

第四章 语句和声明

- ↓ 第四章 语句和声明
 - ↓ 4.1 简单语句
 - ↓ 4.2 混合语句
 - ↓ 4.2.1 if 和 else 语句
 - ↓ 4.3 循环语句
 - ↓ 4.3.1 while 和 until 语句
 - ↓ 4.3.2 for 循环
 - ↓ 4.3.3 foreach 循环
 - ↓ 4.3.4 循环控制
 - ↓ 4.4 光块
 - ↓ 4.4.1 分支 (case) 结构
 - ↓ 4.5 goto
 - ↓ 4.6 全局声明
 - ↓ 4.7 范围声明
 - ↓ 4.7.1 范围变量声明
 - ↓ 4.7.2 词法范围的变量: my
 - ↓ 4.7.3 词法范围全局声明: our
 - ↓ 4.7.4 动态范围变量: local
 - ↓ 4.8 用法 (pragmas)
 - ↓ 4.8.1 控制警告
 - ↓ 4.8.2 控制全局变量的使用

一个 Perl 程序由一系列声明和语句组成。一个声明可以放在任何可以放语句的地方，但是它的主要作用发生在编译时。有几个声明类似语句，有双重身份，但是大多数在运行时是完全透明的。编译完之后，语句的主序列只执行一次。

和许多编程语言不同，Perl 并不要求明确的变量声明；变量只在第一次使用的时候才存在，不管你是否曾声明它们。如果你试图从一个从未赋值的变量里面获取一个值，当你把它当作数字时 Perl 会被悄悄地把它当 0 看待，而当作字符串时会把它当作 ""（空字符串），或者做逻辑值用的时候就是假。如果你喜欢在误把未定义的值用做真字符串或数字时收到警告，或者你更愿意把这样用当作错误，那么 `use warning` 声明会处理这一切；参阅本章末尾的“用法”节。

如果你喜欢的话，你可以在变量名前用 `my` 或 `our` 声明你的变量。你甚至可以把使用未声明变量处理为错误。这样的限制是好的，但你必须声明你需要这样的限制。通常，对你的编程习惯，Perl 只管自己的事，但是如果使用 `use strict` 声明，未定义的变量就会在编译时被了解。同样，参阅“用法”节。

4.1 简单语句

一个简单语句是一个表达式，因为副作用而计算。每条简单语句都必须以分号结尾，除非它是一个块中的最后语句。这种情况下，分号是可选的——Perl 知道你肯定已经完成语句了，因为你已经结束块了。但是如果是在多个块的结尾，那你最好还是把分号加上，因为你最后可能还是要另加一行。

虽然象 `eval{}`，`do{}`，和 `sub{}` 这样的操作符看起来象组合语句，其实它们不是。的确，它们允许在它们内部放多个语句，但是那不算数。从外部看，这些操作符只是一个表达式里的项，因此如果把它们用做语句中的最后一个项，则需要一个明确的分号结束。

任何简单语句的后面都允许跟着一条简单的修饰词，紧接着是结束的分号（或块结束）。可能的修饰词有：

```
if EXPR
unless EXPR
```

```
while EXPR
until EXPR
foreach LIST
```

if 和 **unless** 修饰词和他们在英语里的作用类似:

```
$trash->take('out') if $you_love_me;
shutup() unless $you_want_me_to_leave;
```

while 和 **until** 修饰词重复计算。如你所想, **while** 修饰词将不停地执行表达式, 只要表达式的值为真, 或者 **until** 里只要表达式为假则不断执行表达式。

```
$expresion++ while -e "$file$expression";
kiss('me') until $I_die;
```

foreach 修饰词 (也拼为 **for**) 为在其 **LIST** 里的每个元素计算一次, 而 **\$_** 是当前元素的别名:

```
s/java/perl/ for @resumes;
print "field: $_ \n" foreach split /:/, $dataline;
```

while 和 **until** 修饰词有普通的 **while** 循环的语意 (首先计算条件), 只有用于 **do BLOCK** (或者现在已经过时的 **do SUBROUTINE** 语句) 里是个例外, 在此情况下, 在计算条件之前, 先执行一次语句块。这样你就可以写下面这样的循环:

```
do {
    $line = <STDIN>
    ...
} until $line eq ".\n"
```

参考第二十九章, 函数, 里的三种不同的 **do** 入口, 还请注意我们稍后讲的循环控制操作符在这个构造中无法使用, 因为修饰词不接受循环标记。你总是可以在它周围放一些额外的 (花括弧) 块提前结束它, 或者在其内部放先行运行——就象我们稍后将在“光块”里描述的那样。或者你可以写一个内部带有多重循环控制的真正的循环。说到真正的循环, 我们下面要谈谈混合语句。

4.2混合语句

在一个范围 (注: 范围和名字空间在第二章, 集腋成裘, 里描述, 在“名字”节) 里的一个语句序列称之为一个块。有时候, 范围是整个文件, 比如一个 **required** 文件或者包含你的主程序的那个文件。有时候, 范围是一个用 **eval** 计算的字符串。但是, 通常来说, 一个块是一个用花括弧 (**{}**) 包围的语句体。当我们说到范围的时候, 我们的意思就是上面三种之一。当我们说一个带花括弧的块时, 我们会用术语 **BLOCK**。

混合语句是用表达式和 **BLOCK** 一起构造的。表达式是由项和操作符组成的。我们将在语法描述里用 **EXPR** 表示那些你可以使用任意标量表达式的地方。要表示一个表达式是在列表环境里计算的, 我们就用 **LIST**。

下面的语句可以用于控制 **BLOCK** 的条件和重复执行。(**LABEL** 部分是可选的。)

```
if (EXPR) BLOCK
if (EXPR) BLOCK else BLOCK
if (EXPR) BLOCK elsif (EXPR) BLOCK ...
if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK

unless (EXPR) BLOCK
unless (EXPR) BLOCK else BLOCK
unless (EXPR) BLOCK elsif (EXPR) BLOCK ...
```

```
unless (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK
```

```
LABEL while (EXPR) BLOCK
```

```
LABEL while (EXPR) BLOCK continue BLOCK
```

```
LABEL until (EXPR) BLOCK
```

```
LABEL until (EXPR) BLOCK continue BLOCK
```

```
LABEL for (EXPR; EXPR; EXPR) BLOCK
```

```
LABEL foreach (LIST) BLOCK
```

```
LABEL foreach VAR (LIST) BLOCK
```

```
LABEL foreach VAR (LIST) BLOCK continue BLOCK
```

```
LABEL BLOCK
```

```
LABEL BLOCK continue BLOCK
```

请注意和 **C** 及 **Java** 不同的是, 这些语句是根据 **BLOCK** 而不是根据语句定义的。这就意味着花括弧是必须的 —— 不允许有虚悬的语句。如果你想不带圆括弧写条件, 可以有若干种处理方法。下面的语句作用相同:

```
unless (open(FOO, $foo))    {die "Can't open $foo: $!" }
if(!open(FOO, $foo))       {die "Can't open $foo: $!" }

die "Can't open $foo: $!"   unless open(FOO, $foo);
die "Can't open $foo: $!"   if !open(FOO, $foo);

open(FOO, $foo)             || die "Can't open $foo: $!";
open(FOO, $foo)             or die "Can't open $foo: $!";
```

