3 : $rootname = "parent";
4 : %tree = ("parentleft", "child1",
5 :          "parentright", "child2",
6 :          "child1left", "grandchild1",
7 :          "child1right", "grandchild2",
8 :          "child2left", "grandchild3",
9 :          "child2right", "grandchild4");
10: # traverse tree, printing its elements
11: &print_tree($rootname);
12:
13: sub print_tree {
14:     local ($nodename) = @_;
15:     local ($leftchildname, $rightchildname);
```

```
16:
17:     $leftchildname = $nodename . "left";
18:     $rightchildname = $nodename . "right";
19:     if ($tree{$leftchildname} ne "") {
20:       &print_tree($tree{$leftchildname});
21:     }
22:     print ("$nodename\n");
23:     if ($tree{$rightchildname} ne "") {
24:       &print_tree($tree{$rightchildname});
25:     }
26: }
```

结果输出如下：

```
grandchild1
child1
grandchild2
parent
grandchild3
child2
grandchild4
```

该程序



注意函数 print_tree()以次序“左子树、节点、右子树”来输出各节点的名字，这种遍历次序称为“左序遍历”。如果把第 22 行移到 19 行前，先输出节点明，再输出左子树、右子树，则为“中序遍历”，如果把第 22 行移到 25 行后，输出次序为左子树、右子树、节点，则为“右序遍历”。

可以用同样的方法，即连接字符串构成下标，来创建其它的数据结构，如数据库等。

第十章 格式化输出

我们已经见过用 print 函数将原始的未格式化的文本输出到文件，本章讲述如何用函数 write 和打印格式来生成格式化的输出。

二、显示打印格式

打印格式的显示有两步：

1、将系统变量$~设成所要使用的格式

2、调用函数 write

例如：

```
1 : #!/usr/local/bin/perl
2 :
3 : $~ = "MYFORMAT";
4 : write;
5 :
6 : format MYFORMAT =
7 : ====================================
8 : Here is the text I want to display.
9 : ====================================
10: .
```

结果输出如下：

```
$ program
====================================
Here is the text I want to display.
```

```
====================================
$
```

如果不用$~指定打印格式，Perl解释器就假定要使用的格式名与要写入的文件变量同名，在本例中，如果不指定使用MYFORMAT，则Perl解释器试图使用名为STDOUT的打印格式。

三、在打印格式中显示值

我们使用打印格式的主要原因当然是格式化存贮在简单变量或数组变量中的值从而生成可读性好的输出，这一目的用“值域”来实现。每个值域指定一个值，如变量或表达式，调用write函数时，该值就以值域指定的格式显示。

1、通用的打印格式

打印格式的一个缺点是定义中包含了变量名，例如：

```
format MYFORMAT =
==================================================
========
The winning number is @<<<<<<!
$winnum
==================================================
========
.
```

当调用write输出此格式时，必须记着它使用了变量$winnum。用子程序和局域变量就可以创建更通用的打印格式。下例从STDIN输入一个文件并输出五个出现频率最高的字母及出现次数。

```
1 : #!/usr/local/bin/perl
2 :
3 : while ($line = ) {
4 :     $line =~ tr/A-Z/a-z/;
5 :     $line =~ s/[^a-z]//g;
6 :     @letters = split(//, $line);
7 :     foreach $letter (@letters) {
8 :       $lettercount{$letter} += 1;
9 :     }
```

```
10: }
11:
12: $~ = "WRITEHEADER";
13: write;
14: $count = 0;
15: foreach $letter (reverse sort occurrences
16:         (keys(%lettercount))) {
17:     &write_letter($letter, $lettercount{$letter});
18:     last if (++$count == 5);
19: }
20:
21: sub occurrences {
22:     $lettercount{$a} <=> $lettercount{$b};
23: }
24: sub write_letter {
25:     local($letter, $value) = @_;
26:
27:     $~ = "WRITELETTER";
28:     write;
29: }
30: format WRITEHEADER =
31: The five most frequently occurring letters are:
32: .
33: format WRITELETTER =
34:     @: @<<<<<<
35:     $letter, $value
36: .
```

运行结果如下：

```
$ program
This is a test file.
This test file contains some input.
The quick brown fox jumped over the lazy dog.
^D
The five most frequently occurring letters are:
```

```
t: 10
e: 9
i: 8
s: 7
o: 6
$
```

2、格式和局域变量

在上例中，你可能已经注意到子程序 write_letter 调用 write 输出字母及其出现次数，即使格式定义在子程序外部仍能正常工作。在第 17 行中将字母及其出现次数传递给该子程序，在子程序中，打印格式使用局域变量$letter 和$value，这样保证了在 foreach 循环中每次输出当前的字母和值。

然而要注意的是，使用 my 定义的局域变量要求格式定义在子程序内部，否则就不会输出，因此，用 write 输出的局域变量一定要用 local 定义。（local 和 my 详见《子程序》一章）

注：Perl4 中没有 my 函数，故不会有此问题。

3、选择值域格式

我们已经知道了打印格式和 write 函数怎么工作，现在来看看值域的格式，见下表：

格式	值域含义
@<<<	左对齐输出
@>>>	右对齐输出
@|||	中对齐输出
@##.##	固定精度数字
@*	多行文本

每个值域的第一个字符是行填充符，当使用@字符时，不做文本格式化。对文本的格式化稍后来讲。

在上表中，除了多行值域@*，域宽都等于其指定的包含字符@在内的字符个数，例如：

```
@###.##
```

表示七个字符宽，小数点前四个，小数点后两个。

4、输出值域字符

在打印格式里，特定字符如@、<和>被看作值域定义，那么如何将它们输出呢？方法如下：

```
format SPECIAL =
This line contains the special character @.
"@"
.
```

四、输出到其它文件

缺省地，函数 write 将结果输出到标准输出文件 STDOUT，我们也可以使它将结果输出到任意其它的文件中。最简单的方法就是把文件变量作为参数传递给 write，如：

```
write (MYFILE);
```

这样，write 就用缺省的名为 MYFILE 的打印格式输出到文件 MYFILE 中，但是这样就不能用$~变量来改变所使用的打印格式。系统变量$~只对缺省文件变量起作用，我们可以改变缺省文件变量，改变$~，再调用 write，例如：

```
select (MYFILE);
$~ = "MYFORMAT";
write;
```

当 select 改变缺省文件变量时，它返回当前缺省文件变量的内部表示，这样我们就可以创建子程序，按自己的想法输出，又不影响程序的其它部分，如下：

```
        sub write_to_stdout {
  local ($savefile, $saveformat);
  $savefile = select(STDOUT);
  $saveformat = $~;
  $~ = "MYFORMAT";
  write;
  $~ = $saveformat;
  select($savefile);
}
```

五、分页

在输出到打印机时，可以在每页顶部输出相应的信息，这样的特殊文本叫页眉。定义页眉实际上就是定义名为 filename_TOP 的打印格式，例如给标准输出文件定义页眉如下：

```
format STDOUT_TOP =
Consolidated Widgets Inc. 1994 Annual Report
.
```

在页眉的定义中也可以包含值域，页眉中经常使用的一个特殊值是当前页码，存贮在系统变量$%中，如：

```
format STDOUT_TOP =
Page @<<.
$%
.
```

我们也可以通过改变系统变量$^改变定义页眉的打印格式名，与$~一样，$^只对当前缺省文件起作用，因此可以与 select 函数结合使用。

缺省情况下，每页长度为 60 行，可以通过改变$=来改变页长，如：

```
$= = 66; #页长设为 66 行
```

此赋值语句必须出现在第一个 write 语句前。

注：一般使用分页机制时不用 print 函数，因为当用 write 输出时，Perl 解释器跟踪每页的当前行号。如果必须使用 print 而又不打乱页计数，可以调整系统变量$-。$-的含义是当前行到页末之间的行数，当$-达到零时，就开始新的一页，调整方法如：

```
print ("Here is a line of output\n");
$- -= 1;
```

六、格式化长字符串

我们已经学过值域@*可以输出多行文本，但它完全将字符串原样输出，不加以格式化。在 Perl 中对长字符串（包含换行）进行格式化的值域定义很简单，只需把打头的@字符换成^就行了，这种文本格式化中，Perl 解释器在一行中放置尽可能多的单词。每当输出一行文本，被输出的子串就从变量中删除，再次在域值中使用该变量就把剩下的字符串继续按格式输出。当内容已输出完毕，该变量就成了空串，再输出就会输出空行，为避免输出空行，可以在值域格式行首加一个~字符。见下例：

```
1：#!/usr/local/bin/perl
2：
3：@quotation = <STDIN>;
```

```
4 : $quotation = join("", @quotation);
5 : $~ = "QUOTATION";
6 : write;
7 :
8 : format QUOTATION =
9 : Quotation for the day:
10: ----------------------------
11:
~   ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
<<<<
12: $quotation
13:
~   ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
<<<<
14: $quotation
15:
~   ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
<<<<
16: $quotation
17: ----------------------------
18: .
```

运行结果如下：

```
$ program
Any sufficiently advanced programming
language is indistinguishable from magic.
^D
Quotation for the day:
----------------------------
   Any sufficiently advanced programming language is
   indistinguishable from magic.
----------------------------
$
```

如果把打印格式中行首的~字符去掉，上面的输出结果中就会多一行空行。很明显，当字符串长度不明确时，这种用法很不方便，

原因就在于它指明了输出的行数上限，超过这一上限的字符就不会被输出，解决方法很简单，就是在域值格式行首加两个~字符，这样就会持续按格式输出文本直到输出完毕，用此方法把上述程序改写如下：

```
#!/usr/local/bin/perl

@quotation = <STDIN>;
$quotation = join("", @quotation);
$~ = "QUOTATION";
write;

format QUOTATION =
Quotation for the day:
------------------------------
~~ ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
$quotation
------------------------------
.
```

这样运行结果相同。

七、用 printf 格式化输出

还有一种容易掌握和使用的格式化输出函数，那就是 printf，它与 C 语言中的 printf 基本上是相同的。printf 有两个参数，一个是字符串，其中含有一个或多个域值形式，另一个是与各域值相对应的变量值按一定格式替换，如：

```
printf("The number I want to print is %d.\n", $number);
```

各种域值形式如下表：

域值	含义
%c	单个字符
%d	十进制整数
%e	科学计数法形式的浮点数
%f	普通形式（定点）浮点数

%g	紧缩形式浮点数
%o	八进制整数
%s	字符串
%u	无符号整数
%x	十六进制整数

一些使用细节如下：

1、在格式 d、o、u 或 x 中，如果整数值较大或可能较大，可加个 l 字符，意为长整型，如%ld。

2、%字符后加正整数表示该域的最小宽度，如果输出结果宽度不足，则向右对齐，前面用空格补足，如果该正整数以数字 0 打头，则补足字符为 0。若%字符后为负整数，则结果向右对齐。

3、浮点数域值(%c、%f 和%g)中可以指定小数点前后的宽度，如%8.3f 意为总宽度为 8 个字符，小数点后（即小数部分）为 3 个字符，多出的小数部分四舍五入。

4、在整数、字符或字符串的值域中使用如上的小数形式 n.m，整数部分 n 为总宽度，小数部分 m 为输出结果的最大宽度，这样就保证了输出结果前至少有 n-m 个空格。

第十一章 文件系统

一、文件输入/输出函数

1、基本 I/O 函数

1)open 函数

2)用 open 重定向输入

3)文件重定向

4)指定读写权限

5)close 函数

6)print, printf 和 write 函数

7)select 函数

8)eof 函数

9)间接文件变量

2、跳过和重读数据

本章所讲的函数多数使用了 UNIX 操作系统的特性，在非 UNIX 系统中，一些函数可能没有定义或有不同的工作方式，使用时请查看 Perl 联机文档。

一、文件输入/输出函数

本节讲述从文件中读取信息和向文件写入信息的内置库函数。

1、基本 I/O 函数

一些 I/O 函数在前面的章节中已有讲述，如

- open：允许程序访问文件
- close：终止文件访问
- print：文件写入字符串
- write：向文件写入格式化信息
- printf：格式化字符串并输出到文件

这里简单回顾一下，再讲一些前面未提到的函数。

1)open 函数

open 函数将文件变量与某文件联系起来，提供访问文件的接口，例如：open(MYVAR, "/u/file"); 如果文件打开成功，则返回非零值，否则返回零。缺省地，open 打开文件用以读取其内容，若想打开文件以写入内容，则在文件名前加个大于号：open(MYVAR, ">/u/file"); 向已有的文件末尾添加内容用两个大于号：open(MYVAR, ">>/u/file"); 若想打开文件作为数据导向的命令，则在命令前加上管道符(|)：open(MAIL, "|mail dave");

2)用 open 重定向输入

可以把打开的文件句柄用作向程序输入数据的命令，方法是在命令后加管道符(|)，如：

open(CAT, "cat file*|");

对 open 的调用运行命令 cat file* ，此命令创建一个临时文件，这个文件的内容是所有以 file 打头的文件的内容连接而成，此文件看作输入文件，可用文件变量 CAT 访问，如：

$input = ;

下面的例子使用命令 w 的输出来列出当前登录的所有用户名。

