Perl 基本语法

by Jian Lee

标量

标量是 Perl 中最简单的数据类型。大多数的标量是数字(如 255 或 3.25e20)或者字符串(如 hello 或者盖茨堡地址)。

数字

perl 中所有数字内部的格式都是双精度浮点数。

浮点数

```
1.25
255.000
255.0

7.25e45 #7.25x10   的45   次   一个大整数

-6.5e24 # -6.5x10   的24 次方   一个大的负数

-12e-24 #- -12x10   的24 次方   很小的负数

-1.2E-23 #  指数符号可以大写(E)
```

整数

```
0
2001
-40
255
61298040283768
```

其中 61298040283768 也可以写作：

```
61_298_040_283_768
```

非十进制整数

```
0377     #  八进制数字3 7 7,  等于十进制数字2 5 5

0xff      #  十六进制数字F F ,  等于十进制数字2  55

0b11111111 #  等于十进制数字2 5 5
```

可以用下划线表示：

```
0x1377_0B77
0x50_65_72_7C
```

数字操作符

```
2+3      #2+3,5
5.1-2.4  #5.1-2.4,2.7
3*12     #3*12,36
14/2      #14/2,7
10.2/0.3  #10.2/0.3,34

10/3      #  通常是浮点除3 .33333... ...
```

字符串

单引号字符串

```
'fred' #四个字符 f,r,e,d

'' #  空字符   没有字符
'hello\n'

'\'\\' #单引号') 跟着反斜线(\)  字符串
```

单引号中的 "\n" 不会被当作换行符来处理。

双引号字符串

```
"barney"        #等同于'barney'

"hello world\n"  #hello world, 换行
```

字符串操作符

链接操作符 "."

```
"hello"."world"     #  同于"helloworld"

"hello"."".."world"  #  同"hello world"

'hello world'."\n"  #  同"hello world\n"
```

重复操作符 "x"

```
"fred" x 3   # "fredfredfred"

5 x 4        #  等于"5" x 4, "5555"
```

数字和字符串之间的自动转换

大多数情况下,Perl 将在需要的时候自动在数字和字符串之间转换。它怎样知道什么时候需要字符串,什么时候需要数字呢?这完全依赖于标量值之间的的操作符。如果操作符(如+)需要数字,Perl 将把操作数当作数字看待。如果操作符需要字符串(如 .), Perl 将把操作数当作字符串看待。不必担心数字和字符串的区别;使用恰当的操作符,Perl 将为你做剩下的事。

```
"12" * "3"  # * 操作符需要数字，所以结果为36

"12fred34" * " 3" #  结果仍然是36 ,  后面的非数字部分和前面的空格都过滤掉。

"Z" . 5 * 7 #  等"Z".35,  或Z35"
```

Perl 内嵌的警告

使用 -w 参数可以打开警告:

```
$ perl -w perl  程序  #  命令行执行警告

#!/usr/bin/perl -w  #  源码中使用警告
```

标量变量

标量变量可以存放一个标量值。标量变量的名字由一个美圆符号($)后接 Perl 标识符:由字母或下划线开头,后接字母,数字,或者下划线。或者说由字母,数字和下划线组成,但不能由数字开头。大小写是严格区分的:变量$Fred 和变量 $fred 是不同的。任意字母,数字,下划线都有意义,如:

```
$a_very_long_variable_that_ends_in_1
$a_very_long_variable_that_ends_in_2
```

标量赋值

```
$fred = 17;
$barney = "hello";

$barney = $fred + 3;#  将fred  的值加上三赋给$barney (20)

$barney= $barney*2;#  将变量barney  乘  2  赋给barney (40)
```

二元赋值操作符

```
$fred = $fred + 5; #  没用二元赋值操作符

$fred+=5;          #  利用二元赋值操作符
$barney = $barney*3;
$barney*=3;

$str = str . ""; #$str  接空格

$str .= "";    #同上
```

print 输出

```
print "hello world\n";   # 输出hello world, 后接换行符
print "The answer is", 6*7, ".\n"
```

字符串中引用标量变量

```
$meal = "brontosaurus steak" ;
$barney = "fred ate a $meal";

$barney = 'fred ate a'.$meal;    #  同上
```

if 控制结构

```
if ($name gt 'fred') {
    print "$name’comes after 'fred' in sorted order.\n";
}
```

Boolean 值

perl 没有专门的 Boolean 值， 真假值这样判断：

- 如果值为数字,0 是 false;其余为真
- 如果值为字符串,则空串(')为 false;其余为真
- 如果值的类型既不是数字又不是字符串,则将其转换为数字或字符串后再利用上述规则

这些规则中有一个特殊的地方。由于字符串’0’ 和数字 0 有相同的标量值,Perl 将它们相同看待。也就是说字符串 ’0’ 是唯一一个非空但值为 0 的串。

用户的输入 <STDIN>

chomp 操作

```
$text = "a line of text\n";    # 也可以由 STDIN> 输

chomp($text);                  #  去掉换行符\n)。
```

一步执行：

```
chomp ($text = <STDIN>); #  读入、但含换行符
```

chomp 是一个函数。作为一个函数,它有一个返回值,为移除的字符的个数。这个数字基本上没什么用：

```
$food = <STDIN>;

$betty = chomp $food; #得到值1
```