在大多数情况下, 我们都建议使用最后一对儿。这种形式看着整齐一点, 尤其是 **"or die"** 版本。而用 **||** 形式时你必须习惯虔诚地使用圆括弧, 而如果用 **or** 版本, 即使你忘了也用不着担心。

不过我们喜欢最后一种形式的原因是它把语句里重要的部分放到行的前面, 这样你会先看到它们。错误控制部分挪到了边上, 这样除非必要的时候, 你用不着注意它们。如果你每次都把所有 **"or die"** 检查放到右边同一行, 那就很容易读了:

```
chdir $dir                or die "chdir $dir: $!";
open FOO, $file            or die "open $file: $!";
@lines = <FOO>             or die "$file is empty?";
close FOO                 or die "close $file: $!";
```

4.2.1 if 和 else 语句

if 语句比较直接了当。因为 **BLOCK** 们总是用花括弧包围, 所以从不会出现混淆哪一个 **if** 和 **else** 或 **elsif** 有效的情况。在给出的任意 **if/elsif/else BLOCK** 里, 只有第一个条件 **BLOCK** 才执行。如果没有一个为真, 而且存在 **else BLOCK**, 则执行之。一个好习惯是在一个 **elsif** 链的最后放上一个 **else** 以捕捉漏掉的情况。

如果你用 **unless** 取代 **if**, 那么它的测试是相反的, 也就是说:

```
unless ($x == 1) ...
```

等效于

```
if($x != 1) ...
```

或者是难看的:

```
if(!($x == 1)) ...
```

在控制条件里定义的变量，其范围只扩展到其余条件的范围，包括任何随后可能存在的 **elsif** 和 **else** 子句，但是不会超过这个范围:

```
if (( my $color = <STDIN> ) =~ /red/i ) {
    $value = 0xff0000;
}
else ($color =~ /green/i) {
    $value = 0x00ff00;
}
else if ($color =~ /blue /i ){
    $value = 0x0000ff;
}
else {
    warn "unknown RGB component `$color', using black instead\n";
    $value = 0x000000;
}
```

在 **else** 以后，**\$color** 变量将不再位于范围之内。如果你想把范围扩大一些，那么请在条件之前定义该变量。

4.3 循环语句

所有循环语句在它们正式语法里有可选的 **LABEL** (标记)。(你可以在任何语句里放上标记，但是它们对循环有特殊含义。)如果有标记，它由一个标识后面跟着一个冒号组成。通常标记都用大写以避免与保留字冲突，并且这样它也比较醒目。同时，尽管 **Perl** 不会因为你使用一个已经有含义的词做标记 (比如 **if** 和 **open**) 而晕菜，但你的读者却是会的，所以 ...。

4.3.1 while 和 until 语句

while 语句在 **EXPR** (表达式) 为真的时候不断执行语句块。如果 **while** 被 **until** 取代，那么条件测试的取向就变反了；也就是说，它只有在 **EXPR** 为假的时候才执行语句块。不过，在第一个执行文本之前先要测试条件。

while 和 **until** 语句可以有一个额外的块：**continue** 块。这个块在整个块每继续一次都执行一次，不管是退出第一个块还是用一个明确的 **next** (一个循环控制操作符，它控制进入下一句文本)。在实际中 **continue** 块用得并不多，但是有它可以让我们严格地定义下一节里的 **for** 循环。

和我们稍后将看到的 **foreach** 循环不同的是，一个 **while** 循环从不在它的测试条件里隐含地局部化任何变量。这种特性在 **while** 循环使用全局量做循环变量时可能会有“很有趣”的结果。尤其是你可以看看第二章“行输入 (尖角) 操作符”里关于在某些 **while** 循环里是如何隐含地给全局的 **\$_** 赋值的例子，以及如何如何明确地局部化 **\$_** 的例子。不过，对于其他循环变量来说，你最好象我们下一个例子那样用 **my** 定义它们。

在一个 **while** 或者 **until** 语句的测试条件里定义的变量只是在该语句块或由该测试条件控制的语句块中可见。它可不属于任何周围的语句块的范围。比如：

```
while (my $line = <STDIN>) {
    $line = lc $line;
}
continue {
```

```

    print $line;    # 仍然可见
}
# $line现在超出范围外了

```

这里的 `$line` 的范围从它在控制表达式里定义开始一直延伸到循环构造的其他部分，包括 `continue` 语句块，但是不再超出该范围。如果你想把范围扩得更大，请在循环之前定义该变量。

4.3.2 for循环

分成三部分的 `for` 循环在其圆括弧里有三个用分号隔离的表达式。这些表达式分别代表循环的初始化，条件和再初始化表达式。所有三个表达式都是可选的（不过分号不是）；如果省略了它们，则条件总是为真。因此三部分的 `for` 表达式可以用对应的 `while` 循环来代替。下面这样的：

```

LABEL:
    for( my $i = i; $i <= 10; $i++ ) {
        ...
    }

```

和

```

{
    my $i =1;
    LABEL:
        while ($i <= 10 ){
            ...
        }
        continue {
            $i++;
        }
}

```

是一样的，只不过实际上没有外层的块。（我们在这里放一个只是为了显示 `my` 的范围是如何被限制的。）

如果你想同时使用两个变量，只需要用逗号分隔平行的表达式即可：

```

for ( $i = 0, $bit = 0; $i < 32; $i++, $bit <=>1) {
    print "Bit $i is set\n" if $ mask & $bit;
}
# $i 和 $bit里的值超越循环继续存在

```

或者把变量定义在只有 `for` 循环里可见：

```

for (my ($i, $bit) = (0, 1); @i < 32; $i ++, $bit <=>1 ) {
    print "Bit $i is set \n" if $mask & $bit;
}
# 循环版本的$i和$bit现在超出范围了

```

除了通常用于数组索引循环外，`for` 还可以把自身借给许多其他感兴趣的应用。它甚至不需要明确的循环变量。下面是一个这样的例子，这个例子可以避免你在明确地测试一个交互的文件描述符的文件结束符（EOF）的时候导致的程序的挂起。

```

$on_a_tty = -t STDIN && -t STDOUT;
sub prompt {print "yes?" if $on_a_tty }
for ( prompt(); <STDIN>; prompt() ) {
    # 处理一些事情
}

```

```
}
```

三部分 **for** 的另外一个传统应用源自其所有三个表达式都是可选的，而且缺省为真这样的一个特点。如果你省略的所有三个表达式，那么你实际上写了一个无限循环：

```
for(;;) {
    ...
}
```

