Perl 语言编程

目录

第一章 Perl 概述

1.1 从头开始

我们认为 Perl 是一种容易学习和使用的语言，而且我们希望能证明我们是对的．Perl 比较简单的一个方面是你用不着在说想说的东西之前先说很多其他东西。在很多其他编程语言里，你必须首先定义类型，变量，以及你需要用到的子过程，然后才能开始写你要执行的第一行程序。虽然对于那些需要复杂数据结构的复杂问题而言，声明变量是一个好主意.但是对于很多简单的日常问题，你肯定喜欢这样的一种编程语言，你只需简单说：

```
print "Howdy, World!\n";
```

程序就能得到你所需的结果.

Perl 就是这样的一种语言。实际上，上面这个例子是一个完整的程序，如果你将它输入到 Perl 解释器里，它就会在你的屏幕上打印出 "Howdy, world!"（例子中的 \n 在输出中产生一个新行。）

同样，你也用不着在说完之后说很多其他东西。和其他语言不同的是，Perl 认为程序的结尾就是一种退出程序的正常途径，如果你愿意的话，你当然可以明确地调用 exit 函数来退出程序。就象你可以声明一些你所用的变量，或者甚至可以强迫自己声明所用的所有变量，但这只是你的决定。用 Perl 你可以自由的做那些正确的事，不过你要仔细的定义它们。

关于 Perl 的容易使用还有很多其他理由，但是不可能全在这里列出来，因为这是这本书余下部分说要讨论的内容。语言的细节是很难理解的，但是 Perl 试图能把你从这些细节中解放出来。在每一个层次，Perl 都能够帮助你以最小的忙乱获得最大的享受和进步，这就是为什么这么多 Perl 程序员能够如此悠闲的原因吧。

本章是 Perl 的一个概述，所以我们不准备介绍得过于深入，同时我们也不追求描述的完整性和逻辑性。那些是下面章节所要做的事情。如果你等不及了，或者你是比较死板的人，你可以直接进入到第二章，集腋成裘，获取最大限度的信息密度。另外如果你需要一个更详细的教程,你可以去找 Randal 的 Learning Perl （由 O'Reilly&Associates 出版）。

不管你喜欢把 Perl 称做想象力丰富的，艺术色彩浓厚的，富有激情的还是仅仅是具有很好的灵活性的东西，我们都会在本章中给你展现 Perl 的另一个方面。到本章结束时，我们将给你展现 Perl 的不同方面，并帮助你建立起一个 Perl 的清晰完整的印象。

1.2 自然语言与人工语言

语言最早是人类发明出来方便自身的东西。但在计算机科学的历史中，这个事实偶尔会（注：更准确地说，人们会偶尔记起这个事实）被人们忘记。因为 Perl 碰巧是由一个语言学家设计的（可以这么说吧），因此它被设计成一个可以象自然语言那样使用的编程语言。通常，做到这一点要处理很多方面的事情，因为自然语言可以同时在几个不同的层次做得非常好。我们可以列举出很多语言设计上的原则，但是我们认为语言设计最重要的原则就是：处理简单的事情必须容易，并且能够处理困难的事情（其实这是两个原则）。这对你来说也许显而易见，但是有很多计算机语言在其中的某个方面做得不好。

自然语言在上述两个方面都做得很好，因为人们总是需要表达简单的事情和复杂的事情,所以语言进化成能够同时处理这两种情况。Perl 首先被设计成可以进化，并且实际上也已经进化了。在这个进化过程中，很多人做出了很多贡献。我们经常开玩笑说：骆驼（Perl）是一匹委员会设计的马，但是如果你想一想，骆驼非常适应沙漠中的生活。骆驼已经进化成为相当能自给自足(另一方面，骆驼闻起来不怎么样，Perl 也一样），这也是我们选择骆驼作为 Perl 的吉祥物众多原因中的一个，而和语言学没有什么关系。

现在，当有人提起"语言学"的时候，一些人关注于字，另一些人则关注句子。但是词和句子只是拆分一大段话的两个简单方法。它们要么可以拆分成可以更小的表意部分，要么可以并成更大的表意部分。任何部分所表达的意思很大程度上依赖于语法，语义以及所处的环境。自然语言由不同词性的词：名词，动词等等组成。在一个隔离的环境中说"狗"的时候，我们认为它是一个名词，但是你也可以以不同的方式使用同一个词。在 "If you dog a dog during the dog days of summer, you will be a dog tired dogcather"（如果你在三伏天追赶一只狗，你就会成为疲劳的捕狗人。）这个句子中，dog 这个名词在这个环境里可以作为动词，形容词，和副词。（注：你看了这句话可能都对这些贫嘴的狗词汇都烦了。不过我们只是想让你理解为什么 Perl 和其他典型的计算机语言不同， TMD！）

Perl 也根据不同的环境来处理词，在下面的章节中我们将会了解到 Perl 是如何进行处理的。现在我们只需要记住 Perl 象一个好听众那样努力理解你说的话. 你只需要说你的意思，Perl 就能理解你的意思(除非你在胡说，当然 Perl 解释器更容易听懂 Perl，而不是英语或斯瓦希里语。）

回到名词，一个名词可以命名一个特定的对象，或者它可以命名非特指的某类对象. 绝大多数计算机语言将上述两个方面区别开来。只有我们把特定对象当作值而把泛指的对象当做变

量。值保存在一个地方，而变量和一个或多个值关联。因此无论是谁解释变量都必须保持跟踪这个关联。这个解释器也许是你的大脑或者是你的计算机.

1.2.1 变量语法

一个变量就是用来保存一些东西的地方，这个地方有一个名字，这样当你需要使用这些东西的时候就知道从哪里找到它。在日常生活中有很多不同地方用来储存东西，有些是很秘密的，有些则是公开的。有些是暂时性的，有些则更为永久。计算机学家很喜欢讨论变量范围。Perl 有不同于其他语言的简便方法来处理范围问题。你可以在本书后面适当的时候学习到相关的知识（如果你很急迫地知道这些知识，你可以参阅二十九章，函数，或第四章，语句和声明，里"范围声明"。）

一个区分变量类型最直接的方法是看变量里面保存了何种数据。象英语一样，Perl 变量类型之间区别主要是单数和复数，字符串和数字是单个数据，而一组数字和字符串是复数(当我们接触到面对对象编程时，对象就象一个班的学生一样，从外部看，它是一个单数，而从内部看则是一个复数)我们叫把单数变量称为标量，而把复数变量称为数组。我们可以将第一个例子程序改写成一个稍微长一些的版本:

```
$phrase = "Howdy, world!\n";

print $phrase;
```

请注意,在 Perl 中我们不必事先定义 $phrase 是什么类型的变量，$ 符号告诉 Perl，phrase 是一个标量，也就是包含单个数值的变量。与此对应的数组变量使用 @ 开头。(可以将 $ 理解成代表 "s" 或 "scalar"（标量），而 @ 表示 "a" 或 "array" （数组）来帮助你记忆。）

Perl 还有象"散列"，"句柄"，"类型团"等其他一些变量类型，与标量和数组一样，这些变量类型也是前导趣味字符，下面是你将会碰到的所有趣味字符：

类型	字符	例子	用于哪种名字
标量	$	$cents	一个独立的数值（数字或字串）
数组	@	@large	一列数值，用编号做键字
散列	%	%interest	一组数值，用字串做键字
子过程	&	&how	一段可以调用的 Perl 代码
类型团	*	*struck	所有叫 struck 的东西

一些纯粹语言主义者将这些古怪的字符作为一个理由来指责 Perl。这只是表面现象。这些字符有很多好处，不用额外的语法变量就可以代换成字串只是最简单的一个。Perl 脚本也很容易阅读（当然是对那些花时间学习了 Perl 的人！）因为名词和动词分离，这样我们就

可以向语言中增加新的动词而不会破坏旧脚本。（我们告诉过你 Perl 被设计成是可以进化的语言。）名词也一样，在英语和其他语言中为了满足语法需要有很多前缀。这就是我们的想法。

1.2.2 单数变量

从我们前面的例子中可以看到，Perl 中的标量象其他语言一样，可以用 = 对它进行赋值，在 Perl 中标量可以赋予：整数，浮点数，字符串，甚至指向其他变量或对象的引用这样深奥的东西。有很多方法可以对标量进行赋值。

与 Unix（注：我们在这里和其他所有地方提到 Unix 的时候，我们指的是所有类 Unix 系统，包括 BSD，Linux，当然还包括 Unix）中的 shell 编程相似，你可以使用不同的引号来获得不同的值，双引号进行变量代换（注：有时候 shell 程序员叫"替换"，不过，我们宁愿在 Perl 里把这个词保留给其他用途。所以请把它称之为"代换"。我们使用的是它的字面意思（"这段话是某种宗教代换"），而不是其数学含义（"图上这点是另外两点的插值"））和反斜扛代换（比如把 \n 转换成新行）．而反勾号（那个向左歪的引号）将执行外部程序并且返回程序的输出，因此你可以把它当做包含所有输出行的单个字串.

- $answer = 42; # 一个整数
- $pi = 3.14159265 # 一个"实"数
- $pet = "Camel"; # 字串
- $sign = "I ove my $pet"; # 带代换的字串
- $cose = 'It cose $100'; # 不带代换的字串
- $thence = $whence; # 另一个变量的数值
- $salsa = $moles * $avocados; # 一个胃化学表达式
- $exit = system("vi $file"); # 一条命令的数字状态
- $cwd = `pwd`; # 从一个命令输出的字串

上面我们没有涉及到其他一些有趣的值，我们应该先指出知道标量也可以保存对其他数据结构的引用，包括子程序和对象.

如果你使用了一个尚未赋值的变量，这个未初始化的变量会在需要的时候自动存在．遵循最小意外的原则，该变量按照常规初始化为空值，"" 或 0．根据使用的地方的不同，变量会被自动解释成字符串，数字或"真"和"假"（通常称布尔值）．我们知道在人类语言中语言环境是十分重要的．在 Perl 中不同的操作符会要求特定类型的单数值作为参数．我们就称之为这个操作符给这些参数提供了一个标量的环境．有时还会更明确，比如说操作符会给这些

参数提供一个数字环境，字符串环境或布尔环境．（稍后我们会讲到与标量环境相对的数组环境）Perl 会根据环境自动将数据转换成正确的形式．例如：

```
$camels = '123';

print $camels +1, "\n";
```

$camels 最初的值是字符串，但是它被转换成数字然后加一，最后有被转换回字符串，打印出 124．"\n" 表示的新行同样也在字符串环境中，但是由于它本来就是一个字符串，因此就没有必要做转换了．但是要注意我们在这里必须使用双引号，如果使用单引号 '\n'，这就表示这是由反斜扛和 n 两个字符组成的字符串，而不表示一个新行．

所以，从某种意义上来说，使用单引号和双引号也是另外一种提供不同环境方法．不同引号里面的内容根据所用的引号有不同的解释．（稍后，我们会看到一些和引号类似的操作符，但是这些操作符以一些特殊的方法使用字符串，例如模式匹配，替换．这些都象双引号字符串一样工作．双引号环境在 Perl 中称为代换环境．并且很多其他的操作符也提供代换环境．）

同样的，一个引用在"解引用"环境表现为一个引用，否则就象一个普通标量一样工作，比如，我们可以说：

```
$fido    = new Camel "Amelia";

if (not $fido) { die "dead camel"; }

$fido->saddle();
```

在这里，我们首先创建了一个指向 Camel 对象的引用，并将它赋给变量 $fido，在第二行中，我们将 $fido 当成一个布尔量来判断它是否为真，如果它不为真，程序将抛出一个例外．在这个例子中，这将意味着 new Camel 构造函数创建 Camel 对象失败．最后一行，我们将 $fido 作为一个引用，并调用 Camel 对象的 saddle() 方法．今后我们还将讲述更多关于环境的内容．现在你只需记住环境在 Perl 中是十分重要的，Perl 将根据环境来判断你想要什么，而不用象其他编程语言一样必须明确地告诉它．

1.2.3 复数变量

一些变量类型保存多个逻辑上联系在一起的值．Perl 有两种类型的多值变量：数组和散列，在很多方面，它们和标量很相似，比如它们也会在需要时自动存在．但是，当你给它们赋值时，它们就和标量就不一样了．它们提供一个列表环境而不是标量环境．

数组和散列也互不相同．当你想通过编号来查找东西的时候，你要用数组．而如果你想通过名称来查找东西，那么你应该用散列．这两种概念是互补的．你经常会看到人们用数组实现月份数到月份名的翻译，而用对应的散列实现将月份名翻译成月份数．（然而散列不仅仅局限于保存数字，比如，你可以有一个散列用于将月名翻译成诞生石的名字．）

数组．一个数组是多个标量的有序列表，可以用标量在列表中的位置来访问（注：也可以说是索引，脚标定位，查找，你喜欢用哪个就用哪个）其中的标量，列表中可以包含数字，字符串或同时包含这两者．（同时也可以包括对数组和散列的引用），要对一个数组赋值，你只需简单的将这些值排列在一起，并用大括弧括起来．

```
@home = ("couch", "chair", "table", "stove");
```

相反，如果你在列表环境中使用 @home，例如在一个列表赋值的右边，你将得到与你放进数组时同样的列表．所以可以你象下面那样从数组给四个标量赋值：

```
($potato, $lift, $tennis, $pipe) = @home;
```

他们被称为列表赋值，他们逻辑上平行发生，因此你可以象下面一样交换两个变量：

```
($alpha, $omega) = ( $omega, $alpha);
```

和 C 里一样，数组是以 0 为基的，你可以用下标 0 到 3 来表示数组的第一到第四个元素．（注：如果你觉得不好记，那么就把脚标当做偏移量，也就是它前面的元素个数．显然，第一个元素前面没有任何元素，因此偏移量是 0．计算机就是这么想的．）数组下标使用中括弧包围[象这样]，因此如果你想选用独立的数组元素，你可以表示为 $home[n]，这里 n 是下标（元素编码减一），参考下面的这个例子．因为我们处理的这个数组元素是标量，因此在他前面总是前缀 $．

如果你想一次对一个数组元素赋值，你可以使用下面的方法：

```
$home[0] = "couch";

$home[1] = "chair";

$home[2] = "table";

$home[3] = "stove";
```

因为数组是有序的，所以你可以在它上面做很多很有用操作．例如堆栈操作 push 和 pop，堆栈就是一个有序的列表，有一个开始和一个结尾．特别是有一个结尾．Perl 将你数组的结尾当成堆栈的顶端（也有很多的 Perl 程序员认为数组是水平的，因此堆栈的顶端在数组的右侧．）

散列，散列是一组无序标量，可以通过和每个标量关联的字符串进行访问．因为这个原因，散列经常被称为关联数组．但是这个名字太长了，因为会经常提到它,我们决定给它起一个简短的名字．我们称之为散列的另外一个原因是为了强调它们是无序的．（在 Perl 的内部实现中，散列的操作是通过对一个散列表查找完成的，这就是散列为什么这么快的原因，而且无论你在散列中存储多少数据，它总是很快）．然而你不能 push 或 pop 一个散列，因为这样做没有意义．一个散列没有开始也没有结束．不管怎么样，散列的确非常有用而且强大．如果你不能理解散列的概念，那你还不能算真正的了解 Perl．图 1-1 显示了一个数组中有序的元素和一个散列中无序但是有名字的元素．

因为散列不是根据位置来访问的，因此你在构建散列时必须同时指定数值和键字，你仍然可以象一个普通的数组那样给散列赋值，但是在列表中的每一对元素都会被解释为一个键字和一个数值．因为我们是在处理一对元素，因此散列使用 % 这个趣味字符来标志散列名字（如果你仔细观察 %，你会发现斜扛两边的键字和数值．这样理解可能会帮助记忆．）

假设你想把简写的星期名称转换成全称，你可以使用下面的赋值语句:

```
%longday = ("Sun", "Sunday", "Mon", "Monday", "Tue", "Tuesday",
                    "Wed", "Wednesday", "Thu", "Thursday", "Fri",
                    "Friday", "Sat", "Saturday");
```

不过上面地写法非常难读懂，因此 Perl 提供了=>（等号和大于号地组合）来做逗号的替代操作符．使用这种表示方法，可以非常容易地看出哪个字符串是关键字，哪个是关联的值．

```
%longday = (
                "Sun" => "Sunday",
                "Mon" => "Monday",
                "Tue" => "Tuesday",
                "Wed" => "Wednesday",
                "Thu" => "Thursday",
                "Fri" => "Friday",
                "Sat" => "Saturday",
                );
```

象我们在上边做的，你可以将一个列表赋值给一个散列，同样如果你在一个列表环境中使用散列，Perl 能将散列以一种奇怪的顺序转换回的键字/数值列表．通常人们使用 keys 函数来抽取散列的关键字．但抽取出来的键字也是无序的．但是你用 sort 函数可以很容易地对它进行排序．然后你可以使用排过序的键字以你想要的顺序获取数值．

因为散列是一种特殊的数组，你可以通过 {} 来获取单个的散列元素．比如，如果你想找出与关键字 Wed 对应的值，你应该使用 $longday{"Wed"}．注意因为你在处理标量，因此你在 longday 前面使用 $，而不是 %，% 代表整个散列．

通俗的讲，散列包含的关系是所有格的，象英文里面的 of 或者 's．例如 Adam 的妻子是 Eve，所以我们用下面的表达式：

```
$wife{"Adam"} = "Eve";
```

1.2.4 复杂数据结构

数组和散列是易用、简单平面的数据结构，很不幸，现实总不能如人所愿。 很多时候你需要使用很难、复杂而且非平面的数据结构。Perl 能使复杂的事情变得简单。方法是让你假装那些复杂的数值实际上是简单的东西．换句话说，Perl 让你可以操作简单的标量，而这些标量碰巧是指向复杂数组和散列的引用．在自然语言中，我们总是用简单的单个名词来表示复杂的难以理解的实体，比如，用"政府"来代表一个关系复杂的硬壳等等．

继续讨论上个例子，假设我们想讨论 Jacob 的妻子而不是 Adam 的，而 Jacob 有四个妻子（自己可别这么干）。为了在 Perl 中表示这个数据结构，我们会希望能将 Jocob 的四个妻子当成一个来处理，但是我们会遇到一些问题。你可能认为我们可以用下面的语句来表示：

```
$wife{"Jacob"} = ("Leah", "Rachel", 'Bilhah", "Zilpah");   # 错
```

但是这并不能象你希望的那样运转，因为在 Perl 中括弧和逗号还不够强大，还不能将一个列表转换成为标量（在语法中，圆括弧用于分组，逗号用于分隔）。你需要明确地告诉 Perl 你想将一个列表当成一个标量。[] 中括弧能够实现这个转换：

```
$wife{"Jacob"} = ["Leah", "Rachel", "Bilhah", "Zilpah"];   # 正确
```

这个语句创建了一个未命名的数组，并将这个数组的引用放入散列的元素 $wife{"Jacob"} 中．因此我们有了一个命名的散列，其中包含了一个未命名的数组．这就是 Perl 处理多维数组和嵌套数据类型的方法.同普通数组和散列的赋值方法一样,你可以单独对其进行赋值：

```
$wife{"Jacob"}[0] = "Leah";
```

```
$wife{"Jacob"}[1] = "Rachel";

$wife{"Jacob"}[2] = "Bilhah";

$wife{"Jacob"}[3] = "Zilpah";
```

你可以从上边看出，这看起来象一个具有一个字符串下标和一个数字下标的多维数组. 为了更多了解树状结构，如嵌套数据结构，假设我们希望不仅能列出 Jocob 的妻子，而且同时能列出每个妻子的所生的儿子。这种情况下，我们希望将散列结构也当成一个标量，我们可以使用花括弧来完成（在每个散列值中，象上个例子一样用中括弧表示数组,现在我们有了一个在散列中的散列中的数组）。

```
 $kids_of_wife{"Jacob"} = {

      "Leah"   => ["Reuben", "Simeon", "Levi", "Judah", "Issachar",
"Zebulun"],

      "Rachel" => ["Joseph", "Benjamin"],

      "Bilhah" => ["Dan", "Naphtali"],

      "Zilpah" => ["Gad", "Asher"],};
```

同样，我们也可以象下面这样表示：

```
$kids_of_wife{"Jacob"}{"Leah"}[0]   = "Reuben";

$kids_of_wife{"Jacob"}{"Leah"}[1]   = "Simeon";

$kids_of_wife{"Jacob"}{"Leah"}[2]   = "Levi";

$kids_of_wife{"Jacob"}{"Leah"}[3]   = "Judah";

$kids_of_wife{"Jacob"}{"Leah"}[4]   = "Issachar";

$kids_of_wife{"Jacob"}{"Leah"}[5]   = "Zebulun";

$kids_of_wife{"Jacob"}{"Rachel"}[0] = "Joseph";

$kids_of_wife{"Jacob"}{"Rachel"}[1] = "Benjamin";

$kids_of_wife{"Jacob"}{"Bilhah"}[0] = "Dan";

$kids_of_wife{"Jacob"}{"Bilhah"}[1] = "Naphtali";

$kids_of_wife{"Jacob"}{"Zilpah"}[0] = "Gad";
```

```
$kids_of_wife{"Jacob"}{"Zilpah"}[1] = "Asher";
```

可以从上面看出，在嵌套数据结构中增加一层，就像在多维数组中增加了一维。在 Perl 内部表示是一样的，但是你可以用任何一种方法来理解。

这里最重要的一点就是，Perl 可以用简单的标量来代表复杂数据结构。Perl 利用这种简单的封装方法构建了基于对象的结构。当我们用下面的方法调用 Camel 对象的构造函数的时候：

```
$fido = new Camel "Amelia";
```

我们创建了一个 Camel 对象，并用标量 $fido 来代表。但是在 Camel 对象里面是很复杂的。作为优秀的面向对象的程序员，我们不想关心 Camel 对象里面的细节（除非我们是实现 Camel 类方法的人）。但是一般说来，一个对象的组成中会有一个包含对象属性的散列。例如它的名字（本例子中，是“Amelia”而不是“fido”），还有驼峰的数量（在这里我们没有明确定义，因此使用缺省值 1，和封面一样）。

1.2.5 简单数据结构

阅读完上一节，你一定会感到有点头晕，否则你一定不简单。通常人们不喜欢处理复杂数据结构。因此在自然语言中，我们有很多方法来消除复杂性。很多其中的方法都归结到“主题化”这个范畴，主题化是一个语言学概念，指在谈论某方面事情时，谈论双方保持一致。主题化可以在语言的各个级别出现，在较高的级别中，我们可以根据不同的感兴趣的子话题将自己分成不同的文化类型，同时建立一些专有语言来讨论这些特定的话题。就象在医生办公室中的语言（“不可溶解窒息物”）和在巧克力厂中的语言（“永久块止动器”）肯定是有差异的一样。所幸，我们能够在语言环境发生转换时能够自动适应新的语言环境。