如上,在使用 chomp 时,可以使用或不使用括号()。这又是 Perl 中的一条通用规则:除非移除它们时含义会变,否则括号是可以省略的。

while 控制结构

```
$count = 0;
while ($count < 10) {
   $count + = 2;
   print "count is now $count\n";
}
```

undef 值

变量被赋值之前使用它会有什么情况发生呢?通常不会有什么严重的后果。变量在第一次赋值前有一个特殊值 undef, 按照 Perl 来说就是:"这里什么也没有,请继续"。如果这里的“什么也没有”是一些“数字”,则表现为 0。如果是“字符串”,则表现为空串。但 undef 既非数字也非字符串,它是另一种标量类型。

defined 函数

能返回 undef 的操作之一是行输入操作,<STDIN>。通常,它会返回文本中的一行。但如果没有更多的输入,如到了文件的结尾,则返回 undef。要分辨其是 undef 还是空串,可以使用 defined 函数, ， 如果其参数是 undef 值就返回 false，其他值返回 true。

```
$madonna = <STDIN>;
If ($defined ($madonna)){
```

```
   print "The input was $madonna";
} else {
   print "No input available!\n;
}
```

如果想声明自己的 undef 值,可以使用 undef:

```
$madonna = undef ; #  同$madonna  未被初始化一样。
```

列表和数组

```
#!/usr/bin/env perl -w
```

```
$fred[0] = "yabba";
$fred[1] = "dabba";
$fred[2] = "doo";

print @fred;
#print @fred."\n";
```

qw 简写

```
qw ! fred barney betty wilma dino !

qw#  fred barney betty wilma dino # #  有类似注释
qw(   fred barney betty wilma dino )
...
```

列表赋值

```
($fred, $barney, $dino) = ("flintstone", "rubble",
undef);
($fred, $barney) = qw <flintstone rubble slate granite>;

#  两个值被忽略了
```

```
($rocks[0],$rocks[1],$rocks[2],$rocks[3]) = qw/talc mica
feldspar quartz/;
```

当想引用这个数组时, Perl 有一种简单的写法。在数组名前加@(后没有中括号)来引用整个数组。 你可以把他读作 "all of the "(所有的)",所以@rocks 可以读作 "all of the rocks(所有的石头)"。其在赋值运算符左右均有效:

```
@rocks = qw / bedrock slate lava /;

@tiny = ();                          #空表
```

```
@giant = 1..1e5;              #  包含100,000  个元素的表

@stuff = (@giant, undef, @giant);      #包含200,001  个元素的表
@dino = "granite";
@quarry = (@rocks, "crushed rock", @tiny, $dino);
```

pop 和 push 操作

```
@array = 5..9;

$fred = pop(@array);      #$fred 得到9,@array  现为(5,6,7,8)

$barney = pop @array;   #$barney gets 8, @array  现为(5,6,7)

pop @array;                  #@array  现在为 5,6)(7  被弃了
```

```
push(@array,0);                #@array  现为5,6,0)

push @array,8;                 #@array  现为5,6,0,8)

push @array,1..10;             #@array  现多了10  个元素
@others =qw/9 0 2 1 0 /;
```

```
push @array,@others;          #@array 现在又多了5 个元素 其
有19 个
```

shift 和 unshift 操作

push 和 pop 对数组的末尾进行操作(或者说数组右边有最大下标的元素,这依赖于你是怎样思考的)。相应的, unshift 和 shift 对一个数组的开头进行操作(数组的左端有最小下标的元素) 。下面是一些例子:

```
@array = qw# dino fred barney #;

$m = shift (@array);       #$m 得到"dino", @array 现在为
("fred", "barney")

$n = shift @array;         #$n 得到"fred", @array 现在为
("barney")

shift @array;                #@array 现在为空

$o = shift @array;         #$o 得到undef, @arry 仍为空

unshift(@array,5);         #@array 现在为 5)

unshift @array,4;          #@array 现在为 4,5)
@others = 1..3;

unshift @array, @others;  #array 现在为1 , 2,3,4,5)
```

和 pop 类似,如果其数组变量为空,则返回 undef。

字符串中引用数组

和标量类似,数组也可以插入双引号的字符串中。插入的数组元素会自动由空格分开:

```
@rocks = qw{ flintstone slate rubble };

print "quartz @rocks limestone\n";       # 输出为 5 种
rocks 由空格分开
```

foreach 控制结构

```
foreach $rock (qw/ bedrock slate lava /) {

    print "One rock is $rock.\n" ;          # 打印出3 种
rocks
}
```

这里的$rock 不是这些列表元素中的一个拷贝而是这些元素本身

最常用的默认变量 : $_

如果在 foreach 循环中省略了控制变量,那 Perl 会使用其默认的变量:$_。除了其不寻常的名字外,这和普通变量类似,如下面代码所示:

```
foreach(1..10){                  # 使用默认的变量$_
   print "I can count to $_!\n";
}
$_ = "Yabba dabba doo\n";

print;                                    # 打印出默认变量$_
```

reverse 操作

reverse(逆转)操作将输入的一串列表(可能是数组)按相反的顺序返回。

```
@fred = 6 .. 10;

@barney = reverse (@fred);  # 得到10,9,8,7,6

@wilma = reverse 6 . .10;   # 同上, 没有使用额外的数组

@fred = reverse @fred;      # 将逆转过的字符串存回去
```

sort 操作

```
@rocks = qw/ bedrock slate rubble granite /;

@sorted = sort(@rocks);                    # 得到 bedrock,
granite, rubble, slate
```

标量和列表上下文

```
42 + something              #something 必须是标量

sort something              #something 必须是列表
@people = qw( fred barney betty );

@sorted = sort @people;           # 列表内容: barney ,
betty, fred

$number = 42 + @people;           # 标量内容 4 2 +3, 得到 4 5
```

另一个例子是 reverse。在列表 context 中,它返回反转的列表。在标量 context 中,返回反转的字符串(或者将反转的结果串成一个字符串):

```
@backwards = reverse qw / yabba dabba doo /;

# 返回 doo, dabba, yabba
$backwards = reverse qw/ yabba  dabba doo /;

# 返回 doodabbadabbay
```

在列表 Context 中使用 Scalar-Producing 表达式

如果一个表达式不是列表值,则标量值自动转换为一个元素的列表:

```
@fred = 6*7;
@barney = "hello" . '' . "world";
```