上面这样等效于写：

```
while (1) {
    ...
}
```

如果你因为无限循环的表示法而烦恼，那我们要指出的是你总是可以在任何一点退出循环，只需要明确的使用一个象 **last** 这样的循环控制操作符即可，当然，如果你为巡航导弹写代码，你就可能不用什么明确的循环退出了。因为循环会在合适的时间自动退出。（译注：导弹掉下来就...）

4.3.3 foreach 循环

foreach 循环通过把控制变量（**VAR**）设置为每个后继的列表元素来循环通过一系列数值：

```
foreach VAR (LIST) {
    ...
}
```

foreach 键字只是 **for** 键字的一个同义词，所以，只要你觉得哪个可读性更好，你就可以互换地使用 **for** 和 **foreach**。如果省略 **VAR**，则使用全局 **\$_** 变量。（别担心，Perl 可以很容易地区分 **for** (**@ARGV**) 和 **for** (**\$i=0; \$i<\$#ARGV;\$i++**)，因为后者包含分号。）下面是一些例子：

```
$sum = 0; foreach $value (@array) { $sum += $value }

for $count (10,9,8,7,6,5,4,3,2,1,'BOOM') { # 倒计时
    print "$count\n"; sleep(1);
}

for (reverse 'BOOM', 1 .. 10) { # 一样的东西
    print "$_\n"; sleep(1);
}

for $field (split /\:/, $data) { # 任何 LIST 表达式
    print "Field contains: `$field'\n";
}

foreach $key (sort keys %hash) {
    print "$key => $hash{$key}\n";
}
```

最后一句是打印一个排了序的散列数组的规范的方法。参阅第二十九章里的键字和排序记录获取更详细的例子。

在 **foreach** 里，你没有办法获知你位于列表的何处。你可以通过把前面的元素保留在一个变量里，然后用相邻的元素与之进行比较来获取当前位置，但是有时候你就是得用一个带脚标的三部分 **for** 循环来获取当前位置。毕竟，**for** 循环还是有其特点的。

如果 **LIST** 完全包含可赋值元素（通常也就是说变量，而不是枚举常量），你可以通过修改循环内的

VAR 来修改每个变量。这是因为 **foreach** 循环的索引变量隐含地是你正在逐一取出的列表的每个元素的别名。你不仅可以现场修改单个列表，你还可以修改在一个列表里的多个数组和散列：

```
foreach $pay (@salaries) {      # 赋予 8%的提升
    $pay *= 1.08;
}

for (@christmas, @easter) {
    s/ham/turkey/;              # 修改菜单（译注：这可真的是菜单）
}

s/ham/turkey/ for @christmas, @easter;    # 和上面一样的东西

for ($scalar, @array, values %hash) {
    s/^\s+//;                  #删除开头的空白
    s/\s+$//;                  #删除结尾的空白
}
```

循环变量只在循环的动态或者词法范围内有效。如果该变量事先用 **my** 定义，那么它隐含地是在词法范围里。这样，对于任何在词法范围之外定义的函数它都是不可见的，即使是在循环里调用的函数也看不到这个变量。不过，如果在范围里没有词法定义，那么循环变量就是局部化的（动态范围）全局变量；这样就允许循环内调用的函数访问它。另外，循环变量在循环前拥有的值将在循环退出之后自动恢复。

如果你愿意，你可以明确地声明使用哪种变量（词法或全局）。这样让你的代码的维护人员能够比较容易理解你的代码；否则，他们就得在一个封闭的范围里往回找，看看该变量在前面声明为什么：

```
for my $i (1 .. 10) { ... }    # $i总是在词法范围
for our $Tick (1.. 10) { ...}  # $Tick 总是全局
```

当定义伴随着循环变量时，短些的 **for** 比 **foreach** 要好，因为它更符合英文的阅读习惯。

下面是一个 **C** 或者 **JAVA** 程序员在使用 **Perl** 表示某种算法时首先会想到的代码：

```
for ( $i = 0; $i < @ary1; $i++) {
    for ( $j=0; $j <@ary2; $j++) {
        if($ary1[$i] > $ary2[$j]) {
            last;          # 没法到外层循环 :- (
        }
        $ary1[$i] += $ary2[$j];
    }
    # 这里是last把我带到的地方
}
```

而这里是老练的 **Perl** 程序员干的：

```
WID: foreach $this (@ary1) {
    JET: foreach $that (@ary2) {
        next WID if $this > $that;
        $this += $that;
    }
}
```

看看，用合乎 **Perl** 习惯的语法是多简单？这样更干净，更安全，更快。更干净是因为它较少代码。更安全是因为代码在内层和后面的外层循环之间做加法；第二段代码不会被意外地执行，因为 **next**（下面解释）明确地执行外层循环而不只是简单地退出内层。更快是因为 **Perl** 在执行 **foreach** 语句

时比等效的 `for` 循环快，因为元素是直接访问的，而不是通过下标来访问的。

当然，你熟悉哪个就用哪个。要知道“回字有四种写法”。

和 `while` 语句相似，`foreach` 语句也可以有一个 `continue` 块。这样就允许你在每个循环之后执行一小段代码，不管你是用普通的方法到达那里还是用一个 `next` 到达的。

现在我们可以说 `next` 就是“下一个”。

4.3.4 循环控制

我们说过，你可以在一个循环上放一个 `LABEL`（标记），这样就给它一个名字。循环的 `LABEL` 为循环的循环控制操作符 `next`，`last`，和 `redo` 标识循环。`LABEL` 是给整个循环取名，而不仅仅是循环的顶端。因此，一个引用了循环（标记）的循环控制操作符实际上并不是“`go to`”（跑到）循环标记本身。就计算机而言，该标记完全可以放在循环的尾部。但是不知什么原因人们喜欢把标记放在东西的顶部。

循环的典型命名是命名为每次循环所处理的项目。这样和循环控制操作符交互地很好，因为循环控制操作符的设计原则是：当合适的标记和语句修饰词一起使用时，它读起来应该象英文。如果循环原型是处理行的，那循环标记原型就是 `LINE:`，因此循环控制操作符就是类似这样的东西：

```
next LINE if / ^#/;    # 丢弃注释
```

循环控制操作符的语法是：

```
last LABEL
next LABEL
redo LABEL
```

`LABEL` 是可选的；如果忽略了，该操作符就选用最内层的封闭循环。但是如果你想跳过比一层多的（循环），那你就必须用一个 `LABEL` 以命名你想影响的循环。`LABEL` 不一定非在你的词法范围内不可（尽管它应该在词法范围里）。实际上，`LABEL` 可以在你的动态范围的任何地方。如果它的位置强制你的跳转超出了一个 `eval` 或者子过程的范围，Perl（会根据需要）发出一个警告。

就象在一个函数里你可以想要几个 `return` 就要几个 `return` 一样，你也可以在循环里想要几个循环控制语句就要几个。在早期的结构化编程的日子里，有些人坚持认为循环和子过程只能有一个入口和一个出口。单入口的表示法是个好主意，可单出口的想法却让我们写了许多非自然的代码。许多程序都包括在决策树里漫游的问题。一个决策树通常从一个树干开始，但通常以许多叶子结束。把你的循环退出代码（还有函数返回代码）写成具有多个出口是解决你的问题的很自然的做法。如果你在合适的范围里定义你的变量，那么所有东西在合适的时候都会被清理干净，不管你是如何退出该语句块的。