在对话级别中，环境转换必须更加明确，因此语言让我们能用很多的方式来表达同一个意思。我们在书和章节的开头加上题目。在我们的句子中，我们会用“根据你最近的查询”或“对于所有的 X”来表示后面的讨论主题。

Perl 也有一些主题化的方法，最主要的就是使用 package 声明。例如你想在 Perl 中讨论 Camels，你会在 Camel 模块中以下面的方法开头：

```
package Camel;
```

这个开头有几个值得注意的效果，其中之一就是从这里开始，Perl 认为所有没有特别指出的动词和名词都是关于 Camels 的，Perl 通过在全局名字前添加模块名字“Camel::”来实现，因此当你使用下面的方法：

```
package Camel;
```

这里，$fido 的真实名字是 $Camel::fido（&fetch 的真实名字是 &Camel::fetch）。这就意味着如果别人在其他模块中使用：

```
package Dog;

$fido = &fetch();
```

Perl 不会被迷惑，因为这里 $fido 的真实名字是 $Dog::fido，而不是 $Camel::fido。计算机科学家称之为一个 package 建立了一个名字空间。你可以建立很多的名字空间，但是在同一时间你只能在一个名字空间中，这样你就可以假装其他名字空间不存在。这就是名字空间如何为你简化实际工作的方法。简化是基于假设的（当然，这是否会过于简化，这正是我们写这一章的原因）

保持动词的简洁和上面讨论保持名词的简洁同样重要。在 Camel 和 Dog 名字空间中，&Camel::fetch 不会与 &Dog::fetch 混淆，但包的真正好处在于它们能够将你的动词分类，这样你就可以在其他包中使用它们。当我们使用：

```
$fido = new Camel "Amelia";
```

我们实际上调用了 Camel 包中的 &new，它的全名是 &Camel::new。并且当我们使用：

```
$fido->saddle();
```

的时候，我们调用了 &Camel::saddle 过程，因为 $fido 记得它是指向一个 Camel 对象的。这就是一个面向对象程序的工作方法。

当你说 package Camel 的时候，你实际上是开始了一个新包。但是有时候你只是想借用其他已有包的名词和动词。Perl 中你可以用 use 声明来实现，use 声明不仅可以让你使用其他包的动词，同时也检查磁盘上载入的模块名称。实际上，你必须先使用：

```
use Camel;
```

然后才能使用：

```
$fido = new Camel "Amelia";
```

不然的话，Perl 将不知道 Camel 是什么东西。

有趣的是，你自己并不需要真正知道 Camel 是什么，你可以让另外一个人去写 Camel 模块。当然最好是已经有人为你编写了 Camel 模块。可能 Perl 最强大的东西并不在 Perl 本身，而在于 CPAN(Comprehensive Perl Archive Network)，CPAN 包含无数的用于实现不同任务模块。你不需要知道如何实现这些任务，只需要下载这些模块，并简单用下面的方法来使用它们：

```
use Some::Cool::Module;
```

然后你就可以使用模块中的动词。

因此，象自然语言中的主题化一样，Perl 中的主题化能够"歪曲"使用处到程序结束中的 Perl 语言。实际上，一些内部模块并没有动词，只是简单地以不同的有用方法来封装 Perl 语言。我们称这些模块为用法。比如，你经常看到很多人使用 strict：

```
use strict;
```

strict 模块干的事是更加严格地约束 Perl 中的一些规则，这样你在很多方面必须更明确，而不是让 Perl 去猜，例如如何确定变量的作用范围。使事情更加明确有助于使大工程更容易操作。缺省的 Perl 是为小程序优化的，有了 strict，Perl 对于那些需要更多维护的大型工程也是相当好的。由于你可以在任何时候加入 strict 用法，所以你可以容易地将小型工程发展成大型工程。即使你并不想这么做，但是现实生活中你经常能碰到这种情况。

1.2.6 动词

和典型的祈使性计算机语言中常用一样，Perl 中的很多动词就是命令：它们告诉 Perl 解释器执行某个动作。另一方面，类似于自然语言，Perl 的动词能试图根据不同的环境以不同方向执行。一个以动词开头的语句通常是纯祈使句，并完全为其副作用进行计算。（我们有时候称这些动词过程，尤其当它们是用户定义的时候。）一个常用的内建命令（实际上，你在前面已经看到）是 print：

```
print "Adam's wife is $wife{'Adam'}.\n"
```

它的副作用就是生成下面的输出：

```
Adam's wife is Eve。
```

但是除了祈使语气以外，动词还有其他一些语气。有些动词是询问问题并在条件下十分有用，例如 if 语句。其他的一些动词只是接受输入参数并且返回返回值，就象一个处方告诉你如何将原材料做成可以吃的东西。我们习惯叫这些动词为函数，为了顺从那些不知道英语中"functional"意思的数学家们的习惯.

下面是内建函数的一个例子，这就是指数函数：

```
$e = exp(1);        # 2.718281828459 或者类似的数值
```

在 Perl 中过程和函数并没有硬性的区别。你将发现这两个概念经常能够互换。我们经常称动词为操作符（内建）或者是子过程（用户自定义）（注：历史上，Perl 要求你在调用的

任何用户定义子过程的前面加一个与号（&）（参阅早先时候的 $fido = &fetch();）。但是到了 Perl 版本 5，这个与号是可选的了，所以用户定义动词现在可以和内建动词相同的方法进行调用了（$fido = fetch();）。在讲到关于过程的名字的时候，我们仍然使用与号，比如当我们用一个引用指向过程名字的时候（$fetcher = \&fetch；）。从语言学上来讲，你可以把与号形式的 &fetch 当作不定词"to fetch"，或者类似的形式"to fetch"。但是如果我们可以只说"fetch"的时候，我们很少说"do fetch"。这才是我们在 Perl 5 里去掉那个命令性的与号的原因。），但是你把它们称做任何你喜欢的东西，它们都返回一个值，这个值可能有意义，也可能没有什么意义。你可以简单的省略掉。

当我们继续学习的时候，你可以看到很多其他的例子说明 Perl 和自然语言一样工作。但还可以用其他方面来看 Perl。我们已经从数学语言中借用了一些概念，例如下标，加法和指数函数。而且 Perl 也是一种控制语言，一种连接语言，原型语言，文本处理语言，列表处理语言以及面向对象的语言。

但是 Perl 同样也是一种平常古老的计算机语言，这就是我们下面要观察的角度。

1.3 一个平均值例子

假如你在一个班中教授 Perl 语言，并且你正在想如何给你的学生评分的方法。你有全班所有人每次考试的成绩，它们是随机的顺序，你可能需要一个所有同学的等级列表，加上他们的平均分。你有一份象下面一样的文本文件（假设名字为 grades）：

```
No&#235;l 25

Ben 76

Clementine 49

Norm 66

Chris 92

Doug 42

Carol 25

Ben 12

Clementine 0

Norm 66

...
```

你可以用下面所示的脚本将所有的成绩收集在一起，同时计算出每个学生的平均分数，并将它们按照字母顺序打印出来。这个程序天真地假设在你的班级中没有重名的学生，比如没有两个名为 Carol 的学生。如果班级中有两个 Carol，文件中所有以 Carol 开头的条目，程序都会认为这是第一个 Carol 的成绩（但是不会跟 Noel 的成绩混淆）。

顺便说一句，下面程序里的行号并不是程序的一部分，任何与 BASIC 类似的东西都是站不住脚的。

```
 1  #!/usr/bin/perl
 2
 3  open(GRADES, "grades") or die "Can't open grades: $!\n";
 4  while ($line = <GRADES>) {
 5      ($student, $grade) = split(" ", $line);
 6      $grades{$student} .= $grade . " ";
 7  }
 8
 9  foreach $student (sort keys %grades) {
10      $scores = 0;
11      $total = 0;
12      @grades = split(" ", $grades{$student});
13      foreach $grade (@grades) {
14          $total += $grade;
15          $scores++;
16      }
17      $average = $total / $scores;
18      print "$student: $grades{$student}\tAverage: $average\n";
19  }
```

在你离开本例子程序之前，我们需要指出这个程序演示了我们前面涉及到的许多内容，还要加上我们马上要解释的一些内容。你可以猜测下面将要讲述的内容，我们会告诉你你的猜测是否正确。

可能你还不知道如何运行这个 Perl 程序，在下一小节我们会告诉你。

1.3.1 如何运行

刚才你肯定想知道如何运行一个 Perl 程序。最简单的回答就是你可以将程序送进 Perl 语言解释器程序，通常它的名字就是 perl。另一个稍微长一些的答案就是：回字有四种写法。（注：There's More Than One Way To Do It. TMTOWTDI，这是 Perl 的口号，你可能都听烦了，除非你是当地的专家，那样的话你就是说烦了。有时候我们缩写成 TMOTOWTDI，念做"tim-today"。不过你可以用你喜欢的方式发音，别忘了，TMTOWTDI 。译注：这里借用鲁迅先生的名言"回字有四种写法"好象不算过分吧？不管怎样，回字是有四种写法。）

第一种运行 perl 的方法（也是大多数操作系统都能用的法子）就是在命令行中明确地调用 perl 解释器（注：假设你的操作系统提供一个命令行接口。如果你运行的是老的 Mac，那么你可能需要升级到一个 BSD 的版本，比如 Mac OS X）。如果你正在做一些非常简单的事情，你可以直接使用 -e 选项开关（下面例子中的 % 表示标准的 shell 提示符，所以在你运行的时候，不需要输入它们）。在 Unix 中，你可以输入下面的内容：

```
%perl -e 'print "Hello, world!\n";'
```

在其它的操作系统中，你可能需要修改一下这几个引号才能使用。但是基本规则都是一样的：你必须将 Perl 需要知道的东西塞进 80 列当中。（注：这种类型的脚本通常称做"单行程序"。如果你曾经和其他 Perl 程序员交流过，那么你就会发现我们中有些人非常喜欢写这样的单行程序。因为这个，有时候 Perl 还被诽谤成只写语言。）

对于长一些的脚本，你可以使用你熟悉的文本编辑器（可以是任何文本编辑器），将你所有的命令放进一个文件当中，假设你把这个脚本命名为 gradation （不要和 graduation （毕业）混淆），你可以这样用：

```
%perl gradation
```

你现在仍然在明确地调用 Perl 解释器，但至少你不需要每次都在命令行中输入所有东西。而且你也不需要为各个 shell 之间引号的使用方法不同而大伤脑筋。

执行脚本最方便的方法就是只需要直接输入脚本的名字（或者点击它），然后操作系统帮你找到正确的解释器。在一些系统中，可能有方法将文件后缀或目录和特定的应用程序关联起

来。在这些系统中，你可以将 Perl 脚本与 perl 解释器关联起来。在 Unix 系统中支持 #!"shebang"标志（现在，大多数 Unix 都支持），你可以使你的脚本第一行变得具有特殊功能，因此操作系统知道会需要运行哪个程序。在我们例子中，用下面的语句作为第一行：

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

（如果 Perl 不在 /usr/bin 目录下，你需要根据实际情况修改 #! 行），然后你只需要简单地输入：

```
%gradation
```

当然，这样不能运转，因为你忘了确定脚本是否是可执行的（参看 chmod (1) 手册页）以及程序是否在你的运行路径下（通常用环境变量 PATH 定义）。如果不在你的环境变量 PATH 下，你需要提供带路径的完整文件名，只有这样操作系统才知道到什么地方找你的脚本。就象下面这样：

```
%/home/sharon/bin/gradation
```

最后，如果你在古老的 Unix 上工作，它不支持 #!，或者你的解释器的路径超过 32 个字符（在很多系统上的一个内置限制），你也许可以使用下面的方法使你的脚本工作：

```
#! /bin/sh -- # perl，用于停止循环

eval 'exec /usr/bin/perl -S $0 ${1+"$@"}'

    if 0;
```

不同的操作系统和不同的命令行解释器，如 /bin/csh，DCL，COMMAND.COM，需要不同的方法来符合不同需要。或者你还有一些其它的缺省命令行解释器，你可以咨询你身边的专家。

本书中，我们只使用 #!/usr/bin/perl 来代表所有的其它的标志，但是你要知道我们真正的意思。

另外，当你写测试脚本时，不要将你的脚本命名为 test，因为 Unix 系统有一个内建的命令叫 test 会优先运行，而不是运行你的脚本。用 try 做名字。

还有，当你在学习 Perl 的时候，甚至在你认为已经掌握了 Perl 后，我们建议你使用 -w 开关，尤其在你开发的过程中。这个选项会打开所有有用的和有趣的警告信息，你可以象下面例子中一样，将 -w 开关加入到 shenbang 行中：

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

现在你已经知道如何运行你自己的 Perl 程序（不要和 perl 解释器混淆），让我们回到例子上来。

1.4 文件句柄

除非你在用人工智能来制作唯我主义哲学家的模型，否则你的程序肯定需要和外边的世界进行通讯的途径。在计算平均分的例子第 3, 4 行，你可以看到 GRADES 这个词，它是 Perl 另一个数据：类型文件句柄的例子。文件句柄只是你给文件，设备，网络套接字或管道起的一个名字，这样可以帮助你分清你正在和那个文件或设备通讯，同时掩藏了如缓冲等复杂性。（在内部实现中，文件句柄近似于 C++ 中的流，或者 BASIC 中的 I/O 通道）。

文件句柄能帮助你容易地从不同的地方接收输入，并输出到不同的地方。Perl 能成为一种好的连接语言部分也归功于它能很容易地与很多文件和进程通讯。有很好的符号名字来表示各种不同的外部对象是好的连接语言的一个要求。（注：其他一些令 Perl 成为优秀连接语言的方面包括：8 位无关，可嵌入，以及你可以通过扩展模块嵌入其他语言。它的一致性，以及它的"网络"易用性。它与环境相关性。你可以用许多不同的方法调用它（正如我们前面看到的）。但最重要的是，这门语言本身没有僵化的结构要求，搞得你无法让它 "绕开"你的问题。我们又回到"回字有四种写法"的话题上来了。）

你可以使用 open 创建并关联一个文件。open 函数需要至少两个参数：文件句柄和你希望与文件句柄关联的文件名。Perl 也给你一些预定义（并且预先打开）的文件句柄。STDIN 是我们程序的标准输入，STDOUT 是标准输出。STDERR 是一个额外的输出途径，这样就允许你在将输入转换到你的输出上的时候进行旁路。（注：通常这些文件句柄附着在你的终端上，这样你就可以向你的程序输入并且观察结果，但是它们也可以附着在文件（之类）上。 Perl 能给你这些预定义的句柄是因为你的操作系统已经通过某种方式提供它们了。在 Unix 里，进程从它们的父进程那里继承标准输入，标准输出和标准错误，通常父进程是 shell。 shell 的一个职责就是设置这些 I/O 流，好让这些子进程不用担心它们。）

因为你可以用 open 函数创建用于不同用途（输入，输出，管道）的文件句柄，因此你必须指定你需要哪种类型。 象在命令行中一样，你只需简单地在文件名中加入特定的字符。

```
open(SESAME, "filename")          # 从现存文件中读取

open(SESAME, "<filename")          # （一样的东西，明确地做）

open(SESAME, ">filename")          # 创建文件并写入

open(SESAME, ">>filename")          # 附加在现存文件后面

open(SESAME, "| output-pipe-command")    # 设置一个输出过滤器
```

```
open(SESAME, "input-pipe-command |")      # 设置一个输入过滤器
```

象你看到的一样，文件句柄可以使用任何名字。一旦打开，文件句柄 SESAME 可以被用来访问相应的文件或管道，直到它被明确地关闭（也许你可以猜到使用 close(SESAME) 来关闭文件句柄）或者另外一个 open 语句将文件句柄同别的文件关联起来。（注：打开一个已经打开的文件句柄隐含地关闭第一个文件，让它不能用该文件句柄访问，然后再打开另外一个文件。你必须小心地检查这个是否你要的动作，有时候这种情况会偶然发生，比如当你说 open($handle, $file) 的时候，$handle 碰巧包含着一个常量字串。确保设置 $handle 为某些唯一的东西，否则你就是在同样的文件句柄上打开了一个新文件。或者你可以让 $handle 是未定义，Perl 自然会给你填充它。）

一旦你打开一个用于接受输入的文件句柄，你可以使用读行操作符 <> 来读入一行，因为它是由尖括弧组成，所以也称为尖角操作符。读行操作符用于括住与你需要读入文件相关联的文件句柄（）。当使用空的读行操作符时，将读入命令行上指定的所有文件，当命令行未指定时，读入标准输入 STDIN。（这是很多过滤程序标准的动作）。一个使用 STDIN 文件句柄获取用户输入答案的例子如下：

```
print STDOUT "Enter a number: ";      # 请求一个数字

$number = <STDIN>;                    # 输入数字

print STDOUT "The number is $number.\n";   # 打印该数字
```

在这些 print 语句中的 STDOUT 起什么作用? 实际上这只是你使用输出文件句柄的一种方法。文件句柄可以作为 print 语句的第一个参数，这样，程序就知道将输出送到哪里，在这个例子中，文件句柄是多余的，因为输出无论如何都能输出到 STDOUT。就象 STDIN 是缺省输入一样，STDOUT 是缺省的输出途径。（在我们求平均成绩的例子中，第 18 行我们没有使用 STDOUT 免得使你糊涂）。

如果你测试上一个例子，你会注意到输出中有一个多余的空行。这是因为读行操作符不能自动将新行符从你的输入中删除。（比如，你的输入是 "9\n"）。为了方便你去除新行符，Perl 提供了 chop 和 chomp 函数，chop 不加区别地去处字符串地最后一个字符，并将结果返回，而 chomp 仅删除结束标记（通常是"\n"）同时返回被删除的字符数。你经常能看见这样来处理单行输入：

```
chop($number = <STDIN>);      # 输入数字并删除新行
```

还有另外一种写法：

```
$number = <STDIN>;            # 输入数字

chop($number);                # 删除新行
```

1.5 操作符

正象我们以前提过的一样，Perl 也是一种数学语言。这可以从几个层次上来说明，从基于位的逻辑操作符，到数字运算，以至各种抽象。我们都学过数学，也都知道数学家们喜欢使用各种奇怪的符号。而且更糟的是，计算机学家建立了一套他们自己的奇怪符号。Perl 也有很多这些奇怪符号，幸好大多数符号都是直接取自 C，FORTRAN，sed (1) 和 awk (1)，至少使用这些语言的用户对它们应该比较熟悉。另外，值得庆幸的也许是，在 Perl 中学习这些奇怪符号，可以为你学习其它奇怪语言开一个好头。Perl 内置的操作符可以根据操作数的数目分为单目，双目和三目操作符。也可以根据操作符的位置分为前置（放在操作符前面）和嵌入操作符（在操作符中间）。也可以根据对操作对象不同分类，如数字，字符串，或者文件。稍后我们会提供一个列出所有操作符的表格，但现在我们需要先学习一些简单常用的操作符。

1.5.1 双目算术操作符

算术操作符和我们在学校中学到的没有什么区别。它们对数字执行一些数学运算。比如：

例子	名字	结果
$a + $b	加法	将 $a 和 $b 相加
$a * $b	乘法	$a 和 $b 的积
$a % $b	模	$a 被 $b 除的余数
$a ** $b	幂	取 $a 的 $b 次幂

在这里我们没有提及减法和除法，我们认为你能够知道它们是怎样工作的。自己试一下并看看是不是和你想象的一样（或者直接阅读第三章，单目和双目操作符），算术操作符按照数学老师教我们的顺序执行（幂先于乘法，乘法先于加法）。同样你可以用括弧来是顺序更加明确。

1.5.2 字符串操作符

Perl 中有一个"加号"来串联（将字符串连接起来）字符串。与其它语言不一样，Perl 定义了一个分隔操作符 (.) 来完成字符串的串联，这样就不会跟数字的加号相混淆。

```
$a = 123;

$b = 456;

print $a + $b;      # 打印 579
```

```
print $a . $b;       # 打印 123456
```

同样，字符串中也有“乘号”，叫做“重复”操作符。类似的，采用分隔操作符(x)同数字乘法相区别：

```
$a = 123;

$b = 3;

print $a * $b;       # 打印 369

print $a x $b;       # 打印 123123123
```

这些字符串操作符和它们对应的算术操作符关系紧密。重复操作符一般不会用一个字符串作为左边参数，一个数字作为右边参数。同样需要注意的是 Perl 是怎样将一个数字自动转换成字符串的。你可以将上边所有的数字都用引号括起来，但是它们仍然能够产生同样的输出。在内部，它们已经以正确的方向转换了（从字符串到数字）。

另外一些需要说明的是，字符串连接符同我们前面提到过的双引号一样，里面的表达式中的变量将使用它所包含的内容。并且当你打印一串值时，你同样得到已经连接过的字符串，下面三个语句产生同样的输出：

```
print $a . ' is equal to ' . $b . ".\n";    # 点操作符

print $a, ' is equal to ', $b, ".\n";       # 列表

print "$a is equal to $b.\n";       # 代换
```

什么时候使用哪种写法完全取决于你（但是代换的写法是最容易读懂的）。

x 操作符初看起来没什么用处，但是在有些时候它确实非常有用处，例如下边的例子：

```
print "-" x $scrwid, "\n";
```

如果 $scrwid 是你屏幕的宽度，那么程序就在你的屏幕上画一条线。

1.5.3 赋值操作符

我们已经多次使用了简单的赋值操作符 =，虽然准确地说它不是一个数学操作符。你可以将 = 理解为“设为”而不是“等于”（数学等于操作符 == 才表示等于，如果你现在就开始理解它们的不同之处，你将会省掉日后的许多烦恼。== 操作符相当于一个返回布尔值的函数，而 = 则相当与一个用于修改变量值的过程）。

根据我们在前面已经提到过操作符的分类，赋值操作符属于双目中缀操作符，这就是意味它们在操作符的两边都有一个操作数。操作符的右边可以是任何表达式，而左边的操作数则必须是一个有效的存储空间（也就是一个有效的存储空间，比如一个变量或者是数组中的一个位置）。最普通的赋值操作符就是简单赋值，它计算右边表达式的值，然后将左边的变量设置成这个值：

```
$a = $b;

$a = $b + 5;

$a = $a * 3;
```

