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'stuff，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：

子例程定义的位置不同，my定义的词汇的作用域会有什么区别？

{

my $state = 0;

# $state变量会有效？

      sub on { $state = 1}

      sub off { $state = 0}

      sub toggle { $state =!$state }

}

测试程序：

#!/usr/local/bin/perl

#

#

use strict;

my $var = 0;

 print " begin var = $var \n";   #结果０

 &on;　＃　打印１

 print "after on var = $var \n";　＃打印０

 &off;　　＃打印０

 sub on {my  $var = 1; print " on :var = $var\n";}  # 子例程中单独定义会临时覆盖外边的$var定义

 sub off { $var = 0 ; print "off : var = $var \n"}

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。（闭包的真实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 strice 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"watier"

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掉的 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。