`last` 操作符立即退出当前循环。如果有 `continue` 块也不会执行。下面的例子在第一个空白行撤出循环：

```
LINE: while (<STDIN>) {
    last LINE if / ^$/;    # 当处理完邮件头后退出
}
```

`next` 操作符忽略当前循环的余下的语句然后开始一次新的循环。如果循环里有 `continue` 子句，那它将在重新计算条件之前执行，就象三部分的 `for` 循环的第三个部件一样。因此 `continue` 语句可以用于增加循环变量——即使是在循环的部分语句被 `next` 终止了的情况下：

```
LINE: while (<STDIN>) {
    next LINE if / ^#/;    # 忽略注释
    next LINE if / ^$/;    # 忽略空白行
}
```



```
...
} continue {
    $count++;
}
```

redo 操作符在不重新计算循环条件的情况下重新开始循环语句块。如果存在 **continue** 块也不会执行。这个操作符通常用于那些希望在刚输入的东西上耍点小伎俩的程序。假设你正在处理这样的一个文件，这个文件里你时不时要处理带有反斜杠续行符的行。下面就是你如何利用 **redo** 来干这件事：

```
while (<>) {
    chomp;
    if (s/\\$//) {
        $_ .= <>;
        redo unless eof;      # 不超过每个文件的eof
    }
    # 现在处理 $_
}
```

上面的代码是下面这样更明确（也更冗长）的 **Perl** 代码的缩写版本：

```
LINE: while (defined($line = <ARGV>)) {
    chomp($line);
    if ($line =~ s/\\$//) {
        $line .= <ARGV>;
        redo LINE unless eof(ARGV);
    }
    # 现在处理$line
}
```

下面是一段真实的代码，它使用了所有三种循环控制操作符。因为有了 **Getopts::*** 模块后，现在我们不常用下面这样的方法分析命令行参数，但是它仍然是一个在命名的嵌套循环上使用循环控制操作符的好例子：

```
ARG: while (@ARGV && $ARGV[0] =~ s/^(?=.)//) {
    OPT: for (shift @ARGV) {
        m/^\$/      && do {                                next ARG; };
        m/^-$/      && do {                                last ARG; };
        s/^\d//     && do { $Debug_Level++;                redo OPT; };
        s/^\l//     && do { $Generate_Listing++;           redo OPT; };
        s/^\i(.*)// && do { $In_Place = $1 || ".bak";      next ARG; };
        say_usage("Unknown option: $_");
    }
}
```

还有一个关于循环控制操作符的要点。你可能已经注意到我们没有把它们称为“语句”。那是因为它们不是语句——尽管和其他表达式一样，它们可以当语句用。你甚至可以把它们看作那种只是导致控制流改变的单目操作符。因此，你可以在表达式里任何有意义的地方使用它们。实际上，你甚至可以在它们没有意义的地方使用它们。我们有时候看到这样的错误代码：

```
open FILE, $file
    or warn "Can't open $file: $!\n", next FILE; # 错
```

这样做的目的是好的，但是 **next FILE** 会被分析为 **warn** 的一个参数。所以是在 **warn** 获得发出警告的机会之前先执行 **next**。这种情况，我们很容易用一些圆括弧通过把 **warn** 列表操作符转换成 **warn** 函数来修补这个错误：

```
open FILE, $file
    or warn( "Can't open $file: $! \n"), next FILE; # 对了
```

不过, 你会觉得下面的更好读:

```
unless(open FILE, $file) {
    warn "Can't open $file: $!\n";
    next FILE;
}
```

4.4 光块

一个 **BLOCK** 本身（带标记或者不带标记）等效于一个执行一次的循环。所以你可以用 **last** 退出一个块或者用 **redo** 重新运行块（注：相比之下，**next** 也退出这种一次性的块。不过有点小区别：**next** 会执行一个 **continue** 块，而 **last** 不会。）。不过请注意，对于 **eval{}**，**sub{}** 或者更让人吃惊的是 **do{}** 这些构造而言，情况就不一样了。这哥仨不是循环块，因为它们自己就不是 **BLOCK**；它们前面的键字把它们变成了表达式中的项，只不过碰巧包含一个语句块罢了。因为它们不是循环块，所以不能给它们打上标记用以循环控制等用途。循环控制只能用于真正的循环，就象 **return** 只能用于子过程（或者 **eval**）一样。

循环控制也不能在一个 **if** 或 **unless** 里运行，因为它们也不是循环。但是你总是可以引入一付额外的花括弧，这样就有了一个光块，而光块的确是一个循环：

```
if ( /pattern/) {{
    last if /alpha/;
    last if /beta/;
    last if /gamma/;
    # 在这里处理一些只在if()里处理的事情
}}
```

下面是如何利用一个光块实现在 **do{}** 构造里面使用循环控制操作符的例子。要 **next** 或 **redo** 一个 **do**，在它里面放一个光块：

```
do {{
    next if $x == $y;
    # 在这处理一些事务
}} until $x++ > $z;
```

对于 **last** 而言，你得更仔细：

```
{
    do {
        last if $x = $y ** 2;
        # 在这里处理一些事务
    }while $x++ <= $z;
}
```

如果你想同时使用两种循环控制，那你就得在那些块上放上标记，以便区分它们：

```
DO_LAST:{
    do {
        DO_NEXT:    {
            next DO_NEXT if $x == $y;
            last DO_LAST if $x = $y ** 2;
            # 在这里处理一些事务
```

```

    }
    }while $x++ <= $z;
}

```

不过就这个例子而言, 你还是用一个在末尾有 **last** 的普通的无限循环比较好:

```

for (;;) {
    next if $x == $y;
    last if $x = $y ** 2;
    # 在这里处理一些事务
    last unless $x++ <= $z;
}

```

4.4.1 分支 (case) 结构

和其他一些编程语言不同的是, Perl 没有正式的 **switch** 或者 **case** 语句。这是因为 Perl 不需要这样的东西, 它有很多方法可以做同样的事情。一个光块就是做条件结构 (多路分支) 的一个很方便的方法。下面是一个例子:

```

SWITCH: {
    if (/^abc/)    { $abc = 1; last SWITCH; }
    if (/^def/)    { $def = 1; last SWITCH; }
    if (/^xyz/)    { $xyz = 1; last SWITCH; }
    $nothing = 1;
}

```

这里是另外一个:

```

SWITCH: {
    /^abc/    && do { $abc = 1;    last SWITCH; };
    /^def/    && do { $def = 1;    last SWITCH; };
    /^xyz/    && do { $xyz = 1;    last SWITCH; };
}

```

或者是把每个分支都格式化得更明显:

```

SWITCH: {
    /^abc/    && do {
        $abc = 1;
        last SWITCH;
    };
    /^def/    && do {
        $def = 1;
        last SWITCH;
    };
    /^xyz/    && do {
        $xyz = 1;
        last SWITCH;
    };
}