注意在最后的这个例子中，赋值操作符使用了同一个变量两次：一次为了计算，一次为了赋值。还有一种方法（从 c 语言中借鉴过来）能起到同样的作用，而且写法更简洁：

```
lvalue operator= expression
```

和下面的这种写法是一样的：

```
lvalue = lvalue operator expression
```

区别只是 lvaule 没有处理两次（只有当给 lvalue 赋值时有副作用，这两种方法才会有差异。但是如果它们的确有差异，通常也是得到你所希望得到的结果。所以你不需要为这个担心）

因此，例如你可以将上面的例子写成：

```
$a *= 3;
```

你可读成"用 3 乘 $a"。Perl 中大多数的双目操作符都可以这么使用，甚至有些你在 c 语言中不能使用的也可以在 Perl 使用：

```
$line .= "\n";       # 给 $line 附加一个新行

$fill x=80;       # 把字串变成自填充 80 遍

$val ||= "2";       # 如果 $val 不为真则把它设置为 2
```

在我们求平均成绩的例子中（注：我们还没忘呢，你呢？），第 6 行包含两个字符串连接，其中一个是赋值操作符。同时第 14 行有一个 +=。

不管你使用哪种赋值操作符，最左边变量的值被返回作为整个赋值表达式的值。（注：这一点和象 Pascal 这样的语言不同，在那样的语言里赋值是一个语句，不返回值。我们前面

说过赋值类似一个过程，但是请记住在 Perl 里，每个过程都返回值。）C 语言的程序员不会感到奇怪，因为他们已经知道用下面的方法来使变量清零：

```
$a = $b = $c =0;
```

你也会经常看到赋值语句在 while 循环中作为条件，例如求平均成绩的例子中第 4 行。

真正能使 c 程序员惊讶的是在 Perl 中，赋值语句返回实际的变量作为 lvalue。因此你可以在同一个语句中多次改变同一个变量的值。例如可以使用下面的语句：

```
($temp -= 32) *= 5/9
```

将华氏温度转换成摄氏温度。这也是为什么在本章的前面我们能使用下面的语句：

```
chop ($number = <STDIN>);
```

上面的语句能将 $number 最后的值进行 chop 操作。通常，当你想在拷贝的同时进行一些其它操作。你就可以利用这个特性。

1.5.4 单目算术操作符

如果你觉得 $variable += 1 还是不够精简，同 c 一样，Perl 有一个更短的方法自增变量。使用自增(自减)操作符可以将变量的值简单地加上（减去）一。你可以将操作符放在变量的任何一边，这取决于你希望操作符什么时候被执行：

例子	名字	结果
++$a, $a++	自增	向 $a 加一
--$a, $a--	自减	从 $a 中减一

如果你将自增（减）操作符放在变量的前边，变量就成为"预增"变量。变量的值在它被引用前改变。如果放在变量的后边，被称为"后增变量"，它在被引用后改变。如：

```
$a = 5;          # 给 $a 赋予 5

$b = ++$a;       # $b 被赋予 $a 自增之后的值，6

$c = $a--;       # $c 被赋予 6，然后 $a 自减为 5
```

平均成绩例子中第 15 行增加成绩个数，这样我们就知道我们统计了多少个成绩。这里使用了一个后增操作符（$scores++），但是在这个例子中，实际上无所谓使用哪一种，因为表达式在一个空环境里，在这种环境中，表达式只是为了得到增加变量的值这个副作用，

而把返回的值丢弃了。（注：优化器会注意到这些并且把后增操作符优化成预增操作符，因为它执行起来略微快一些。（你不必知道这些，我们只是希望你听到这个后会开心些。））

1.5.5 逻辑操作符

逻辑操作符，也称为"短路"操作符，允许程序不使用嵌套 if 语句，而根据多个条件来决定执行流程。他们之所以被称为"短路"操作符，是因为当认为左边的参数能够提供足够的信息来决定整个值的时候，它们跳过（短路）执行右边的参数。这不仅仅是为了效率。你可以依靠这种"短路"的特性来避免执行为左边代码做防护的右边的代码。比如你可以说 "California or bust!"，在 Perl 中将不会有 bust（假设你已经到达 California）。

实际上 Perl 有两组逻辑操作符，一组传统操作符是借鉴了 C 中的操作符，另外一组比较新（或者更旧的）低级操作符借鉴了 BASIC 中的操作符。两组操作符在使用适当的情况下都很易读。如果你希望逻辑操作符比逗号优先级更高，那么 c 的符号操作符运转得很好，而如果你希望逻辑操作符比逗号优先级更低时，BASIC 的基于词的操作符比较适合。在很多情况下，它们得到相同的结果。使用哪一组你可以根据你自己的喜好来选择。（你可以在第三章中的"逻辑与，或，非和异或"小节找到对比例子）。虽然由于优先级不同，这两组操作符不能互换，但是一旦它们已经被解析，操作符的效果是一样的，优先级只能在它们参数的范围内起作用。表 1-1 列出了逻辑操作符。

表 1-1 逻辑操作符

例子	名字	结果
$a && $b	与	如果 $a 为假则为 $a，否则为 $b
$a	$b	或	如果 $a 为真则为 $a，否则为 $b
! $a	非	如果 $a 为假则为真
$a and $b	与	如果 $a 为假则为 $a，否则为 $b
$a or $b	或	如果 $a 为真则为 $a，否则为 $b
not $a	非	如果 $a 为假则为真
$a xor $b	异或	如果 $a 或 $b 为真，但不能同时为真

因为逻辑操作符有"短路"的特性，因此它们经常被使用在条件执行代码里面. 下面的代码（平均分数例子中的第 4 行）试图打开文件 grades：

```
open(GRADES, "grades") or die "Can't open file grades: $!\en";
```

如果成功打开了文件，将跳到下一行继续执行，如果不能打开文件，程序将打印一个错误信息并停止执行.

从字面意义上看,这行代码表示 "Open gradues or bust",短路操作符保留了虚拟流程.重要的动作在操作符的左边，右边藏着第二个动作（$! 变量包含了操作系统返回的错误信息，参看 28 章，特殊名字），当然，这些逻辑操作符也可以用在传统的条件判断中，例如 if 和 while 语句中. 赋值

1.5.6 比较操作符

比较操作符告诉我们两个标量值（数字或字符串）之间的比较关系. 这里也有两组关系比较操作符，一组用于数字比较，另一组用于字符串比较. （两组操作符都要求所有的参数都必须先转换成合适的类型），假设两个参数是 $a 和 $b，我们可以：

比较	数字	字串	返回值
等于	==	eq	如果 $a 等于 $b 返回真
不等于	=	ne	如果 $a 不等于 $b 返回真
小于	<	lt	如果 $a 小于 $b 返回真
大于	>	gt	如果 $a 大于 $b 返回真
小于或等于	<= </td>	le	如果 $a 不大于 $b 返回真
比较	<=>	cmp	相等时为 0，如果 $a 大为 1 如果 $b 大为 -1

最后一对操作符（<=> 和 cmp）完全是多余的，但是在 sort 函数中，它们非常有用（参看 29 章）. （注：有些家伙认为这样的冗余是错误的，因为它让语言无法变得最小，或者正交. 不过 Perl 不是正交的语言；它是斜交的语言. 我们的意思是 Perl 并没有强迫你总是走直角. 有时候你就是想走三角形斜边到你的目的. TMTOWTDI 与短路有关，而短路与效率有关.

1.5.7 文件测试操作符

在你盲目地对文件操作之前，你可以使用文件测试操作符来测试文件的特定属性. 最普通的文件属性就是文件是否存在. 例如，在你试图打开新的邮件别名文件的之前，最好还是确定一下你的邮件别名文件是否存在. 下面是一些文件测试操作符：

例子	名字	结果
-e $a	存在	如果在 $a 中命名的文件存在则为真
-r $a	可读	如果在 $a 中命名的文件可读则为真
-w $a	可写	如果在 $a 中命名的文件可写则为真

-d $a	目录	如果在 $a 中命名的文件是目录则为真
-f $a	文件	如果在 $a 中命名的文件是普通文件则为真
-T $a	文本文件	如果在 $a 中命名的文件是文本文件则为真

你可以象下面例子中一样使用它们：

-e "/usr/bin/perl" or warn "Perl is improperly installed\n"; -f "/vmlinuz" and print "I see you are a friend of Linus\n";

注意普通文件并不等同于文本文件，二进制文件 /vmlinuz 是普通文件，而不是文本文件．文本文件与二进制文件对应．而普通文件则是和目录及设备等非普通文件相对应．

在 Perl 中有很多文件测试操作符，很多没有在这里被列出来．这些操作符大多数都是单目的布尔操作符，它们只有一个操作数（一个指向文件名或文件句柄的标量），并返回一个真或假的布尔值．其中一小部分操作符返回一些其它有趣的东西，例如文件的大小和存在的时间，但是你可以在你需要用的时候查阅第三章的"命名单目操作符和文件测试操作符"小节．

1.6 流程控制

到目前为止，除了一个大的例子外，所有其它的例子都是只有线性代码；我们按顺序执行代码.我们也已经接触到了一些例子中,利用"短路"操作符来使得单个命令被(或不被)执行.虽然你可以写出很多有用的线性代码（很多的 CGI 脚本就是这个类型的），但是使用条件表达式和循环机制可以写出很多更加强大的程序．它们在一起被称为控制结构．因此你也可以认为 Perl 是一种控制语言．

为了能够控制，你必须能够做决定，为了做决定，你必须知道假和真之间的差别．

1.6.1 什么是真

我们前面已经涉及到真的概念，（注：严格说，我们这么说并不正确）．同时我们也提到了一些返回真或假的操作符．在进行更深入的讨论之前，我们应该给所提到真和假准确的定义．Perl 中真的判断与大多数的计算机语言中的稍微有些不同，但是通过一段时间的使用，你会对它有更多的认识（实际上,我们希望你能通过阅读下面的内容获得很多的认识）．

基本上，Perl 以自明的方式处理真．这是一种灵活的方法，你可以确定出几乎所有事物的真值．Perl 使用非常实用的方法来定义真，真的定义依赖于你所处理的事物的类型．事实上，真值的种类要比不真的种类多得多．

在 Perl 中，真总是在标量环境中处理．除此外没有任何的类型强制要求．下面是标量可以表示的不同种类的真值：

- 除了""和"0"，所有字符串为真
- 除了 0，所有数字为真
- 所有引用为真
- 所有未定义的值为假．

1.6.2 If 和 unless 语句

早些时候，我们已经看到一个逻辑操作符如何起一个条件的作用．一个比逻辑操作符稍微复杂一些的形式是 if 语句。if 语句计算一个真假条件（布尔表达式）并且在条件为真时执行一个代码段。

```
if ($debug_level > 0) {
    # Something has gone wrong.  Tell the user.
    print "Debug: Danger, Will Robinson, danger!\en";
    print "Debug: Answer was '54', expected '42'.\en";
}
```

一个代码段是由一对花括弧括在一起的一些语句。因为 if 语句执行代码段，因此花括弧是必须的。如果你对一些其它语言比较熟悉，例如 C，你会发现它们的不同之处，在 C 中，如果你只有一条语句，那么你可以省略花括弧。但是在 Perl 中，花括弧是必须的。

有些时候，当一个条件满足后执行一个代码段并不能满足要求，你可能要求在条件不满足的情况下执行另外一个代码段。当然你可以用两个 if 语句来完成，其中一个正好和另一个相反，为此 Perl 提供了更好的方法，在第一个代码段后边，if 有一个可选的条件叫 else，当条件为假时运行其后的代码段。（经验丰富的程序员对此不会感到惊讶。）

当你有多于两个选择的时候，你可以使用 elsif 条件表达式来表示另一个可能选择，（经验丰富的程序员可能对"elsif"的拼写感到惊讶，不过我们这里没有人准备道歉，抱歉。）

```
if ($city eq "New York") {
    print "New York is northeast of Washington, D.C.\en";
```

```
}
elsif ($city eq "Chicago") {
   print "Chicago is northwest of Washington, D.C.\en";
}
elsif ($city eq "Miami") {
   print "Miami is south of Washington, D.C.  And much warmer!\en";
}
else {
   print "I don't know where $city is, sorry.\en";
}
```

if 和 elsif 子句按顺序执行，直到其中的一个条件被发现是真的或到了 else 条件为止，当发现其中的一个条件是真的，就执行它的代码段，然后跳过所有其余的分支。有时候，你可能希望在条件为假的时候执行代码，而不想在条件为真时执行任何代码。使用一个带 else 的空 if 语句看起来会比较零乱。而用否定的 if 会难以理解，这就象在英语中说"如果这不是真的，就做某事"一样古怪。在这种情况下，你可以使用 unless 语句：

```
unless ($destination eq $home) { print "I'm not going home.\en"; }
```

但是，没有 elsunless。这通常解释为 if 的一个特性。

1.6.3 循环

Perl 有四种循环语句的类型：while，until，for 和 foreach。这些语句可以允许一个 Perl 程序重复执行同一些代码。

1.6.3.1 while 和 until 语句

除了是重复执行代码段以外，While 和 until 语句之间的关系就象 if 和 unless 的关系一样。首先，检查条件部分，如果条件满足（while 语句是真，until 是假），执行下面代码段，例如：

```
while ($tickets_sold < 10000) {
```

```
    $available = 10000 - $tickets_sold;
    print "$available tickets are available.  How many would you like: ";
    $purchase = <STDIN>;
    chomp($purchase);
    $tickets_sold += $purchase;
     }
```

注意如果没满足最初的条件，就根本不会进入循环。例如，假设我们已经卖出 10000 张票，我们可能希望执行下面的代码：

```
print "This show is sold out, please come back later.\en";
```

在我们前面的"平均分"例子中，第 4 行：

```
while ($line = <GRADES>) {
```

这句代码将文件的下一行内容赋值给变量 $line 并且返回 $line 的值，因此 while 语句的条件表达式为真，你也许想知道，当遇到空白行时，Perl 是不是返回假并且过早地退出循环。答案是否定的，如果你还记得我们先前学过的内容，那么理由是非常明显的。读行操作符在字符串最后并不去掉新行符，因此空白行的值是"\n"。并且我们知道"\n"不是假值。因此条件表达式为真，并且循环将继续。

另一方面，当我们最后达到文件结束的时候，读行操作符将返回未定义值，未定义值总是被解释为假。并且循环结束，正好是我们希望结束的时候。在 Perl 中不需要明确地测试 eof 的返回值，因为输入操作符被设计成可以在条件环境下很好的工作。

实际上，几乎所有东西都设计成在条件环境下能很好工作，如果在一个标量环境中使用一个数组，就会返回数组的长度。因此你使用下面的代码处理命令行参数：

```
while (@ARGV) {
    process(shift @ARGV);
}
```

每次循环，shift 操作符都从参数数组中删除一个元素（同时返回这个元素），当数组 @ARGV 用完时循环自动退出，这时候数组长度变为 0，而在 Perl 中认为 0 为假。所以数组本身已经变为"假"。（注：这是 Perl 程序员的看法，因此我们没有比较拿 0 和 0 比较以证实它是否为假。但是其他语言却强迫你这么做，如果你不写 while(@ARGV = 0) 就

不退出。这样做不论对你还是对计算机还是以后维护你的代码的人来说，都是效率低下的做法。）

1.6.3.2 for 语句

另外一个循环语句就是 for 循环。for 循环和 while 循环非常相似，但是看起来有很多不同之处。（C 语言程序员会觉得和 C 中的 for 循环非常相似。）

```
for ($sold = 0; $sold < 10000; $sold += $purchase) {
    $available = 10000 - $sold;
    print "$available tickets are available.  How many would you like: ";
    $purchase = <STDIN>;
    chomp($purchase);
        }
```

for 循环在园括弧中有三个表达式：一个表达式初始化循环变量，一个对循环变量进行条件判断，还有一个表达式修改条件变量。当一个 for 循环开始时，设置初始状态并且检查条件判断，如果条件判断为真，就执行循环体。当循环体中的语句，修改表达式执行。并且再次检查条件判断，如果为真，循环体返回下一个值，如果条件判断值为真，循环体和修改表达式将一直执行。（注意只有中间的条件判断才求值，第一个和第三个表达式只是修改了变量的值，并将结果直接丢弃！）

1.6.3.3 foreach 语句

Perl 中最后一种循环语句就是 foreach 语句，它是用来针对一组标量中的每一个标量运行同一段程序，例如一个数组：

```
foreach $user (@users) {
    if (-f "$home{$user}/.nexrc") {
        print "$user is cool\&\.\.\.  they use a perl-aware vi!\en";
    }
}
```

不同于 if 和 while 语句，foreach 的条件表达式时在列表环境中，而不是标量环境。因此表达式用于生成一个列表（即使列表中只有一个标量）。然后列表中的每个元素按顺序作为循环变量，同时循环体代码针对每个元素执行一次。注意循环变量直接指向元素本身，而不是它的一个拷贝，因此，修改循环变量，就是修改原始数组。

你将发现在 Perl 程序中，foreach 循环比 for 循环要多得多，这是因为 Perl 经常使用一些需要使用 foreach 遍历的各种列表。你经常会看到使用下面的代码来遍历散列的关键字：

```
foreach $key (sort keys %hash) {
```

实际上"平均分"例子中的第 9 行也用到了它。

1.6.3.4 跳出控制结构: next 和 last

next 和 last 操作符允许你在循环中改变程序执行的方向。你可能会经常遇到一些的特殊情况，碰到这种情况时你希望跳过它，或者想退出循环。比如当你处理 Unix 账号时，你也许希望跳过系统账号（比如 root 或 lp），next 操作符允许你将跳至本次循环的结束，开始下一个循环。而 last 操作符允许你跳至整个循环的结束，如同循环条件表达式为假时发生的情况一样。例如在下面例子中，你正在查找某个特殊账号，并且希望找到后立即退出循环，这时候，last 就非常有用了：

```
foreach $user (@users) {
    if ($user eq "root" or $user eq "lp") {
        next;
    }
    if ($user eq "special") {
        print "Found the special account.\en";
        # 做些处理
        last;
    }
}
```

当在循环中做上标记，并且指定了希望退出的循环，next 和 last 就能退出多重循环。结合语句修饰词（我们稍后会谈到的条件表达式的另外一种形式），能写出非常具有可读性的退出循环代码（如果你认为英语是非常容易读懂的）：

```
LINE: while ($line = <ARTICLE>) {
    last LINE if $line eq "\en"; # 在第一个空白行处停止
    next LINE if $line =~ /^#/; # 忽略注释行
    # 你的东西放在这里
}
```

你也许会说：稍等，在双斜杠内的 ^# 看起来并不象英语。没错，这就是包含了一个正则表达式的模式匹配（虽然这是一个很简单的正则表达式）。在下一节中，我们将讲述正则表达式。Perl 是最好的文本处理语言，而正则表达式是 Perl 文本处理的核心。

1.7 正则表达式

正则表达式（也可以表示为 regexes，regexps 或 Res）广泛使用在很多搜索程序里，比如：grep 和 findstr，文本处理程序如：sed 和 awk，和编辑器程序，如：vi 和 emacs。一个正则表达式就是一种方法，这种方法能够描述一组字符串，但不用列出所有的字符串。（注：一本关于正则表达式的概念的好书是 Jeffrey Friedl 的“Mastering Regular Expressions”（O'Reilly & Associates）

其它的一些计算机语言也提供正则表达式（其中的一些甚至宣扬“支持 Perl5 正则表达式”）但是没有一种能象 Perl 一样将正则表达式和语言结合成一体。正则表达式有几种使用方法，第一种，也是最常用的一种，就是确定一个字符串中是否匹配某个模式，因为在一个布尔环境中它们返回真或假。因此当看见 /foo/ 这样的语句出现在一个条件表达式中，我们就知道这是一个普通的模式匹配操作符：

```
if (/Windows 95/) { print "Time to upgrade?\n" }
```

第二种方法，如果你能将一个模式在字符串中定位，你就可以用别的东西来替换它。因此当看见 s/foo/bar/ 这样的语句，我们就知道这表示将 foo 替换成 bar。我们叫这是替换操作符。同样，它根据是否替换成功返回真或假。但是一般我们需要的就是它的副作用：

```
s/Windows/Linux/;
```

最后，模式不仅可以声明某地方是什么，同样也可以声明某地方不是什么。因此 split 操作符使用了一个正则表达式来声明哪些地方不能匹配。在 split 中，正则表达式定义了各

个数据域之间定界的分隔符。在我们的"平均分"例子中，我们在第 5 和 12 行使用了两次 split，将字符串用空格分界以返回一列词。当然你可以用正则表达式给 split 指定任何分界符：

```
($good, $bad, $ugly) = split(/,/, "vi,emacs,teco");
```

（Perl 中有很多修饰符可以让我们能轻松完成一些古怪的任务，例如在字符匹配中忽略大小写。我们将在下面的章节中讲述这些复杂的细节）

正则表达式最简单的应用就是匹配一个文字表达式。象上面的例子中，我们匹配单个的逗号。但如果你在一行中匹配多个字符，它们必须按顺序匹配。也就是模式将寻找你希望要的子串。下面例子要完成的任务是，我们想显示一个 html 文件中所有包含 HTTP 连接的行。我们假设我们是第一次接触 html，而且我们知道所有的这些连接都是有"http:"，因此我们写出下面的循环：

```
while ($line = <FILE>) {
   if ($line =~ /http:/) {
      print $line;
   }
}
```

在这里，=~ 符号（模式绑定操作符）告诉 Perl 在 $line 中寻找匹配正则表达式"http:"，如果发现了该表达式，操作符返回真并且执行代码段（一个打印语句）。（注：非常类似于 Unix 命令 grep 'http:' file 做的事情，在 MS-DOS 里你可以用 find 命令，但是它不知道如何做更复杂的正则表达式。（不过，Windows NT 里名字错误的 findstr 程序知道正则表达式。））另外，如果你不是用 =~ 操作符，Perl 会对缺省字符串进行操作。这就是你说"Eek，帮我找一下我的联系镜头!"，别人就会自动在你周围寻找，而不用你明确告诉他们。同样，Perl 也知道当你没有告诉它在那里寻找的时候，它会在一个缺省的地方寻找。这个缺省的字符串就是 $_ 这个特殊标量。实际上，$_ 并不是仅仅是模式匹配的缺省字符串。其它的一些操作符缺省也使用 $_ 变量。因此一个有经验的 Perl 程序员会将上个例子写成：

```
while (<FILE>) {
   print if /http:/;
}
```