```
1 : #!/usr/local/bin/perl
2 :
3 : open (WOUT, "w|");
4 : $time = <WOUT>;
5 : $time =~ s/^ *//;
6 : $time =~ s/ .*//;
7 : ; # skip headings line
8 : @users = ;
9 : close (WOUT);
10: foreach $user (@users) {
11:     $user =~ s/ .*//;
12: }
13: print ("Current time: $time");
14: print ("Users logged on:\n");
15: $prevuser = "";
16: foreach $user (sort @users) {
17:     if ($user ne $prevuser) {
```

```
18:       print ("\t$user");
19:       $prevuser = $user;
20:    }
21: }
```

结果输出如下：

```
Current time: 4:25pm
Users logged on:
  dave
  kilroy
  root
  zarquon
```

w 命令列出当前时间、系统负载和登录的用户，以及每个用户的作业时间和当前运行的命令，如：

```
          4:25pm  up 1 day,  6:37,  6 users,  load average: 0.79, 0.36, 0.28
          User      tty         login@  idle   JCPU   PCPU what
          dave      ttyp0       2:26pm             27     3 w
          kilroy    ttyp1       9:01am  2:27    1:04   11 -csh
          kilroy    ttyp2       9:02am    43    1:46   27 rn
          root      ttyp3       4:22pm     2           -csh
          zarquon   ttyp4       1:26pm     4      43   16 cc myprog.c
          kilroy    ttyp5       9:03am          2:14   48 /usr/games/hack
```

上例中从 w 命令的输出中取出所需的信息：当前时间和登录的用户名。第 3 行运行 w 命令，此处对 open 的调用指定 w 的输出用作程序的输入，用文件变量 WOUT 来访问该输入。第 4 行读取第一行信息，即：

4:25pm up 1 day, 6:37, 6 users, load average: 0.79, 0.36, 0.28

接下来的两行从这行中抽取出时间。首先，第 5 行删除起始的空格，然后第 6

行删去除时间和结尾换行符之间的所有字符，存入变量$time。

第 7 行从 WOUT 读取第二行，这行中无有用信息，故不作处理。第 8 行把剩下的行赋给数组@users，然后第 9 行关闭 WOUT，终止运行 w 命令的进程。

@users 中的每个元素都是一行用户信息，因为本程序只需要每行的第一个单词，即用户名，故 10~12 行去掉除换行符外的其它字符，这一循环结束后，@users 中只剩下用户名的列表。

第 13 行输出存贮在$time 中的时间，注意这时 print 不需要加上换行符，因为$time 中有。16~21 行对@users 中的用户名排序并输出。因为同一个用户可以多次登录，所以用$preuser 存贮输出的最后一个用户名，下次输出数组元素$user 时，如果其与$preser 相等，则不输出。

3)文件重定向

许多 UNIX shell 可以把标准输出文件(STDOUT)和标准错误文件(STDERR)都重定向到同一个文件，例如在 Bourne Shell（sh）中，命令

$ foo > file1 2>&1

运行命令 foo 并把输出到标准输出文件和标准错误文件的内容存贮到文件 file1 中。下面是用 Perl 实现这一功能的例子：

```
1: #!/usr/local/bin/perl
2:
3: open (STDOUT, ">file1") || die ("open STDOUT failed");
4: open (STDERR, ">&STDOUT") || die ("open STDERR failed");
5: print STDOUT ("line 1\n");
6: print STDERR ("line 2\n");
7: close (STDOUT);
8: close (STDERR);
```

运行后，文件 file1 中的内容为：

line 2

line 1

可以看到，这两行并未按我们想象的顺序存贮，为什么呢？我们来分析一下这段程序。

第 3 行重定向标准输出文件，方法是打开文件 file1 将它与文件变量 STDOUT 关联，这也关闭了标准输出文件。第 4 行重定向标准错误文件，参数>&STDOUT 告诉 Perl 解释器使用已打开并与 STDOUT 关联的文件，即文件变量 STDERR 指向与 STDOUT 相同的文件。第 5、6 行分别向 STDOUT 和 STDERR 写入数据，

因为这两个文件变量指向同一个文件，故两行字符串均写到文件 file1 中，但顺序却是错误的，怎么回事呢？

问题在于 UNIX 对输出的处理上。当使用 print（或其它函数）写入 STDOUT 等文件时，UNIX 操作系统真正所做的是把数据拷贝到一片特殊的内存即缓冲区中，接下来的输出操作继续写入缓冲区直到写满，当缓冲区满了，就把全部数据实际输出。象这样先写入缓冲区再把整个缓冲区的内容输出比每次都实际输出所花费的时间要少得多，因为一般来说，I/O 比内存操作慢得多。

程序结束时，任何非空的缓冲区都被输出，然而，系统为 STDOUT 和 STDERR 分别维护一片缓冲区，并且先输出 STDERR 的内容，因此存贮在 STDERR 的缓冲区中的内容 line 2 出现在存贮在 STDOUT 的缓冲区中的内容 line 1 之前。

为了解决这个问题，可以告诉 Perl 解释器不对文件使用缓冲，方法为：

1、用 select 函数选择文件

2、把值 1 赋给系统变量$|

系统变量$|指定文件是否进行缓冲而不管其是否应该使用缓冲。如果$|为非零值则不使用缓冲。$|与系统变量$~和$^协同工作，当未调用 select 函数时，$|影响当前缺省文件。下例保证了输出的次序：

```
1 : #!/usr/local/bin/perl
2 :
3 : open (STDOUT, ">file1") || die ("open STDOUT failed");
4 : open (STDERR, ">&STDOUT") || die ("open STDERR failed");
5 : $| = 1;
6 : select (STDERR);
7 : $| = 1;
8 : print STDOUT ("line 1\n");
9 : print STDERR ("line 2\n");
10: close (STDOUT);
11: close (STDERR);
```

程序运行后，文件 file1 中内容为：

```
line 1
line 2
```

第 5 行将$|赋成 1，告诉 Perl 解释器当前缺省文件不进行缓冲，因为未调用 select，当前的缺省文件为重定向到文件 file1 的 STDOUT。第 6 行将当前缺省文件设为 STDERR，第 7 行又设置

$|为 1，关掉了重定向到 file1 的标准错误文件的缓冲。由于 STDOUT 和 STDERR 的缓冲均被关掉，向其的输出立刻被写到文件中，因此 line 1 出现在第一行。

4)指定读写权限

打开一个既可读又可写的文件方法是在文件名前加上"+>"，如下：

```
open (READWRITE, "+>file1");
```

此语句打开既可读又可写的文件 file1，即可以重写其中的内容。文件读写操作最好与库函数 seek 和 tell 一起使用，这样可以跳到文件任何一点。

注：也可用前缀"+<"指定可读写权限。

5)close 函数

用于关闭打开的文件。当用 close 关闭管道，即重定向的命令时，程序等待重定向的命令结束，如：

```
open (MYPIPE, "cat file*|");
close (MYPIPE);
```

当关闭此文件变量时，程序暂停运行，直到命令 cat file*运行完毕。

6)print, printf 和 write 函数

print 是这三个函数中最简单的，它向指定的文件输出，如果未指定，则输出到当前缺省文件中，如：

```
print ("Hello, there!\n");
print OUTFILE ("Hello, there!\n");
```

第一句输出到当前缺省文件中，若未调用 select，则为 STDOUT。第二句输出到由文件变量 OUTFILE 指定的文件中。

printf 函数先格式化字符串再输出到指定文件或当前缺省文件中，如：

```
printf OUTFILE (“You owe me %8.2f", $owing);
```

此语句取出变量$owing 的值并替换掉串中的%8.2f，%8.2f 是域格式的例子，把$owing 的值看作浮点数。

write 函数使用输出格式把信息输出到文件中，如：

```
select (OUTFILE);
$~ = "MYFORMAT";
write;
```

关于 printf 和 write，详见《第 x 章　格式化输出》。

7)select 函数

select 函数将通过参数传递的文件变量指定为新的当前缺省文件，如：

```
select (MYFILE);
```

这样，MYFILE 就成了当前缺省文件，当对 print、write 和 printf 的调用未指定文件时，就输出到 MYFILE 中。

8)eof 函数

eof 函数查看最后一次读文件操作是否为文件最后一个记录，如果是，则返回非零值，如果文件还有内容，返回零。

一般情况下，对 eof 的调用不加括号，因为 eof 和 eof()是等效的，但与<>操作符一起使用时，eof 和 eof()就不同了。现在我们来创建两个文件，分别叫做 file1 和 file2。file1 的内容为：

```
This is a line from the first file.
Here is the last line of the first file.
```

file2 的内容为：

```
This is a line from the second and last file.
Here is the last line of the last file.
```

下面就来看一下 eof 和 eof()的区别，第一个程序为：

```
1: #!/usr/local/bin/perl
2:
3: while ($line = <>) {
4:     print ($line);
5:     if (eof) {
6:        print ("-- end of current file --\n");
7:     }
8: }
```

运行结果如下：

```
$ program file1 file2
This is a line from the first file.
Here is the last line of the first file.
-- end of current file --
```

```
This is a line from the second and last file.
Here is the last line of the last file.
-- end of current file --
$
```

下面把 eof 改为 eof()，第二个程序为：

```
1: #!/usr/local/bin/perl
2:
3: while ($line = <>) {
4:     print ($line);
5:     if (eof()) {
6:         print ("-- end of output --\n");
7:     }
8: }
```

运行结果如下：

```
$ program file1 file2
This is a line from the first file.
Here is the last line of the first file.
This is a line from the second and last file.
Here is the last line of the last file.
-- end of output --$
```

这时，只有所有文件都读过了，eof()才返回真，如果只是多个文件中前几个的末尾，返回值为假，因为还有要读取的输入。

9)间接文件变量

对于上述各函数 open, close, print, printf, write, select 和 eof，都可以用简单变量来代替文件变量，这时，简单变量中所存贮的字符串就被看作文件变量名，下面就是这样一个例子，此例很简单，就不解释了。需要指出的是，函数 open, close, write, select 和 eof 还允许用表达式来替代文件变量，表达式的值必须是字符串，被用作文件变量名。