```

甚至可以是更恐怖的:

```

if (/^ac/) { $abc = 1 }
elseif (/^def/) { $def = 1 }
elseif (/^xyz/) { $xyz = 1 }

```

```
else          {$nothing = 1}
```

在下面的例子里，请注意 **last** 操作符是如何忽略并非循环的 **do{}** 块，并且直接退出 **for** 循环的：

```
for ($very_nasty_long_names[$i++][$j++]->method()) {
    /this pattern/    and do { push @flags, '-e'; last; };
    /that one/       and do { push @flags, '-h'; last; };
    /something else/ and do {          last; };
    die "unknown value: `$_'";
}
```

只对单个值进行循环，可能你看起来有点奇怪，因为你只是走过循环一次。但是这里利用 **for/foreach** 的别名能力做一个临时的，局部的 **\$_** 赋值非常方便。在与同一个很长的数值进行重复地比较的时候，这样做更容易敲键而且更不容易敲错。这样做避免了再次计算表达式的时候可能出现的副作用。并且和本章相关的是，这样的结构也是实现 **switch** 或 **case** 结构最常见最标准的习惯用法。

?: 操作符的级联使用也可以起到简单的分支作用。这里我们再次使用 **for** 的别名功能，把重复比较变得更清晰：

```
for ($user_color_preference) {
    $value = /red/      ? 0xff0000:
            /green/    ? 0xff0000:
            /blue/     ? 0x0000ff:
            0x000000;   # 全不是用黑色
}
```

对于最后一种情况，有时候更好的方法是给自己建一个散列数组，然后通过索引快速取出结果。和我们刚刚看到的级联条件不同的是，散列可以扩展为无限数量的记录，而且查找第一个比查找最后一个的时间不会差到哪儿去，缺点是只能做精确匹配，不能做模式匹配。如果你有这样的散列数组：

```
%color_map = (
    azure      => 0xF0FFFF,
    chartreuse => 0x7FFF00,
    lavender   => 0xE6E6FA,
    magenta    => 0xFF00FF,
    turquoise  => 0x40E0D0,
);
```

那么精确的字串查找跑得飞快：

```
$value = $color_map{ lc $user_color_preference } || 0x000000;
```

甚至连复杂的多路分支语句（每个分支都涉及多个不同语句的执行）都可以转化成快速的查找。你只需要用一个函数引用的散列表就可以实现这些。参阅第九章，数据结构，里的“函数散列”获取如何操作这些的信息。

4.5 goto

尽管不是想吓唬你（当然也不是想安慰你），Perl 的确支持 **goto** 操作符。有三种 **goto** 形式：**goto LABEL**，**goto EXPR**，和 **goto &NAME**。

goto LABEL 形式找出标记为 **LABEL** 的语句并且从那里重新执行。它不能用于跳进任何需要初始化的构造，比如子过程或者 **foreach** 循环。它也不能跳进一个已经优化了的构造（参阅第十八章，编译）。除了这两个地方之外，**goto** 几乎可以用于跳转到当前块的任何地方或者你的动态范围（就是说，一个调用你的块）的任何地方。你甚至可以 **goto** 到子过程外边，不过通常还有其他更好的构造

可用。Perl 的作者从来不觉得需要用这种形式的 `goto`（在 Perl 里就是这样——C 就单说了）。

`goto EXPR` 形式只是 `goto LABEL` 的一般形式。它期待表达式生成一个标记名称，这个标记名称显然可以由分析器动态地解释。这样允许象 **FORTRAN** 那样计算 `goto`，但是如果你为了保持可维护性，我们建议你还是不要这么做：

```
goto(("FOO", "BAR", "GLARCH")[$i]);      # 希望0<=i <3

@loop_label = qw/FOO BAR GLARCH/;
goto $loop_label[rand @loop_label];      # 随机端口
```

几乎在所有类似这样的例子中，通常远比这种做法更好的方法是使用结构化的 `next`，`last`，或 `redo` 等流控制机制，而不是用这样的 `goto`。对于某些应用，一个函数引用的散列或者是 `eval` 和 `die` 构造的例外捕获-抛出对也是很不错的解决方法。

`goto &NAME` 形式非常神奇，它卓有成效地消灭了传统的 `goto` 的使用，令那些使用 `goto` 的用户免于惨遭批判。它把正在运行着的子过程替换为一个对命名子过程的调用。这个特性被 **AUTOLOAD** 子过程用于装载其它子过程，然后假装是那些子过程先被调用的。`autouse`，`AutoLoader`，和 [SelfLoader](#)² 模块都是用这个方法在函数头一次被调用的时候定义这些函数，然后跳到那些函数里，而我们谁都不知道这些函数实际上不是一开始就是在那里的。

4.6 全局声明

子过程和格式声明是全局声明。不管你把它们放在哪里，它们声明的东西都是全局的（对包而言是局部的，但是包对程序而言是全局的，所以包里面的任何东西在任何地方都可见）。全局声明可以放在任何可以出现语句的地方，不过它们对语句的主要执行顺序没有影响--声明只在编译时起作用。

这意味着你不能做条件的子过程和/或格式声明。你可能会想用运行时的 `if` 来屏蔽你的声明，使之不为编译器所见，但这是不可能的，因为只有解释器关心那些条件。不管出现在什么地方，子过程和格式声明（还有 `use` 和 `no` 声明）都只有编译器能够看到。

全局声明通常出现在你的程序的开头或者结尾，或者是放在其它的文件里。不过，如果你声明的是词法范围的变量（见下节），而且你还希望你的格式或者子过程能够访问某些私有变量，那你就得保证你的声明落在这些变量声明的范围之内。

请注意我们偷偷地从讲声明转移到了讲定义。有时候，把子过程的声明和定义分开能帮我们忙。这两个概念语义上的唯一的区别是定义包含一个要执行的代码块 **BLOCK**，而声明没有。（如果一个子过程没有声明部分，那么它的定义就是它的声明。）把定义从声明里面剥离，就允许你把子过程的声明放在开头而把其定义放在后面（而你的词法范围的变量声明放在中间）：

```
sub count (@);      # 现在编译器知道如何调用 count()。
my $x;              # 现在编译器知道词法变量
$x = count(3,2,1);  # 编译器可以核实函数调用
sub count (@) { @_ } # 现在编译器知道 count() 是什么意思了
```

正如上面例子显示的那样，子过程在调用它们之前不用定义也能编译，（实际上，如果你使用自动装载技术，定义甚至可以推迟到首次调用），但是声明子过程可以以各种方式协助编译器，并且给你更多调用它们的选择。

声明一个子过程后允许你不带圆括弧使用它，好象它是一个内建的操作符一样。（我们在上面的例子里用了圆括弧，实际上我们可以不用。）你可以只声明而不定义子过程，只需：

```
sub myname;
$me = myname $0      or die "can't get myname";
```