（这里我们又提到另外一个语句修饰词。阴险的小动物。）

上边的例子十分简洁，但是如果我们想找出所有连接类型而不是只是 http 连接时怎么办？我们可以给出很多连接类型：象“http:”，“ftp:”，“mailto:”等等。我们可以使用下面的代码来完成，但是当我们需要加进一种新的连接类型时怎么办？

```
while (<FILE>) {
    print if /http:/;
    print if /ftp:/;
    print if /mailto:/;
    # 下一个是什么？
}
```

因为正则表达式是一组字符串的抽象，我们可以只描述我们要找的东西：后面跟着一个冒号的一些字符。用正则表达式表示为 /[a-zA-Z]+:/，这里方括弧定义了一个字符表，a-z 和 A-Z 代表所有的字母字符（划线表示从开头的字符到结尾字符中间的所有字母字符）。+ 是一个特殊字符，表示匹配“+ 前边内容一次或多次”。我们称之为量词，这是表示允许重复多少次的符号。（这里反斜杠不是正则表达式的一部分，但是它是模式匹配操作符的一部分。在这里，反斜杠与包含正则表达式的双引号起同样的作用）。

因为字母字符这种类型经常要用到，因此 Perl 定义了下面一些简写的方式：

名字	ASCII 定义	代码
空白	[\t\n\r\f]	\s
词	[a-zA-Z_0-9]	\w
数字	[0-9]	\d

注意这些简写都只匹配单个字符，比如一个 \w 匹配任何单个字符，而不是整个词。（还记得量词 + 吗？你可以使用 \w+ 来匹配一个词）。在 Perl 中，这些通配符的大写方式代表的意思和小写方式刚好相反。例如你可以使用 \D 表示左右非数字字符。

我们需要额外注意的是，\w 并不是总等于 [a-zA-Z0-9]（而且 \d 也不总等于 [0-9]），这是因为有些系统自定义了一些 ASCII 外的额外的字符，\w 就代表所有的这些字符。较新版本的 Perl 也支持 Unicode 字符和数字特性，并且根据这些特性来处理 Unicode 字符。（Perl 认为 \w 代表表意文字）。

还有一种非常特别的字符类型，用“.”来表示，这将匹配所有的字符。（注：除了通常它不会匹配一个新行之外。如果你有怀疑，点“.”在 grep (1) 里通常也不匹配新行。）例如，/a./ 将会匹配所有含有一个“a”并且“a”不是最后一个字符的字符串。因而它将匹配“at” 或“am”

甚至“a!”，但不匹配“a”，因为没有别的字母在“a”的后面。同时因为它在字符串的任何地方进行匹配，所以它将匹配“oasis”和“camel”，但不匹配“sheba”。它将匹配 “caravan”中的第一个”a”。它能和第二个“a”匹配，但它在找到第一个合适的匹配后就停止了。查找方向是由左向右。

1.7.1 量词

刚才我们讨论的字符和字符类型都只能匹配单个字符，我们提到过你可以用 \w+ 来匹配多个 “文本”字符。这里 + 就是量词，当然还有其它一些。所有的量词都放在需要多重匹配的东西后边。

最普通的量词就是指定最少和最多的匹配次数。你可以将两个数字用花括弧括起来，并用逗号分开。例如，你想匹配北美地区的电话号码，使用 \d{7,11} 将匹配最少 7 位数字，但不会多于 11 位数字。如果在括弧中只有一个数字，这个数字就指定了最少和最多匹配次数，也就是指定了准确的匹配次数（其它没有使用量词的项我们可以认为使用了{1}）。

如果你的花括弧中有最少次数和逗号但省略了最大次数，那么最大次数将被当作无限次数。也就是说，该正则表达式将最少匹配指定的最少次数，并尽可能多地匹配后面的字符串。例如 \d{7} 将匹配开始的七位号码（一个本地北美电话号码，或者一个较长电话号码的前七位），但是当你使用 \d{7,} 将会匹配任何电话号码，甚至一个国际长途号码（除非它少于七位数字）。你也可以使用这种表达式来表示“最多”这个含义，例如。{0,5} 表示至多五个任意字符。

一些特殊的最少和最多地经常会出现，因此 Perl 定义了一些特殊的运算符来表示他们。象我们看到过的 +，代表 {1,}，意思为“最少一次”。还有 *，表示 {0,}，表示“零次或多次”. ? 表示 {0,1}，表示“零或一次”。

对于量词而言，你需要注意以下一些问题。首先，在缺省状态下，Perl 量词都是贪婪的，也就是他们将尽可能多地匹配一个字符串中最大数量的字符，例如，如果你使用 /\d+/ 来匹配字符串“1234567890”，那么正则表达式将匹配整个字符串。当你使用“.”时特别需要注意，例如有下边一个字符串：

```
larry:JYHtPh0./NJTU:100:10:Larry Wall:/home/larry:/bin/tcsh
```

并且想用 /.+:/ 来匹配“larry:”，但是因为 + 是贪婪的，这个模式将匹配一直到 /home/larry: 为止。因为它尽可能多地匹配直到最后出现的一个冒号。有时候你可以使用反向的字符类来避免上边的情况，比如使用 /[^:]+:/，表示匹配一个或多个不是冒号的的字符（也是尽可能多），这样正则表达式匹配至第一个冒号。这里的 ^ 表示后边的字符表的反集。（注：抱歉，我们不是有意选用这个名词的，所以别骂我们。这也是 Unix 里写

反字符表的习惯方式。）另外需要仔细观察的就是，正则表达式将尽早进行匹配。甚至在它变得贪婪以前。因为字符串扫描是从左向右的，这就意味着，模式将尽可能在左边得到匹配。尽管也许在后边也能得到匹配。（正则表达式也许贪婪，但不会错过满足条件的机会）。例如，假设你在使用替换命令（s///）处理缺省字符串（变量 $_），并且你希望删除中间的所有的 x。如果你说：

```
$_ = "fred xxxxxxx barney";

s/x*//;
```

但是上面的代码并没有达到预想的目的，这是因为 x*（表示零次或多次"x"）在字符串的开始匹配了空字符串，因为空字符串具有零字符宽度，并且在 fred 的 f 字符前正好有一个空字串。（注：千万不要感觉不爽，即使是作者也经常惨遭毒手。）

还有一件你必须知道的事情，缺省时量词作用在它前面的单个字符上，因此 /bam{2}/ 将匹配 "bamm"而不是"bambam"。如果你要对多于一个字符使用量词，你需要使用圆括弧，因此为了匹配"bambam"需要使用 /(bam){2}/。

1.7.2 最小匹配

如果你在使用老版本的 Perl 并且你不想使用贪婪匹配，你必须使用相反的字符表（实际上，你还是在使用不同形式的贪婪匹配）。

在新版本 Perl 中，你可以强制进行非贪婪匹配。在量词后面加上一个问号来表示最小匹配。我们同样的用户名匹配就可以写成 /.*?:/。这里的 .*? 现在尽可能少地匹配字符，而不是尽可能多的匹配字符。所以它将停止在第一个冒号而不是最后一个。

1.7.3 把钉子敲牢

你无论什么时候匹配一个模式，正则表达式都尝试在每个地方进行匹配直到找到一个匹配为止。一个锚点允许你限制模式能在什么地方匹配。基本来说，锚点匹配一些"无形"的东西，这些东西依赖于周边的特殊环境。你可以称他们为规则，约束或断言。不管你怎么称呼它，它都试图匹配一些零宽度的东西，或成功或失败（失败仅仅意味着这个模式用这种特殊的方法不能匹配。如果还有其它方法可以试的话，该模式会继续用其它方法进行匹配。）

特殊符号 \b 匹配单词边界，就是位于单词字符 （\w）和非单词字符（\W）之间的零宽度的地方。（字符串的开始和结尾也被认为是非单词字符）。例如: /\bFred\b/ 将会匹配"The Great Fred"和"Fred the Great"中的 Fred，但不能匹配"Frederick the Great"，因为在"Frederick"中的"d"后面没有跟着非单词字符。

同理，也有表示字符串开始和结尾的锚点，^ 如果放在模式中的第一个字符，将匹配字符串的开始。因此，模式 /^Fred/ 将匹配"Frederick the Great"中的"Fred"，但不配 "The Great Fred"中的"Fred"。相反，/Fred^/ 两者都不匹配。（实际上，它也没有什么意义。）美元符号（$）类似于^，但是 $ 匹配字符串的结尾而不是开头。（注：这么说有点过于简单了，因为我们在这里假设你的字串不包含新行；^ 和 $ 实际上是用于行的开头和结尾，而不是用于字串的。我们将在第五章，模式匹配里通篇强调这一点（做我们做得到的强调）。）

现在你肯定已经理解下面的代码：

```
next LINE if $line =~ /^#/;
```

这里我们想做的是"当遇到以 # 开头的行，则跳至 LINE 循环的下一次循环。"

早些时候，我们提到过 \d{7,11} 将匹配一个长度为 7 到 11 位的数字。但是严格来讲，这个语句并不十分正确：当你在一个真正的模式匹配操作符中使用它的时候，如 /\d{7,11}/，它并不排除在 11 位匹配的数字外的数字。因此你通常需要在量词两头使用锚点来获取你所要的东西。

1.7.4 反引用

我们曾经提到过可以用圆括弧来为量词包围一些字符。同样，你也可以使用圆括弧来记住匹配到的东西。正则表达式中的一对圆括弧使得这部分匹配到的东西将被记住以供以后使用。它不会改变匹配的方式，因此 /\d+/ 和 /(\d+)/ 仍然会尽可能多地匹配数字。但后边的写法能够把匹配到的数字保存到一个特殊变量中，以供以后反向引用。

如何反向引用保存下来的匹配部分决定于你在什么地方使用它，如果在同一个正则表达式中，你可以使用反斜杠加上一个整数。数字代表从左边开始计数左圆括弧的个数（从一开始计数）。例如，为了匹配 HTML 中 的标记如"**Bold**"，你可以使用 /<(.*?)>.*?<\/\1>/。这样强制模式的两个部分都匹配同样的字符串。在此，这个字符串为"B".

如果不在同一个正则表达式，例如在替换的置换部分中使用反引用，你可以使用 $ 后边跟一个整数。看起来是一个以数字命名的普通标量变量。因此，如果你想将一个字符串的前两个词互相调换，你可以使用下面的代码：

```
/(\S+)\s+(\S+)/$2 $1/
```

上边代码的右边部分（第二第三个反斜杠之间）几乎就是一个双引号字符串，在这里可以代换变量。包括反向引用。这是一个强大的概念：代换（在有所控制的环境中）是 Perl 成为一种优秀的文本处理语言的重要原因之一。另外一个原因就是模式匹配，当然，正则表达式方便将所需要的东西分离出来，而代换可以方便将这些东西放回字符串。

1.8 列表处理

本章早些时候，我们提过 Perl 有两种主要的环境：标量环境（处理单个的事物）和列表环境（处理复数个事物）。我们描述过的很多传统操作符都是严格在标量环境下执行。他们总是有单数的参数（或者象双目操作符一样有一对单数的参数）并且产生一个单数的返回值。甚至在列表环境中亦如此。当你使用下面的代码：

```
@array = (1 + 2, 3 - 4, 5 * 6, 7 / 8);
```

你知道右边的的列表中包含四个值，因为普通数学操作符总是产生标量，即使是在给一个数组赋值这样的列表环境中。

但是,有一些 Perl 操作符能根据不同的环境产生一个标量或列表环境。他们知道程序需要标量环境还是列表环境。但是你如何才能知道？下面是一些关键的概念，当你理解这些概念之后，你就能很容易地知道需要标量还是列表了。

首先，列表环境必须是周围的事物提供的，在上个例子中，列表赋值提供了列表环境。早些时候，我们看到过 foreach 循环也能提供列表环境。还有 print 操作符也能提供。但是你不必逐个学习他们。

如果你通读本书其余部分种不同的语法说明，你会看到一些操作符定义为使用 LIST 作为参数。这就是提供列表环境的操作符。在本书中，LIST 作为一种特殊的技术概念表示"提供列表环境的句法"。例如，你观察 sort，你可以总结为：

```
sort LIST
```

这表示，sort 给它的参数提供了一个列表环境。

其次，在编译的时候（当 Perl 分析你的程序，并翻译成内部操作码的时候），任何使用 LIST 的操作符给 LIST 的每个语法元素提供了列表环境。因此，在编译的时候，每个顶层操作符和 LIST 中的每个元素都知道 Perl 假设它们使用自己知道的方法生成最好的列表。例如当你使用下面的代码：

```
sort @dudes, @chicks, other();
```

那么 @dudes，@chicks，和 other() 都知道在编译的时候 Perl 假设它们都产生一个列表值而不是一个标量值。因此编译器产生反映上述内容的内部操作码。

其后，在运行时候（当内部执行码被实际解释的时候），每个 LIST 成员按顺序产生列表，然后将所有单独的列表连接在一起（这很重要），形成一个单独的列表。并且这个平面的一维列表最后由那些需要 LIST 的函数使用。因此如果 @dudes 包含（Fred, Barney），

@chicks 包含 (Wilma, Betty)，而 other() 函数返回只有一个元素的列表 (Dino)，那么 LIST 看起来就象下面一样：

```
(Fred, Barney, Wilma, Betty, Dino)
```

sort 返回的 LIST：

```
(Barney, Betty, Dino, Fred, Wilma)
```

一些操作符产生列表（如 keys），而一些操作符使用列表（如 print），还有其它一些操作符将列表串进其它的列表（如 sort）。最后的这类操作符可以认为是筛选器。同 shell 不一样，数据流是从右到左，因为列表操作符从右开始操作参数，你可以在一行中堆叠几个列表操作符：

```
print reverse sort map {lc} keys %hash;
```

这行代码获取 %hash 的关键字并将它们返回给 map 函数，map 函数使用 lc 将所有的关键字转换成小写，并将处理后的结果传给 sort 函数进行排序，然后再传给 reverse 函数， reverse 函数将列表元素颠倒顺序后，传给 print 函数打印出来。

正如你看到的一样，使用 Perl 描述比使用英语要简单的多。

在列表处理方面还有很多方法可以写出很多更自然的代码。在这里我们无法列举所有方法。但是作为一个例子，让我们回到正则表达式，我们曾经谈到在标量中使用一个模式来看是否匹配。但是如果你在一个列表环境中使用模式，它将做一些其它的事情：它将获得所有的反引用作为一个列表。假设你在一个日志文件或邮箱中搜索。并且希望分析一些包含象 “12:59:59 am” 这样形式时间的字符串，你可以使用下面的写法：

```
($hour, $min, $sec, $ampm) = /(\ed+):(\ed+):(\ed+) *(\ew+)/;
```

这是一种同时设置多个变量的简便方法,但是你也可以简单的写:

```
@hmsa = /(\ed+):(\ed+):(\ed+) *(\ew+)/;
```

这里将所有四个值放进了一个数组。奇秒的，通过从 Perl 表达式能力中分离正则表达式的能力，列表环境增加了语言的能力。有些人可能不同意，但是 Perl 除了是一种斜交语言外，它还的确是一种正交语言.

1.9 你不知道但不伤害你的东西(很多)

最后，请允许我们再次回顾 Perl 是一种自然语言的概念。自然语言允许使用者有不同的技巧级别，使用语言不同的子集，并且边学边用。通常在知道语言的全部内容之前,他们就可

以很好地运用语言。你不知道 Perl 的所有内容，正象你不知道英语的所有内容一样。但这在 Perl 文化中是明确支持的。即使我们还没有告诉如何写自己的子过程也这样，但是你能够使用 Perl 来完成你的工作。我们还没有开始解释如何来看待 Perl 是一种系统管理语言，或者一种原型语言，或者一们网络语言或面向对象的语言，我们可以写一整章关于这些方面的内容（我们已经写了）。但是最后，你必须建立起你对 Perl 的看法.就象画家自己造成创造力的痛苦一样。我们能教你我们怎么画，但是我们不能教你该化什么。并且可能有不同的方法去做同一件事。

第二章 集腋成裘

因为我们准备从小处开始，所以我们在随后几章里将逐步从小到大。也就是说，我们将发起一次从零开始的长征，从 Perl 程序里最小的构件开始，逐步把它们更精细的组件，就象分子是由原子组成的那样。这样做的缺点是你在卷入细节的洪流之前没有获得必要的全景蓝图。这样做的好处是你能随着我们的进展理解我们的例子。（当然，如果你是那种从宏观到微观的人，你完全可以把书反过来，从后面开始向前看。）

每一章都是建筑在前面章节的基础之上的（如果你从后向前看，就是后面几章），所以如果你喜欢这看看那看看，那你最好看得仔细一点。

我们还建议你在看书的过程中也抽空看看本书末尾的参考资料。（这可不算随处翻看。）尤其是任何以打字机字体（黑体）隔离的文字可能都会在第二十九章，函数，里面找到。同时，尽管我们力图做到与操作系统无关，但如果你对一些 Unix 术语不熟悉并且碰到一些看起来没有意义的词语，你就应该检查一下这些词语是否出现在术语表里。如果术语表里没有，那可能在索引里。

2.1 原子

尽管在我们现在解说的事物背后还有许多看不见的事物在发生作用，通常你在 Perl 里面用到的最小的东西是字符。这里我们确实说的是字符，在历史上，Perl 很自由地混淆字节和字符这两个概念，不过在如今的全球网络化的时代，我们必须仔细地区分这两个概念。

Perl 当然可以完全用 7 位（bit）的 ASCII 字符集组成。Perl 同样还允许你用任何 8 位或 16 位字符集书写程序，不管这些字符集是国家字符集还是其他什么传统的字符集。不过，如果你准备用这些古老的非 ASCII 字符集书写程序，可能在你的字符串的文字里只能

使用非 ASCII 字符集。你必须负责保证你的程序的语意和你选择的国家字符集是一致的。比如说，如果你正在将 16 位的编码用于亚洲国家字符集，那么你要记住 Perl 会认为你的每个字符是两个字节，而不是一个字符。

象我们在第十五章，Unicode，里面描述的那样，我们最近为 Perl 增加了 Unicode 的支持（注：尽管我们对能支持 Unicode 感到非常激动，我们的大部分例子仍然是 ASCII 编码的，因为不是每个人都有一个好的 Unicode 编辑器。）。这个支持是遍及这门语言全身的：你可以用 Unicode 字符做标识符（比如说变量名），就象在文本串里使用那样。当你使用 Unicode 的时候，用不着担心一个字符是由几个位或者几个字节组成的。Perl 只是假装所有 Unicode 字符都是相同大小（也就是说，尺寸为 1），甚至任意字符在内部都是由多个字节表示的。Perl 通常在内部把 Unicode 表示为 UTF-8 —— 一种变长编码方式。（比如，一个 Unicode 的笑脸字符，U-263A，在内部会表现为一个三字符序列。）

如果让我们把与物理元素的类比进行得更深入一些，字符是基本粒子，就象不同元素里面独立的原子一样。的确，字符是由位和字节这些更小的粒子组成的，但是如果你把一个字符分割开（当然是在一个字符加速器里），这些独立的位和字节就会完全失去那些赖以区分字符的化学属性。就象中子是铀-238 原子的实现细节一样，字节也是 U-236A 字符的实现细节。

所以当我们提及字符时，我们会小心地用"字符"这个字眼，而提及字节时，我们会用"字节"。不过我们不是想吓唬你——你仍然可以很容易的做那些老风格的字节处理。你要干的事只是告诉 Perl 你依然想把字节当作字符处理。你可以用 use bytes 用法来实现这个目的（参阅第三十一章，实用模块）。不过即使你不这样做，Perl 还是能很不错地在你需要的时候把小字符保存在 8 个位里面。

所以不要担心这些小的方面。让我们向更大和更好的东西前进。

2.2 分子

Perl 是自由格式语言，不过也不意味着 Perl 就完全是自由格式。象计算机工作者常说的那样，自由格式的语言就是说你可以在你喜欢的任何地方放空白，制表符和新行等字符（除了不能放的地方以外）。

有一个显然不能放空白的地方就是在记号里。一个记号就是有独立含义的一个字符序列，非常象自然语言中的单字。不过和典型的字不一样的地方是记号可能含有除了字符以外的其他字符——只要它们连在一起形成独立的含义。（从这个意义来看，记号更象分子，因为分子不一定要由一种特定的原子组成。）例如，数字和数学操作符都是记号。一个标识符是由字母或下划线开头的，只包括字母，数字，和下划线。记号里面不能有空白，因为那样会把它分成两个记号，就象在英文里面空白把单词分成两个单词一样。（注：聪明的读者可能会说

在文本串里可以包含空白字符。但是字符串只有在两边都用双引号括起来才能保证空白不会漏出去。）

尽管允许空白出现在任何两个记号中间，只有在两个记号放在一起会被误认为是一个记号的时候才要求一定要在中间放空白。用于这个目的的时候所有空白都是一样的。新行只有在引号括起的字串，格式（串）和一些面向行的格式的引用中才和空白或（水平）制表符（tab）不一样。具体来说，换行符不象某些语言一样（比如 FORTRAN 或 Python）是语句的结束符。Perl 里的语句是用分号结束的，就象在 C 里面和 C 的许多其他变种一样。

Unicode 的空白允许出现在 Unicode 的 Perl 程序里面，不过你要小心处理。如果你应用了特殊的 Unicode 段落和分行符，请注意 Perl 可能会和你的文本编辑器计算出不同的行数来，因此错误信息可能会变得更难处理。最好是依然使用老风格的新行符。

记号的识别是贪多为胜的；如果在某一点让 Perl 的分析器在一长一短两个记号之间做选择，她会选择长的那个。如果你的意思是两个记号，那就在它们中间插入一些空白。（为了增加可读性我们总是在大多数操作符周围放上额外的空白。）

注释用 # 字符标记，从这个字符开始到行尾。一个注释会被当作分隔记号的空白。Perl 语言对你放到注释里面的东西不附加任何特殊含义。（注：实际上，这里撒了个小谎，不过无伤大雅。Perl 的分析器的确在 #! 开头的行里查找命令行开关（参阅第十九章，命令行接口）。它还可以分析各种预处理器产生的各种行数标识（参阅第二十四章，普通实践，的"在其他语言里生成 Perl"节）。

另外一个例外是，语句里任何地方如果存在以 = 开头的行都是合法的，Perl 将忽略从这一行开始直到下一个由 =cut 开头的行。被忽略的文本将被认为是 pod，或"plain old documentation（简单的旧文档）"。Perl 的发布版本里面有一个程序可以从 Perl 模块里抽取 pod 注释输出到一个平面文本文件，手册页，LATEX，HTML，或者（将来某一天）XML 文档里。Perl 分析器会从 Perl 模块里面抽取 Perl 代码并且忽略 pod。因此你可以把这个方法当作一种可用的多行注释方法。你也完全可以认为它是一种让人头疼的东西，不过这样做可以使 Perl 模块不会丢失它的文档。参阅第二十六章，简单旧文档，那里有关于 pod 的细节，包括关于如何有效的在 Perl 里面使用多行注释的描述。