强制转换为标量 Context

偶尔,你可能需要标量 context 而 Perl 期望的是列表。这种情况下,可以使用函数 scalar。它不是一个真实的函数因为其仅是告诉 Perl 提供一个标量 context:

```
@rocks = qw(talc quartz jade obsidian);
print "How many rocks do you have?\n;

print "I have " @rocks, "rocks!\n";    # 错误 输出 rocks 的名字

print "I have " scalar @rocks, "rocks!\n;    # 正确 输出其数字
```

<STDIN> 在列表 Context 中

```
@lines = <STDIN>; # 将输入读入列表 context 中

chomp (@lines = <STDIN>); # 读入所有的行 包括换行符
```

子程序

使用 sub 定义子程序

```
sub marine {

       $n + = 1; # 全局变量 n
       print "Hello, sailor number $n!\n";
}
```

调用子程序

```
&marine; # 输出 Hello, sailor number 1!

&marine; # 输出 Hello, sailor number 2!

&marine; # 输出 Hello, sailor number 3!
```

```
&marine; #  输出Hello, sailor number 4!
```

通常有括号,即便参数为空。子程序将继承调用者的 @_ 的值。

参数

```
$n = &max(10,15);     # 此子程序有 2  个参数
```

此参数列表被传到子程序中;这些参数可以被子程序使用。当然,这些参存放在某个地方,在 Perl 中,会自动将此参数列表(此参数列表的另一个名字)自动存放在一个叫做@_的数组中。子程序可以访问次数组变量来确定此参数的个数以及其值。这也就是说此子程序参数的第一个值存放在$_[0]中,第二个存放在$_[1],依次类推。但必须强调的是这些变量和 $_ 这个变量没有任何关系,如$dino[3](数组 @dino 的一个元素)和$dino 的关系一样。这些参数必须存放在某个数组变量中,Perl 存放在@_这个变量中。

```
sub max{
    if($_[0] > $_[1]) {
       $_[0];
       } else {
       $_[1];
    }
}
```

my 变量

```
foreach (1..10){

   my($square) = $_*$_; #本循环中的私有变量
   print "$_ squared is $squrare.\n";
}
```

变量$square 是私有的,仅在此块中可见;在本例中,此块为 foreach 循环块。

当然,my 操作不会改变赋值参数的 context:

```
my ($num) = @_;         # 列表context,  同($sum) = @_;

my $num = @_;           #  量context,  同$num = @_;
```

使用 strict Pragma

```
use strict; #  迫使采用更严格的检测
```

省略 &

有些地方调用子程序可以不要 &

```
my @cards = shuffle(@deck_of_cards); # &是不必要的
```

输入和输出

从标准输入设备输入

从标准输入设备输入是容易的。使用<STDIN>。在标量 context 中它将返回输入的下一行:

```
$line = <STDIN>;     #  读下一行

chomp($line);        #  去掉结尾的换行符

chomp($line=<STDIN>)  #  同  更常用的方法
```

于,行输入操作在到达文件的结尾时将返回 undef,这对于从循环退出时非常方便的:

```
while (defined($line = <STDIN>)) {
   print "I saw $line";
```

```
}
```

从 <> 输入

尖括号操作(<>)是一种特殊的行输入操作。其输入可由用户选择

```
$n = 0;
while (defined($line = <>)) {
    $n += 1;
    chomp($line);
    print "$n  $line\n";
}
```

```
while (<>) {
  chomp;
  print "It was $_ that I saw!\n";
}
```

调用参数

技术上讲，<>从数组@ARGV 中得到调用参数。这个数组是 Perl 中的一个特殊数组,其包含调用参数的列表。 换句话说，这和一般数组没什么两样 (除了其名字有些特别: 全为大写字母) ,程序开始运行时,调用参数已被存在@ARGV 之中了。

输出到标准输出设备

```
print @array;      #  打印出元素的列表

print "@array";    #  打印一个字符串 包含一个内插的数组
```

第一个语句打印出所有的元素,一个接着一个,其中没有空格。第二个打印出一个元素,它为@array 的所有元素,其被存在一个字符串中。也就是说,打印出@array 的所有元素,并由空格分开。如果@array 包含 qw /fred barney betty /,则第一个例子输出为:fredbarneybetty,而第二个例子输出为 fred barney betty(由空格分开)。

使用 printf 格式化输出

```
printf "Hello, %s :  your password expires in %d days!\n",
   $user, $days_to_die;

printf "%6f\n" 42;          #   输出为o o o o 42 (o  此指代空格
printf "%23\n",2e3+1.95;  # 2001
```

数组和 printf

```
my @items = qw( wilma dino pebbles );
my $format = "The items are:\n". ("%10s\n" x @items);
printf $format, @items;
```

等同:

```
printf "The items are:\n". ("%10s\n"x @items), @items;
```

本处 @items 有两个不同的 context （上下文），第一次表示元素个数，第二次表示列表中的所有元素。

句柄 （即文件描述符）

Perl 自身有六个文件句柄 ： STDIN,STDOUT,STDERR,DATA,ARGV,ARGVOUT

文件句柄的打开

```
open CONFIG, "dino" ;
open CONFIG, "<dino"    ;
open BEDROCK, ">fred"   ;
open LOG, ">>logfile" ;
```

Perl 的新版本中(从 Perl5.6 开始)，open 支持“3 参数”类型:

```
open CONFIG, "<",  "dino";
open BEDROCK, ">", $file_name;
open LOG, ">>", &logfile_name();
```

Bad 文件句柄

关闭文件句柄

```
close BEDROCK;
```

严重错误和 die

可以和 C 中使用 perror 类似，用 die 函数：

```
if (!open LOG, ">>logfile") {
   die "Cannot create logfile:$!";
}
```