这样的空定义把函数定义成一个列表操作符，但不是单目操作符，所以上面用了 **or** 而不是

。操作符

和列表操作符之间的联系太紧密了，以至于基本上只能用于列表操作符后面，当然，你还是可以在列表操作符参数周围放上圆括弧，让列表操作符表现得更加象函数调用来解决这个问题。另外，你可以用原型 (**\$**) 把子过程转化成单目操作符：

```
sub myname ($);  
$me = myname $0 || die "can't get myname";
```

这样就会按照你想象的那样分析了，不过你还是应该养成在这种情况下用 **or** 的习惯。有关原型的更多内容，参阅第六章，子过程。

有时候你的确需要定义子过程，否则你在运行时会收到一个错误，说你调用了没有定义的子过程。除了自己定义子过程外，还有几个方法从其它地方引入定义。

你可以用简单的 **require** 语句从其它文件装载定义，在 **Perl 4** 里，这是装载文件的最好方法，但是这种方法有两个问题。首先，其他文件通常会向一个它们自己选定的包（一个符号表）里插入子过程名，而不是向你的包里插。其次，**require** 在运行时起作用，这对调用它起声明作用的文件来说有点太晚了。不过，有时候你要的就是推迟的装载。

引入声明和定义的更好的办法是使用 **use** 声明，它可以在编译时就 **require** 各模块（因为 **use** 算做 **BEGIN** 块），然后你就可以把一些模块的声明引入到你的程序里面来了。所以可以把 **use** 看成某种类型的全局声明，因为它在编译时把名字输入到你自己的（全局）包里面，就好像你是自己声明的一样。参阅第十章，包，的“符号表”一节，看看包之间的传输运做的低层机制；第十一章，模块，看看如何设置一个模块的输入和输出；以及第十八章，看看 **BEGIN** 和它的表兄弟 **CHECK**，**INIT**，和 **END** 的解释。它们在某种程度上也是全局声明，因为它们编译是做处理，而且具有全局影响。

4.7 范围声明

和全局声明类似，词法范围声明也是在编译时起作用的。和全局声明不同的是，词法范围声明的作用范围是从声明开始到闭合范围的最里层（块，文件，或者 **eval--**以先到者为准）。这也是为什么我们称它为词法范围，尽管“文本范围”可能更准确些，因为词法范围这个词实在和词法没什么关系。但是全世界的计算机科学家都知道“词法范围”是什么意思，所以在这里我们还是用这个词。

Perl 还支持动态范围声明。动态范围同样也伸展到最里层的闭合块，但是这里的“闭合”是运行时动态定义的，而不是象文本那样在编译时定义。用另外一种方式来说，语句块通过调用其他语句块实现动态地嵌套，而不是通过包含其他语句块来实现嵌套。这样的动态嵌套可能在某种程度上和嵌套的文本范围相关，但是这两者通常是不一样的，尤其是在调用子过程的时候。

我们曾经说过 **use** 的一些方面可以认为是全局声明，但是 **use** 的其他方面却是词法范围的。特别是，**use** 不仅输入包的符号，而且还实现了许多让人不可思议的编译器暗示，也就是我们说的用法（**pragmas**）。大多数用法是词法范围的，包括 **use strict 'vars'** 用法，这个用法强制你在使用前先声明变量。参阅后面的“用法”节。

很有意思的是，尽管包是一个全局入口，包声明本身是词法范围的。但是一个 **package** 声明只是为闭合块的余下部分声明此缺省包的身份。**Perl** 会到这个包中查找未声明的，未修饰的变量名（注：还有未定义的子过程，文件句柄，目录句柄和格式）。换句话说，实际上从来没有声明什么包，只是当你引用了某些属于那些包的东西的时候才突然出现。当然这就是 **Perl** 的风格。

4.7.1 范围变量声明

本章剩下的大部分内容是关于使用全局变量的。或者换句话说，是关于‘不’使用全局变量的。有各

种各样的声明可以帮助你不使用全局变量——或者至少不会愚蠢地使用它们。

我们已经提到过 **package** 定义，它在很早以前就引入 Perl 了，这样就允许全局量可以分别放到独立的包里。对于某些变量来说，这个方法非常不错。库，模块和类都用包来存储它们的接口数据（以及一些它们的半私有数据）以避免和你的主程序或者其他模块的变量或者函数冲突。如果你看到某人写到 **\$Some::stuff**（注：或者 **\$Some'sstuff**，不过我们不鼓励这么写），他们是在使用来自包 **Some** 的标量变量 **\$stuff**。参阅第十章。

如果这么干活的话，Perl 程序随着变量的增长会很快变得不好用。好在 Perl 的三种范围声明让它很容易做下面这些事：创建私有变量（用 **my**），进行有选择地访问全局变量（用 **our**），和给全局变量提供临时的值（用 **local**）：

```
my $nose;
our $House;
local $TV_channel;
```

如果列出多于一个变量，那么列表必须放在圆括弧里。就 **my** 和 **our** 而言，元素只能是简单的标量，数组或者散列变量。就 **local** 而言，其构造可以更宽松：你还可以局部化整个类型团和独立的变量或者数组和散列的片段：

```
my($nose, @eyes, %teeth);
our ($House, @Autos, %Kids);
local (*Spouse, $phone{HOME});
```

上面每种修饰词都给它们修饰的变量做出某种不同类型的“限制”。简单说：**our** 把名字限于一个范围，**local** 把值限于一个范围以及 **my** 把名字和值都限于一个范围。

这些构造都是可以赋值的，当然它们对值的实际处理是不同的，因为它们有不同的存储值的机制。如果你不给它们赋任何值（象我们上面那样），它们也有一些区别：**my** 和 **local** 把涉及的变量初始化为 **undef** 或 **()**，另一方面，**our** 不修改与之相联的全局变量的当前值。

从语义上来讲，**my**，**our** 和 **local** 都只是简单的左值表达式的修饰词（类似形容词）。当你给一个被修饰的左值赋值时，修饰词并不改变左值是标量状态还是列表状态。想判断赋值将按照什么样的方式运行，你只要假设修饰词不存在就行了。所以：

```
my ($foo) = <STDIN>;
my @array = <STDIN>;
```

给右手边提供了一个列表环境，而：

```
my $foo = <STDIN>;
```

提供了一个标量环境。

修饰词比逗号绑定得更紧密（也有更高优先级）。下面的例子错误地声明了一个变量，而不是两个，因为跟在列表后面的修饰词没有用圆括弧包围。

```
my $foo, $bar = 1; #错
```

上面和下面的东西效果一样：

```
my $foo; $bar = 1;
```

如果打开警告的话，你会收到一个关于这个错误的警告。你可以用 **-w** 或 **-W** 命令行开关打开警告，或者用后面在“用法”里解释的 **use warning** 声明。

通常，应尽可能在变量所适合的最小范围内定义它们。因为在流控制语句里面定义的变量只能在该语

句控制的块里面可见，因此，它们的可视性就降低了。同样，这样的英文读起来也更通顺。

```
sub check_warehouse {
    for my $widget (our @Current_Inventory) {
        print "I have a $widget in stock today.\n";
    }
}
```