不过可不要小看普通的注释字符。用一列整齐的 # 注释的多行文本可以有舒服的视觉效果。它马上就告诉你的眼睛："这些不是代码。"你可能注意到即使象 C 这样有多行注释机制的语言里，人们都总是在他们注释的左边放上一行 * 字符。通常外观要比仅仅出现更重要。

在 Perl 里，就象在化学和语言里一样，你可以从小的东西开始建造越来越大的结构。我们已经提到过语句了；它只是组成一条命令，就是说，一个祈使句。你可以用花括弧把一系列语句组成块。小块可以组装成大块。若干块组成子函数，然后子函数可以组成模块，模块可

以合并到程序里面去。不过我们这里已经走得太远了 ——那些是未来若干章的内容。先让我们从字符里面组建更多的记号开始吧。

2.3 内置的数据类型

在我们开始讲述各种各样的用字符组建的记号之前，我们先要做一些抽象。具体说来，就是我们需要三种数据类型。

计算机语言因它们支持的数据类型的多寡和类别而不同。一些常用的语言为类似的数值提供了许多让人易混的数据类型， Perl 不一样，它只提供了少数几种内建的数据类型。让我们看看 C，在 C 里你可能会碰到 char，short，int，long， long long，bool，wchar_t，size_t，off_t，regex_t，uid_t，u_longlong_t，pthread_key_t，fp_exception_field_type 等等类型。这些都是某种类型的整型！然后还有浮点数，指针和字符串等。

所有的这些复杂的类型都只对应 Perl 里面的一种类型：标量（你通常需要的只是 Perl 的简单数据类型，如果不是的话，你可以利用 Perl 的面向对象的特性自由地定义动态类型——参阅第十二章，对象。） Perl 的三种基本数据类型是：标量，标量数组和标量散列（hash）（也被称做联合数组）。有些人喜欢把这些称做数据结构，而不是类型。那也行。

标量是建造更复杂类型的基本类型。一个标量存储单一的简单值——通常是一个字串或者一个数字。这种简单类型的元素可以组成两种聚集类型的任何一种。一个数组是是一个标量的有序排列，你可以通过一个整型脚标（或者索引）访问。Perl 里的所有索引都从 0 开始。不过，和许多编程语言不一样的是，Perl 认为负数脚标也是合法的：负数脚标是从后向前记数你的数组。（这一点对许多子字串和子数组操作符以及正则表达式都适用。）另一方面，一个散列（hash）数组是一个无序的键字/数值对，你可以用字串（就是键字）当作脚标来访问对应一个键字的标量（就是数值）。变量总是这三种类型之一。（除变量外，还有一些其他的 Perl 抽象你也可以认为是数据类型，比如文件句柄，目录句柄，格式串，子过程（子函数），符号表和符号表入口等。）

抽象是好东西，我们一边学习一边会收集到更多的抽象，不过从某个角度来看，抽象也是没什么用的东西。你直接用抽象不能做任何事情。因此计算机语言就要有语法。我们要告诉你各种各样的语法术语，这样你就可以把抽象数据组成表达式。在我们谈到这些语法单元的时候，我们喜欢使用技术术语"项"这个词。（哦，这里的措辞可能有点模糊。不过只要记住数学老师在讲数学等式里用到的"项"这个词，你就不会犯大错。）

就象在数学等式里的项一样，Perl 里的大多数项的目的也是为加号或乘号等操作符生成数值。不过，和数学等式不一样的是，Perl 对它计算的数值要做些处理，而不仅仅是拿着一支笔在手里思考等式两边是否相等。对一个数值最常做的事情就是把它存放在某个地方：

```
$x = $y;
```

上面是赋值操作符（不是数字相等操作符，这个操作符在 Perl 里叫 ==）的例子。这条语句把 $y 的值赋予 $x。请注意我们不是用项 $x 做为其数值，而是作为其位置（$x 的原先的值被赋值语句删除。）我们把 $x 称为一个 lvalue（左值），意思是我们可以用在赋值语句左边的存储位置。我们把 $y 称为一个 rvalue（右值），因为它是用在右边的。

还有第三种数值，叫临时值，如果你想知道 Perl 是如何处理你的左值和右值的，你就得理解这个临时值。如果我们做一些实际的数学运算并说：

```
$x = $y + 1;
```

Perl 拿出右值 $y 并且给它加上右值 1，生成一个临时变量，最后临时变量赋予左值 $x。如果我们告诉你 Perl 把这些临时变量存储在一个叫堆栈的内部结构里面，（注：堆栈就象餐馆小卖部里用发条上紧的自动售货机，你可以在栈顶压入盘子，或者你也可以把他们弹出来。）你可能就能想象内部发生什么事。一个表达式的项（我们在这一章里要谈的内容）会向堆栈里压入数据，当表达式里的操作符（我们在下一章讨论）试图把它们从堆栈里面弹出时，可能会在堆栈里面保留另外一个临时结果给下一个操作符处理。当表达式处理完成时，压栈和弹出相互平衡，堆栈完全清空（或者和开始的时候一样）。后面还有更多关于临时变量的介绍。

有些项只能做右值，比如上面的 1，而其他的既可以做左值也可以做右值。尤其是象上面的赋值语句演示的那样，一个变量就可以同时做左值和右值。那是我们下一章的内容。

2.4 变量

不用说，有三种变量类型和我们前面提到的三种抽象数据类型对应. 每种类型都由我们称之为趣味字符（ funny character）（注：这是计算机科学的另一个技术术语。（如果以前它不是，那现在它就是了）做前缀. 标量变量的开头总是 $，甚至连引用一个数组或者散列中的一个标量也如此. 它有点象英文里的单词 " the " . 所以，我们有下表：

构造	含义
$days	简单标量值 $days
$days[28]	数组 @days 的第二十九个元素
$days{'Feb'}	散列 %days 的“Feb”值

请注意我们可以对 $days，@days，和 %days 使用相同的名字而不用担心 Perl 会混淆它们。

还有其他一些爱好者使用的标量术语，在一些我们一时半会还接触不到的地方非常有用。他们看起来象：

构造	含义
${days}	和 $days 一样，不过在字母数字前面不易混淆
$Dog::days	在 Dog 包里面的不同的 $days 变量
$#days	数组 @days 的最后一个索引
$days->[28]	$days 一个引用指向的数组的第二十九个元素
$days[0][2]	多维数组
$days{200}{'Feb'}	多维散列
$days{2000,'Feb'}	多维散列枚举

整个数组（或者数组和散列的片段）带趣味字符 @ 命名，很象单词"这些"或"那些"的作用：

构造	含义
@days	包含（$days[0],$days[1],...$days[n]）的数组
@days[3,4,5]	包含（$days[3],$days[4],$days[5]）数组片段的数组
@days[3..5]	包含（$days[3],$days[4],$days[5]）数组片段的数组
@days{'Jan','Feb'}	包含（$days{'Jan'},$days{'Feb'}）片段的散列

每个散列都用 % 命名：

构造	含义
%days	（Jan=>31，Feb=>$leap?29:28,...）

任何这些构造都可以作为左值使用，声明一个你可以赋值的位置。对于数组，散列或者数组和散列的片段，这个左值提供了可以赋值的多个位置，因此你可以一次给所有这些位置赋值：

```
@days = 1..7;
```

2.5 名字

我们已经谈过在变量里面保存数值了，但是变量本身（它们的名字和它们相关的定义）也需要存储在某个位置。在抽象（的范畴里），这些地方被称为名字空间。Perl 提供两种类型的名字空间，通常被称为符号表和词法范围（注：当我们谈论 Perl 详细实现时，我们还把它们称做包（packages）和垫（pads），不过那些更长的词是纯正工业术语，所以我们只能用它们，抱歉）。你可以拥有任意数量的符号表和词法范围，但是你定义的任何一个名字都将存储在其中的某一个里面。我们将随着介绍的深入探讨这两种名字空间。目前我们只能说符号表是全局的散列，用于存储存放全局变量的符号表的入口（包括用于其他符号表的散

列）。相比之下，词法范围只是未命名的中间结果暂存器，不会存在于任何符号表，只是附着在你的程序的一块代码后面。它们（词法范围）包含只能被该块所见的变量。（那也是我们说"范围"的含义）。"词法"两个字只是说："它们必须以文本方式处理"，可不是通常字典赋予它们的含义。可别批评我们。

在任何名字空间里（不管是全局还是局部），每个变量类型都有自己的由趣味字符确定的子名字空间。你可以把同一个名字用于一个标量变量，一个数组或者一个散列（或者，说到这份上，可以是一个文件句柄，一个子过程名，一个标签，甚至你的宠物骆驼）也不用担心会混淆。这就意味着 $foo 和 @foo 是两个不同的变量。加上以前的规则，这还意味着 $foo 是 @foo 的一个元素，它和标量变量 $foot 完全没有关系。这些看起来有点怪异，不过也没啥，因为它就是怪异（译注：我是流氓我怕谁？）。

子过程可以用一个 & 开头命名，不过调用子过程的时候这个趣味字符是可选的。子过程通常不认为是左值，不过最近版本的 Perl 允许你从一个子过程返回一个左值并且赋予该子过程，这样看起来可能就象你在给那个子过程赋值。

有时候你想命名一个"所有叫 foo 的东西"，而不管它的趣味字符是什么。因此符号表入口可以用一个前缀的 * 命名，这里的星号（*）代表所有其他趣味字符。我们把这些东西称为类型团（typeglobs），而且它们有好几种用途。它们也可以用做左值。给一个类型团（typeglobs）赋值就是 Perl 从一个符号表向另外一个输入符号的实现。我们后面还会有更多内容讲这些。

和大多数计算机语言类似，Perl 有一个保留字列表，它把这个表里的字看作特殊关键字。不过，由于变量名总是以趣味字符开头，实际上保留字并不和变量名冲突。不过，有些其他类型的名字不带趣味字符，比如标签和文件句柄。即使这样，你也用不着担心与保留字冲突。因为绝大多数保留字都是完全小写，我们推荐你使用带大写字符的名字做标签和文件句柄。例如，如果你说 open(LOG,logfile)，而不是 open(log,"logfile")，你就不会让 Perl 误以为你正在与内建的 log 操作符（它处理对数运算，不是树干（译注：英文 "log" 有树干的含义。））交谈。使用大写的文件句柄也改善了可读性（注：Perl 的一个设计原则是：不同的东西看起来应该不同。和那些试图强制把不同的东西变得看起来一样的语言比较一下，看看可读性的好坏。）并且防止你和我们今后可能会增加的保留字的冲突。处于同样的考虑，用户定义的模块通常都是用首字母大写的名字命名的，这样就不会和内建的模块（叫用法（pragmas））冲突，因为内建模块都是以小写字母命名的。到了面向对象命名的时候，你就会发现类的名称同样都是首字母大写的。

你也许能从前面的段落中推导出这样的结论了，就是标识符是大小写敏感的——FOO，Foo，和 foo 在 Perl 里面都是不同的名字。标识符以字母或下划线开头，可以包含任意长度（这个"任意"值的范围是 1 到 251 之间）个字母，数字和下划线。这里包括 Unicode 字母

和数字。Unicode 象形文字也包括在内，不过我们可不建议你使用它们，除非你能够阅读它们。参阅第十五章。

严格说来，跟在趣味字符后面的名字一定是标识符。他们可以以数字开头，这时候后面只能跟着更多数字，比如 $123。如果一个名字开头不是字母，数字或下划线，这样的名字（通常）限于一个字符（比如 $? 或 $$），而且通常对 Perl 有预定的意义，比如，就象在 Bourne shell 里一样，$$ 是当前进程 ID 而 $? 是你的上一个子进程的退出状态。

到了版本 5.6，Perl 还有一套用于内部变量的可扩展语法。任何形如 ${^NAME} 这样的变量都是保留为 Perl 使用的。所有这些非标识符名字都被强制存放于主符号表。参阅第二十八章，特殊名字，那里有一些例子。

我们容易认为名字和标识符是一样的东西，不过，当我们说名字的时候，通常是指其全称，也就是说，表明自己位于哪个符号表的名字。这样的名字可能是一个由标记 :: 分隔的标识符的序列：

```
$Santa::Helper::Reindeer::Rudolph::nose
```

就象一个路径名里面的目录和文件一样：

```
/Santa/Helper/Reindeer/Rudolph/nose
```

在 Perl 的那个表示法里面，所有前导的标识符都是嵌套的符号表名字，并且最后一个标识符就是变量所在的最里层的符号表的名字。比如，上面的变量里，符号表的名字是 Santa::Helper::Reindeer::Rudolph::，而位于此符号表的实际变量是 $nose。（当然，该变量的值是"red"。）

Perl 里的符号表也被称为包（package），因此这些变量常被称为包变量。包变量名义上是其所属包的私有成员，但实际上是全局的，因为包本身就是全局的。也就是说，任何人都可以命名包以获取该变量；但就是不容易碰巧做到这些。比如，任何提到 $Dog::bert 的程序都是获取位于 Dog:: 包的变量 $bert。但它是与 $Cat::bert 完全不同的变量。参阅第十章，包。

附着在词法范围上的变量不属于任何包，因此词法范围变量名字可能不包含 :: 序列。（词法范围变量都是用 my 定义式定义的。）

2.5.1 名字查找

那问题就来了，名字里有什么？如果你光是说 $bert，Perl 是怎样了解你的意思的？问得好。下面是在一定环境里 Perl 分析器为了理解一个非全称的名字时用到的规则：

1。首先，Perl 预先在最先结束的块里面查找，看看该变量是否有用 my（或则 our）定义在该代码块里（参考那些第二十九章的内容和第四章，语句和声明和，里面的“范围声明”节）。如果存在 my 定义，那么该变量是词法范围内的而不存在于任何包里——它只存在于那个词法范围（也就是在该代码块的暂时缓存器里）。因为词法范围是非命名的，在那块程序之外的任何人甚至都看不到你的变量。（注：如果你用的是 our 定义而非 my，这样只是给一个包变量定义了一个词法范围的别名（外号），而不象 my 定义里面真正地定义了一个词法范围变量。代码外部仍然可以通过变量的包获取其值，但除此以外，our 定义和 my 定义的特点是一样的。如果你想在和 use strict 一起使用（参阅第三十一章里的 strict progma ），限制自己的全局变量的使用时很有用。不过如果你不需要全局变量时你应该优先使用 my。）

2。如果上面过程失败，Perl 尝试在包围该代码段的块里查找，仍然是在这个更大的代码块里查找词法范围的变量。同样，如果 Perl 找到一个，那么就象我们刚刚在第一步里说到的变量一样，该变量只属于从其定义开始到其定义块结束为止的词法范围——包括任何嵌套的代码块。如果 Perl 没有发现定义，那么它将重复第二步直到用光所有上层闭合块。

3。当 Perl 用光上层闭合块后，它检查整个编辑单元，把它当作代码块寻找声明（一个编辑单元就是整个当前文件，或者一个被 eval STRING 操作符编译过的当前字串。）如果编辑单元是一个文件，那就是最大的词法范围，这样 Perl 将不再查找词法范围变量，于是进入第四步。不过，如果编辑单元是一个字串，那事情就有趣了。一个当作运行时的 Perl 代码编译的字串会假装它是在一个 eval STRING 运行着的词法范围里面的一个块，即使其实际词法范围只是包含代码的字串而并非任何真实的花括弧也如此。所以如果 Perl 没有在字串的词法范围找到变量，那么我们假装 eval STRING 是一个块并且回到第 2 步，这回我们才检查 eval STRING 的词法范围而不是其内部字串的词法范围。

4。如果我们走到这一步，说明 Perl 没有找到你的变量的任何声明（my 或 our）。Perl 现在放弃词法范围并假设你的变量是一个包变量。如果 strict pragma 用法有效，你现在会收到一个错误，除非该变量是一个 Perl 预定义的变量或者已经输入到当前包里面。这是因为 strict 用法不允许使用非全称的全局名字。不过，我们还是没有完成词法范围的处理。Perl 再次搜索词法范围，就象在第 1 步到第 3 步里一样，不过这次它找的是 package（包）声明而不是变量声明。如果它找到这样的包声明，那它就知道找到的这些代码是为有问题的包编译的于是就在变量前面追加这个声明包的名字。

5。如果在任何词法范围内都没有包声明，Perl 就在未命名的顶层包里面查找，这个包正好就是 main——只要它没有不带名字到处乱跑。因此相对于任何缺失的声明，$bert 和 $::bert 的含义相同，也和 $main:bert 相同。（不过，因为 main 只是在顶层未命名包中的另一个包，该变量也是$::main::bert，和$main::main::bert，$::main::main::bert 等等。这些可以看作没用的特性。参考第 10 章里的“符号表”。）

这些搜索规则里面还有几个不太明显的暗示，我们这里明确一下。

1。因为文件是可能的最大词法范围，所以一个词法范围变量不可能在定义其的文件之外可见。文件范围不嵌套。

2。不管多大的 Perl 都编译成至少一个词法范围和一个包范围。这个必须的词法范围当然就是文件本身。附加的词法范围由每个封闭的块提供。所有 Perl 代码同样编译进一个包里面，并且尽管声明在哪个包里属于词法范围，包本身并不受词法范围约束。也就是说，它们是全局的。

3。因此可能而在许多词法范围内查找一个非全称变量，但只是在一个包范围内，不管是哪个当前有效的包（这是词汇定义的）。

4。一个变量值只能附着在一个范围上。尽管在你的程序的任何地方都至少有两个不同的范围（词法和包），一个变量仍然只能存在于这些范围之一。

5。因此一个非全长变量名可以被分析为唯一的一个存储位置，要么在定义它的第一个封闭的词法范围里，要么在当前包里——但不是同时在两个范围里。只要解析了存储位置，那么搜索就马上停止，并且如果搜索继续进行，它找到的任何存储位置都被有效地隐藏起来。

6。典型变量名的位置在编译时完全可以决定。

尽管你已经知道关于 Perl 编译器如何处理名字的所有内容，有时候你还是有这样的问题：在编译时你并不知道你想要的名字是什么。有时候你希望间接地命名一些东西；我们把这个问题叫间接（indirection）。因此 Perl 提供了一个机制：你总是可以用一个表达式块代替字母数字的变量名，这个表达式返回一个真正数据的引用。比如，你不说：

```
$bert
```

而可能说：

```
${some_expression()}
```

如果 some_expression() 函数返回一个变量 $bert 的引用（甚至是字串，“bert”），它都会象第一行里的 $bert 一样。另一方面，如果这个函数返回 $ernie 的引用，那你就会得到这个变量。这里显示的是间接的最常用（至少是最清晰）的形式，不过我们会在第 8 章，引用，里介绍几个变体。

2.6 标量值

不管是直接命名还是间接命名，不管它是在变量里还是一个数组元素里或者只是一个临时值，一个变量总是包含单一值。这个值可以是一个数字，一个字串或者是另一片数据的引用。或着它里面可以完全没有值，这时我们称其为未定义（undefined）。尽管我们会说标量"包含"着一个数字或者字串，标量本身是无类型的：你用不着把标量定义为整型或浮点型或字符串或者其他的什么东西。（注：将来的 Perl 版本将允许你插入 int，num，和 str 类型声明，这样做不是为了加强类型，而是给优化器一些它自己发现不了的暗示。通常，这个特性用于那些必须运行得非常快的代码，所以我们不准备告诉你如何使用。可选的类型同样还可以用于伪散列的机制，在这种情况下，它们可以表现得象大多数强类型语言里的类型一样。参阅第八章获取更多信息。）

Perl 把字串当作一个字符序列保存，对长度和内容没有任何限制。用我们人类的话来说，你用不着事先判断你的字串会有多长，而且字串里可以有任何字符，包括空（null）字节。Perl 在可能地情况下把数字保存为符号整数，或者是本机格式的双精度浮点数。浮点数值是有限精度的。你一定要记住这条，因为象 (10/3==1/3*10) 这样的比较会莫名其妙的失败。

Perl 根据需要在各种子类型之间做转换，所以你可以把一个数字当作字串或者反过来，Perl 会正确处理这些的。为了把字串转换成数字，Perl 内部使用类似 C 函数库的 atof(3) 函数。在把数字转换成字串的时候，它在大多数机器上做相当于带有格式"%.14g"的 sprintf(3) 处理。象把 foo 转换成数字这样的不当转换会把数字当成 0；如果你打开警告，上面这样做会触发警告，否则没有任何信息。参阅第五章，模式匹配，看看如何判断一个字串里面有什么东西的例子。

尽管字串和数字在几乎所有场合都可以互换，引用却有些不同。引用是强类型的，不可转换的指针；内建了引用计数和析构调用。也就是说，你可以用它们创建复杂的数据类型，包括用户定义对象。但尽管如此，它们仍然是标量，因为不管一个数据结构有多复杂，你通常还是希望把它当作一个值看待。

我们这里说的不可转换，意思是说你不能把一个引用转换成一个数组或者散列。引用不能转换成其他指针类型。不过，如果你拿一个引用当作一个数字或者字串来用，你会收到一个数字或者字串值，我们保证这个值保持引用的唯一性，即使你从真正的引用中拷贝过来而丢失了该"引用"值也如此（唯一）。你可以比较这样的数值或者抽取它们的类型。不过除此之外你对这种类型干不了什么，因为你没法把数字或字串转换回引用。通常，着不是个问题，因为 Perl 不强迫你使用指针运算——甚至都不允许。参阅第八章读取更多引用的信息。

2.6.1 数字文本

数字文本是用任意常用浮点或整数格式声明的：（注：在 Unix 文化中的习惯格式。如果你来自不同文化，欢迎来到我们中间！）

```
$x = 12345;        # 整数

$x = 12345.67;       # 浮点

$x = 6.02e23;       # 科学记数

$x = 4_294_967_296;    # 提高可读性的下划线

$x = 03777;        # 八进制

$x = 0xffff;       # 十六进制

$x = 0b1100_0000;    # 二进制
```

因为 Perl 使用逗号作为数组分隔符，所以你不能用它分隔千位或者更大位。不过 Perl 允许你用下划线代替。这样的下划线只能用于你程序里面声明的数字文本，而不能用于从其他地方读取的用做数字的字串。类似地，十六进制前面的 0x，二进制前面的 0b，八进制的 0 也都只适用于数字文本。从字串到数字的自动转换并不识别这些前缀，你必须用 oct 函数做显式转换（注：有时候人们认为 Perl 应该对所有输入数据做这个转换。不过我们这个世界里面有太多带有前导零的十进制数会让 Perl 做这种自动转换。比如，O'Reilly & Associates 在麻省剑桥的办公室的邮政编码是 02140。如果你的邮件标签程序把 02140 变成十进制 1120 会让邮递员不知所措的。） oct 也可以转换十六进制或二进制数据，前提是你要在字串前面加 0x 或 0b。