```
1: #!/usr/local/bin/perl
2:
```

```
3: &open_file("INFILE", "", "file1");
4: &open_file("OUTFILE", ">", "file2");
5: while ($line = &read_from_file("INFILE")) {
6:     &print_to_file("OUTFILE", $line);
7: }
8:
9: sub open_file {
10:     local ($filevar, $filemode, $filename) = @_;
11:
12:     open ($filevar, $filemode . $filename) ||
13:        die ("Can't open $filename");
14: }
15: sub read_from_file {
16:     local ($filevar) = @_;
17:
18:     <$filevar>;
19: }
20: sub print_to_file {
21:     local ($filevar, $line) = @_;
22:
23:     print $filevar ($line);
24: }
```

2、跳过和重读数据

函数名	seek
调用语法	seek (filevar, distance, relative_to);
解说	在文件中向前/后移动，有三个参数： 1、filevar，文件变量 2、distance，移动的字节数，正数向前移动，负数往回移动 3、reletive_to，值可为 0、1 或 2。为 0 时，从文件头开始移动，为 1 时，相对于当前位置（将要读的下一行）移动，为 2 时，相对于文件末尾移动。 运行成功返回真（非零值），失败则返回零，常与 tell 函数

	合用。

函数名	tell
调用语法	tell (filevar);
解说	返回从文件头到当前位置的距离。 注意： 1、seek 和 tell 不能用于指向管道的文件变量。 2、seek 和 tell 中文件变量参数可使用表达式。

3、系统读写函数

函数名	read
调用语法	read (filevar, result, length, skipval);
解说	read 函数设计得与 UNIX 的 fread 函数等效，可以读取任意长度的字符（字节）存入一个简单变量。其参数有四个： 1、filevar：文件变量 2、result：存贮结果的简单变量（或数组元素） 3、length：读取的字节数 4、skipval：可选项，指定读文件之前跳过的字节数。 返回值为实际读取的字节数，如果已到了文件末尾，则返回零，如果出错，则返回空串。

函数名	sysread
调用语法	sysread (filevar, result, length, skipval);
解说	更快的读取数据，与 UNIX 函数 read 等效，参数与 read 相同。

函数名	syswrite
调用语法	syswrite (filevar, data, length, skipval);
解说	更快的写入数据，与 UNIX 函数 write 等效，参数： 1、filevar：将要写入的文件 2、data：存贮要写入数据的变量 3、length：要写入的字节数

	4、skipval 写操作之前跳过的字节数。

4、用 getc 读取字符

函数名	getc
调用语法	$char = getc (infile);
解说	从文件中读取单个字符。

5、用 binmode 读取二进制文件

函数名	binmode
调用语法	binmode (filevar);
解说	当你的系统（如类 DOS 系统）对文本文件和二进制文件有所区别时使用。必须在打开文件后、读取文件前使用。

二、目录处理函数

函数名	mkdir
调用语法	mkdir (dirname, permissions);
解说	创建新目录，参数为： 1、dirname：将要创建的目录名，可以为字符串或表达式 2、permissions：8 进制数，指定目录的访问权限，其值和意义见下表，权限的组合方法为将相应的值相加。

值	权限
4000	运行时设置用户 ID
2000	运行时设置组 ID
1000	粘贴位
0400	拥有者读权限
0200	拥有者写权限
0100	拥有者执行权限
0040	组读权限
0020	组写权限

0010	组执行权限
0004	所有人读权限
0002	所有人写权限
0001	所有人执行权限

函数名	chdir
调用语法	chdir (dirname);
解说	改变当前工作目录。参数 dirname 可以为字符串，也可以为表达式。

函数名	opendir
调用语法	opendir (dirvar, dirname);
解说	打开目录，与下面几个函数合用，可查看某目录中文件列表。参数为： 1、dirvar：目录变量，与文件变量类似 2、dirname：目录名，可为字符串或表达式 成功返回真值，失败返回假。 注：程序中可用同名的目录变量和文件变量，根据环境确定取成分。

函数名	closedir
调用语法	closedir (mydir);
解说	关闭打开的目录。

函数名	readdir
调用语法	readdir (mydir);
解说	赋给简单变量时，每次赋予一个文件或子目录名，对数组则赋予全部文件和子目录名。

函数名	telldir

调用语法	location = telldir (mydir);
解说	象在文件中前后移动一样，telldir 和下面的 seekdir 用于在目录列表中前后移动。

函数名	seekdir
调用语法	seekdir(mydir, location);
解说	location 必须为 telldir 返回的值。

函数名	rewinddir
调用语法	rewinddir (mydir);
解说	将读取目录的位置重置回开头，从而可以重读目录列表。

函数名	rmdir
调用语法	rmdir (dirname);
解说	删除空目录。成功则返回真（非零值），失败返回假（零值）。

三、文件属性函数

1、文件重定位函数

函数名	rename
调用语法	rename (oldname, newname);
解说	改变文件名或移动到另一个目录中，参数可为字符串或表达式。

函数名	unlink
调用语法	num = unlink (filelist);
解说	删除文件。参数为文件名列表，返回值为实际删除的文件数目。

	此函数之所以叫 unlink 而不叫 delete 是因为它实际所做的是删除文件的链接。

2、链接和符号链接函数

函数名	link
调用语法	link (newlink, file);
解说	创建现有文件的链接--硬链接，file 是被链接的文件，newlink 是被创建的链接。 成功返回真，失败返回假。 当删除这两个链接中的一个时，还可以用另一个来访问该文件。

函数名	symlink
调用语法	symlink (newlink, file);
解说	创建现有文件的符号链接，即指向文件名，而不是指向文件本身。参数和返回值同上。 当原文件被删除（如：被 unlinke 函数删除），则被创建链接不可用，除非再创建一个与原被链接的文件同名的文件。

函数名	readlink
调用语法	filename = readlink (linkname);
解说	如果 linkname 为符号链接文件，返回其实际指向的文件。否则返回空串。

3、文件许可权函数

函数名	chmod
调用语法	chmod (permissions, filelist);
解说	改变文件的访问权限。参数为： 1、permissions 为将要设置的权限，其含义见上述 mkdir 中权限表

	2、filelist 为欲改变权限的文件列表

函数名	chown
调用语法	chown (userid, groupid, filelist);
解说	改变文件的属主，有三个参数： 1、userid：新属主的(数字)ID 号 2、groupid：新的组(数字)ID 号，-1 为保留原组 3、filelist：欲改变属主的文件列表

函数名	umask
调用语法	oldmaskval = umask (maskval);
解说	设置文件访问权限掩码，返回值为当前掩码。

4、其它属性函数

函数名	truncate
调用语法	truncate (filename, length);
解说	将文件的长度减少到 length 字节。如果文件长度已经小于 length，则不做任何事。其中 filename 可以为文件名，也可以为文件变量

函数名	stat
调用语法	stat (file);
解说	获取文件状态。参数 file 可为文件名也可为文件变量。返回列表元素依次为： • 文件所在设备 • 内部参考号(inode)

	• 访问权限 • 硬链接数 • 属主的(数字)ID • 所属组的(数字)ID • 设备类型（如果 file 是设备的话） • 文件大小（字节数） • 最后访问时间 • 最后修改时间最后改变状态时间 • I/O 操作最佳块大小 • 分配给该文件的块数

函数名	lstat
调用语法	lstat (file);
解说	与 stat 类似，区别是将 file 看作是符号链接。

函数名	time
调用语法	currtime = time();
解说	返回从 1970 年 1 月 1 日起累计秒数。

函数名	gmtime
调用语法	timelist = gmtime (timeval);
解说	将由 time, stat 或 -A 和 -M 文件测试操作符返回的时间转换成格林威治时间。返回列表元素依次为： • 秒 • 分钟 • 小时，0~23 • 日期 • 月份，0~11(一月~十二月) • 年份 • 星期，0~6(周日~周六) • 一年中的日期，0~364 • 是否夏令时的标志

	详见 UNIX 的 gmtime 帮助。

函数名	localtime
调用语法	timelist = localtime (timeval);
解说	与 gmtime 类似，区别为将时间值转换为本地时间。

函数名	utime
调用语法	utime (acctime, modtime, filelist);
解说	改变文件的最后访问时间和最后更改时间。 例如： $acctime = -A "file1"; $modtime = -M "file1"; @filelist = ("file2", "file3"); utime ($acctime, $modtime, @filelist);

函数名	fileno
调用语法	filedesc = fileno (filevar);
解说	返回文件的内部 UNIX 文件描述。参数 filevar 为文件变量。

函数名	fcntl flock
调用语法	fcntl (filevar, fcntlrtn, value); flock (filevar, flockop);
解说	详见同名 UNIX 函数帮助。

四、使用 DBM 文件

Perl 中可用关联数组来访问 DBM 文件，所用函数为 dbmopen 和 dbmclose，在 Perl5 中，已用 tie 和 untie 代替。

函数名	dbmopen
调用语法	dbmopen (array, dbmfilename, permissions);
解说	将关联数组与 DBM 文件相关联。参数为：

	1、array：所用关联数组 2、dbmfilename：将打开的 DBM 文件名 3、访问权限，详见 mkdir
函数名	dbmclose
调用语法	dbmclose (array);
解说	关闭 DBM 文件，拆除关联数组与之的关系。

第十二章 Perl5 中的引用/指针

一、引用简介

引用就是指针，可以指向变量、数组、哈希表（也叫关联数组）甚至子程序。Pascal 或 C 程序员应该对引用（即指针）的概念很熟悉，引用就是某值的地址，对其的使用则取决于程序员和语言的规定。在 Perl 中，可以把引用称为指针，二者是通用的，无差别的。引用在创建复杂数据方面十分有用。

Perl5 中的两种引用类型为硬引用和符号引用。符号引用含有变量的名字，它对运行时创建变量名并定位很有用，基本上，符号引用就象文件名或 UNIX 系统中的软链接。而硬引用则象文件系统中的硬链接。

Perl4 只允许符号引用，给使用造成一些困难。例如，只允许通过名字对包的符号名哈希表（名为_main{}）建立索引。Perl5

则允许数据的硬引用，方便多了。

硬引用跟踪引用的计数，当其数为零时，Perl 自动将被引用的项目释放，如果该项目是对象，则析构释放到内存池中。Perl 本身就是个面向对象的语言，因为 Perl 中的任何东西都是对象，包和模块使得对象更易于使用。

简单变量的硬引用很简单，对于非简单变量的引用，你必须显式地解除引用并告诉其应如何做，详见《第 章 Perl 中的面向对象编程》。

二、使用引用

本章中，简单变量指像$pointer 这样的变量，$pointer 仅含一个数据项，其可以为数字、字符串或地址。

任何简单变量均可保存硬引用。因为数组和哈希表含有多个简单变量，所以可以建立多种组合而成的复杂的数据结构，如数组的数组、哈希表的数组、子程序的哈希表等等。只要你理解其实只是在用简单变量在工作，就应该可以正确的在最复杂的结构中正确地解除引用。

首先来看一些基本要点。

如果$pointer 的值为一个数组的指针，则通过形式@$pointer 来访问数组中的元素。形式@$pointer 的意义为“取出$pointer 中的地址值当作数组使用”。类似的，%$pointer 为指向哈希表中第一个元素的引用。

有多种构建引用的方法，几乎可以对任何数据建立引用，如数组、简单变量、子程序、文件句柄，以及--C 程序员会感兴趣的--引用。Perl 使你有能力写出把自己都搞糊涂的极其复杂的代码。:)

下面看看 Perl 中创建和使用引用的方法。

三、使用反斜线(\)操作符

反斜线操作符与 C 语言中传递地址的操作符&功能类似。一般是用\创建变量又一个新的引用。下面为创建简单变量的引用的例子：

```
$variavle = 22;
$pointer = \$variable;
$ice = "jello";
$iceprt = \$ice;
```

引用$pointer指向存有$variable值的位置，引用$iceptr指向"jello"。即使最初的引用$variable销毁了，仍然可以通过$pointer访问该值，这是一个硬引用，所以必须同时销毁$pointer和$variable以便该空间释放到内存池中。

在上面的例子中，引用变量$pointer存的是$variable的地址，而不是值本身，要获得值，形式为两个$符号，如下：

```
#!/usr/bin/perl
$value = 10;
$pointer = \$value;
printf "\n Pointer Address $pointer of $value \n";
printf "\n What Pointer *($pointer) points to $$pointer\n";
```

结果输出如下：

```
Pointer Address SCALAR(0x806c520) of 10
What Pointer *(SCALAR(0x806c520)) points to 10
```

每次运行，输出结果中的地址会有所改变，但可以看到$pointer给出地址，而$$pointer给出$variable的值。

看一下地址的显示，SCALAR后面一串十六进制，SCALAR说明该地址指向简单变量（即标量），后面的数字是实际存贮值的地址。

注意：指针就是地址，通过指针可以访问该地址处存贮的数据。如果指针指向了无效的地址，就会得到不正确的数据。通常情况下，Perl会返回NULL值，但不该依赖于此，一定要在程序中把所有的指针正确地初始化，指向有效的数据项。

四、引用和数组

关于Perl语言应该记住的最重要的一点可能是：Perl中的数组和哈希表始终是一维的。因此，数组和哈希表只保存标量值，不直接存贮数组或其它的复杂数据结构。数组的成员要么是数（或字符串）要么是引用。

对数组和哈希表可以象对简单变量一样使用反斜线操作符，数组的引用如下：

```
1  #!/usr/bin/perl
2  #
3  # Using Array references
```

```
4  #
5  $pointer = \@ARGV;
6  printf "\n Pointer Address of ARGV = $pointer\n";
7  $i = scalar(@$pointer);
8  printf "\n Number of arguments : $i \n";
9  $i = 0;
10 foreach (@$pointer) {
11     printf "$i : $$pointer[$i++]; \n";
12 }
```

运行结果如下：

```
$ test 1 2 3 4
Pointer Address of ARGV = ARRAY(0x806c378)
Number of arguments : 4
0 : 1;
1 : 2;
2 : 3;
3 : 4;
```

第 5 行将引用$pointer 指向数组@ARGV，第 6 行输出 ARGV 的地址。$pointer 返回数组第一个元素的地址，这与 C 语言中的数组指针是类似的。第 7 行调用函数 scalar()获得数组的元素个数，该参数亦可为@ARGV，但用指针则必须用@$pointer 的形式指定其类型为数组，$pointer 给出地址，@符号说明传递的地址为数组的第一个元素的地址。第 10 行与第 7 行类似，第 11 行用形式$$pointer[$i]列出所有元素。

对关联数组使用反斜线操作符的方法是一样的--把所有关联数组名换成引用$poniter。注意数组和简单变量（标量）的引用显示时均带有类型--ARRAY 和 SCALAR，哈希表（关联数组）和函数也一样，分别为 HASH 和 CODE。下面是哈希表的引用的例子。

```
#!/usr/bin/perl
1  #
2  # Using Associative Array references
3  #
```

```
4   %month = (
5     '01', 'Jan',
6     '02', 'Feb',
7     '03', 'Mar',
8     '04', 'Apr',
9     '05', 'May',
10    '06', 'Jun',
11    '07', 'Jul',
12    '08', 'Aug',
13    '09', 'Sep',
14    '10', 'Oct',
15    '11', 'Nov',
16    '12', 'Dec',
17    );
18
19 $pointer = \%month;
20
21 printf "\n Address of hash = $pointer\n ";
22
23 #
24 # The following lines would be used to print out the
25 # contents of the associative array if %month was used.
26 #
27 # foreach $i (sort keys %month) {
28 # printf "\n $i $$pointer{$i} ";
29 # }
30
31 #
32 # The reference to the associative array via $pointer
33 #
34 foreach $i (sort keys %$pointer) {
35     printf "$i is $$pointer{$i} \n";
36 }
```

结果输出如下：