最常见的声明形式是 **my**，它定义词法范围的变量；这些变量的名字和值都存储在当前范围的临时中间变量暂存器里，而且不能全局访问。与之相近的是 **our** 声明，它在当前范围输入一个词法范围的名字，就象 **my** 一样，但是实际上却引用一个全局变量，任何人如果想看地话都可以访问。换句话说，就是伪装成词汇的全局变量。

另外一种形式的范围，动态范围，应用于 **local** 变量，这种变量实际上是全局变量，除了 “**local**（局部）” 的字眼以外和局部的中间变量暂存器没有任何关系。

4.7.2 词法范围的变量：my

为你帮助你摆脱维护全局变量的痛苦，Perl 提供了词法范围的变量，通常简称为词汇。和全局变量不同，词汇保证你的隐私。假如你没有分发这些私有变量的引用（引用可以间接地处理这些私有变量），你就可以确信对这些私有变量的访问仅限于你的程序里面的一个分离的，容易标识的代码段。这也是为什么我们使用关键字 **my** 的原因。

一个语句序列可以包含词法范围变量的声明。习惯上这样的声明放在语句序列的开头，但我们并不要求这样做。除了在编译时声明变量名字以外，声明所起的作用就象普通的运行时语句：它们每一句都被仔细地放在语句序列中，就好象它们是没有修饰词的普通语句一样：

```
my $name = "fred";
my @stuff = ("car", "house", "club");
my ($vehicle, $home, $tool) = @stuff;
```

这些词法变量对它们所处的最近的闭合范围以外的世界而言是完全不可见的。和 **local** 的动态范围效果（参阅下一节）不同的是，词汇对任何在它的范围内调用的子过程都是不可见的。甚至相同的子过程调用自身或者从别处调用也如此——每个子过程的实例都得到自己的词法变量“中间变量暂存器”。

和块范围不同的是，文件范围不能嵌套；也没有“闭合”的东西——至少没有文本上的闭合。如果你用 **do**，**require** 或者 **use** 从一个独立的文件装载代码，那么在那个文件里的代码无法访问你的词汇，同样你也不能访问那个文件的词汇。

但是，任何一个文件内部的范围（甚至文件本身）都是平等的。通常，一个比子过程定义大一些的范围对你很有用，因为这样你就可以在有限的一个子过程集合里共享私有变量。这就是你创建在 **C** 里被认为是 “**static**”（静态）的变量的方法：

```
{
    my $state = 0;

    sub on      { $state = 1 }
    sub off     { $state = 0 }
    sub toggle  { $state = !$state }
}
```

eval **STRING** 操作符同样也作为嵌套范围运行，因为 **eval** 里的代码可以看到其调用者的词汇（只要其名字不被 **eval** 自己范围里的相同声明隐藏）。匿名子过程也可以在它们的闭合范围内访问任意词汇；如果是这样，那么这些匿名子过程就是所谓的闭包（注：一个记忆用词，表示在“闭合范围”和

“闭包”之间的普通元素。（闭包的真正定义源自一个数学概念，该概念考虑数值集合和对那些数值的操作的完整性。）结合这两种概念，如果一个块 **eval** 了一个创建一个匿名子过程的字符串，该子过程就成为可以同时访问 **eval** 和该块的闭包，甚至在 **eval** 和该块退出后也是如此。参阅第八章，引用，里的“闭包”节。

新声明的变量（或者是值--如果你使用的是 **local**）在声明语句之后才可见。因此你可以用下面的方法给一个变量做镜像：

```
my $x = $x;
```

这句话把新的内部 **\$x** 初始化为当前值 **\$x**，不管 **\$x** 的当前含义是全局还是词汇。（如果你没有初始化新变量，那么它从一个未定义或者空值开始。）

定义一个任意名字的词汇变量隐藏了任何以前定义的同名词汇。它也同时隐藏任何同名无修饰全局变量，不过你总是可以通过明确声明全局变量所处的包的方法来访问全局变量，比如，

\$PackageName::varname。

4.7.3 词法范围全局声明：our

有一个访问全局变量的更好的方法就是 **our** 声明，尤其那些在 **use strict** 声明下运行的程序和模块。这个声明也是词法范围内的，因为它的应用范围只扩展到当前范围的结尾。但与词法范围的 **my** 或动态范围的 **local** 不同的是：**our** 并不隔离当前词法或者动态范围里的任何东西。相反，它在当前环境里提供一个访问全局变量的途径，它把所有同名词汇隐藏起来（否则这些词汇会为你隐藏全局变量）。在这个方面，**our** 变量和 **my** 变量作用相同。

如果你把 **our** 声明放在任何花括弧分隔的块的外面，它的范围就延续到当前编译单元的结尾。通常，人们只是把它放在一个子过程定义的顶端以表明他们在访问全局变量：

```
sub check_warehouse {
    our @Current_Inventory;
    my $widget;

    foreach $widget (@Current_Inventory) {
        print "I have a $widget in stock today.\n";
    }
}
```

因为全局变量比私有变量有更长的生命期和更广的可见范围，所以与临时变量相比我们喜欢为它们使用更长和更鲜明的名字。如果你有意遵循这个习惯，它可以象 **use strict** 一样起到制约全局量使用的效果，尤其是对那些不愿意敲字的人。

重复的 **our** 声明并不意味着嵌套。每个嵌套的 **my** 会生成一个新变量，每个嵌套的 **local** 也生成一个新变量。但是每次你使用 **our** 的时候，你实际上是说同一个变量，不管你有没有嵌套。当你给一个 **our** 变量赋值时，其作用在整个声明范围都起作用。这是因为 **our** 从不创建数值；它只是提供一种有限制地访问全局量的形式，该形式永远存活：

```
our $PROGRAM_NAME = "waiter";
{
    our $PROGRAM_NAME = "server";
    # 这里调用的代码看到的是"server"
}
# 这里执行的代码看到的仍然是"server".
```

而对于 **my** 和 **local** 来说，在块之后，外层变量或值再次可见：

```
my $i = 10;
```

```

{
    my $i = 99;
    ...
}
# 这里编译的代码看到外层变量。

local $PROGRAM_NAME = "waiter";
{
    local $PROGRAM_NAME = "server";
    # 这里的代码看到"server".
    ...
}
# 这里执行的代码再次看到"waiter"

```

通常只给 **our** 赋值一次，可能是在程序或者模块的非常顶端的位置，或者是很少见地用 **local** 前缀 **our**，获取一个 **local** 自己的变量：

```

{
    local our @Current_Inventory = qw(bananas);
    check_warehouse();    # 我们有香蕉 (bananas)
}