2.6.2 字串文本

字串文本通常被单引号或者双引号包围。这些引号的作用很象 Unix shell 里的引号：双引号包围的字串文本会做反斜杠和变量替换，而单引号包围的字串文本不会（除了\'和\\以外，因此你可以在单引号包围的字串里使用单引号和反斜杠）。如果你想嵌入其他反斜杠序列，比如 \n（换行符），你就必须用双引号的形式。（反斜杠序列也被称为逃逸序列，因为你暂时"逃离"了通常的字符替换。）

一个单引号包围的字串必须和前面的单词之间有一个空白分隔，因为单引号在标识符里是个有效的字符（尽管有些老旧）。它的用法已经被更清晰的 :: 序列取代了。这就意味着 $main'val 和 $main::var 是一样的，只是我们认为后者更为易读。

双引号字串要遭受各种字符替换，其他语言的程序员对很多这样的替换非常熟悉。我们把它们列出在表 2-1

表 2-1 反斜杠的字符逃逸

代码	含义
\n	换行符（常作 LF）
\r	回车（常作 CR）
\t	水平制表符
\f	进纸
\b	退格
\a	警报（响铃）
\e	ESC 字符
\033	八进制的 ESC
\x7f	十六进制 DEL
\cC	Control-C
\x{263a}	Unicode（笑脸）
\N{NAME}	命名字符

上面的 \N{NAME} 符号只有在与第三十一章描述的 user charnames 用法一起使用时才有效。这样允许你象征性地声明字符，象在 \N{GREEK SMALL LETTER SIGMA}，\n{greek:Sigma}，或 \N{sigma} 里的一样——取决于你如何调用这个用法。参阅第十五章。

还有用来改变大小写或者对随后的字符"以下皆同"的操作的逃逸序列。见表 2-2

表 2-2。引起逃逸

代码	含义
\u	强迫下一个字符为大写（Unicode 里的"标题"）
\l	强制下一个字符小写
\U	强制后面所有字符大写
\L	强制后面所有字符小写
\Q	所有后面的非字母数字字符加反斜杠
\E	结束\U，\L，或\Q。

你也可以直接在你的字串里面嵌入换行符;也就是说字串可以在另一行里。这样通常很有用，不过也意味着如果你忘了最后的引号字符，Perl 会直到找到另外一个包含引号字符的行时才报错，这是可能已经远在脚本的其他位置了。好在这样使用通常会在同一行立即产生一个语法错，而且如果 Perl 认为有一个字串开了头，它就会很聪明地警告你你可能有字串没有封闭。

除了上面列出的反斜杠逃逸，双引号字串还要经受标量或数组值的变量代换。这就意味着你可以把某个变量的值直接插入字串文本里。这也是一个字串连接的好办法。（注：如果打开了警告，在使用连接或联合操作时，Perl 可能会报告说有未定义的数值插入到字串中，即使你实际上没有在那里使用那些操作也如此。那是编译器给你创建的。）可以做的变量代换的有标量变量，整个数组（不过散列不行），数组或散列的单个元素，或者片段。其它的东西都不转换。换而言之，你只能代换以 $ 或 @ 开头的表达式，因为它们是字串分析器要找的两个字符（还有反斜杠）。在字串里，如果一个 @ 后面跟着一个字母数字字符，而它又不是数组或片段的标识符，那就必须用反斜杠逃逸（\@），否则会导致一个编译错误。尽管带 % 的完整的散列可能不会代换进入字串，单个散列值或散列片段却是会的，因为它们分别以 $ 和 @ 开头。

下面的代码段打印 "The Price is $100.":

```
$Price = '$100';          # 不替换

print "The price is $Price.\n";      # 替换
```

和一些 shell 相似，你可以在标识符周围放花括弧，使之与后面的字母数字区分开来："How ${verb}able!"。一个位于这样的花括弧里面的标识符强制为字串，就象任何散列脚标里面的单引号标识符一样。比如：

```
$days{'Feb'}
```

可以写做：

```
$days{Feb}
```

并且假设有引号。脚标里任何更复杂的东西都被认为是一个表达式，因此你用不着放在引号里：

```
$days{'February 29th'}    # 正确

$days{"February 29th"}    # 也正确""不必代换

$days{February 29th}    # 错，产生一个分析错误
```

尤其是你应该总是在类似下面的片段里面使用引号：

```
@days{'Jan','Feb'}        # Ok.

@days{"Jan","Feb"}        # Also ok.

@days{ Jan,  Feb }        # Kinda wrong (breaks under use strict)
```

除了被代换的数组和散列变量的脚标以外，没有其它的多层代换。与 shell 程序员预期地相反，在双引号里面的反勾号不做代换，在双引号里面的单引号也不会阻止变量计算。Perl 里面的代换非常强大，同时也得到严格的控制。它只发生在双引号里，以及我们下一章要描述的一些"类双引号"的操作里：

```
print "\n";       # 正确，打印一个新行

print \n;       # 错了，不是可替换的环境。
```

2.6.3 选择自己的引号

尽管我们认为引起是文本值，但在 Perl 里他们的作用更象操作符，提供了多种多样的代换和模式匹配功能。Perl 为这些操作提供了常用的引起字符，还提供了更通用的客户化方法，让你可以为上面任意操作选择你自己的引起字符。在表 2-3 里，任意非字母数字，非空白分隔符都可以放在 / 的位置。（新行和空格字符不再允许做分隔符了，尽管老版本的 Perl 曾经一度允许这么做。）

表 2-3。引起构造

常用	通用	含义	替换
' '	q//	文本字串	否
" "	qq//	文本字串	是
` `	qx//	执行命令	是
()	qw//	单词数组	否
//	m//	模式匹配	是
s///	s///	模式替换	是
y///	tr///	字符转换	否
" "	qr//	正则表达式	是

这里的有些东西只是构成"语法调味剂"，以避免你在引起字串里输入太多的反斜杠，尤其是在模式匹配里，在那里，普通斜杠和反斜杠很容易混在一起。

如果你选用单引号做分隔符，那么不会出现变量代换，甚至那些正常状态需要代换的构造也不发生代换。如果起始分隔符是一个起始圆括弧，花括弧，方括弧，那么终止分隔符就是对应终止字符。（嵌入的分隔符必须成对出现。）比如：

```
$single = q!I said, "You said, 'she sad it.'"!;
```

```
$double =qq(can't we get some "good" $variable?);


$chunk_of_code = q {
    if ($condition) {
        print "Gotcha!";
    }
};
```

最后一个例子表明，你可以在引起声明字符和其起始包围字符之间使用空白。对于象 s/// 和 tr///这样的两元素构造而言，如果第一对引起是括弧对，那第二部分获取自己的引起字符。实际上，第二部分不必与第一对一样。所以你可以用象 s(bar)或者 tr(a-f)[A-f]这样的东西。因为在两个内部的引起字符之间允许使用空白，所以你可以把上面最后一个例子写做：

```
tr (a-f)
[A-F];
```

不过，如果用 # 做为引起字符，就不允许出现空白。q#foo# 被分析为字串 'foo'，而 q #foo# 引起操作符 q 后面跟着一个注释。其分隔符将从下一行获取。在两个元素的构造中间也可以出现注释，允许你这样写：

```
s{foo}  # 把 foo
 {bar}  # 换为 bar。


tr [a-f]  # 把小写十六进制
   [A-F];  # 换为大写
```

2.6.4 要么就完全不管引起

一个语法里没有其他解释的名字会被当作一个引起字串看待。我们叫它们光字。（注：我们认为变量名，文件句柄，标签等等不是光字，因为它们有被前面的或后面的（或两边的）语句强制的含义。预定义的名字，比如子过程，也不是光字。只有分析器丝毫不知的东西才是

光字。）和文件句柄和标签一样，完全由小写字符组成的光字在将来也可能有和保留字冲突的危险。如果你打开了警告，Perl 会就光字对你发出警告。比如：

```
@days = (Mon,Tue,Wed,Thu,Fri);

print STDOUT hello, ' ', world, "\n";
```

给数组 @days 设置了周日的短缩写以及在 STDOUT 上打印一个 "hello world" 和一个换行。如果你不写文件句柄， Perl 就试图把 hello 解释成一个文件句柄，结果是语法错。因为这样非常容易出错，有些人就可能希望完全避免光字。前面列出的引用操作符提供了许多方便的构形，包括 qw// "单词引用"这样的可以很好地引用一个空白分隔的数组的构造：

```
@days = qw(Mon Tue Wed Thu Fri);

print STDOUT "hello world\n";
```

你可以一直用到完全废止光字。如果你说：

```
use strict 'subs';
```

那么任何光字都会产生一个编译时错误。此约束维持到此闭合范围结束。一个内部范围可以用下面命令反制：

```
no strict 'subs';
```

请注意在类似：

```
"${verb}able"

$days{Feb}
```

这样的构造里面的空标识符不会被认为是光字，因为它们被明确规则批准，而不是说"在语法里没有其他解释"。

一个不带引号的以双冒号结尾的名字，比如 main::或 Dog::，总是被当作包名字看待。Perl 在编译时把可能的光字 Camel:: 转换成字串 "Camel"，这样，这个用法就不会被 use strict 指责。

2.6.5 代换数组数值

数组变量通过使用在 $" 变量（缺省时包含一个空格）（注：如果你使用和 Perl 捆绑的 English 模块，那么就是 $LIST_SEPARATOR）里声明的分隔符将所有数组元素替换为双引号包围的字串。下面的东西是一样的：

```
$temp = join( $", @ARGV ); print $temp;

print "@ARGV";
```

在搜索模式里（也要进行双引号类似的代换）有一个不巧的歧义：/$foo[bar]/ 是被替换为 /${foo}[bar]/（这时候 [bar] 是用于正则表达式的字符表）还是 /${foo[bar]}/（这里 [bar] 是数组 @foo 的脚标）？如果 @foot 不存在，它很显然是个字符表。如果 @foo 存在，Perl 则猜测 [bar] 的用途，并且几乎总是正确的（注：全面描述猜测机制太乏味了，基本上就是对所有看来象字符表（a-z，\w，开头的^)和看来象表达式（变量或者保留字）的东西进行加权平均）。如果它猜错了，或者是你变态，那你可以用上面描述的花括弧强制正确的代换。就算你只是为了谨慎，那不算是个坏主意。

2.6.6“此处”文档

有一种面向行的引起是以 Unix shell 的“此处文档”语法为基础的。说它面向行是因为它的分隔符是行而不是字符。起始分隔符是当前行，结束分隔符是一个包含你声明的字串的行。你所声明的用以结束引起材料的字串跟在一个 << 后面，所有当前行到结束行（不包括）之间的行都是字串的内容。结束字串可以是一个标识符（一个单词）或者某些引起的文本。如果它也被引起，引起的类型决定文本的变换，就象普通的引起一样。没有引起的标识符当作用双引号引起对待。反斜杠转意的标识符当作用单引号引起（为与 shell 语法兼容）。在 << 和未引起的标识符之间不能有空白，不过如果你用一个带引号的字串做标识符，则可以有空白。（如果你插入了空白，它会被当作一个空标识符，这样做是允许的但我们不赞成这么用，它会和第一个空白行匹配——参阅下面第一个 Hurrah! 例子。）结束字串必须在终止行独立出现——不带引号以及两边没有多余的空白。（译注：常见的错误是为了美观在结束字串前面加 \t 之类的空白，结果却导致错误．）

```
print <<EOF;    # 和前面的例子一样

The price is $Price.

EOF


print <<"EOF";    # 和上面一样，显式的引起

The price is $Price.
```

```
EOF

print <<'EOF';   # 单引号引起
（略）
EOF

print << x 10;   # 打印下面行 10 次
The Camels are coming! Hurrah! Hurrah!

print <<"" x 10;   # 实现上面内容的比较好的方法
The Camels are coming! Hurrah! Hurrah!

print <<`EOC`;   # 执行命令
echo hi there
echo lo there
EOC

print <<"dromedary", <<"camelid";   # 你可以堆叠
I said bactrian.
dromedary
She said llama.
camelid

funkshun(<<"THIS",23,<<'THAT');   # 在不在圆括弧里无所谓
```

```
    Here's a line
    ro two.
    THIS
    And here's another.
    THAT
```

不过别忘记在最后放分号以结束语句，因为 Perl 不知道你不是做这样的试验：

```
print <<'odd'
                                    1. odd +10000; #打印 12345
```

如果你的此处文档在你的其他代码里是缩进的，你就得手工从每行删除开头的空白：

```
($quote = <<'QUOTE') =~ s/^\s+//gm; The Road goes ever on and on, down from the door where it began. QUOTE
```

你甚至还可以用类似下面的方法用一个此处文档的行填充一个数组：

```
@sauces = <<End_Lines =~ m/(\S.*\S)/g;
    normal tomato
    spicy tomato
    green chile
    pesto
    white wine
End_Lines
```

2.6.7 V-字串文本

一个以 v 开头，后面跟着一个或多个用句点分隔的整数的文本，会被当作一个字串文本；该字串的字符的自然数对应 v 文本里的数值：

```
$crlf = v13.10; # ASCII 码回车，换行
```

这些就是所谓 v-字串，“向量字串”（vector strings）或“版本字串”（version strings）或者任何你能想象得出来的以“v”开头而且处理整数数组的东西的缩写。当你想为每个字符直接声明其数字值时，v-字串给你一种可用的而且更清晰的构造这类字串的方法。因此，v1.20.300.4000 是比用下面的方法构造同一个字串的更迷人的手段：

```
"\x{1}\x{14}\x{12c}\x{fa0}"

pack("U*", 1, 20, 300, 4000)

chr(1) . chr(20) . chr(300) . chr(4000)
```

如果这样的文本有两个或更多句点（三组或者更多整数），开头的 v 就可以忽略。

```
print v9786;        # 打印 UTF-8 编码的笑脸“\x{263a}”

print v120.111.111;    # 打印"foo"


use 5.6.0;        # 要求特定 Perl 版本（或更新）


$ipaddr = 204.148.40.9;    # oreilly.com 的 IPV4 地址
```

v-字串在表示 IP 地址和版本号的时候很有用。尤其是在字符可以拥有大于 255 的数值现代，v-字串提供了一个可以表示任意大小的版本并且用简单字符串比较可以得到正确结果的方法。

存储在 v-字串里的版本号和 IP 地址是人类不可读的，因为每个字符都是以任意字符保存的。要获取可读的东西，可以在 printf 的掩码里使用 v 标志，比如 "%vd"，这些在第二十九章的 sprintf 部分有描述。有关 Unicode 字串的信息，请参阅第十五章和第三十一章的 use bytes 用法；关于利用字串比较操作符比较版本字串的内容，参阅第二十八章的 $^V；有关 IPV4 地址的表示方面的内容，见第二十九章 gethostbyaddr。

2.6.8 其他文本记号

你应该把任何以双下划线开头和结束的标识符看作由 Perl 保留做特殊语法处理的记号。其中有两个这类特殊文本是 ***LINE*** 和 __FILE__，分别意味着在你的程序某点的当前行号和文件名。它们只能用做独立的记号；它们不能被代换为字串。与之类似，__PACKAGE__ 是当前代码所编译进入的包的名字。如果没有当前包（因为有一个空的 package; 指示），__PACKAGE__ 就是未定义值。记号 ***END*** （或者是一个 Control-D 或 Control-Z 字

符）可以用于在真正的文件结束符之前表示脚本的逻辑结束。任何后面的文本都被忽略，不过可以通过 DATA 文件句柄读取。

DATA 记号的作用类似 ***END*** 记号，不过它是在当前包的名字空间打开 DATA 文件句柄，因此你所 require 的所有文件可以同时打开，每个文件都拥有自己的 DATA 文件句柄。更多信息请看第二十八章里的 DATA。

2.7 环境

到现在为止，我们已经看到了一些会产生标量值的项。在我们进一步讨论项之前，我们要先讨论带环境（context）的术语。

2.7.1 标量和列表环境

你在 Perl 脚本里激活的每个操作（注：这里我们用"操作"统称操作符或项。当你开始讨论那些分析起来类似项而看起来象操作符的函数时，这两个概念间的界限就模糊了。）都是在特定的环境里进行的，并且该操作的运转可能依赖于那个环境的要求。存在两种主要的环境：标量和列表。比如，给一个标量变量，一个数组或散列的标量元素赋值，在右手边就会以标量环境计算：

```
$x          = funkshun();  # scalar context
$x[1]       = funkshun();  # scalar context
$x{"ray"}   = funkshun();  # scalar context
```

但是，如果给一个数组或者散列，或者它们的片段赋值，在右手边就会以列表环境进行计算，即便是该片段只选出了的一个元素：

```
@x          = funkshun();  # list context
@x[1]       = funkshun();  # list context
@x{"ray"}   = funkshun();  # list context
%x          = funkshun();  # list context
```

即使你用 my 或 our 修改项的定义，这些规则也不会改变：

```
my $x       = funkshun();  # scalar context
my @x       = funkshun();  # list context
```

```
my %x       = funkshun();  # list context

my ($x)     = funkshun();  # list context
```

在你正确理解标量和列表环境的区别之前你都会觉得很痛苦，因为有些操作符（比如我们上面虚构的 funkshun()函数）知道它们处于什么环境中，所以就能在列表环境中返回列表，在标量环境中返回标量。（如果这里提到的东西对于某操作成立，那么在那个操作的文档里面应该提到这一点。）用计算机行话来说，这些操作重载了它们的返回类型。不过这是一种非常简单的重载，只是以单数和复数之间的区别为基础，别的就没有了。

如果某些操作符对环境敏感，那么很显然必须有什么东西给它们提供环境。我们已经显示了赋值可以给它的右操作数提供环境，不过这个例子不难理解，因为所有操作符都给它的每个操作数提供环境。你真正感兴趣的应该是一个操作符会给它的操作数提供哪个环境。这样，你可以很容易地找出哪个提供了列表环境，因为在它们的语法描述部分都有 LIST。其他的都提供标量环境。通常，这是很直观的。（注：不过请注意，列表环境可以通过子过程调用传播，因此，观察某个语句会在标量还是列表环境里面计算并不总是很直观。程序可以在子过程里面用 wantarray 函数找出它的环境。）如果必要，你可以用伪函数 scalar 给一个 LIST 中间的参数强制一个标量环境。Perl 没有提供强制列表环境成标量环境的方法，因为在任何一个你需要列表环境的地方，都会已经通过一些控制函数提供了 LIST。

标量环境可以进一步分类成字串环境，数字环境和无所谓环境。和我们刚刚说的标量与列表环境的区别不同，操作从来不关心它们处于那种标量环境。它们只是想返回的标量值，然后让 Perl 在字串环境中把数字转换成字串，以及在数字环境中把字串转换成数字。有些标量环境不关心返回的是字串还是数字还是引用，因此就不会发生转换。这个现象会发生在你给另外一个变量赋值的时候。新的变量只能接受和旧值一样的子类型。

2.7.2 布尔环境

另外一个特殊的无所谓标量环境是布尔环境。布尔环境就是那些要对一个表达式进行计算，看看它是真还是假的地方。当我们在本书中说到"真"或"假"的时候，我们指的是 Perl 用的技术定义：如果一个标量不是空字串 "" 或者数字 0 （或者它的等效字串，"0" ）那么就是真。一个引用总是真，因为它代表一个地址，而地址从不可能是 0。一个未定义值（常称做 undef）总是假，因为它看起来象 "" 或者 0——取决于你把它当作字串还是数字。（列表值没有布尔值，因为列表值从来不会产生标量环境！）

因为布尔环境是一个无所谓环境，它从不会导致任何标量转换的发生，当然，标量环境本身施加在任何参与的操作数上。并且对于许多相关的操作数，它们在标量环境里产生的标量代表一个合理的布尔值。也就是说，许多在列表环境里会产生一个列表的操作符可以在布尔环

境里用于真/假测试。比如，在一个由 unlink 操作符提供的列表环境里，一个数组名产生一列值：

unlink @files; # 删除所有文件，忽略错误。

但是，如果你在一个条件里（也就是说，在布尔环境里）使用数组，数组就会知道它正处于一个标量环境并且返回数组里的元素个数，只要数组里面还有元素，通常就是真。因此，如果你想获取每个没有正确删除的文件的警告，你可能就会这样写一个循环：

```
while (@files) {
    my $file = shift @files;
    unlink $file or warn "Can't delete $file: $!\n";
}
```

这里的 @files 是在由 while 语句提供的布尔环境里计算的，因此 Perl 就计算数组本身，看看它是"真数组"还是 "假数组"。只要里面还有文件名，它就是真数组，不过一旦最后一个文件被移出，它就变成假数组。请注意我们早先说过的依然有效。虽然数组包含（和可以产生）一列数值，我们在标量环境里并不计算列表值。我们只是告诉数组这里是标量，然后问它觉得自己是什么。

不要试图在这里用 defined @files。那样没用，因为 defined 函数是询问一个标量是否为 undef，而一个数组不是标量。简单的布尔测试就够用了。

2.7.3 空（void）环境

另外一个特殊的标量环境是空环境（void context）。这个环境不仅不在乎返回值的类型，它甚至连返回值都不想要。从函数如何运行的角度看，这和普通标量环境没有区别。但是如果你打开了警告，如果你在一个不需要值的地方，比如说在一个不返回值的语句里，使用了一个没有副作用的表达式，Perl 的编译器就会警告你，比如，如果你用一个字串当作语句：

"Camel Lot";

你会收到这样的警告：

Useless use of a constant in void context in myprog line 123;

2.7.4 代换环境

我们早先说过双引号文本做反斜杠代换和变量代换，不过代换文本（通常称做“双引号文本”）不仅仅适用于双引号字串。其他的一些双引号类构造是：通用的反勾号操作符 qx，模式匹配操作符 m//，替换操作符 s///，和引起正则表达式操作符，qr//。替换操作符在处理模式匹配之前在它的左边做代换动作，然后每次匹配左边时做右边的代换工作。

代换环境只发生在引起里，或者象引起那样的地方，也许我们把它当作与标量及列表环境一样的概念来讲并不恰当。（不过也许是对的。）

2.8 列表值和数组

既然我们谈到环境，那我们可以谈谈列表文本和它们在环境里的性质。你已经看到过一些列表文本。列表文本是用逗号分隔的独立数值表示的（当有优先级要求的时候用圆括弧包围）。因为使用圆括弧几乎从不会造成损失，所以列表值的语法图通常象下面这样说明：