```
$ mth
Address of hash = HASH(0x806c52c)
01 is Jan
02 is Feb
03 is Mar
04 is Apr
05 is May
06 is Jun
07 is Jul
08 is Aug
09 is Sep
10 is Oct
11 is Nov
12 is Dec
```

与数组类似，通过引用访问哈希表的元素形式为$$pointer{$index}，当然，$index是哈希表的键值，而不仅是数字。还有几种访问形式，此外，构建哈希表还可以用=>操作符，可读性更好些。下面再看一个例子：

```
1  #!/usr/bin/perl
2  #
3  # Using Array references
4  #
5  %weekday = (
6     '01' => 'Mon',
7     '02' => 'Tue',
8     '03' => 'Wed',
9     '04' => 'Thu',
10    '05' => 'Fri',
11    '06' => 'Sat',
12    '07' => 'Sun',
13    );
14 $pointer = \%weekday;
```

```
15 $i = '05';
16 printf "\n ================== start test
================== \n";
17 #
18 # These next two lines should show an output
19 #
20    printf '$$pointer{$i} is ';
21    printf "$$pointer{$i} \n";
22    printf '${$pointer}{$i} is ';
23    printf "${$pointer}{$i} \n";
24    printf '$pointer->{$i} is ';
25
26    printf "$pointer->{$i}\n";
27 #
28 # These next two lines should not show anything 29 #
30    printf '${$pointer{$i}} is ';
31    printf "${$pointer{$i}} \n";
32    printf '${$pointer->{$i}} is ';
33    printf "${$pointer->{$i}}";
34 printf "\n ================== end of test
================== \n";
35
```

结果输出如下：

```
================== start test ==================
$$pointer{$i} is Fri
${$pointer}{$i} is Fri
$pointer->{$i} is Fri
${$pointer{$i}} is
${$pointer->{$i}} is
================== end of test ==================
```

可以看到，前三种形式的输出显示了预期的结果，而后两种则没有。当你不清楚是否正确时，就输出结果看看。在 Perl 中，有不明确的代码就用 print 语句输出来实验一下，这能使你清楚 Perl 是怎样解释你的代码的。

五、多维数组

语句@array = list;可以创建数组的引用，中括号可以创建匿名数组的引用。下面语句为用于画图的三维数组的例子：

$line = ['solid' , 'black' , ['1','2','3'] , ['4','5','6']];

此语句建立了一个含四个元素的三维数组，变量$line指向该数组。前两个元素是标量，存贮线条的类型和颜色，后两个元素是匿名数组的引用，存贮线条的起点和终点。访问其元素语法如下：

$arrayReference->[$index] single-dimensional array
$arrayReference->[$index1][$index2] two-dimensional array
$arrayReference->[$index1][$index2][$index3] three-dimensional array

可以创建在你的智力、设计经验和计算机的内存允许的情况下极尽复杂的结构，但最好对可能读到或管理你的代码的人友好一些--尽量使代码简单些。另一方面，如果你想向别人炫耀你的编程能力，Perl给你足够的机会和能力编写连自己都难免糊涂的代码。:)

建议：当你想使用多于三维的数组时，最好考虑使用其它数据结构来简化代码。

下面为创建和使用二维数组的例子：

```
1  #!/usr/bin/perl
2  #
3  # Using Multi-dimensional Array references
4  #
5  $line = ['solid', 'black', ['1','2','3'] , ['4', '5', '6']];
6  print "\$line->[0] = $line->[0] \n";
7  print "\$line->[1] = $line->[1] \n";
8  print "\$line->[2][0] = $line->[2][0] \n";
9  print "\$line->[2][1] = $line->[2][1] \n";
10 print "\$line->[2][2] = $line->[2][2] \n";
11 print "\$line->[3][0] = $line->[3][0] \n";
```

```
12 print "\$line->[3][1] = $line->[3][1] \n";
13 print "\$line->[3][2] = $line->[3][2] \n";
14 print "\n"; # The obligatory output beautifier.
```

结果输出如下：

```
$line->[0] = solid
$line->[1] = black
$line->[2][0] = 1
$line->[2][1] = 2
$line->[2][2] = 3
$line->[3][0] = 4
$line->[3][1] = 5
$line->[3][2] = 6
```

那么三维数组又如何呢？下面是上例略为改动的版本。

```
1  #!/usr/bin/perl
2  #
3  # Using Multi-dimensional Array references again
4  #
5  $line = ['solid', 'black', ['1','2','3', ['4', '5', '6']]];
6  print "\$line->[0] = $line->[0] \n";
7  print "\$line->[1] = $line->[1] \n";
8  print "\$line->[2][0] = $line->[2][0] \n";
9  print "\$line->[2][1] = $line->[2][1] \n";
10 print "\$line->[2][2] = $line->[2][2] \n";
11 print "\$line->[2][3][0] = $line->[2][3][0] \n";
12 print "\$line->[2][3][1] = $line->[2][3][1] \n";
13 print "\$line->[2][3][2] = $line->[2][3][2] \n";
14 print "\n";
```

结果输出如下：

```
$line->[0] = solid
$line->[1] = black
$line->[2][0] = 1
```

```
$line->[2][1] = 2
$line->[2][2] = 3
$line->[2][3][0] = 4
$line->[2][3][1] = 5
$line->[2][3][2] = 6
```

访问第三层元素的方式形如$line->[2][3][0]，类似于C语言中的Array_pointer[2][3][0]。本例中，下标均为数字，当然亦可用变量代替。用这种方法可以把数组和哈希表结合起来构成复杂的结构，如下：

```
1 #!/usr/bin/perl
2 #
3 # Using Multi-dimensional Array and Hash references
4 #
5 %cube = (
6 '0', ['0', '0', '0'],
7 '1', ['0', '0', '1'],
8 '2', ['0', '1', '0'],
9 '3', ['0', '1', '1'],
10 '4', ['1', '0', '0'],
11 '5', ['1', '0', '1'],
12 '6', ['1', '1', '0'],
13 '7', ['1', '1', '1']
14 );
15 $pointer = \%cube;
16 print "\n Da Cube \n";
17 foreach $i (sort keys %$pointer) {
18 $list = $$pointer{$i};
19 $x = $list->[0];
20 $y = $list->[1];
21 $z = $list->[2];
22 printf " Point $i = $x,$y,$z \n";
23 }
```

结果输出如下：

```
Da Cube
Point 0 = 0,0,0
Point 1 = 0,0,1
Point 2 = 0,1,0
Point 3 = 0,1,1
Point 4 = 1,0,0
Point 5 = 1,0,1
Point 6 = 1,1,0
Point 7 = 1,1,1
```

这是一个定义立方体的例子。%cube 中保存的是点号和坐标，坐标是个含三个数字的数组。变量$list 获取坐标数组的引用：$list = $$ pointer{$i}; 然后访问各坐标值：$x = $list->[0]; ... 也可用如下方法给$x、$y 和$z 赋值：($x,$y,$z) = @$list;

使用哈希表和数组时，用$和用->是类似的，对数组而言下面两个语句等效：

```
$$names[0] = "kamran";
$names->[0] = "kamran";
```

对哈希表而言下面两个语句等效：

```
$$lastnames{"kamran"} = "Husain";
$lastnames->{"kamran"} = "Husain";
```

Perl 中的数组可以在运行中创建和扩展。当数组的引用第一次在等式左边出现时，该数组自动被创建，简单变量和多维数组也是一样。如下句，如果数组 contours 不存在，则被创建：

```
$contours[$x][$y][$z] = &xlate($mouseX, $mouseY);
```

六、子程序的引用

perl 中子程序的引用与 C 中函数的指针类似，构造方法如下：

```
$pointer_to_sub = sub {... declaration of sub ...};
```

通过所构造的引用调用子程序的方法为：

```
&$pointer_to_sub(parameters);
```

- 子程序模板

子程序的返回值不仅限于数据，还可以返回子程序的引用。返回的子程序在调用处执行，但却是在最初被创建的调用处被设

置，这是由 Perl 对 Closure 处理的方式决定的。Closure 意即如果你定义了一个函数，它就以最初定义的内容运行。(Closure 详见 OOP 的参考书)下面的例子中，设置了多个错误信息显示子程序，这样的子程序定义方法可用于创建模板。

```
            #!/usr/bin/perl
sub errorMsg {
  my $lvl = shift;
  #
  # define the subroutine to run when called.
  #
  return sub {
    my $msg = shift; # Define the error type now.
    print "Err Level $lvl:$msg\n"; }; # print later.
  }
$severe = errorMsg("Severe");
$fatal = errorMsg("Fatal");
$annoy = errorMsg("Annoying");

&$severe("Divide by zero");
&$fatal("Did you forget to use a semi-colon?");
&$annoy("Uninitialized variable in use");
```

结果输出如下：

```
Err Level Severe:Divide by zero
Err Level Fatal:Did you forget to use a semi-colon?
Err Level Annoying:Uninitialized variable in use
```

上例中，子程序 errorMsg 使用了局域变量$lvl，用于返回给调用者。当 errorMsg 被调用时，$lvl 的值设置到返回的子程序内容中，虽然是用的 my 函数。三次调用设置了三个不同的$lvl 变量值。当 errorMsg 返回时，$lvl 的值保存到每次被声明时所产生的子程序代码中。最后三句对产生的子程序引用进行调用时$msg 的值被替换，但$lvl 的值仍是相应子程序代码创建时的值。

很混淆是吗？是的，所以这样的代码在 Perl 程序中很少见。

七、数组与子程序

数组利于管理相关数据，本节讨论如何向子程序传递多个数组。前面我们讲过用@_传递子程序的参数，但是@_是一个单维数组，不管你传递的参数是多少个数组，都按序存贮在@_中，故用形如 my(@a,@b)=@_; 的语句来获取参数值时，全部值都赋给了@a，而@b 为空。那么怎么把一个以上的数组传递给子程序呢？方法是用引用。见下例：

```
#!/usr/bin/perl
@names = (mickey, goofy, daffy );
@phones = (5551234, 5554321, 666 );
$i = 0;
sub listem {
  my ($a,$b) = @_;
  foreach (@$a) {
    print "a[$i] = " . @$a[$i] . " " . "\tb[$i] = " . @$b[$i] ."\n";
    $i++;
  }
}
&listem(\@names, \@phones);
```

结果输出如下：

```
        a[0] = mickey       b[0] = 5551234
        a[1] = goofy        b[1] = 5554321
        a[2] = daffy        b[2] = 666
```

注意：

1、当想传递给子程序的参数是多于一个的数组时一定要使用引用。

2、一定不要在子程序中使用形如 (@variable)=@_; 的语句处理参数，除非你想把所有参数集中到一个长的数组中。

八、文件句柄的引用

有时，必须将同一信息输出到不同的文件，例如，某程序可能在一个实例中输出到屏幕，另一个输出到打印机，再一个输出到记录文件，甚至同时输出到这三个文件。相比较于每种处理写一个单独的语句，可以有更好的实现方式如下：

```
    spitOut(\*STDIN);
    spitOut(\*LPHANDLE);
    spitOut(\*LOGHANDLE);
```

其中子程序 spitOut 的代码如下：

```
            sub spitOut {
  my $fh = shift;
  print $fh "Gee Wilbur, I like this lettuce\n";
}
```

注意其中文件句柄引用的语法为*FILEHANDLE。

第十三章 Perl 的面向对象编程

本章介绍如何使用 Perl 的面向对象编程(OOP)特性及如何构建对象，还包括继承、方法重载和数据封装等内容。

一、模块简介

模块(module)就是 Perl 包(pachage)。Perl 中的对象基于对包中数据项的引用。（引用见第 x 章引用）。

详见 http://www.metronet.com 的 perlmod 和 perlobj。

在用其它语言进行面向对象编程时，先声明一个类然后创建该类的对象（实例），特定类所有对象的行为方式是相同的，由类方法确定，可以通过定义新类或从现存类继承来创建类。已熟悉面向对象编程的人可以在此遇到许多熟悉的术语。Perl 一直是一个面向对象的语言，在 Perl5 中，语法略有变动，更规范化了对象的使用。

下面三个定义对理解对象、类和方法在 Perl 中如何工作至关重要。

.类是一个 Perl 包，其中含提供对象方法的类。

.方法是一个 Perl 子程序，类名是其第一个参数。

.对象是对类中数据项的引用。

二、Perl 中的类

再强调一下，一个 Perl 类是仅是一个包而已。当你看到 Perl 文档中提到“类”时，把它看作“包”就行了。Perl5 的语法可以创建类，如果你已熟悉 C++，那么大部分语法你已经掌握了。与 Perl4 不同的概念是用双冒号(::)来标识基本类和继承类（子类）。

面向对象的一个重要特性是继承。Perl 中的继承特性与其它面向对象语言不完全一样，它只继承方法，你必须用自己的机制来实现数据的继承。

因为每个类是一个包，所以它有自己的名字空间及自己的符号名关联数组（详见第 x 章关联数组），每个类因而可以使用自己的独立符号名集。与包的引用结合，可以用单引号(')操作符来定位类中的变量，类中成员的定位形式如：$class'$member。在 Perl5 中，可用双冒号替代单引号来获得引用，如：$class'$member 与 $class::$member 相同。

三、创建类。

本节介绍创建一个新类的必要步骤。下面使用的例子是创建一个称为 Cocoa 的简单的类，其功能是输出一个简单的 Java 应用的源码的必要部分。放心，这个例子不需要你有 Java 的知识，但也不会使你成为 Java 专家，其目的是讲述创建类的概念。

首先，创建一个名为 Cocoa.pm 的包文件(扩展名 pm 是包的缺省扩展名，意为 Perl Module)。一个模块就是一个包，一个包就是一个类。在做其它事之前，先加入“1;”这样一行，当你增加其

它行时，记住保留“1;”为最后一行。这是 Perl 包的必需条件，否则该包就不会被 Perl 处理。下面是该文件的基本结构。