使用文件句柄

```
if (!open PASSWD, "/etc/passwd") {
    die "How did you get logged in?($!)";
}

while (<PASSWD>) {
    chomp;
    ...
}
```

哈希

什么是哈希

和 Python 的字典一样

哈希元素的存取

```
$hash {$some_key}
```

作为整体的 hash

要引用整个 hash,使用百分号(“%”)作为前缀。

```
%some_hash = ("foo",35, "bar", 12.4, 2.5, "hello",
"wilma", 1.72e30, "betty", "bye\n");
```

hash 的值(在列表 context 中)是一个 key/value 对的列表 ：

```
@array_array = %some_hash;
```

哈希赋值

```
%new_hash = %old_hash;
%inverse_hash = reverse %any_hash;
```

大箭头符号 (=>)

```
my %last_name = (
  "fred" => "flintstone",
  "dino" => undef,
  "barney" => "rubble",
  "betty" => "rubble",
);
```

哈希函数

keys 和 values

```
my %hash = ("a" => 1, "b" => 2, "c" => 3);
my @k = keys %hash;
my @v = values %hash;
```

each 函数

```
while (($key, $value) = each %hash) {
   print "$key => $value\n";
}
```

exists 函数

```
if (exists $books{$dino}) {
    print "Hey, there's a libaray card for dino!\n";
}
```

delete 函数

```
my $person = "betty";

delete $books{$person};   #   将person   的借书卡删除掉
```

正则表达式

简单的模式

```
$_ = "yabba dabba doo";
if (/abba/) {
    print "It matched!\n";
}
```

所有在双引号中的转义字符在模式中均有效,因此你可以使用 /coke\tsprite/ 来匹配 11 个字符的字符串 coke, tab(制表符),sprite。

元字符

```
.
?
+
*
```

模式中的分组

```
/fred+/      #   能匹配fredddddd 等

/(fred)+/    #   能匹配 fredfredfred   等

/(fred)*/    #   可以匹配"hello,world" ， 因为* 是匹配前面的0或多次
```

选择符（|）

```
/fred|barney|betty/
/fred( |\t)+barney/
```

字符类

指 [] 中的一列字符。

字符类的简写

```
\d    ==   [0-9]
\w    ==   [A-Za-z0-9_]

\s    ==   [\f\t\n\r]  # 格式符form-feed)、制表符 tab)  、换行符、回车
```

简写形式的补集

```
\D  ==   ^[\d]
\W  ==  [^\w]
\S  ==  [^\s]
```

可以组合：

```
[\d\D]   #   任何数字和任何非数字，可以匹配所有字符！比如.  是不能匹配所有字符的

[^\d\D]   #   无用
```

正则表达式的应用

使用 m// 匹配

同 qw // 一样，可以使用任何成对字符，比如可以使用 m(fred), m<fred>, m{fred}, m[fred],或者 m,fred,, m!fred!, m^fred^。

如果使用 / 作为分隔符，可以省略前面的 m

如果使用配对的分隔符，那不用当心模式内部会出现这些分隔符，因为通常模式内部的分隔符也是配对的。 因此，m(fred(.*)barney)，m{\w{2,}}，m[wilma[\n \t]+betty]是正确的。对于尖括号(<和>),它们通常不是配对的。如模式 m{(\d+)\s*>=?\s*(\d+)},如果使用尖括号,模式中的尖括号前因当使用反斜线(\)，以免模式被过早的结束掉。

可选的修饰符

不区分大小写： /i

```
if (/yes/i) {           #大小写无关
   print "In that case, I recommend that you go
bowling.\n";
}
```

匹配任何字符： /s

使用/s 这个修饰符,它将模式中点（.）的行为变成同字符类 [\d\D] 的行为类似：可以匹配任何字符,包括换行符。从下例中可见其区别：

```
$_ = "I saw Barney\ndown at the bowing alley\nwith
Fred\nlast night.\n;
if (/Barney.*Fred/s) {
   print "That string mentions Fred after Barney!\n";
}
```

添加空格： /x

/x 修饰符,允许你在模式中加入任何数量的空白,以方便阅读：

```
/-?\d+\.?\d*/       #   这是什么含义

/ -? \d+ \.? \d* /x  #   要好些
```

Perl 中,注释可以被作为空白,因此使用/x,可以在模式中加上注释：

```
/
-?  #  可选的负号
d+ #  小数点前一个或多个十进制数字
\.? #  可选的小数点
\d* #  小数点后一些可选的十进制数字
/x #  模式结束
```

使用多个修饰符

```
if (/barney.*fred/is/){ # /i   和/s
  print "That string mentions Fred after Barney!\n";
}
```

锚定

```
^    开头
$    结尾
```

注意：/^fred$/能同时匹配上 "fred" 和 "fred\n"。

```
/^\s*$/   #   匹配空行
```

词锚定： \b

/\bfred\b/ 可以匹配单词 "fred"，但不能匹配 "frederick"

绑定操作符： =~

对 $_ 进行匹配只是默认的行为，使用绑定操作符（=~）将告诉 Perl 将右边的模式在左边的字符串上进行匹配,而非对 $_ 匹配。

```
my $some_other = "I dream of betty rubble.";
if ($some_other =~ /\brub/) {
    print "Aye, there's the rub.\n";
```

```
}
```

绑定操作符优先级较高：

```
my $likes_perl = <STDIN> =~ /\byes\b/i;
```

匹配变量

```
$_ = "Hello there, neighbor";

if (/\s(\w+),/) {             #  空格和逗号之间的词
    print "The word was $1\n";
}
```