(LIST)

我们早先说过在语法描述里的 LIST 表示某个东西给它的参数提供了列表环境，不过只有列表文本自身会部分违反这条规则，就是说只有在列表和操作符全部处于列表环境里才会提供真正的列表环境。列表文本在列表环境里的内容只是顺序声明的参数值。作为一种表达式里的特殊的项，一个列表文本只是把一系列临时值压到 Perl 的堆栈里，当操作符需要的时候再从堆栈里弹出来。

不过，在标量环境里，列表文本并不真正表现得象一个列表（LIST），因为它并没有给它的值提供列表环境。相反，它只是在标量环境里计算它的每个参数，并且返回最后一个元素的值。这是因为它实际上就是伪装的 C 逗号操作符，逗号操作符是一个两目操作符，它会丢弃左边的值并且返回右边的值。用我们前面讨论过的术语来说，逗号操作符的左边实际上提供了一个空环境。因为逗号操作符是左关联的，如果你有一系列逗号分隔的数值，那你总是得到最后一个数值，因为最后一个逗号会丢弃任何前面逗号生成的东西。因此，要比较这两种环境，列表赋值：

```
@stuff = ( "one", "two", "three");
```

给数组@stuff，赋予了整个列表的值。但是标量赋值：

```
$stuff = ( "one", "two", "three");
```

只是把值 "three" 赋予了变量 $stuff。和我们早先提到的 @files 数组一样，逗号操作符知道它是处于标量还是列表环境，并且根据相应环境选择其动作。

值得说明的一点是列表值和数组是不一样的。一个真正的数组变量还知道它的环境，处于列表环境时，它会象一个列表文本那样返回其内部列表。但是当处于标量环境时，它只返回数组长度。下面的东西给 $stuff 赋值 3：

@stuff = ("one", "two", "three"); $stuff = @stuff;

如果你希望它获取值 "three"，那你可能是认为 Perl 使用逗号操作符的规则，把 @stuff 放在堆栈里的临时值都丢掉，只留下一个交给 $stuff。不过实际上不是这样。@stuff 数组从来不把它的所有值都放在堆栈里。实际上，它从来不在堆栈上放任何值。它只是在堆栈里放一个数值——数组长度，因为它知道自己处于标量环境。没有任何项或者操作符会在标量环境里把列表放入堆栈。相反，它会在堆栈里放一个标量，一个它喜欢的值，而这个值不太可能是列表的最后一个值（就是那个在列表环境里返回的值），因为最后一个值看起来不象时在标量环境里最有用的值。你的明白？（如果还不明白，你最好重新阅读本自然段，因为它很重要。）

现在回到真正的 LIST（列表环境）。直到现在我们都假设列表文本只是一个文本列表。不过，正如字串文本可能代换其他子字串一样，一个列表文本也可以代换其他子列表。任何返回值的表达式都可以在一个列表中使用。所使用的值可以是标量值或列表值，但它们都成为新列表值的一部分，因为 LIST 会做子列表的自动代换。也就是说，在计算一个 LIST 时，列表中的每个元素都在一个列表环境中计算，并且生成的列表值都被代换进 LIST，就好象每个独立的元素都是 LIST 的成员一样。因此数组在一个 LIST 中失去它们的标识（注：有些人会觉得这是个问题，但实际上不是。如果你不想失去数组的标识，那么你总是可以用一个引用代换数组。参阅第八章。）。列表：

(@stuff,@nonsense,funkshun())

包含元素 @stuff，跟着是元素 @nonsense，最后是在列表环境里调用子过程 &funkshun 时它的返回值。请注意她们的任意一个或者全部都可能被代换为一个空列表，这时就象在该点没有代换过数组或者函数调用一样。空列表本身由文本 () 代表。对于空数组，它会被代换为一个空列表因而可以忽略，把空列表代换为另一个列表没有什么作用。所以，((),(),())等于()。

这条规则的一个推论就是你可以在任意列表值结尾放一个可选的逗号。这样，以后你回过头来在最后一个元素后面增加更多元素会简单些：

```
@releases = (
    "alpha",
    "beta",
```

```
    "gamma",);
```

或者你可以完全不用逗号：另一个声明文本列表的方法是用我们早先提到过的 qw（引起字）语法。这样的构造等效于在空白的地方用单引号分隔。例如：

```
@froots = qw(

    apple       banana      carambola

    coconut     guava       kumquat

    mandarin    nectarine   peach

    pear        persimmon   plum);
```

（请注意那些圆括弧的作用和引起字符一样，不是普通的圆括弧。我们也可以很容易地使用尖括弧或者花括弧或者斜杠。但是圆括弧看起来比较漂亮。）

一个列表值也可以象一个普通数组那样使用脚标。你必须把列表放到一个圆括弧（真的）里面以避免混淆。我们经常用到从一个列表里抓取一个值，但这时候实际上是抓了列表的一个片段，所以语法是：

(LIST)[LIST]

例子：

```
# Stat 返回列表值

$modification_time = (stat($file))[9];


# 语法错误

$modification_time = stat($file)[9];   # 忘记括弧了。


# 找一个十六进制位

$hexdigit = ('a','b','c','d','e','f')[$digit-10];
```

```
# 一个“反转的逗号操作符”。
return (pop(@foo),pop(@foo))[0];


# 把多个值当作一个片段
($day, $month, $year) = (localtime)[3,4,5];
```

2.8.1 列表赋值

只有给列表赋值的每一个元素都合法时，才能给整个列表赋值：

```
($a, $b, $c) = (1, 2, 3); ($map{red}, ${map{green}, $map{blue}) =
(0xff0000, 0x00ff00, 0x0000ff);
```

你可以给一个列表里的 undef 赋值。这一招可以很有效地把一个函数的某些返回值抛弃：

```
($dev, $ino, undef, undef, $uid, $gid) = stat($file);
```

最后一个列表元素可以是一个数组或散列：

```
($a, $b, @rest) = split;
my ($a, $b, %rest) = @arg_list;
```

实际上你可以在赋值的列表里的任何地方放一个数组或散列，只是第一个数组或散列会吸收所有剩余的数值，而且任何在它们后面的东西都会被设置为未定义值。这样可能在 local 或 my 里面比较有用，因为这些地方你可能希望数组初始化为空。

你甚至可以给空列表赋值：

```
() = funkshun();
```

这样会导致在列表环境里调用你的函数，但是把返回值丢弃。如果你在没有赋值（语句）的情况下调用了此函数，那它就会在一个空环境里被调用，而空环境是标量环境，因此可能令此函数的行为完全不同。

在标量环境里的列表赋值返回赋值表达式右边生成的元素的个数：

```
$x = (($a,$b)=(7,7,7));      # 把 $x 设为 3，不是 2
$x = ( ($a, $b) = funk());   # 把 $x 设为 funk() 的返回数
```

```
$x = ( () = funk() );       # 同样把$x 设为 funk() 的返回数
```

这样你在一个布尔环境里做列表赋值就很有用了，因为大多数列表函数在结束的时候返回一个空（null）列表，空列表在赋值时生成一个 0，也就是假。下面就是你可能在一个 while 语句里使用的情景：

```
while (($login, $password) = getpwent) {
    if (crypt($login, $password) eq $password) {
        print "$login has an insecure password!\n";
    }
}
```

2.8.2 数组长度

你可以通过在标量环境里计算数组 @days 而获取数组 @days 里面的元素的个数，比如：

@days + 0; # 隐含地把 @days 处于标量环境 scalar(@days) # 明确强制 @days 处于标量环境

请注意此招只对数组有效。并不是一般性地对列表值都有效。正如我们早先提到的，一个逗号分隔的列表在标量环境里返回最后一个值，就象 C 的逗号操作符一样。但是因为你几乎从来都不需要知道 Perl 里列表的长度，所以这不是个问题。

和 @days 的标量计算有紧密联系的是 $#days。这样会返回数组里最后一个元素的脚标，或者说长度减一，因为（通常）存在第零个元素。给 $#days 赋值则修改数组长度。用这个方法缩短数组的长度会删除插入的数值。你在一个数组扩大之前预先伸展可以获得一定的效能提升。（你还可以通过给超出数组长度之外的元素赋值的方法来扩展一个数组。）你还可以通过给数组赋空列表 () 把它裁断为什么都没有。下面的两个语句是等效的：

@whatever = (); $#whatever = -1;

而且下面的表达式总是真：

scalar(@whatever) == $#whatever + 1;

截断一个数组并不回收其内存。你必须 undef(@whatever) 来把它的内存释放回你的进程的内存池里。你可能无法把它释放回你的系统的内存池，因为几乎没有那种操作系统支持这样做。

2.9 散列

如前所述，散列只是一种有趣的数组类型，在散列里你是用字串而不是数字来取出数值。散列定义键字和值之间的关联，因此散列通常被那些打字不偷懒的人称做关联数组。

在 Perl 里实际上是没有叫什么散列文本的东西的，但是如果你给一个散列赋一个普通列表的值，列表里的每一对值将被当作一对键字/数值关联：

```
%map = ('red', 0xff0000,'green', 0x00ff00,'blue',0x0000ff);
```

上面形式和下面的形式作用相同：

```
%map = ();       # 先清除散列
$map{red} = 0xfff0000;
$map{green} = 0x00ff00;
$map{blue} = 0x0000ff;
```

通常在键字/数值之间使用 => 操作符会有更好的可读性。=> 操作符只是逗号的同义词，不过却有更好的视觉区分效果，并且还把任何空标识符引起在其左边（就象上面的花括弧里面的标识符），这样，它在若干种操作中就显得非常方便，包括初始化散列变量：

```
%map = (
    red => 0xff0000,
    green => 0x00ff00,
    blue => 0x0000ff,
        );
```

或者初始化任何当作记录使用的匿名散列引用：

```
$rec = {
    NAME => 'John Simth',
```

```
    RANK => 'Captain',
    SERNO => '951413',
};
```

或者用命名的参数激活复杂的函数：

```
$fiels = radio_group(
        NAME => 'animals'
        VALUES   =>['camel','llama','ram','wolf'],
        DEFAULT   =>'camel',
        LINEBREAD   => 'true',
        LABELS   =>\%animal_names,
    );
```

不过这里我们又走的太远了。先回到散列。

你可以在一个列表环境里使用散列变量（%hash），这种情况下它把它的键字/数值对转换成列表。但是，并不意味着以某种顺序初始化的散列就应该同样的顺序恢复出来。散列在系统内部实现上是使用散列表来达到高速查找，这意味着记录存储的顺序和内部用于计算记录在散列表的里的位置的散列函数有关，而与任何其它事情无关。因此，记录恢复出来的时候看起来是随机的顺序。（当然，每一对键字/数值是以正确的顺序取出来的。）关于如何获得排序输出的例子，可以参考第二十九章的 keys 函数。

当你在标量环境里计算散列变量的数值的时候，它只有在散列包含任意键字/数值对时才返回真。如果散列里存在键字/数值对，返回的值是一个用斜线分隔的已用空间和分配的总空间的值组成的字串。这个特点可以用于检查 Perl 的（编译好的）散列算法在你的数据集里面性能是否太差。比如，你把 10,000 个东西放到一个散列里面，但是在标量环境里面计算 %HASH 得出"1/8"，意味着总共八个桶里只用了一个桶。大概是一个桶里存放了 10,000 个条目。这可是不应该发生的事情。

要算出一个散列里面的键字的数量，在标量环境里使用 keys 函数：
scalar(keys(%HASH)).

你可以通过在花括弧里面声明用逗号分隔的，超过一个键字的方法仿真多维数组。列出的键字连接到一起，由 $;（$SUBSCRIPT_SEPARATOR）（缺省值是 chr(28)）的内容分隔。结果字串用做散列的真实键字。下面两行效果相同：

```
$people{ $state, $country } = $census_results;

$people{ join $; =>$state, $county} = $census_results;
```

这个特性最初是为了支持 a2p（awk 到 perl 转换器）而实现的。现在，你通常会只使用第九章，数据结构，里写的一个真的（或者，更真实一些）的多维数组。旧风格依然有用的一个地方是与 DBM 文件捆绑在一起的散列（参阅第三十二章，标准模块，里的 DB_File），因为它不支持多维键字。

请不要把多维散列仿真和片段混淆起来。前者表示一个标量数值，后者则是一个列表数值：

```
$hash{ $x, $y, $z}      # 单个数值

@hash{ $x, $y, $z}      # 一个三个值的片段
```

2.10 型团（typeglob）和文件句柄

Perl 里面有种特殊的类型叫类型团（typeglob）用以保留整个符号表记录。（符号表记录 *foo 包括 $foo， @foo， %foo，&foo 和其他几个 foo 的简单解释值。）类型团（typeglob）的类型前缀上一个 *，因为它代表所有类型。

类型团（typeglob）（或由此的引用）的一个用途是是用于传递或者存储文件句柄。如果你想保存一个文件句柄，你可以这么干：

```
$fh = *STDOUT;
```

或者作为一个真的引用，象这样：

```
$fh = \*STDOUT;
```

这也是创建一个本地文件句柄的方法，比如：

```
sub newopen {
    my  $path = shift;
    local *FH;      # 不是 my() 或 our ()
    open(FH,$path ) or return undef;
```

```
    return *FH;       # 不是\*FH!
}

$fh = newopen('/etc/passwd');
```

参阅 open 函数获取另外一个生成新文件句柄的方法。

类型团如今的主要用途是把一个符号表取另一个符号表名字做别名。别名就是外号，如果你说：

```
*foo = *bar;
```

那所有叫"foo"的东西都是每个对应的叫"bar"的同意词。你也可以通过给类型团赋予引用实现只给某一个变量取别名：

```
*foo = \$bar;
```

这样 $foo 就是 $bar 的一个别名，而没有把 @foo 做成 @bar 的别名，或者把 %foo 做成 %bar 的别名。所有这些都只影响全局（包）变量；词法不能通过符号表记录访问。象这样给全局变量别名看起来可能有点愚蠢，不过事实是整个模块的输入/输出机制都是建筑在这个特性上的，因为没有人要求你正在当别名用的符号必须在你的名字空间里。因此：

```
local *Here::blue = \$There::green;
```

临时为 $There::green 做了一个叫 $Here::blue 的别名，但是不要给 @There:green 做一个叫 @Here::blue 的别名，或者给 %There::green 做一个 %Here::blue 的别名。幸运的是，所有这些复杂的类型团操作都隐藏在你不必关心的地方。参阅第八章的"句柄参考"和"符号表参考"，第十章的"符号表"，和第十一章，模块，看看更多的关于类型团的讨论和重点。

2.11 输入操作符

这里我们要讨论几个操作符，因为他们被当作项分析。有时候我们称它们为伪文本，因为它们在很多方面象引起的字串。（象 print 这样的输出操作符被当作列表操作符分析并将在第二十九章讨论。）

2.11.1 命令输入（反勾号）操作符

首先，我们有命令输入操作符，也叫反勾号操作符，因为它看起来象这样：

$info = `finger $user`;

一个用反勾号（技术上叫重音号）引起的字串首先进行变量替换，就象一个双引号引起的字串一样。得到的结果然后被系统当作一个命令行，而且那个命令的输出成为伪文本的值。（这是一个类似 Unix shell 的模块。）在标量环境里，返回一个包含所有输出的字串。在列表环境里，返回一列值，每行输出一个值。（你可以通过设置 $/ 来使用不同的行结束符。）

每次计算伪文本的时候，该命令都得以执行。该命令的数字状态值保存在 $?（参阅第二十八章获取 $? 的解释，也被称为 $CHILD_ERROR ）。和这条命令的 csh 版本不同的是，对返回数据不做任何转换——换行符仍然是换行符。和所有 shell 不同的是，Perl 里的单引号不会隐藏命令行上的变量，使之避免代换。要给 shell 传递一个 $，你必须用反斜杠把它隐藏起来。我们上面的 finger 例子里的 $user 被 Perl 代换，而不是被 shell。（因为该命令 shell 处理，参阅第二十三章，安全，看看与安全有关的内容。）

反勾号的一般形式是 qx//（意思是"引起的执行"），但这个操作符的作用完全和普通的反勾号一样。你只要选择你的引起字符就行了。有一点和引起的伪函数类似：如果你碰巧选择了单引号做你的分隔符，那命令行就不会进行双引号代换：

$perl_info = qx(ps $$); # 这里 $$ 是 Perl 的处理对象 $perl_info = qx'ps $$'; # 这里 $$ 是 shell 的处理对象

2.11.2 行输入（尖角）操作符

最频繁使用的是行输入操作符，也叫尖角操作符或者 readline 函数（因为那是我们内部的叫法）。计算一个放在尖括弧里面的文件句柄（比如 STDIN）将导致从相关的文件句柄读取下一行。（包括新行，所以根据 Perl 的真值标准，一个新输入的行总是真，直到文件结束，这时返回一个未定义值，而未定义值习惯是为假。）通常，你会把输入值赋予一个变量，但是有一种情况会发生自动赋值的现象。当且仅当行输入操作符是一个 while 循环的唯一一个条件的时候，其值自动赋予特殊变量$_。然后就对这个赋值进行测试，看看它是否定义了。（这些东西看起来可能有点奇怪，但是你会非常频繁地使用到这个构造，所以值得花些时间学习。）因此，下面行是一样的：

```
while (defined($_ = <STDIN>)) {print $_; }   # 最长的方法

while ($_ = <STDIN) { pirnt; }          # 明确使用 $_

while (<STDIN>) { PRINT ;}          # 短形式

for (;<STDIN>;) { print;}          # 不喜欢用 while 循环

print $_ while defined( $_ = <STDIN>);      # 长的语句修改
```

```
print while $_ = <STDIN>;          # 明确$_

print while <STDIN>;               # 短的语句修改
```

请记住这样的特殊技巧要求一个 while 循环。如果你在其他的什么地方使用这样的输入操作符，你必须明确地把结果赋给变量以保留其值：

```
while(<FH1>&& <fh2>) { ... }          # 错误：两个输入都丢弃

if (<STDIN>)   { print; }          # 错误：打印$_原先的值

if ($_=<STDIN>)   {PRINT; }          # 有小问题：没有测试是否定义

if (defined($_=<STDIN>)) { print;}      # 最好
```

当你在一个 $_ 循环里隐含的给 $_ 赋值的时候，你赋值的对象是同名全局变量，而不是 while 循环里的那只局部的。你可以用下面方法保护一个现存的 $_ 的值：

```
while(local $_=) { print; } # 使用局部 $_
```

当循环完成后，恢复到原来的值。不过，$_ 仍然是一个全局变量，所以，不管有意无意，从那个循环里调用的函数仍然能够访问它。当然你也可以避免这些事情的发生，只要定义一个文本变量就行了：

```
while (my $line = ) { print $line;} # 现在是私有的了
```

（这里的两个 while 循环仍然隐含地进行测试，看赋值结果是否已定义，因为 my 和 local 并不改变分析器看到的赋值。）文件句柄 STDIN，STDOUT，和 STDERR 都是预定义和预先打开的。额外的文件句柄可以用 open 或 sysopen 函数创建。参阅第二十九章里面那些函数的文档获取详细信息。

在上面的 while 循环里，我们是在一个标量环境里计算行输入操作符，所以该操作符分别返回每一行。不过，如果你在一个列表环境里使用这个操作符，则返回一个包括所有其余输入行的列表，每个列表元素一行。用这个方法你会很容易就使用一个很大的数据空间，所以一定要小心使用这个特性：

```
$one_line =; # 获取第一行 $all_lines =; # 获取文件其余部分。
```

没有哪种 while 处理和输入操作符的列表形式相关联，因为 while 循环的条件总是提供一个标量环境（就象在任何其他条件语句里一样）。

在一个尖角操作符里面使用空（null）文件句柄是一种特殊用法；它仿真典型的 Unix 的命令行过滤程序（象 sed 和 awk）的特性。当你从一个 <> 读取数据行的时候，它会魔术

般的把所有你在命令行上提到的所有文件的所有数据行都交给你。如果你没有（在命令行）上声明文件，它就把标准输入交给你，这样你的程序就可以很容易地插入到一个管道或者一个进程中。

下面是其工作原理的说明：当第一次计算 <> 时，先检查 @ARGV 数组，如果它是空（null），则 $ARGV[0] 设置为 "-"，这样当你打开它的时候就是标准输入。然后 @ARGV 数组被当作一个文件名列表处理。更明确地说，下面循环：

```
while (<>) { ... # 处理每行的代码 }
```

等效于下面的类 Perl 的伪代码：

```
@ARGV = ('-')  unless @ARGV;      # 若为空则假设为 STDIN
while( @ARGV) {
   $ARGV = shift @ARGV;       # 每次缩短@ARGV
   if( !open(ARGV, $ARGV)) {
      warn "Can't open $ARGV: $!\n";
      next;
   }
   while (<ARGV>) {
      ...          # 处理每行的代码
   }
}
```

第一段代码除了没有那么唠叨以外，实际上是一样的。它实际上也移动 @ARGV，然后把当前文件名放到全局变量 $ARGV 里面。它也在内部使用了特殊的文件句柄 ARGV——<> 只是更明确的写法 <ARGV>（也是一个特殊文件句柄）的一个同义词，（上面的伪代码不能运行，因为它把 当作一个普通句柄使用。）

你可以在第一个 <> 语句之前修改 @ARGV，直到数组最后包含你真正需要的文件名列表为止。因为 Perl 在这里使用普通的 open 函数，所以如果碰到一个"-"的文件名，就会把它当作标准输入，而其他更深奥的 open 特性是是 Perl 自动提供给你的（比如打开一个名字是"gzip -dc <file.gz|"的文件）。行号（$.）是连续的，就好象你打开的文件是一个

大文件一样。（不过你可以重置行号，参阅第二十九章看看当到了 eof 时怎样为每个文件重置行号。）

如果你想把 @ARGV 设置为你自己的文件列表，直接用：

如果没有给出 args 则缺省为 README @ARGV = ("README") unless @ARGV;

如果你想给你的脚本传递开关，你可以用 Getopt::* 模块或者在开头放一个下面这样的循环：

```
while( @ARGV and $ARGV[0] =~ /^-/) {
    $_ = shift;
    last if /^--$/;
    if (/^-D(.*)/) {$debug = $1 }
    if (/^-v/)       { $verbose++}
    ...       # 其他开关
}
while(<>){
    ...       # 处理每行的代码
}
```