```
package Cocoa;

#
# Put "require" statements in for all required,imported packages
#

#
# Just add code here
#

1; # terminate the package with the required 1;
```

接下来，我们往包里添加方法使之成为一个类。第一个需添加的方法是 new()，它是创建对象时必须被调用的，new()方法是对象的构造函数。

四、构造函数

构造函数是类的子程序，它返回与类名相关的一个引用。将类名与引用相结合称为“祝福”一个对象，因为建立该结合的函数名为 bless()，其语法为：

bless YeReference [,classname]

YeReference 是对被“祝福”的对象的引用，classname 是可选项，指定对象获取方法的包名，其缺省值为当前包名。

创建一个构建函数的方法为返回已与该类结合的内部结构的引用，如：

```
sub new {
  my $this = {}; # Create an anonymous hash, and #self points to it.
  bless $this; # Connect the hash to the package Cocoa.
  return $this; # Return the reference to the hash.
}
```

```
1;
```

{}创建一个对不含键/值对的哈希表（即关联数组）的引用，返回值被赋给局域变量$this。函数 bless()取出该引用，告诉对象它引用的是 Cocoa，最后返回该引用。函数的返回值现在指向这个匿名哈希表。

从 new()函数返回后，$this 引用被销毁，但调用函数保存了对该哈希表的引用，因此该哈希表的引用数不会为零，从而使 Perl 在内存中保存该哈希表。创建对象可如下调用：

```
$cup = new Cocoa;
```

下面语句为使用该包创建对象的例子：

```
1 #!/usr/bin/perl
2 push (@INC,'pwd');
3 use Cocoa;
4 $cup = new Cocoa;
```

第一行指出 Perl 解释器的位置，第二行中，将当前目录加到路径寻找列表@INC 中供寻找包时使用。你也可以在不同的目录中创建你的模块并指出该绝对路径。例如，如果在 /home/test/scripts/创建包，第二行就应该如下：

```
push (@INC , "/home/test/scripts");
```

在第三行中，包含上包 Cocoa.pm 以获取脚本中所需功能。use 语句告诉 Perl 在@INC 路径寻找文件 Cocoa.pm 并包含到解析的源文件拷贝中。use 语句是使用类必须的。第四行调用 new 函数创建对象，这是 Perl 的妙处，也是其易混淆之处，也是其强大之处。创建对象的方法有多种，可以这样写：

```
$cup = cocoa->new();
```

如果你是 C 程序员，可以用双冒号强制使用 Cocoa 包中的 new()函数，如：

```
$cup = Cocoa::new();
```

可以在构造函数中加入更多的代码，如在 Cocoa.pm 中，可以在每个对象创建时输出一个简单声明，还可以用构造函数初始化

变量或设置数组或指针。
注意：

1、一定要在构造函数中初始化变量；
2、一定要用 my 函数在方法中创建变量；
3、一定不要在方法中使用 local，除非真的想把变量传递给其它子程序；
4、一定不要在类模块中使用全局变量。

加上声明的 Cocoa 构造函数如下：

```
sub new {
  my $this = {};
  print "\n /* \n ** Created by Cocoa.pm \n ** Use at own risk";
  print "\n ** Did this code even get pass the javac compiler? ";
  print "\n **/ \n";
  bless $this;
  return $this;
}
```

也可以简单地调用包内或包外的其它函数来做更多的初始化工作，如：

```
sub new {
  my $this = {}
  bless $this;
  $this->doInitialization();
  return $this;
}
```

创建类时，应该允许它可被继承，应该可以把类名作为第一个参数来调用 new 函数，那么 new 函数就象下面的语句：

```
sub new {
  my $class = shift; # Get the request class name
  my $this = {};
```

```
  bless $this, $class # Use class name to bless() reference
  $this->doInitialization(); return $this;
}
```

此方法使用户可以下列三种方式之一来进行调用：

- Cocoa::new()
- Cocoa->new()
- new Cocoa

可以多次 bless 一个引用对象，然而，新的将被 bless 的类必然把对象已被 bless 的引用去掉，对 C 和 Pascal 程序员来说，这就象把一个指针赋给分配的一块内存，再把同一指针赋给另一块内存而不释放掉前一块内存。总之，一个 Perl 对象每一时刻只能属于一个类。

对象和引用的真正区别是什么呢？Perl 对象被 bless 以属于某类，引用则不然，如果引用被 bless，它将属于一个类，也便成了对象。对象知道自己属于哪个类，引用则不属于任何类。

- 实例变量

作为构造函数的 new()函数的参数叫做实例变量。实例变量在创建对象的每个实例时用于初始化，例如可以用 new()函数为对象的每个实例起个名字。

可以用匿名哈希表或匿名数组来保存实例变量。

用哈希表的代码如下：

```
sub new {
my $type = shift;
my %parm = @_;
my $this = {};
$this->{'Name'} = $parm{'Name'};
$this->{'x'} = $parm{'x'};
$this->{'y'} = $parm{'y'};
bless $this, $type;
            }
```

用数组保存的代码如下：

```
sub new {
my $type = shift;
my %parm = @_;
my $this = [];
$this->[0] = $parm{'Name'};
$this->[1] = $parm{'x'};
$this->[2] = $parm{'y'};
bless $this, $type;
            }
```

构造对象时，可以如下传递参数：

```
$mug = Cocoa::new( 'Name' => 'top','x' => 10,'y' => 20 );
```

操作符=>与逗号操作服功能相同，但=>可读性好。访问方法如下：

```
print "Name=$mug->{'Name'}\n";
print "x=$mug->{'x'}\n";
print "y=$mug->{'y'}\n";
```

五、方法

Perl 类的方法只不过是一个 Perl 子程序而已，也即通常所说的成员函数。Perl 的方法定义不提供任何特殊语法，但规定方法的第一个参数为对象或其被引用的包。Perl 有两种方法：静态方法和虚方法。

静态方法第一个参数为类名，虚方法第一个参数为对象的引用。方法处理第一个参数的方式决定了它是静态的还是虚的。静态方法一般忽略掉第一个参数，因为它们已经知道自己在哪个类了，构造函数即静态方法。虚方法通常首先把第一个参数 shift 到变量 self 或 this 中，然后将该值作普通的引用使用。如：

```
            1. sub nameLister {
2.      my $this = shift;
3.      my ($keys ,$value );
4.      while (($key, $value) = each (%$this)) {
5.          print "\t$key is $value.\n";
6.      }
7. }
```

六、方法的输出

如果你现在想引用 Cocoa.pm 包，将会得到编译错误说未找到方法，这是因为 Cocoa.pm 的方法还没有输出。输出方法需要 Exporter 模块，在包的开始部分加上下列两行：

```
require Exporter;
@ISA = qw (Exporter);
```

这两行包含上 Exporter.pm 模块，并把 Exporter 类名加入@ISA 数组以供查找。接下来把你自己的类方法列在@EXPORT 数组中就可以了。例如想输出方法 closeMain 和 declareMain，语句如下：

```
@EXPORT = qw (declareMain , closeMain);
```

Perl 类的继承是通过@ISA 数组实现的。@ISA 数组不需要在任何包中定义，然而，一旦它被定义，Perl 就把它看作目录名的特殊数组。它与@INC 数组类似，@INC 是包含文件的寻找路径。@ISA 数组含有类(包)名，当一个方法在当前包中未找到时就到@ISA 中的包去寻找。@ISA 中还含有当前类继承的基类名。

类中调用的所有方法必须属于同一个类或@ISA 数组定义的基类。如果一个方法在@ISA 数组中未找到，Perl 就到 AUTOLOAD()子程序中寻找，这个可选的子程序在当前包中用 sub 定义。若使用 AUTOLOAD 子程序，必须用 use Autoload;语句调用 autoload.pm 包。AUTOLOAD 子程序尝试从已安装的 Perl 库中装载调用的方法。如果 AUTOLOAD 也失败了，Perl 再到 UNIVERSAL 类做最后一次尝试，如果仍失败，Perl 就生成关于该无法解析函数的错误。

七、方法的调用

调用一个对象的方法有两种方法，一是通过该对象的引用（虚方法），一是直接使用类名（静态方法）。当然该方法必须已被输出。现在给 Cocoa 类增加一些方法，代码如下：

```
package Cocoa;
require Exporter;
@ISA = qw(Exporter);
@EXPORT = qw(setImports, declareMain, closeMain);
#
# This routine creates the references for imports in Java functions
```

```
#
sub setImports{
  my $class = shift @_;
  my @names = @_;
  foreach (@names) {
    print "import " . $_ . ";\n";
  }
}
#
# This routine declares the main function in a Java script
#
sub declareMain{
  my $class = shift @_;
  my ( $name, $extends, $implements) = @_;
  print "\n public class $name";
  if ($extends) {
    print " extends " . $extends;
  }
  if ($implements) {
    print " implements " . $implements;
  }
  print " { \n";
}
#
# This routine declares the main function in a Java script
#
sub closeMain{
  print "} \n";
}
#
# This subroutine creates the header for the file.
#
sub new {
  my $this = {};
```

```
  print "\n /* \n ** Created by Cocoa.pm \n ** Use at own risk \n
*/ \n";
  bless $this;
  return $this;
}

1;
```

现在，我们写一个简单的 Perl 脚本来使用该类的方法，下面是创建一个 Java applet 源代码骨架的脚本代码：

```
#!/usr/bin/perl
use Cocoa;
$cup = new Cocoa;
$cup->setImports( 'java.io.InputStream', 'java.net.*');
$cup->declareMain( "Msg" , "java.applet.Applet", "Runnable");
$cup->closeMain();
```

这段脚本创建了一个叫做 Msg 的 Java applet，它扩展(extend)了 java.applet.Applet 小应用程序并使之可运行(runnable)，其中最后三行也可以写成如下：

```
Cocoa::setImports($cup, 'java.io.InputStream', 'java.net.*');
Cocoa::declareMain($cup, "Msg" , "java.applet.Applet",
"Runnable");
Cocoa::closeMain($cup);
```

其运行结果如下：

```
/*
** Created by Cocoa.pm
** Use at own risk
*/
import java.io.InputStream;
import java.net.*;
```

public class Msg extends java.applet.Applet implements Runnable
{
}

注意：如果用->操作符调用方法（也叫间接调用），参数必须用括号括起来，如：$cup->setImports('java.io.InputStream', 'java.net.*');而双冒号调用如：Cocoa::setImports($cup, 'java.io.InputStream', 'java.net.*');也可去掉括号写成：Cocoa::setImports $cup, 'java.io.InputStream', 'java.net.*' ;

八、重载

有时需要指定使用哪个类的方法，如两个不同的类有同名方法的时候。假设类 Espresso 和 Qava 都定义了方法 grind,可以用::操作符指定使用 Qava 的方法：

$mess = Qava::grind("whole","lotta","bags");

Qava::grind($mess, "whole","lotta","bags");

可以根据程序的运行情况来选择使用哪个类的方法，这可以通过使用符号引用去调用来实现：

$method = $local ? "Qava::" : "Espresso::";

$cup->{$method}grind(@args);

九、析构函数

Perl 跟踪对象的链接数目,当某对象的最后一个应用释放到内存池时，该对象就自动销毁。对象的析构发生在代码停止后，脚本将要结束时。对于全局变量而言，析构发生在最后一行代码运行之后。

如果你想在对象被释放之前获取控制权,可以定义 DESTROY()方法。DESTROY()在对象将释放前被调用，使你可以做一些清理工作。DESTROY()函数不自动调用其它 DESTROY()函数，Perl 不做内置的析构工作。如果构造函数从基类多次 bless，DESTROY()可能需要调用其它类的 DESTROY()函数。当一个对象被释放时，其内含的所有对象引用自动释放、销毁。

一般来说，不需要定义 DESTROY()函数，如果需要，其形式如下：

sub DESTROY {

#