自动匹配变量

```
$&    # 整个被匹配的部分

$`    # 匹配部分的前一部分存放在 ` 中

$'    # 后一部分被存到 '
```

使用正则表达式处理文件

使用 s/// 进行替换

```
$_ = "He's out bowling with Barney tonight.";

s/Barney/Fred/;             # Barney 被 red 替换掉
print "$_\n";
```

```
#  接上例 现在$_ 为" He's out bowling with Fred tonight."

s/Wilma/Betty/;             #  用Wilma 替换Betty( 败

s/with (\w+)/agaist $1's team/;

print "$_\n";                #  为He's out bowling
against Fred's team tonight.";
```

使用 /g 进行全局替换

```
$_ = "home, sweet home!";
s/home/cave/g;
print "$_\n";            # "cave, sweet cave!"
```

全局替换的一个常用地方是将多个空格用单个空格替换掉：

```
$_ = "Input     data\t may have           extra
whitespace.";

s/\s+/ /g;              # 现在是" I nput data may have extra
whitespace."
```

现在已经知道怎样去掉多余的空格，那怎样去掉开头和结尾的空白呢?这是非常容易的：

```
s/^\s+//;       #将开头的空白去掉

s/\s+$//;       #将结尾的空白去掉

s/^\s+|\s+$//g;   #  将开头，结尾的空白去掉
```

不同的分隔符

如同 m//和 qw//一样，我们也可以改变 s///的分隔符。但这里使用了 3 个分隔符，因此有些不同。

```
s#^https://#http://#;
```

如果使用的是配对的字符，也就是说其左字符和右字符不的，则必需使用两对：一对存放模式，一对存放替换的字符串。此时，分隔符甚至可以是不同的。事实上，分隔符还可以使用普通的符号(非配对的)。下面三例是等价的：

```
s{fred}{barney};
```

```
s[fred](barney);
s<fred>#barney#;
```

可选的修饰符

除了/g 修饰符外，替换操作中还可以使用 /i ， /x ， 和 /s， 这些在普通的模式匹配中已经出现过的修饰符。其顺序是无关紧要的。

```
s#wilma#Wilma#gi;              #    将所有的Wilma, 或者WILMA

等等    由wilma    替换掉

s{_ _END_ _.*}{ }s;            #    将END    标记及其后面的行去掉
```

绑定操作

同 m// 一样,我们也可以通过使用绑定操作符改变 s/// 的替换目标:

```
$file_name =~ s#^.*###s;                # 将file_name    中

所有的Unix    类型的路径去掉
```

大小写替换

有时,希望确保被替换的字符串均是大写的(或者不是,视情况而定）。这在 Perl 中只需使用某些修饰符就能办到。 \U 要求紧接着的均是大写:

```
$_ = "I saw Barney with Fred.";

s/(fred|barney)/\U$1/gi;          # $_    现在是" I  saw
BARNEY with FRED."
```

同样,也可以要求后面的均为小写 \L :

```
s/(fred)|barney/\L$1/gi;        #$_    现在是 "I saw barney
with fred."
```

默认时,会影响到剩余的(替换的)字符串。可以使用 \E 来改变这种影响:

```
s/(\w+) with (\w+)/\U$2\E with $1/I;       # $1    现在是" I
saw FRED with barney."
```

使用小写形式时(\l 和 \u),只作用于下一个字符:

```
s/ (fred|barney)/\u$1/ig;     #$_现在是" I  saw FRED with
Barney."
```

也可以同时使用它们。如使用\u 和\L 表示 "第一个字母大写,其它字母均小写"。\L 和\u 可以按任意顺序出现。Larry 意识到人们有时可能按相反顺序使用它们,因此他将 Perl 设计成,在这两种情况下都是将第一个字母大写,其余的小写。Larry 是个非常好的人。

```
s/(fred|barney)/\u\L$1/ig; #$_    现在为" I  saw Fred with
Barney."
```

这些在替换中出现的大小写转换的修饰符,也可在双引号中使用:

```
print "Hello, \L\u$name\E, would you like to play a
game?\n";
```

split 操作

另一个使用正则表达式的操作是 split ， 它根据某个模式将字符串分割开。这对于由制表符分割开,冒号分割开,空白分割开,或者任意字符分割开的数据是非常有用的。任何可在正则表达式之中（通常，是一个简单的正则表达式）指定分离符（separator）的地方,均可用 split。其形式如下:

```
@fields = split /separtor/, $string;

@fields = split /:/, "abc:def:g:h";    # 返回( "abc",
"def", "g", "h")
```

/\s+/ 这个模式分隔符非常常见:

```
my $some_input = "This    is a \t            test.\n";
```

```
my @args = split /\s+/, $some_input;          # ("This",
"is", "a", "test." )
```

默认时,split 对$_操作,模式为空白:

```
my @fields = split;      #   同split /\s+/, $_;
```

join 函数

join 函数不使用模式,但它完成同 split 相反的操作:split 将一个字符串分割开,而 join 函数将这些分割的部分组合成一个整体。join 函数类似于 :

```
my $result = join $glue, @pieces;
```

join 函数的第一个参数是粘合元素(glue),它可以是任意字符串。剩下的参数是要被粘合的部分。join 将粘合元素添加在这些部分之间,并返回其结果:

```
my $x = join ":", 4, 6, 8, 10, 12; #$x   为4:6:8:10:12"
```

列表上下文中的 m//

在列表 context 中使用模式匹配(m//)时,如果匹配成功返回值为内存变量值的列表;如果匹配失败则为空列表:

```
$_ = "Hello there, neighbor!";
my($first, $second, $third) =/(\S+) (\S+), (\S+)/;
print "$second is my $third\n" ;
```

在 s/// 中介绍的 /g 修饰符也可在 m// 中使用,它允许你在字符串中的多处进行匹配。在这里,由括号括起来的模式将在每一次匹配成功时返回其内存中所存放的值:

```
my $text = "Fred dropped a 5 ton granite block on Mr. Slate";
my @words = ($text =~ /([a-z]+)/ig);
print "Result: @words\n";
#Result: Fred dropped a ton granite block on Mr slate
```