```

4.7.4 动态范围变量: **local**

在一个全局变量上使用 **local** 操作符的时候，在每次执行 **local** 的时候都给该全局量一个临时值，但是这并不影响该变量的全局可视性。当程序抵达动态范围的末尾时，临时值被抛弃然后恢复原来的值。但它仍然是一个全局变量，只是在执行那个块的时候碰巧保存了一个临时值而已。如果你在该全局变量包含临时值时调用其他函数，而且该函数访问了该全局变量，那么它看到的将是临时值，而不是初始值。换句话说，该函数处于你的动态范围，即使它并不处在你的词法范围也如此（注：这就是为什么有时候把词法范围叫做静态范围：这样可以与动态范围相比并且突显它们的编译时决定性。不要把这个术语的用法和 C 或 C++ 里的 **static** 的用法混淆。这个术语用得也太广泛了，也是我们避免使用它的原因。）

如果你有个看起来象这样的 **local**：

```

{
    local $var = $newvalue;
    some_func();
    ...
}

```

你完全可以认为它是运行时的赋值：

```

{
    $oldvalue = $var;
    $var = $newvalue;
    some_func();
    ...
}
continue {
    $var = $oldvalue;
}

```

区别是如果用 **local**，那么不管你是如何退出该块的，变量值都会恢复到原来的，即使你提前从那个范围 **return**（返回）。变量仍然是同样的全局变量，其值则取决于函数是从从哪个范围调用的。这也

是为什么我们称之为动态范围——因为它是在运行时修改。

和 **my** 一样，你也可以用一份同样的全局变量的拷贝来初始化一个 **local**。在子过程执行过程中（以及任何在该过程中的调用，因为显然仍将看到的是动态范围的全局变量）对那个变量的任何改变都会在子过程返回的时候被丢弃。当然，你最好还是对你干的事加注释：

```
# WARNING: Changes are temporary to this dynamic scope.
local $Some_Global = $Some_Global;
```

不管一个全局变量是用 **our** 明确声明的，还是突然出现的，还是它保存一个注定要在范围退出后被丢弃的 **local** 变量，它对你的整个程序而言仍然是完全可见的。对小程序来说，这样挺好；但是对大程序来说，你很快就会忘记代码中在那里使用了全局变量。如果你愿意，你可以禁止随机地使用全局变量，你可以用下一节描述的 **use strict 'vars'** 用法来达到这个目的。

尽管 **my** 和 **local** 都提供了某种程度的保护，总的来说你还是应该优先使用 **my**。当然，有时候你不得不用 **local** 来临时改变一个现有全局变量的值，就象我们在第二十八章，特殊名字，里列出来的那样。只有字母数字标识符才能处于词法范围，而那些特殊变量有许多并不是严格的字母数字。你也需要用 **local** 来对一个包的符号表做临时的修改——象我们在第十章 “符号表” 里显示的那样。最后，你可以把 **local** 用在数组或散列的单个元素或者整个片段上。甚至当数组或散列是词法变量的时候也能这么干，这时候是把 **local** 的动态范围建筑在那些词法（变量）的上层。我们不会在这里就 **local** 的语义讲得太多。参阅第二十九章的 **local** 获取更多知识。

4.8 用法 (pragmas)

许多编程语言允许你给编译器一些提示或暗示。在 **Perl** 里，这些暗示是用 **use** 声明交给编译器的。一些用法是：

```
use warning;
use strict;
use integer;
use bytes;
use constant pi => ( 4* atan2(1,1) );
```

Perl 的用法都在第三十一章，用法模块，里描述。这里我们只是讲几个和本章的内容关系非常密切的用法。

虽然有几个用法是全局声明，它们对全局变量或者对当前包有影响，但其他大部分用法都是词法范围里声明的，其影响范围只是伸展到闭合块的结尾，文件或者 **eval**（先到为准）。一个词法范围的用法可以在内层范围里用 **no** 声明取消，**no** 的用途和 **use** 一样，只是作用正相反。

4.8.1 控制警告

为了显示这些东西如何运行，我们将操纵 **warnings** 用法，告诉 **Perl** 是否就有问题的东西发出警告：

```
use warnings;    # 在这里打开警告，作用到文件结束
...
{
    no warnings;    # 关闭警告，作用到块结束
    ...
}
# 警告在这里自动恢复打开状态
```

一旦打开警告，**Perl** 就会抱怨你的变量只用了一次，变量的声明屏蔽了同范围内的其他声明，字符串到数字的非法转换，把未定义的值用做合法字符串或数字，试图写入一个你以只读方式（或者根本没有打

开) 打开的文件和许多其他的问题。这些问题在第三十三章, 诊断信息, 里有描述。

use warning 用法是优选的控制警告的方法。老的程序可能只用 **-w** 命令行开关或者修改 **\$\$W** 全局变量:

```
{
    local $$W = 0;
    ...
}
```

最好还是用 **use warnings** 和 **no warnings** 用法。用法更好些的原因是, 首先它是在编译时作用; 其次它是词法声明, 所以不会影响它不该影响的代码; 最后, 它提供了对离散的警告范畴精细的控制 (尽管我们没有在这些简单的例子中向你演示这些)。更多关于 **warnings** 用法的知识, 包括如何把一般警告转换成致命错误, 如何把警告全局地打开, 覆盖所有说 **no** 的模块的设置等, 请参阅第三十一章, **use warnings**。

4.8.2 控制全局变量的使用

另外一个常见的声明是 **use strict** 用法, 它有几个功能, 其中之一是控制全局量的使用。通常, Perl 允许你在需要时随时随地创建全局变量 (或者是覆盖旧变量)。就是说缺省时不需要变量声明。因为不加限制地使用全局变量会导致程序或模块维护的困难, 所以有时候你可能想限制全局变量的随机使用。为了避免全局变量的随机使用, 你可以说:

```
use strict 'vars';
```

这意味着从这里开始到闭合范围结尾的这个区间里, 任何变量要么是一个词法范围变量, 要么是一个明确声明允许使用的全局变量。如果两者都不是, 将导致编译错误。如果下列之一为真, 则一个全局变量是明确允许使用的:

- 它是 Perl 的程序范围内的特殊变量之一 (参阅第二十八章)。
- 它带有包括其包名字的全称 (参阅第十章)。
- 它被输入到当前包 (参阅第十一章)。
- 它通过一个 **our** 声明伪装成了一个词法范围的变量。(这是我们向 Perl 中 增加 **our** 声明的主要原因。)

当然, 还有第五种可能——如果该用法让人觉得烦, 只需要在内层块里用下面语句取消掉它:

```
no strict 'vars';
```

你还可以利用这个用法打开对符号解引用和光字的随机使用的严格检查。通常人们只是说:

```
use strict;
```

这样就把三个检查都打开了。参阅第三十一章的 **use strict** 部分获取更多信息。

Revision: r1.5 - 14 Oct 2005 - 04:54 - [TingYu](#)

[Perl](#) > [PerlProgramming3](#) > [StatementsandDeclarations](#)

版权 © 1999-2006 归这里所有作者. [PostgreSQL](#) 的中文文档版权归何伟平所有.

向为这里贡献想法, 文章的人致敬 [PostgreSQL](#) 中文网

[反馈意见](#)