```
# Add code here.
#
}
```

因为多种目的，Perl 使用了简单的、基于引用的垃圾回收系统。任何对象的引用数目必须大于零，否则该对象的内存就被释放。当程序退出时，Perl 的一个彻底的查找并销毁函数进行垃圾回收，进程中的一切被简单地删除。在 UNIX 类的系统中，这像是多余的，但在内嵌式系统或多线程环境中这确实很必要。

十、继承

类方法通过@ISA 数组继承，变量的继承必须明确设定。下例创建两个类 Bean.pm 和 Coffee.pm,其中 Coffee.pm 继承 Bean.pm 的一些功能。此例演示如何从基类（或称超类）继承实例变量，其方法为调用基类的构造函数并把自己的实例变量加到新对象中。

Bean.pm 代码如下

```
package Bean;
require Exporter;
@ISA = qw(Exporter);
@EXPORT = qw(setBeanType);

sub new {
  my $type = shift;
  my $this = {};
  $this->{'Bean'} = 'Colombian';
  bless $this, $type;
  return $this;
}

#
# This subroutine sets the class name
sub setBeanType{
  my ($class, $name) = @_;
  $class->{'Bean'} = $name;
```

```
  print "Set bean to $name \n";
}
1;
```

此类中，用$this 变量设置一个匿名哈希表，将'Bean'类型设为'Colombian'。方法 setBeanType()用于改变'Bean'类型，它使用$class 引用获得对对象哈希表的访问。

Coffee.pm 代码如下：

```
1  #
2  # The Coffee.pm file to illustrate inheritance.
3  #
4  package Coffee;
5  require Exporter;
6  require Bean;
7  @ISA = qw(Exporter, Bean);
8  @EXPORT = qw(setImports, declareMain, closeMain);
9  #
10 # set item
11 #
12 sub setCoffeeType{
13     my ($class,$name) = @_;
14     $class->{'Coffee'} = $name;
15     print "Set coffee type to $name \n";
16     }
17 #
18 # constructor
19 #
20 sub new {
21     my $type = shift;
22     my $this = Bean->new(); ##### <- LOOK HERE!!! ####
23     $this->{'Coffee'} = 'Instant'; # unless told otherwise
24     bless $this, $type;
25     return $this;
```

```
26     }
27 1;
```

第 6 行的 require Bean;语句包含了 Bean.pm 文件和所有相关函数，方法 setCoffeeType()用于设置局域变量$class->{'Coffee'}的值。在构造函数 new()中，$this 指向 Bean.pm 返回的匿名哈希表的指针，而不是在本地创建一个，下面两个语句分别为创建不同的哈希表从而与 Bean.pm 构造函数创建的哈希表无关的情况和继承的情况：

```
my $this = {}; #非继承
my $this = $theSuperClass->new(); #继承
```

下面代码演示如何调用继承的方法：

```
1   #!/usr/bin/perl
2   push (@INC,'pwd');
3   use Coffee;
4   $cup = new Coffee;
5   print "\n ------------------- Initial values ------------ \n";
6   print "Coffee: $cup->{'Coffee'} \n";
7   print "Bean: $cup->{'Bean'} \n";
8   print "\n ------------------- Change Bean Type ---------- \n";
9   $cup->setBeanType('Mixed');
10 print "Bean Type is now $cup->{'Bean'} \n";
11 print "\n ------------------ Change Coffee Type ---------- \n";
12 $cup->setCoffeeType('Instant');
13 print "Type of coffee: $cup->{'Coffee'} \n";
```

该代码的结果输出如下：

```
------------------- Initial values ------------
Coffee: Instant
Bean: Colombian
------------------- Change Bean Type ----------
Set bean to Mixed
Bean Type is now Mixed
```

```
----------------- Change Coffee Type ----------
Set coffee type to Instant
Type of coffee: Instant
```

上述代码中，先输出对象创建时哈希表中索引为'Bean'和'Coffee'的值，然后调用各成员函数改变值后再输出。

方法可以有多个参数，现在向 Coffee.pm 模块增加函数 makeCup()，代码如下

```
sub makeCup {
  my ($class, $cream, $sugar, $dope) = @_;
  print "\n================================== \n";
  print "Making a cup \n";
  print "Add cream \n" if ($cream);
  print "Add $sugar sugar cubes\n" if ($sugar);
  print "Making some really addictive coffee ;-) \n" if ($dope);
  print "================================== \n";
}
```

此函数可有三个参数，不同数目、值的参数产生不同的结果，例如：

```
1  #!/usr/bin/perl
2  push (@INC,'pwd');
3  use Coffee;
4  $cup = new Coffee;
5  #
6  # With no parameters
7  #
8  print "\n Calling with no parameters: \n";
9  $cup->makeCup;
10 #
11 # With one parameter
12 #
13 print "\n Calling with one parameter: \n";
14 $cup->makeCup('1');
```

```
15 #
16 # With two parameters
17 #
18 print "\n Calling with two parameters: \n";
19 $cup->makeCup(1,'2');
20 #
21 # With all three parameters
22 #
23 print "\n Calling with three parameters: \n";
24 $cup->makeCup('1',3,'1');
```

其结果输出如下：

```
Calling with no parameters:
===================================
Making a cup
===================================
Calling with one parameter:
===================================
Making a cup
Add cream
===================================
Calling with two parameters:
===================================
Making a cup
Add cream
Add 2 sugar cubes
===================================
Calling with three parameters:
===================================
Making a cup
Add cream
Add 3 sugar cubes
```

```
Making some really addictive coffee ;-)
==================================
```

在此例中，函数 makeCup()的参数既可为字符串也可为整数，处理结果相同，你也可以把这两种类型的数据处理区分开。在对参数的处理中，可以设置缺省的值，也可以根据实际输入参数值的个数给予不同处理。

十一、子类方法的重载

继承的好处在于可以获得基类输出的方法的功能，而有时需要对基类的方法重载以获得更具体或不同的功能。下面在 Bean.pm 类中加入方法 printType()，代码如下：

```
        sub printType {
  my $class = shift @_;
  print "The type of Bean is $class->{'Bean'} \n";
}
```

然后更新其@EXPORT 数组来输出：

```
 @EXPORT = qw ( setBeanType , printType );
```

现在来调用函数 printType()，有三种调用方法：

```
        $cup->Coffee::printType();
$cup->printType();
$cup->Bean::printType();
```

输出分别如下：

```
        The type of Bean is Mixed
The type of Bean is Mixed
The type of Bean is Mixed
```

为什么都一样呢？因为在子类中没有定义函数 printType()，所以实际均调用了基类中的方法。如果想使子类有其自己的 printType()函数，必须在 Coffee.pm 类中加以定义：

```
        #
# This routine prints the type of $class->{'Coffee'}
#
sub printType {
  my $class = shift @_;
  print "The type of Coffee is $class->{'Coffee'} \n";
}
```

然后更新其@EXPORT 数组：

@EXPORT = qw(setImports, declareMain, closeMain, printType);

现在输出结果变成了：

The type of Coffee is Instant
The type of Coffee is Instant
The type of Bean is Mixed

现在只有当给定了 Bean::时才调用基类的方法，否则直接调用子类的方法。

那么如果不知道基类名该如何调用基类方法呢？方法是使用伪类保留字 SUPER::。在类方法内使用语法如：$this->SUPER::function(...argument list...); ，它将从@ISA 列表中寻找。刚才的语句用 SUPER::替换 Bean::可以写为 $cup->SUPER::printType(); ，其结果输出相同，为：

The type of Bean is Mixed

十二、Perl 类和对象的一些注释

OOP 的最大好处就是代码重用。OOP 用数据封装来隐藏一些复杂的代码，Perl 的包和模块通过 my 函数提供数据封装功能，但是 Perl 并不保证子类一定不会直接访问基类的变量，这确实减少了数据封装的好处，虽然这种动作是可以做到的，但却是个很坏的编程风格。

注意：

1、一定要通过方法来访问类变量。

2、一定不要从模块外部直接访问类变量。

当编写包时，应该保证方法所需的条件已具备或通过参数传递给它。在包内部，应保证对全局变量的访问只用通过方法传递的引用来访问。对于方法要使用的静态或全局数据，应该在基类中用 local()来定义，子类通过调用基类来获取。有时，子类可能需要改变这种数据，这时，基类可能就不知道怎样去寻找新的数据，因此，这时最好定义对该数据的引用，子类和基类都通过引用来改变该数据。

最后，你将看到如下方式来使用对象和类：

use coffee::Bean;

这句语句的含义是“在@INC 数组所有目录的 Coffee 子目录来寻找 Bean.pm”。如果把 Bean.pm 移到./Coffee 目录，上面的例子将用这一 use 语句来工作。这样的好处是有条理地组织类的代码。再如，下面的语句：

use Another::Sub::Menu;

意味着如下子目录树：

./Another/Sub/Menu.pm

第十四章 Perl5 的包和模块

一、require 函数

用 require 函数可以把程序分割成多个文件并创建函数库。例如，在 myfile.pl 中有定义好的 Perl 函数，可用语句 require ("myfile.pl"); 在程序中包含进来。当 Perl 解释器看到这一语句，就在内置数组变量@INC 指定的目录中寻找文件 myfile.pl。如果找到了，该文件中的语句就被执行，否则程序终止并输出错误信

息：

Can't find myfile.pl in @INC

作为子程序调用参数，文件中最后一个表达式的值成为返回值，require 函数查看其是否为零，若为零则终止。例如 myfile.pl 最后的语句是：

print ("hello, world!\n");

$var = 0;

因为最后的语句值为零，Perl 解释器输出下列错误信息并推出：

myfile.pl did not reture true value

可以用简单变量或数组元素等向 require 传递参数，如：

```
@reqlist = ("file1.pl", "file2.pl", "file3.pl");
require ($reqlist[$0]);
require ($reqlist[$1]);
require ($reqlist[$2]);
```

还可以不指定文件名，即：

require;

这时，变量$_的值即作为文件名传递给 require。

注：如果@INC 中有多个目录中含有同一个文件，则只有第一个被包含。

1、require 函数和子程序库

用 require 函数可以创建可用于所有 Perl 程序的子程序库，步骤如下：

a、确定存贮子程序库的目录

b、将子程序抽取放到单独的文件中，将文件放到子程序库目录

c、每个文件末尾加一句非零值的语句，最简单的办法是语句 1;

d、在主程序中用 require 包含一个或多个所需的文件。

e、运行主程序时，用 -I 选项指定子程序库目录，或者在调用 require 前将该目录添加到@INC 数组中。

例如：假设目录/u/perldir 中存有你的 Perl 子程序库，子程序 mysub 存贮在文件 mysub.pl 中。现在来包含上该文件：

unshift (@INC, "/u/perldir");

require ("mysub.pl");

对 unshift 的调用把目录/u/perldir 添加到@INC 数组，对 require 的调用将 mysub.pl 文件的内容包含进来作为程序的一部分。

注意：

1、应该使用 unshift 来向@INC 中添加目录，而不是 push。因为 push 增加到@INC 的末尾，则该目录将被最后搜寻。

2、如果你的库文件名与/usr/local/lib/perl 中的某文件同名，则不会被包含进来，因为 require 只包含同名文件中的第一个。

2、用 require 指定 Perl 版本

Perl 5 中，可以用 require 语句来指定程序运行所需的 Perl 版本。当 Perl 解释器看到 require 后跟着数字时，则只有其版本高于或等于该数字时才运行该程序。例如，下面语句表明只有 Perl 解释器为 5.001 版或更高时才运行该程序：

require 5.001;

二、包

Perl 程序把变量和子程序的名称存贮到符号表中，perl 的符号表中名字的集合就称为包(package)。

1、包的定义

在一个程序中可以定义多个包，每个包有一个单独的符号表，定义语法为：

package mypack;

此语句定义一个名为 mypack 的包，从此以后定义的所有变量和子程序的名字都存贮在该包关联的符号表中，直到遇到另一个 package 语句为止。

每个符号表有其自己的一组变量、子程序名，各组名字是不相关的，因此可以在不同的包中使用相同的变量名，而代表的是不同的变量。如：

```
$var = 14;
package mypack;
$var = 6;
```

第一个语句创建变量$var并存贮在main符号表中，第三个语句创建另一个同名变量$var并存贮在mypack包的符号表中。

2、在包间切换

在程序里可以随时在包间来回切换，如：

```
1: #!/usr/local/bin/perl
2:
3: package pack1;
4: $var = 26;
5: package pack2;
6: $var = 34;
7: package pack1;
8: print ("$var\n");
```

运行结果如下：

```
$ program
26
$
```

第三行定义了包pack1，第四行创建变量$var，存贮在包pack1的符号表中，第五行定义新包pack2，第六行创建另一个变量$var，存贮在包pack2的符号表中。这样就有两个独立的$var，分别存贮在不同的包中。第七行又指定pack1为当前包，因为包pack1已经定义，这样，所有变量和子程序的定义和调用都为该包的符号表中存贮的名字。因此第八行对$var的调用为pack1包中的$var，其值为26。

3、main包

存贮变量和子程序的名字的缺省符号表是与名为main的包相关联的。如果在程序里定义了其它的包，当你想切换回去使用缺省的符号表，可以重新指定main包：

```
package main;
```

这样，接下来的程序就好象从没定义过包一样，变量和子程序的名字象通常那样存贮。

4、包的引用

在一个包中可以引用其它包中的变量或子程序，方法是在变量名前面加上包名和一个单引号，如：