如果有不止一对括号,每一次返回不止一个字符串。例如将字符串放入 hash 中,如下:

```
my $data = "Barney Rubble Fred Flintstone Wilma
Flintstone";
my %last_name = ($data =~ / (\w+)\S+(\w+)/g);
```

每当模式匹配成功时,将返回一对值。这些一对一对的值就成了 hash 中的 key/value 对。

更强大的正则表达式

非贪婪的数量词

各符号对应的非贪婪匹配:

```
+  -->  /fred.+?barney/
*  -->  s#<BOLD>(.*?)</BOLD>#$1#g;
{5,10}?
{8,}?
?  -->  ??
```

在命令行进行修改

```
$ perl -p -i.bak -w -e 's/Randall/Randal/g' fred*.dat
```

-p 要求 Perl 为你写一个程序。它算不上是一个完整的程序;看起来有些像下面的:

```
while(<>) {
    print;
}
```

更多控制结构

unless 控制结构

在 if 控制结构中,只有条件为真时,才执行块中的代码。如果你想在条件为假时执行,可以使用 unless:

```
unless ($fred =~ /^[A-Z_]\w*$/i) {
   print "The value of \$fred doesn't look like a Perl
identifier name.\n";
 }
```

unless 的含义是：除非条件为真，否则执行块中的代码。这和在 if 语句的条件表达式前面加上!(取反)所得结果是一样的。

另一种观点是，可以认为它自身含有 else 语句。如果不太明白 unless 语句，你可以把它用 if 语句来重写(头脑中，或者实际的重写）：

```
if ($fred =~ /^[A-Z_]\w*$/i) {

    #  什么也不做
} else {
    print "The value of \$fred doesn't look like a Perl
identifier name.\n";
 }
```

unless 和 else 一起

```
unless ($mon =~ /^Feb/) {
    print "This month has at least thirty days.\n";
} else {
    print "Do you see what's going on here?\n";
 }
```

也可以同时使用它们

until 控制结构

有时，希望将 while 循环的条件部分取反。此时，可以使用 until：

```
until ($j > $i) {
    $j *= 2;
 }
```

表达式修饰符

为了得到更紧凑的形式，表达式后可以紧接控制修饰语。如，if 修饰语可以像 if 块那样使用：

```
print "$n is a negative number.\n" if $n<0;
```

等同下面代码：

```
if ($n < 0) {
    print "$n is a negative number.\n";
 }
```

虽然被放在后面，条件表达式也是先被求值的。这和通常的的从左到右的顺序相反。要理解 Perl 代码，应当像 Perl 内部的编译器那样，将整个语句读完，来其具体的含义。

还有一些其它的修饰语：

```
&error ("Invalid input") unless &valid($input);
$i *= 2 unitl $i > $j;
print "", ($n += 2) while $n <10;
&greet($_) foreach @person;
```

The Naked Block 控制结构

被称为“裸的”块是指没有关键字或条件的块。假如有一个 while 循环，大致如下：

```
while(condition) {
    body;
    body;
    body;
 }
```

将关键字 while 和条件表达式去掉，则有了一个“裸的”块：

```
{
    body;
    body;
    body;
 }
```

“裸的”块看起来像 while 或 foreach 循环,除了它不循环外:它执行“循环体”一次,然后结束。它根本就没循环!

你也可能在其它地方使用过“裸的”块,这通常是为临时变量提供作用域:

```
{
    print "Please enter a number:";
    chomp(my $n = <STDIN>);

    my $root = sqrt $n;           #计算平方根
    print "The square root of $n is $root.\n";
}
```

elsif 语句

自增和自减 (同 C)

```
my @people = qw{ fred barney fred Wilma dino barney fred
pebbles};
```

```
my %count;                              #  新的空的h a s h

$count{$_}++ foreach @people;           #  根据情况创建新的 keys

和v a l u e s
```

第一次执行 foreach 循环时,$count{$_}自增 1。此时,$count{"fred"} 从 undef (由于之前 hash 中不存在)变成 1。第二次执行 foreach 循环时,$count{"barney"} 变成 1; 接着,$count{"fred"} 变成 2。每执行一次循环,%count 中的某个元素增 1,或者新元素被创建。循环结束时,$count{"fred"} 值为 3。这提供了一种查看某个元素是否存在于列表之中,以及其出现次数的方法。

for 控制结构 (同 C)

```
for (initialization; test; increment) {
  body;
  body;
}
```

foreach 和 for 的关系

对于 Perl 解析器(parser)而言, 关键字 foreach 和 for 是等价的。也就是说, Perl 遇见其中之一时, 和遇见另一个是一样的。Perl 通过括号能理解你的目的。如果其中有两个分号,则是 for 循环(和我们刚讲的类似); 如果没有,则为 foreach 循环:

```
for (1..10) {                                #实际上是f oreach

循环     从到 0
    print "I can count to $_!\n";
}
```

这实际是一个 foreach 循环,但写作 for。除了本例外,本书后面均写作 foreach。在实际代码中,你觉得 Perler 会输入这四个多余的字符吗?除了新手外,一般均写作 for,你也应当像 Perl 那样,通过查看括号内分号个数来判断。