```
package mypack;
$var = 26;
package main;
print ("$mypack'var\n");
```

这里，$mypack'var 为 mypack 包中的变量$var。

注意：在 Perl 5 中，包名和变量名用双冒号隔开，即 $mypack::var。单引号引用的方式仍然支持，但将来的版本中未必支持。

5、指定无当前包

在 Perl 5 中，可以用如下语句指定无当前包：

```
package;
```

这时，所有的变量必须明确指出所属包名，否则就无效--错误。

```
$mypack::var = 21; #ok
$var = 21;     #error - no current package
```

这种情况直到用 package 语句指定当前包为止。

6、包和子程序

包的定义影响到程序中的所有语句，包括子程序，如：

```
package mypack;
subroutine mysub {
  local ($myvar);
  # stuff goes here
}
```

这里，mysub 和 myvar 都是包 mypack 的一部分。在包 mypack 外调用子程序 mysub，则要指定包：$mypack'mysub。

可以在子程序中切换包：

```
package pack1;
subroutine mysub {
  $var1 = 1;
  package pack2;
  $var1 = 2;
}
```

这段代码创建了两个变量$var1，一个在包 pack1 中，一个在包 pack2 中，包中的局域变量只能在其定义的子程序等语句块中使用，像普通的局域变量一样。

7、用包定义私有数据

包最通常的用途是用在含有子程序和子程序所使用的全局变量的文件中，为子程序定义这样的包，可以保证子程序使用的全局变量不可在其它地方使用，这样的数据即为私有数据。更进一步，可以保证包名不可在其它地方使用。私有数据例：

```
1 : package privpack;
2 : $valtoprint = 46;
3 :
4 : package main;
5 : # This function is the link to the outside world.
6 : sub printval {
7 :     &privpack'printval();
8 : }
9 :
10: package privpack;
11: sub printval {
12:     print ("$valtoprint\n");
13: }
14:
15: package main;
16: 1; # return value for require
```

此子程序只有在调用 printval 后才能产生输出。

该文件分为两个部分：与外界联系的部分和私有部分。前者为缺省的 main 包，后者为包 privpack。第 6~8 行定义的子程序 printval 可被其它程序或子程序调用。printval 输出变量 $valtoprint 的值，此变量仅在包 privpack 中定义和使用。第 15、16 行确保其被其它程序用 require 语句包含后工作正常，15 行将当前包设置回缺省包 main，16 行返回非零值使 require 不报错。

8、包和系统变量

下列变量即使从其它包中调用，也在 main 包中起作用：

- 文件变量 STDIN, STDOUT, STDERR 和 ARGV
- 变量%ENV, %INC, @INC, $ARGV 和 @ARGV
- 其它含有特殊字符的系统变量

9、访问符号表

在程序中查找符号表可用数组%_package，此处package为想访问的符号表所属的包名。例如%_main含有缺省的符号表。

通常不需要亲自查找符号表。

三、模块

多数大型程序都分割成多个部件，每一部件通常含有一个或多个子程序及相关的变量，执行特定的一个或多个任务。集合了变量和子程序的部件称为程序模块。

1、创建模块

Perl 5中用包来创建模块，方法是创建包并将之存在同名的文件中。例如，名为Mymodult的包存贮在文件Mymodult.pm中（扩展名.pm表示Perl Module）。下例的模块Mymodult含有子程序myfunc1和myfunc2及变量$myvar1和$myvar2。

```
1：#!/usr/local/bin/perl
2：
3：package Mymodule;
4：require Exporter;
5：@ISA = qw(Exporter);
6：@EXPORT = qw(myfunc1 myfunc2);
7：@EXPORT_OK = qw($myvar1 $myvar2);
8：
9：sub myfunc1 {
10:    $myvar1 += 1;
11: }
12:
13: sub myfunc2 {
14:    $myvar2 += 2;
15: }
```

第3~7行是标准的Perl模块定义方式。第3行定义包，第4行包含内置Perl模块Exporter，6、7行进行子程序和变量的输出以与外界联系。第6行创建名为@EXPORT的特殊数组，该数组中的子程序可以被其它程序调用，这里，myfunc1和myfunc2可以被访问。其它任何在模块中定义但没有赋给数组

@EXPORT 的子程序都是私有的，只能在模块内部调用。第 7 行创建另一个名为@EXPORT_OK 的特殊数组，其中含有可被外部程序访问的变量，这里含有$myvar1 和$myvar2。

2、导入模块

将模块导入你的 Perl 程序中使用 use 语句，如下句导入了 Mymodule 模块：

use Mymodule;

这样，模块 Mymodule 中的子程序和变量就可以使用了。

取消导入模块使用 no 语句，如下句取消了 Mymodule 模块的导入：

no Mymodule;

下面看一个导入模块和取消导入的例子，使用 integer 模块要求所有数字运算基于整数，浮点数在运算前均被转化为整数。

```
1: #!/usr/local/bin/perl
2:
3: use integer;
4: $result = 2.4 + 2.4;
5: print ("$result\n");
6:
7: no integer;
8: $result = 2.4 + 2.4;
9: print ("$result\n");
```

程序输出如下：

```
$ program
4
4.8
$
```

如果 use 或 no 语句出现在语句块中，则只在该块的有效范围内起作用，如：

```
use integer;
$result1 = 2.4 + 2.4;
if ($result1 == 4) {
no integer;
$result2 = 3.4 + 3.4;
```

```
}
$result3 = 4.4 + 4.4;
```

结果输出如下：

```
        4
6.8
8
```

这里，no 语句只在 if 语句中有效，出了 if 语句仍使用 integer 模块，因此 4.4 在做加法前被转化成了 4。

3、预定义模块

Perl 5 提供了许多有用的预定义模块，可以用 use 导入和 no 语句取消。下面是库中最有用的一些模块：

integer	使用整数运算
Diagnostics	输出较多的诊断信息（警告）
English	允许英文名用作系统变量的别名
Env	导入环境变量的 Perl 模块
POSIX	POSIX 标准（IEEE 1003.1）的 Perl 接口
Socket	装载 C 语言的套接字处理机制

Perl 文档中有完整的预定义模块列表。

注：世界各地的 Perl 5 用户写了许多有用的模块，CPAN(Comprehensive Perl Archive Network)的 Perl 文档有其完整的列表。关于 CPAN 的更多信息见其网址：http://www.perl.com/perl/CPAN/README.html。

附录一 函数集(未定稿)

一、进程处理函数

1、进程启动函数

2、进程终止函数

一、进程处理函数

1、进程启动函数

函数名	eval
调用语法	eval(string)
解说	将 string 看作 Perl 语句执行。 正确执行后，系统变量$@为空串，如果有错误，$@中为错误信息。
例子	$print = "print (\"hello,world\\n\");"; eval ($print);
结果输出	hello, world

函数名	system
调用语法	system(list)
解说	list 中第一个元素为程序名，其余为参数。 system 启动一个进程运行程序并等待其结束，程序结束后错误代码左移八位成为返回值。
例子	@proglist = ("echo", "hello,world!"); system(@proglist);
结果输出	hello, world!

函数名	fork
调用语法	procid = fork();
解说	创建程序的两个拷贝--父进程和子进程--同时运行。子进程返回零，父进程返回非零值，此值为子程序的进程 ID 号。

例子	$retval = fork();
	if ($retval == 0) {
	  # this is the child process
	  exit; # this terminates the child process
	} else {
	  # this is the parent process
	}

结果输出	无

函数名	pipe
调用语法	pipe (infile, outfile);
解说	与 fork 合用，给父进程和子进程提供通信的方式。送到 outfile 文件变量的信息可以通过 infile 文件变量读取。步骤： 1、调用 pipe 2、用 fork 将程序分成父进程和子进程 3、一个进程关掉 infile，另一个关掉 outfile

例子	pipe (INPUT, OUTPUT);
	$retval = fork();
	if ($retval != 0) {
	  # this is the parent process
	  close (INPUT);
	  print ("Enter a line of input:\n");
	  $line = <STDIN>;
	  print OUTPUT ($line);

	} else {
	  # this is the child process
	  close (OUTPUT);
	  $line = <INPUT>;
	  print ($line);
	  exit (0);
	}

结果输出	$ program Enter a line of input: Here is a test line Here is a test line $
函数名	exec
调用语法	exec (list);
解说	与 system 类似，区别是启动新进程前结束当前程序。常与 fork 合用，当 fork 分成两个进程后，子进程用 exec 启动另一个程序。
例子	
结果输出	
函数名	syscall
调用语法	syscall (list);
解说	调用系统函数，list 第一个元素是系统调用名，其余为参数。 如果参数是数字，就转化成 C 的整型数(type int)。否则传递字符串的指针。详见 UNIX 的帮助或 Perl 文档。 使用 syscall 必须包含文件 syscall.pl，即：

	require ("syscall.ph");
例子	
结果输出	

2、进程终止函数

函数名	die
调用语法	die (message);
解说	终止程序并向 STDERR 输出错误信息。message 可以为字符串或列表。如果最后一个参数不包含换行符，则程序文件名和行号也被输出。
例子	die ("Cannot open input file");
结果输出	Cannot open input file at myprog line 6.

函数名	warn
调用语法	warn (message);
解说	与 die 类似，区别是不终止程序。
例子	warn("Danger! Danger!\n");
结果输出	Danger! Danger!

函数名	exit
调用语法	exit (retcode);
解说	终止程序并指定返回值。
例子	exit(2);
结果输出	无

函数名	kill
调用语法	kill (signal, proclist);
解说	给一组进程发送信号。 signal 是发送的数字信号，9 为杀掉进程。

	proclist 是进程 ID 列表。详见 kill 的 UNIX 帮助。
例子	
结果输出	

3、进程控制函数

函数名	sleep
调用语法	sleep (time);
解说	将程序暂停一段时间。time 是停止的秒数。返回值为实际停止的秒数。
例子	sleep (5);
结果输出	无

函数名	wait
调用语法	procid = wait();
解说	暂停程序执行，等待子进程终止。 不需要参数，返回值为子进程 ID，如果没有子进程，返回-1。
例子	
结果输出	

函数名	waitpid
调用语法	waitpid (procid, waitflag);
解说	暂停程序执行，等待特定的子进程终止。procid 为等待的进程 ID

例子	$procid = fork();
	if ($procid == 0) {
	  # this is the child process
	  print ("this line is printed first\n");
	  exit(0);

	} else {
	  # this is the parent process
	  waitpid ($procid, 0);
	  print ("this line is printed last\n");
	}

结果输出	$ program this line is printed first this line is printed last $

4、其它控制函数

函数名	caller
调用语法	subinfo = caller();
解说	返回调用者的程序名和行号，用于 Perl Debugger。 返回值为三元素的列表： 1、调用处的包名 2、调用者文件名 3、调用处的行号
例子	
结果输出	

函数名	chroot
调用语法	chroot (dir);
解说	改变程序的根目录，详见 chroot 帮助。
例子	
结果输出	

函数名	local
调用语法	local($variable);
解说	在语句块(由大括号包围的语句集合)中定义局域变量，仅在此语句块中起作用，对其的改变不对块外同名变量造成影响。

	千万不要在循环中使用，否则每次循环都定义一个新的局域变量！
例子	
结果输出	

函数名	times
调用语法	timelist = times
解说	返回该程序及所有子进程消耗的工作时间。 返回值为四个浮点数的列表： 1、程序耗用的用户时间 2、程序耗用的系统时间 3、子进程耗用的用户时间 4、子进程耗用的系统时间
例子	
结果输出	

二、数学函数

函数名	sin
调用语法	retval = sin (value);
解说	参数为弧度值。

函数名	cos
调用语法	retval = cos (value);
解说	参数为弧度值。

函数名	atan2
调用语法	retval = atan2 (value1, value2);
解说	运算并返回 value1 除以 value2 结果的 arctan 值，单位为弧度，范围在-PI~PI。

应用例：
角度转化成弧度子程序。
	sub degrees_to_radians {
	local ($degrees) = @_;
	local ($radians);11:
	$radians = atan2(1,1) * $degrees / 45;
	}

函数名	sqrt
调用语法	retval = sqrt (value);
解说	平方根函数。value 为非负数。

函数名	exp
调用语法	retval = exp (value);
解说	返回 e 的 value 次方。

函数名	log
调用语法	retval = log (value);
解说	以 e 为底的自然对数。

函数名	abs
调用语法	retval = abs (value);
解说	绝对值函数。(Perl 4 中没有)

函数名	rand
调用语法	retval = rand (num);
解说	随机数函数，返回 0 和整数 num 之间的一个浮点数。

函数名	srand
调用语法	srand (value);
解说	初始化随机数生成器。保证每次调用 rand 真正随机。

三、字符串处理函数

函数名	index
调用语法	position = index (string, substring, position);

解说	返回子串 substring 在字符串 string 中的位置，如果不存在则返回-1。参数 position 是可选项，表示匹配之前跳过的字符数，或者说从该位置开始匹配。

函数名	rindex
调用语法	position = rindex (string, substring, position);
解说	与 index 类似，区别是从右端匹配。

函数名	length
调用语法	num = length (string);
解说	返回字符串长度，或者说含有字符的数目。

函数名	pos
调用语法	offset = pos(string);
解说	返回最后一次模式匹配的位置。

函数名	substr
调用语法	substr (expr, skipchars, length)
解说	抽取字符串（或表达式生成的字符串）expr 中的子串，跳过 skipchars 个字符，或者说从位置 skipchars 开始抽取子串（第一个字符位置为 0），子串长度为 length，此参数可忽略，意味着取剩下的全部字符。 当此函数出现在等式左边时，expr 必须为变量或数组元素，此时其中部分子串被等式右边的值替换。

函数名	study
调用语法	study (scalar);

解说	用一种内部格式提高变量的访问速度，同一时刻只对一个变量起作用。

函数名	lc uc
调用语法	retval = lc(string); retval = uc(string);
解说	将字符串全部转换成小/大写字母。

函数名	lcfirst ucfirst
调用语法	retval = lcfirst(string); retval = ucfirst(string);
解说	将第一个字母转换成小/大写。

函数名	quotameta
调用语法	newstring = quotemeta(oldstring);
解说	将非单词的字母前面加上反斜线(\)。 语句 ： $string = quotemeta($string); 等效于：$string =~ s/(\W)/\\$1/g; 常用于模式匹配操作中，确保字符串中没有字符被看作匹配操作符。

函数名	join
调用语法	join (joinstr, list);
解说	把字符串列表(数组)组合成一个长的字符串，在每两个列表元素间插入串 joinstr。

函数名	sprintf
调用语法	sprintf (string, fields);
解说	与 printf 类似，区别是结果不输出到文件，而

	作为返回值赋给变量。
例子	$num = 26; $outstr = sprintf("%d = %x hexadecimal or %o octal\n",$num, $num, $num); print ($outstr);
结果输出	26 = 1a hexadecimal or 32 octal

四、标量转换函数

函数名	chop
调用语法	$lastchar = chop (var);
解说	var 可为变量或数组，当 var 为变量时，最后一个字符被删除并赋给$lastchar，当 var 为数组/列表时，所有元素的最后一个字符被删除，最后一个元素的最后一个字母赋给$lastchar。

函数名	chomp
调用语法	result = chomp(var);
解说	检查字符串或字符串列表中元素的最后一个字符是否为由系统变量$/定义的行分隔符，如果是就删除。返回值为实际删除的字符个数。

函数名	crypt

调用语法	result = crypt (original, salt);
解说	用 DES 算法加密字符串，original 是将要加密的字符串，salt 是两个字符的字符串，定义如何改变 DES 算法，以使更难解码。返回值为加密后的串。

函数名	hex
调用语法	decnum = hex (hexnum);
解说	将十六进制数(字符串形式)转化为十进制数。

函数名	int
调用语法	intnum = int (floatnum);
解说	将浮点数舍去小数部分转化为整型数。

函数名	oct
调用语法	decnum = oct (octnum);
解说	将八进制数(字符串形式)或十六进制数("0x.." 形式)转化为十进制数。

函数名	ord
调用语法	asciival = ord (char);
解说	返回单个字符的 ASCII 值，与 PASCAL 中同名函数类似。

函数名	chr
调用语法	$char = chr (asciival);
解说	返回 ASCII 值的相应字符，与 PASCAL 中同名函数类似。

<table>
<tr><td>函数名</td><td>pack</td></tr>
<tr><td>调用语法</td><td>formatstr = pack(packformat, list);</td></tr>
<tr><td>解说</td><td>把一个列表或数组以在实际机器存贮格式或C等编程语言使用的格式转化（包装）到一个简单变量中。参数packformat包含一个或多个格式字符，列表中每个元素对应一个，各格式字符间可用空格或tab隔开，因为pack忽略空格。
除了格式a、A和@外，重复使用一种格式多次可在其后加个整数，如：
$twoints = pack ("i2", 103, 241);
把同一格式应用于所有的元素则加个*号，如：
$manyints = pack ("i*", 14, 26, 11, 83);
对于a和A而言，其后的整数表示要创建的字符串长度，重复方法如下：
$strings = pack ("a6" x 2, "test1", "test2");
格式@的情况比较特殊，其后必须加个整数，该数表示字符串必须的长度，如果长度不够，则用空字符(null)补足，如：
$output = pack ("a @6 a", "test", "test2");
pack函数最常见的用途是创建可与C程序交互的数据，例如C语言中字符串均以空字符(null)结尾，创建这样的数据可以这样做：
$Cstring = pack ("ax", $mystring);
下表是一些格式字符与C中数据类型的等价关系：
<table>
<tr><td>字符</td><td>等价C数据类型</td></tr>
<tr><td>C</td><td>char</td></tr>
<tr><td>d</td><td>double</td></tr>
<tr><td>f</td><td>float</td></tr>
</table>
</td></tr>
</table>

i	int
I	unsigned int (or unsigned)
l	long
L	unsigned long
s	short
S	unsigned short

完整的格式字符见下表。

格式字符	描述
a	用空字符(null)补足的字符串
A	用空格补足的字符串
b	位串，低位在前
B	位串，高位在前
c	带符号字符（通常-128~127）
C	无符号字符（通常 8 位）
d	双精度浮点数
f	单精度浮点数
h	十六进制数串，低位在前
H	十六进制数串，高位在前
i	带符号整数
I	无符号整数
l	带符号长整数
L	无符号长整数
n	网络序短整数
N	网络序长整数
p	字符串指针
s	带符号短整数
S	无符号短整数
u	转化成 uuencode 格式

v	VAX 序短整数
V	VAX 序长整数
x	一个空字节
X	回退一个字节
@	以空字节(null)填充

函数名	unpack
调用语法	@list = unpack (packformat, formatstr);
解说	unpack 与 pack 功能相反，将以机器格式存贮的值转化成 Perl 中值的列表。其格式字符与 pack 基本相同（即上表），不同的有：A 格式将机器格式字符串转化为 Perl 字符串并去掉尾部所有空格或空字符；x 为跳过一个字节；@为跳过一些字节到指定的位置，如@4 为跳过 4 个字节。下面看一个@和 X 合同的例子： $longrightint = unpack ("@* X4 L", $packstring); 此语句将最后四个字节看作无符号长整数进行转化。下面看一个对 uuencode 文件解码的例子： 1 : #!/usr/local/bin/perl 2 : 3 : open (CODEDFILE, "/u/janedoe/codefile") \|\| 4 : die ("Can't open input file"); 5 : open (OUTFILE, ">outfile") \|\| 6 : die ("Can't open output file"); 7 : while ($line = <CODEDFILE>) { 8 : $decoded = unpack("u", $line); 9 : print OUTFILE ($decoded); 10: } 11: close (OUTFILE);

	12: close (CODEDFILE); 当将 pack 和 unpack 用于 uuencode 时，要记住，虽然它们与 UNIX 中的 uuencode、uudecode 工具算法相同，但并不提供首行和末行，如果想用 uudecode 对由 pack 的输出创建的文件进行解码，必须也把首行和末行输出（详见 UNIX 中 uuencode 帮助）。
函数名	vec
调用语法	retval = vec (vector, index, bits);
解说	顾名思义，vec 即矢量(vector)函数，它把简单变量 vector 的值看作多块(维)数据，每块含一定数目的位，合起来即一个矢量数据。每次的调用访问其中一块数据，可以读取，也可以写入。参数 index 就象数组下标一样，提出访问哪一块，0 为第一块，依次类推，要注意的是访问次序是从右到左的，即第一块在最右边。参数 bits 指定每块中的位数，可以为 1,2,4,8,16 或 32。
例子	1 : #!/usr/local/bin/perl 2 : 3 : $vector = pack ("B*", "11010011"); 4 : $val1 = vec ($vector, 0, 4); 5 : $val2 = vec ($vector, 1, 4); 6 : print ("high-to-low order values: $val1 and $val2\n"); 7 : $vector = pack ("b*", "11010011"); 8 : $val1 = vec ($vector, 0, 4); 9 : $val2 = vec ($vector, 1, 4); 10: print ("low-to-high order values: $val1 and $val2\n");
结果	high-to-low order values: 3 and 13 low-to-high order values: 11 and 12
函数	defined

名	
调用语法	retval = defined (expr);
解说	判断一个变量、数组或数组的一个元素是否已经被赋值。expr 为变量名、数组名或一个数组元素。 如果已定义，返回真，否则返回假。
函数名	undef
调用语法	retval = undef (expr);
解说	取消变量、数组或数组元素甚至子程序的定义，回收其空间。返回值始终为未定义值，此值与空串等效。

五、数组和列表函数

函数名	grep
调用语法	@foundlist = grep (pattern, @searchlist);
解说	与同名的 UNIX 查找工具类似，grep 函数在列表中抽取与指定模式匹配的元素，参数 pattern 为欲查找的模式，返回值是匹配元素的列表。
例子	@list = ("This", "is", "a", "test"); @foundlist = grep(/^[tT]/, @list);
结果	@foundlist = ("This", "test");
函数名	splice
调用	@retval = splice (@array, slipelements, length, @newlist);

语法	
解说	拼接函数可以向列表（数组）中间插入元素、删除子列表或替换子列表。参数 skipelements 是拼接前跳过的元素数目，length 是被替换的元素数，newlist 是将要拼接进来的列表。当 newlist 的长度大于 length 时，后面的元素自动后移，反之则向前缩进。因此，当 length=0 时，就相当于向列表中插入元素，而形如语句 splice (@array, -1, 0, "Hello"); 则向数组末尾添加元素。而当 newlist 为空时就相当于删除子列表，这时，如果 length 为空，就从第 skipelements 个元素后全部删除，而删除最后一个元素则为：splice (@array, -1);这种情况下，返回值为被删去的元素列表。

函数名	shift
调用语法	element = shift (@arrayvar);
解说	删去数组第一个元素，剩下元素前移，返回被删去的元素。不加参数时，缺省地对@ARGV 进行操作。

函数名	unshift
调用语法	count = unshift (@arrayver, elements);
解说	作用与 shift 相反，在数组 arrayvar 开头增加一个或多个元素，返回值为结果(列表)的长度。等价于 splice (@array, 0, 0, elements);

函数名	push
调用	push (@arrayvar, elements);

语法	
解说	在数组末尾增加一个或多个元素。等价于 slice (@array, @array, 0, elements);

函数名	pop
调用语法	element = pop (@arrayvar);
解说	与 push 作用相反，删去列表最后一个元素，并将其作为返回值，当列表已空，则返回“未定义值”(即空串)。

函数名	split
调用语法	@list = split (pattern, string, maxlength);
解说	将字符串分割成一组元素的列表。每匹配一次 pattern，就开始一个新元素，但 pattern 本身不包含在元素中。maxlength 是可选项，当指定它时，达到该长度就不再分割。

函数名	sort
调用语法	@sorted = sort (@list);
解说	按字母次序给列表排序。

函数名	reverse
调用语法	@reversed = reverse (@list);
解说	按字母反序给列表排序。

函数名	map
调用	@resultlist = map (expr, @list);

语法	
解说	此函数在 Perl5 中定义，可以把列表中的各个元素作为表达式 expr 的操作数进行运算，其本身不改变，结果作为返回值。在表达式 expr 中，系统变量$_代表各个元素。
例子	1、@list = (100, 200, 300); @results = map ($_+1, @list); 2、@results = map (&mysub($_), @list);
结果	1、(101, 201, 301) 2、无

函数名	wantarray
调用语法	result = wantarray();
解说	Perl 中，一些内置函数的行为根据其处理简单变量还是数组有所不同，如 chop。自定义的子程序也可以定义这样两种行为。当子程序被期望返回列表时，此函数返回值为非零值(真)，否则为零值(假)。

例子	1 : #!/usr/local/bin/perl
	2 :
	3 : @array = &mysub();
	4 : $scalar = &mysub();
	5 :
	6 : sub mysub {
	7 : if (wantarray()) {
	8 : print ("true\n");
	9 : } else {
	10: print ("false\n");
	11: }
	12: }

结果	$program true false $

六、关联数组函数

函数名	keys
调用语法	@list = keys (%assoc_array);
解说	返回关联数组无序的下标列表。

函数名	values
调用语法	@list = values (%assoc_array);
解说	返回关联数组无序的值列表。

函数名	each
调用语法	@pair = each (%assoc_array);
解说	返回两个元素的列表--键值对（即下标和相应的值），同样无序。当关联数组已空，则返回空列表。

函数名	delete
调用语法	element = delete (assoc_array_item);
解说	删除关联数组中的元素，并将其值作为返回值。
例子	%array = ("foo", 26, "bar", 17"); $retval = delete ($array{"foo"});
结果	$retval = 26;

函数名	exists
调用语法	result = exists (element);
解说	在 Perl5 中定义，判断关联数组中是否存在某元素，若存在，返回非零值(真)，否则返回零值(假)。
例子	$result = exists ($myarray{$mykey});