循环控制

last 操作

last 会立刻结束循环。(这同 C 语言或其它语言中的 "break" 语句类似)。它从循环块中“紧急退出” 。当执行到 last,循环即结束,如下例 :

```
#  输出所有出现f red    的行   直到遇见_  _END_ _   标记
while(<STDIN>){
    if(/_ _ END_ _/){

        #这个标记之后不会有其它输入了
         last;
    }elsif(/fred/){
          print;
     }
}

##last    跳到这里 #
```

next 操作 (同 C 的中 continue)

redo 操作

循环控制的第三个操作是 redo。它会调到当前循环块的顶端,不进行条件表达式判断以及接着本次循环。(在 C 或类似语言中没有这种操作)下面是一个例子:

```
#    输入测试
my @words = qw{ fred barney pebbles dino Wilma betty };
my $errors = 0;
foreach(@words){

    ##redo    跳到这里##
    print "Type the word '$_' : ";
    chomp(my $try = <STDIN>);
    if($try ne $_){
        print "sorry -- That's not right.\n\n";
        $errors++;

        redo;                           #   跳转到循环顶端
     }
}
print "You've completed the test, with $errors errors.\n";
```

标签块

要给循环体加上标签,可在循环前面加上标签和冒号。在循环体内,可以根据需要在 last, next,或 n者 redo 后加上标签名,如:

```
LINE: while(<>){
    foreach (split){

    last LINE if /_ _END_ _/; #    退出LINE 循环
    ...
    }
}
```

逻辑操作符 && 和 ||

短路操作符

和 C(以及类似的语言)不同的地方是,短路操作的结果是最后被执行语句的返回值,而非仅仅是一个 Boolean 值。

```
my $last_name = $last_name{$someone} || '(No last name)';
```

三元操作符 ?:

当 Larry 决定 Perl 应当具有哪些操作符时,他不希望以前的 C 程序员在 Perl 中找不到 C 中出现过的操作符,因此他在 Perl 中实现了所有 C 的操作符。这意味着 C 中最容易混淆的操作符:三元操作符(?:)也被移植过来了。虽然它带来了麻烦,但有时也是非常有用的。

```
Express ? if_true_expr : if_false_expr
my $size =
 ($width < 10 ) ? "small":
 ($width < 20) ? "medium":
 ($width < 50) ? "large":
                 "extra_large"; #default
```

控制结构： 使用部分求值的操作符

前面三个操作符,均有一个共同的特殊性质:根据左侧的值(true 或 false),来判断是否执行右侧代码。有时会被执行,有时不会。由于这个理由,这些操作符有时叫做部分求值(partial-evaluation)操作符,因为有时并非所有的表达式均被执行。部分求值操作符是自动的控制结构。并非 Larry 想引入更多的控制结构到 Perl 中来。而是当他决定将这些部分求值操作符引进 Perl 后,它们自动成了控制结构。毕竟,任何可以激活或解除某块代码的执行即被称作控制结构。

文件校验

文件检测操作

如果程序会建立新的文件,在程序创建新文件之前,我们应先确定是否存在同名的文件,以免重要数据被覆盖掉。对于这种问题,我们可以使用 - 选项,检测是否存在相同名字的文件:

```
die "Oops! A file called '$filename' already exists.\n"
    if -e $filename;
```

如果文件在过去 28 天内都未被修改,输出警告:

```
warn "Config file is looking pretty old!\n"
```

```
  if -M CONFIG > 28;
```

下例首先检查文件列表，找到那些大于 100KB 的文件。如果一个文件仅是很大，我们不一定会将其移到备份磁带上去，除非同时其在最近 90 天内都未被访问。

```
my @ariginal_files = qw/ fred barney betty Wilma pebbles
dino bam-bamm /;

my @big_old_files; #  要到备份磁带上的文件
foreach my $filename (@original_files){
  push @big_old_files, $filename
     if -s $filename > 100_100 and -A $filename > 90;
}
```

-r	文件或目录对此(有效的)用户(effective user)或组是可读的
-w	文件或目录对此(有效的)用户或组是可写的
-x	文件或目录对此(有效的)用户或组是可执行的
-o	文件或目录由本(有效的)用户所有
-R	文件或目录对此用户(real user)或组是可读的
-W	文件或目录对此用户或组是可写的
-X	文件或目录对此用户或组是可执行的
-O	文件或目录由本用户所有
-e	文件或目录名存在
-z	文件存在,大小为 0(目录恒为 false)
-s	文件或目录存在,大小大于 0(值为文件的大小,单位:字节)
-f	为普通文本
-d	为目录
-l	为符号链接
-S	为 socket
-p	为管道(Entry is a named pipe(a "fifo"))
-b	为 block-special 文件(如挂载磁盘)
-c	为 character-special 文件(如 I/O 设备)
-u	setuid 的文件或目录
-g	setgid 的文件或目录
-k	File or directory has the sticky bit set
-t	文件句柄为 TTY(系统函数 isatty()的返回结果;不能对文件名使用这个测试)
-T	文件有些像"文本"文件
-B	文件有些像"二进制"文件
-M	修改的时间(单位:天)
-A	访问的时间(单位:天)
-C	索引节点修改时间(单位:天)

stat 和 lstat 函数

```
my ($dev, $ino, $mode, $nlink, $uid, $gid, $rdev, $size,
$atime, $mtime, $ctime, $blksize, $blockes)
    = stat($filename);
```

localtime 函数

```
my($sec, $min, $hour, $day, $mon, $year, $wday, $yday,
$isdst)
  = localtime $timestamp;
my $timestamp = 1180630098;
```

```
my $date = localtime $timestamp;
```

gmtime

```
my $now = gmtime; #  得到当前的时间
```

位操作

10 & 12	按位与;当操作数相应位均为 1 时结果为 1 (本例结果为 8)
10 竖线 12	按位或;当操作数中相应位有一个为 1 则结果为 1(本例结果为 14)
10 ^ 12	按位异或;当操作数中相应位有且仅有一个为 1,结果才为 1(本例结果 6)
6 << 2	位左移,将左边操作数左移右边操作数所指定的位数,被移出的位置(右边)补 0(结果为 24)
25 >> 2	位右移,将左边操作数右移动右边操作数所指定的位数,丢弃多余的位数(左边) (本例结果位 6)
~10	位取反,也叫做一元位补运算;其返回值为操作数中相应位取反的值(本例为 0Xffffff5,这和具体情况有关)

目录操作

在目录树上移动

```
chdir "/etc" or die "cannot chdir to /etc: $!";
```

由于这是系统请求,错误发生时将给变量$!赋值。通常应当检查$!的值,因为它将告诉你 chdir 失败的原因。

Globbing

通常,shell 将每个命令行中的任何的文件名模式转换成它所匹配的文件名。这被称作 globbing。例如,在 echo 命令后使用了文件名模式*.pm,shell 会将它转换成它所匹配的文件名:

```
$ echo *.pm
barney.pm dino.pm fred.pm wilma.pm
$
$ cat > show_args
foreach $arg (@ARGV) {
  print "one arg is $arg\n";
}
^D
$ perl show_args *
...
```

show-args 不需要知道如何进行 globbing,这些名字已经被处理后存在@ARGV 中了。

在 Perl 程序中也可以使用 Glob 匹配。

```
my @all_files = glob "*" ;
my @pm_files = glob "*.pm";
```

@all_files 得到了当前目录下的所有文件,这些文件按照字母排序的,不包括由点 (.)开头的文件。

Globbing 的替换语法

虽然我们任意的使用 globbing 这个术语,我们也谈论 glob 操作,但在许多使用 globbing 的程序中并没有出现 glob 这个字眼。为什么呢?原因是,许多这类代码在 glob 操作被命名前就写好了。它使用一对尖括号(<>), 和从文件句柄读入操作类似:

```
my @all_files = <*>;     #   基本上同 @all_files = glob "*" 一样
my $dir = "/etc";
my @dir_files = <$dir/* $dir/.*>;
```

目录句柄

从给定目录得到其文件名列表的方法还可以使用目录句柄(directory handle)。目录句柄外形及其行为都很像文件句柄。打开(使用 opendir 而非 open),从中读入(使用 readdir 而非 readline),关闭(使用 closedir 而非 close)。不是读入文件的内容,而是将一个目录中的文件名(以及一些其它东西)读入,如下例:

```
my $dir_to_process = "/etc";
opendir DH, $dir_to_process or die "Cannot open
$dir_to_process: $!";
foreach $file(readdir DH) {
    print "One file in $dir_to_process is $file\n";
}
closedir DH;
```

删除文件 unlink

```
unlink "slate", "bedrock", "lava";
```

这里有一个鲜为人知的 Unix 事实。你可以有一个文件不可读,不可写,不可执行,甚至你并不拥有它,例如属于别人的,但你仍然可以删除它。这时因为 unlink 一个文件的权限同文件本身的权限是没有关系的;它和目录所包含的文件权限有关。

重命名文件

```
rename "over_there/some/place/some_file", "some_file";
```

链接文件

要找出符号连接指向的地方,使用 readlink 函数。它会告诉你符号连接指向的地方,如果参数不是符号连接其返回 undef:

```
my $where = readlink "carroll";        # 得到"dodgson"

my $perl = readlink "/usr/local/bin/perl"  #  可能得到Perl放置的地方
```

创建和删除目录

```
mkdir "fred", 0755 or warn "Cannot make fred directory:
$!";

rmdir glob "fred/*";          #  删除fred/下面所有的空目录
```

修改权限

```
chmod 0755, "fred", "barney";
```

改变所有者

```
my $user = 1004;
my $group = 100;
chown $user, $group, glob "*.o";
```

如果要使用用户名:

```
defined(my $user = getpwnam "merlyn") or die "user bad";
defined(my $group = getprnam "users") or die "group bad";
chown $user, $group, glob "/home/Merlyn/*";
```

defined 函数验证返回值是否为 undef, 如果请求的 user 或 group 不存在,则返回 undef。chown 函数返回其改变的文件的个数,而错误值被设置在$!之中。

改变时间戳

```
my $now = time;

my $ago = $now -24*60*60;  #  一天的秒数

utime $now,$ago,glob "*"   #  设成当前访问的, 一天之前修改的
```

字符串和排序

使用索引寻找子串

```
$where = index($big, $small);
```

Perl 查找子串第一次在大字符串中出现的地方,返回第一个字符的位置。字符位置是从 0 开始编号的。如果子串在字符串的开头处找到,则 index 返回 0。如果一个字符后,则返回 1,依次类推。如果子串不存在,则返回-1。

使用 substr 操作子串

```
$part = substr($string, $initial_position, $length);
```

它有三个参数:一个字符串,一个从 0 开始编号的初始位置(类似于 index 的返回值),以及子串的长度。返回值是一个子串:

```
my $mineral = substr("Fred J. Flintstone", 8, 5);   #   得到" Flint"
my $rock = substr "Fred J. Flintstone", 13, 1000;  #   得到" stone"
my $pebble = substr "Fred J. Flintsone", 13;     #    同上,得到" stone"
my $out = substr ("some very long string", -3, 2);       #   $out   得到"in"
```

substr 选则的子串可以就地修改!

```
#!/usr/bin/perl -w
use strict;

my $string = "Hello, world!";
substr($string,6) = "Jian Lee!";

print $string . "\n";
```

最后输出为: “Hello,Jian Lee!”

使用 sprintf 格式化数据

sprintf 函数的参数和 printf 的参数完全相同(除了可选的文件句柄外), 但它返回的是被请求的字符串,而非打印出来。这对于希望将某个格式的字符串存入变量以供将来使用的情况非常方便,或者你想比 printf 提供的方法,更能控制结果:

```
my $data_tag = sprintf   "%4d/%02d/%02d %02d:%02d:%02d",
$yr, $mo, $da, $h, $m, $s;

rney = $fred / $dino };  #   即被除数为      也会终止
```