符号 <> 将只会返回一次假。如果从这（返回假）以后你再次调用它，它就假设你正在处理另外一个 @ARGV 列表，如果你没有设置 @ARGV，它会从 STDIN 里输入。

如果尖括弧里面的字串是一个标量变量（比如，<$foo>），那么该变量包含一个间接文件句柄，不是你准备从中获取输入的文件句柄的名字就是一个这样的文件句柄的引用。比如：

$fh = *STDIN; $line = <$fh>;

或：

open($fh, "<data.txt"); $line = <$fh>;

2.11.3 文件名聚集操作符

你可能会问：如果我们在尖角操作符里放上一些更有趣的东西，行输入操作符会变成什么呢？答案是它会变异成不同的操作符。如果在尖角操作符里面的字串不是文件句柄名或标量变量（甚至只是多了一个空格），它就会被解释成一个要“聚集”（注：文件团和前面提到的类型团毫无关系，除了它们都把 * 字符用于通配符模式以外。当用做通配符用途时，字符 * 有“聚集”（glob）的别名。对于类型团而言，它是聚集符号表里相同名字的符号。对于文件团而言，它在一个目录里做通配符匹配，就象各种 shell 做的一样。）的文件名模式。这里的文件名模式与当前目录里的（或者作为文件团模式的一部分直接声明的目录）文件名进行匹配，并且匹配的文件名被该操作符返回。对于行输入而言，在标量环境里每次返回一个名字，而在列表环境里则是一起返回。后面一种用法更常见；你常看到这样的东西：

```
@files = <*.xml>;
```

和其他伪文本一样，首先进行一层的变量代换，不过你不能说 <$foo>，因为我们前面已经解释过，那是一种间接文件句柄。在老版本的 Perl 里，程序员可以用插入花括弧的方法来强制它解释成文件团：<${foo}>。现在，我们认为把它当作内部函数 glob($foo) 调用更为清晰，这么做也可能是在第一时间进行干预的正确方法。所以，如果你不想重载尖角操作符（你可以这么干。）你可以这么写：

```
@files = glob("*.xml");
```

不管你用 glob 函数还是老式的尖括弧形式，文件团操作符还是会象行输入操作符那样做 while 特殊处理，把结果赋予 $_。（也是在第一时间重载尖角操作符的基本原理。）比如，如果你想修改你的所有 C 源代码文件的权限，你可以说：

```
while (glob "*.c") { chmod 0644, $_; }
```

等效于：

```
while (<*.c>) { chmod 0644,$_; }
```

最初 glob 函数在老的 Perl 版本里是作为一个 shell 命令实现的（甚至在旧版的 Unix 里也一样），这意味着运行它开销相当大，而且，更糟的是它不是在所有地方运行得都一样。现在它是一个内建的函数，因此更可靠并且快多了。参阅第三十二章里的 File:Glob 模块的描述获取如何修改这个操作符的缺省特性的信息，比如如何让它把操作数（参数）里面的空白当作路径名分隔符，是否扩展发音符或花括弧，是否大小写敏感和是否对返回值排序等等。

当然，处理上面 chmod 命令的最短的和可能最易读的方法是把文件团当作一个列表操作符处理：

```
chmod 0644, <*.c>;
```

文件团只有在开始（处理）一个新列表的时候才计算它（内嵌）的操作数。所有数值必须在该操作符开始处理之前读取。这在列表环境里不算什么问题，因为你自动获取全部数值。不过，在标量环境里时，每次调用操作符都返回下一个值，或者当你的数值用光后返回一个假值。同样，假值只会返回一次。所以如果你预期从文件团里获取单个数值，好些的方法是：

```
($file) = ; # 列表环境
```

上面的方法要比：

```
$fiole = ; # 标量环境
```

好，因为前者返回所有匹配的文件名并重置该操作符，而后者要么返回文件名，要么返回假。

如果你准备使用变量代换功能，那么使用 glob 操作符绝对比使用老式表示法要好，因为老方法会导致与间接文件句柄的混淆。这也是为什么说项和操作符之间的边界线有些模糊的原因：

```
@files = <$dir/*.[ch]>; # 能用，不过应该避免这么用。 @files = glob("dir/*.[ch]"); # 把 glob 当函数用。 @files = glob $some_pattern; # 把 glob 当操作符用。
```

我们在最后一个例子里把圆括弧去掉是为了表明 glob 可以作为函数（一个项）使用或者是一个单目操作符用；也就是说，一个接受一个参数的前缀操作符。glob 操作符是一个命名的单目操作符的例子；是我们下一章将要谈到的操作符。稍后，我们将谈谈模式匹配操作符，它也是分析起来类似项，而作用象操作符。

第三章 单目和双目操作符

在上面一章里，我们讲了各种你可能在表达式里用到的项，不过老实说，把项隔离出来让人觉得有点无聊。因为许多项都是群居动物。它们相互之间有某种关系。年轻的项急于以各种方式表现自己并影响其它项，而且还存在不同类型的社会关系和许多不同层次的义务。在 Perl 里，这种关系是用操作符来表现的。

社会学必须对某些事物有利。

从数学的角度来看，操作符只是带着特殊语法的普通函数。从语言学的角度来说，操作符只是不规则动词。不过，几乎任何语言都会告诉你，在一种语言里的不规则动词很可能是你最

常用的语素。而从信息理论的角度来看，这一点非常重要，因为不规则动词不管是在使用中还是识别上都比较短而且更有效。

从实用角度出发，操作符非常易用。

根据操作符的元数（它们操作数的个数）的不同，它们的优先级（它们从周围的操作符手中夺取操作数的难易）的不同，它们的结合性（当与同优先级的操作符相联时，它们是从左到右处理还是从右到左处理。）的不同，操作符可以分成各种各样类型。

Perl 的操作符有三种元数：单目，双目和三目。单目操作符总是前缀操作符（除自增和自减操作符以外）。（注：你当然可以认为各种各样的引号和括弧是项与项之间分隔的环缀操作符。）其他的都是中缀操作符——除非你把列表操作符也算进来，它可以做任意数量参数的前缀。不过大多数人认为列表操作符只是一种普通的函数，只不过你可以不为它写括弧而已。下面是一些例子：

```
! $x          # 一个单目操作符

$x * $y          # 一个双目操作符

$x ? $y : $z       # 一个三目操作符

print $x, $y, $z    # 一个列表操作符
```

操作符的优先级控制它绑定的松紧度。高优先级的操作符先于低优先级的操作符攫取它们周围的参数。优先级的原理可以直接在基本数学里面找到，在数学里，乘法比加法优先级高：

```
1. + 3 * 4 # 生成 14 而不是 20
```

两个同等优先级的操作符在一起的时候，它们的执行顺序取决于它们的结合性。这些规则在某种程度上仍然遵循数学习惯：

```
2 * 3 * 4       # 意味着(2*3)*4，左结合

2 ** 3 ** 4       # 意味着 2**(3**4)，右结合

2 != 3 != 4       # 非法，不能结合
```

表 3-1 列出了从高优先级到低优先级的 Perl 操作符，以及它们的结合性和元数。

表 3-1。操作符优先级

结合性	元数	优先级表
无	0	项，和列表操作符（左侧）

左	2	->
无	1	++ --
右	2	**
右	1	!~&gt 和单目 + 和 -
左	2	=~ !~
左	2	* / % x
左	2	+ - .
左	2	<< > >
右	0,1	命名单目操作符
无	2	< > <= >= lt gt le ge
无	2	= <=> eq ne cmp
左	2	&
左	2	| ……
左	2	&&
左	2	| |
无	2	.. ...
右	3	?:
右	2	+ += -+ *= 等等
左	2	, =>
右	0+	列表操作符（右侧）
右	1	not
左	2	and
左	2	or xor

看起来好象要记很多的优先级级别。不错，的确很多。幸运的是，有两件事情可以帮助你。首先，这里定义的优先级级别通常遵循你的直觉（前提是你没得精神病）。第二，如果你得了精神病，那你总还是可以放上额外的圆括弧以减轻你的疑虑。

另外一个可以帮助你的线索是，任何从 C 里借来的操作符相互之间仍然保留相同的优先级关系，尽管 C 的优先级有点怪。（这就让那些 C 和 C++ 的爱好者，甚至还包括 JAVA 的爱好者学习起 Perl 来会更容易些。）

随后的各节按照优先级顺序讲述这些操作符。只有极少数例外，所有这样的操作符都只处理标量值，而不处理列表值。我们会在轮到它们出现的时候提到这一点。

尽管引用是标量值，但是在引用上使用大多数操作符没有什么意义，因为一个数字值的引用只是在 Perl 内部才有意义。当然，如果一个引用指向一个允许重载的类里的一个对象，你就可以在这样的对象上调用这些操作符，并且如果该类为那种特定操作符定义了重载，那它也会定义那个操作符应该如何处理该对象。比如，复数在 Perl 里就是这么实现的。有关重载的更多内容，请参考第十三章，重载。

3.1 项和列表操作符（左向）

在 Perl 里，项的优先级最高。项包括变量，引起和类似引起的操作符、大多数圆括弧（或者方括弧或大括弧）内的表达式，以及所有其参数被圆括弧包围的函数。实际上，Perl 里没有这种意义上的函数，只有列表操作符和单目操作符会表现得象函数——当你在它们的参数周围放上圆括弧的时候。不管怎样，第二十九章的名称是函数。

现在请注意听了。下面有几条非常重要的规则，它们能大大简化事情的处理，但是如果你粗心地话，偶尔会产生不那么直观的结果。如果有哪个列表操作符（如 print）或者哪个命名单目操作符（比如 chdir）后面跟着左圆括弧做为下一个记号（忽略空白），那么该操作符和它的用圆括弧包围的参数就获得最高优先级，就好象它是一个普通的函数调用一样。规则是：如果看上去象函数调用，它就是函数调用。你可以把它做得不象函数——在圆括弧前面加一个单目加号操作符即可，（从语意上来说，这个加号什么都没干，它甚至连参数是否数字都不关心）。

例如，因为 || 比 chdir 的优先级低，我们有：

```
chdir $foo || die;    # (chdir $foo) || die

chdir ($foo) || die;    # (chdir $foo) || die

chdir ($foo) || die;    # (chdir $foo) || die

chdir +($foo) || die;    # (chdir $foo) || die
```

不过，因为 * 的优先级比 chdir 高，我们有：

```
chdir $foo * 20;    # chdir ($foo * 20)

chdir ($foo) * 20;    # (chdir $foo) * 20

chdir ($foo) * 20;    # (chidir $foo) * 20

chdir +($foo) * 20     # chdir ($foo * 20 )
```

这种情况对任何命名单目操作符的数字操作符同样有效，比如 rand：

```
rand 10 * 20;               # rand (10 * 20)

rand(10) * 20;              # (rand 10) * 20

rand (10) * 20;             # (rand 10) * 20

rand +(10) * 20;            # rand (10 * 20)
```

当缺少圆括弧时，象 print，sort 或 chmod 这样的列表操作符的优先级要么非常高，要么非常低——取决于你是向操作符左边还是右边看。（这也是为什么我们在本节标题上有"左向"字样的原因。）比如，在：

```
@ary = (1, 3, sort 4, 2);

print @ary;       # 打印 1324
```

在 sort 右边的逗号先于 sort 计算，而在其左边的后其计算。换句话说，一个列表操作符试图收集它后面所有的参数，然后当做一个简单的项和它前面的表达式放在一起。但你还是要注意圆括弧的使用：

```
# 这些在进行 print 前退出

print($foo, exit);    # 显然不是你想要的。

print $foo, exit;     # 也不是这个


# 这些在退出前打印：

(print $foo), exit;   # 这个是你要的。

print ($foo), exit;   # 或者这个。

print ($foo), exit;    # 这个也行。
```

最容易出错的地方是你用圆括弧把数学参数组合起来的时候，但是你却忘记了圆括弧同时用于组合函数参数：

```
print ($foo & 255) + 1, "\n";    # 打印($foo & 255)
```

（译注：这里 print ($foo & 255) 构成函数，函数是一个项，项的优先级最高，因而先执行. ）

这句话可能和你一开始想的结果不同。好在这样的错误通常会生成类似 "Useless use of addition in a void context" 这样的警告——如果你打开了警告。

同样当作项分析的构造还有 do {} 和 eval {} 以及子过程和方法调用，匿名数组和散列构造符 [] 和 {}，还有匿名子过程构造符 {}。

3.2 箭头操作符

和 C 和 C++ 类似，双目操作符 -> 是一个中缀解引用操作符。如果右边是一个 [...] 数组下标、一个 {...} 散列下标、或者一个 (...) 子过程参数列表，那么左边必须是一个对应的数组、散列、或者子过程的应用（硬引用或符号引用都行）。在一个左值（可赋值）环境里，如果左边不是一个引用，那它必须是一个能够保存硬引用的位置，这种情况下这种引用会为你自动激活。有关这方面的更多的信息（以及关于故意自激活的一些警告信息），请参阅第八章，引用。

```
$aref-&gt;[42]      # 一个数组解引用

$href-&gt;{"corned beff"}   # 一个散列解引用

$sref-&gt;(1,2,3)       # 一个子过程解引用
```

要不然，它就是某种类型的方法调用。右边必须是一个方法名（或者一个包含该方法名的简单标量变量），而且左边必须得出一个对象名（一个已赐福引用）或者一个类的名字（也就是说，一个包名字）：

$yogi = Bear->new("Yogi"); # 一个类方法调用 $yogi->swipe($picnic); # 一个对象方法调用

方法名可以用一个包名修饰以标明在哪个包里开始搜索该方法，或者带着特殊包名字，SUPER::，以表示搜索应该从父类开始。参阅第十二章，对象。

3.3 自增和自减操作符

++ 和 -- 操作符的功能和 C 里面一样。就是说，当把它们放在一个变量前面时，它们在返回变量值之前增加或者减少变量值，当放在变量后面时，它们在返回变量值后再对其加一或减一。比如，$a++ 把标量变量 $a 的值加一，在它执行增加之前返回它的值。类似地，--$b{(/(\w+)/)[0]} 把散列 %b 里用缺省的搜索变量（$_）里的第一个"单词"索引的元素先减一，然后返回。（注：哦，这儿可能有点不公平，因为好多东西你还不知道。我们只是想让你专心。该表达式的工作过程是这样的：首先，模式匹配用表达式 \w+ 在 $_ 里找第一个单词。它周围的圆括弧确保此单词作为单元素列表值返回，因为该模式匹配是在列

表环境里进行的。这个列表环境是由列表片段操作符，(...)[0] 提供的，它返回列表的第一个（也是唯一一个）元素。该值用做散列的键字，然后散列记录（值）被判断并返回。通常，如果碰到一个复杂的表达式，你可以从内向外地分析它并找出事情发生的顺序。）

自增操作符有一点额外的内建处理。如果你增加的变量是一个数字，或者该变量在一个数字环境里使用，你得到正常自增的功能。不过，如果该变量从来都是在字串环境里使用，而且值为非空，还匹配模式/^[a-zA-z]*[0-9]*$/，这时自增是以字串方式进行的，每个字符都保留在其范围之内，同时还会进位：

```
print ++($foo = '99');    # 打印'100'

print ++($foo = 'a0');    # 打印'a1'

print ++($foo = 'Az');    # 打印'Ba'

print ++($foo = 'zz');    # 打印'aaa'
```

在我们写这些的时候，自增的额外处理还没有扩展到 Unicode 字符和数字，不过将来也许会的。

不过自减操作符没有额外处理，我们也没有准备给它增加这个处理。

3.4 指数运算

双目 ** 是指数操作符。请注意它甚至比单目操作符的绑定更严格，所以 -2**4 是 -(2**4)，不是 (-2)**4。这个操作符是用 C 的 pow(3) 函数实现的，该函数在内部以浮点数模式运转。它用对数运算进行计算，这就意味着它可以处理小数指数，不过有时候你得到的结果不如直接用乘法得出的准确。

3.5 表意单目操作符

大多数单目操作符只有名字（参阅本章稍后的“命名的单目和文件测试操作符”），不过，有些操作符被认为比较重要，所以赋予它们自己的特殊符号。所有这类操作符好象都和否定操作有关。骂数学家去。

单目 ! 执行逻辑否，就是说，“not”。参阅 not 看看一个在优先级中级别较低的逻辑否。如果操作数为假（数字零，字串"0"，空字串或未定义），则对操作数取否，值为真（1），若操作数为真，则值为假（“”）。

如果操作数是数字，单目 - 执行数学取负。如果操作数是一个标识，则返回一个由负号和标识符连接在一起的字串。否则，如果字串以正号或负号开头，则返回以相反符号开头的字

串。这些规则的一个效果是 -bareword 等于 "-bareword"。这个东西对 Tk 程序员很有用。

单目 ~ 操作符进行按位求反，也就是 1 的补数。从定义上来看，这个是有点不可移植的东西，因为它受限于你的机器。比如，在一台 32 位机器上，~123 是 4294967172，而在一台 64 位的机器上，它是 18446744073709551493。不过你早就知道这个了。

你可能还不知道的是，如果 ~ 的参数是字串而不是数字，则返回等长字串，但是字串的所有位都是互补的。这是同时翻转所有位的最快的方法，而且它还是可移植的翻转位的方法，因为它不依靠你的机器的字大小。稍后我们将谈到按位逻辑操作符，它也有一个面向字串的变体。

单目 + 没有任何语义效果，即使对字串也一样。它在语法上用于把函数名和一个圆括弧表达式分隔开，否则它们会被解释成一个一体的函数参数。（参阅"项和列表操作符"的例子。）如果你向它的一边进行考虑，+ 取消了圆括弧把前缀操作符变成函数的作用。

单目操作符 \ 给它后面的东西创建一个引用。在一个列表上使用时，它创建一列引用。参阅第八章中的"反斜杠操作符"获取详细信息。不要把这个性质和字串里的反斜杠的作用混淆了，虽然两者都有防止下一个东西被转换的模糊的含义。当然这个相似也并不是完全偶然的。

3.6 绑定操作符

双目 =~ 把一个字串和一个模式匹配、替换或者转换绑定在一起。要不然这些操作会搜索或修改包含在 $_（缺省变量）里面的字串。你想绑定的字串放在左边，而操作符本身放在右边。返回值标识右边的操作符的成功或者失败，因为绑定操作符本身实际上不做任何事情。

如果右边的参数是一个表达式而不是模式匹配、子过程或者转换，那运行时该表达式会被解释成一个搜索模式。也就是说，$_ =~ $pat 等效于 $_ =~ /$pat/。这样做要比明确搜索效率低，因为每次计算完表达式后都必须检查模式以及可能还要重新编译模式。你可以通过使用 qr//(引起正则表达式）操作符预编译最初的模式的方法来避免重新编译。

双目 !~ 类似 =~ 操作符，只是返回值是 =~ 的对应返回值的逻辑非。下面的表达式功能上是完全一样的：

```
$string !~ /pattern/

not $string =~ /pattern/
```

我们说返回值标识成功，但是有许多种成功。替换返回成功替换的数量，转换也一样。（实际上，转换操作符常用于字符计数。）因为任何非零值都是真，所以所有的都对。最吸引人

的真值类型是模式的列表赋值：在列表环境下，模式匹配可以返回和模式里圆括弧相匹配的子字串。不过，根据列表赋值的规则，如果有任何东西匹配并且赋了值，列表赋值本身将返回真，否则返回假。因此，有时候你会看到这样的东西：

```
if( ($k, $v) = $string =~ m/(\w+)=(\w*)/) {
   print "KEY $k VALUE $v\n";
}
```

让我们分解这个例子。 =~ 的优先级比 = 高，因此首先计算 =~。 =~ 把字串 $string 绑定与右边的模式进行匹配，右边是扫描你的字串里看起来象 KEY=VALUES 这样的东西。这是在列表环境里，因为它是在一个列表赋值的右边。如果匹配了模式，它返回一个列表并赋值给 $k 和 $v。列表赋值本身是在标量环境，所以它返回 2－－赋值语句右边的数值的个数。而 2 正好又是真——因为标量环境也是一个布尔环境。当匹配失败，没有赋值发生，则返回零，是假。

关于模式规则的更多内容，参阅第五章，模式匹配。

3.7 乘号操作符

Perl 提供类似 C 的操作符 **（乘）、/（除）、和 %（模除）。** 和 / 的运行和你预料的一样，对其两个操作数进行乘或除。除法是以浮点数进行的，除非你用了 integer 用法模块。

% 操作符在用整数除法计算余数前，把它的操作数转换为整数。（不过，如果必要，它会以浮点进行除法，这样你的操作数在大多数 32 位机器上最多可以有（以浮点）15 位。）假设你的两个操作数叫 $b 和 $a。如果 $b 是正数，那么 $a % $b 的结果是 $a 减去 $b 不大于 $a 的最大倍数（也就意味着结果总是在范围 0 .. $b-1 之间）。如果 $b 是负数，那么 $a % $b 的结果是 $a 减去 $b 不小于 $a 的最小倍数（意味着结果介于 $b+1 .. 0 之间）。

当 use integer 在范围里时，% 直接给你由你的 C 编译器实现的模除操作符。这个操作符对负数定义得不是很好，但是执行得更快。

双目 x 是复制操作符。实际上，它是两个操作符，在标量环境里，它返回一个由左操作数重复右操作数的次数连接起来的字串。（为了向下兼容，如果左操作数没有位于圆括弧中，那么它在列表环境里也这样处理。）

```
print '-' x 80;             # 打印一行划线
```

```
print "\t" x ($tab/8), ' ' x ($tab%8);   # 跳过
```

在列表环境里，如果左操作数是在圆括弧中的列表，x 的作用是一个列表复制器，而不是字串复制器。这个功能对初始化一个长度不定的数组的所有值为同一值时很有用：

```
@ones = (1) x80;       # 一个 80 个 1 的列表
@ones = (5) x @ones;       # 把所有元素设置为 5
```

类似，你还可以用 x 初始化数组和散列片段：

```
@keys = qw(perls before swine);
@hash{@keys} = ("") x @keys;
```

如果这些让你迷惑，注意 @keys 被同时当做一个列表在赋值左边使用和当做一个标量值（返回数组长度）在赋值语句右边。前面的例子在 %hash 上有相同的作用：

```
$hash{perls} = "";
$hash{before} = "";
$hash{swine} = "";
```

3.8 附加操作符

很奇怪的是，Perl 还有惯用的 +（加法）和 -（减法）操作符。两种操作符都在必要的时候把它们的参数从字串转换为数字值并且返回一个数字值。

另外，Perl 提供 . 操作符，它做字串连接处理。比如：

```
$almost = "Fred" . "Flitstone"; # 返回 FredFlitstone[2]
```

请注意 Perl 并不在连接的字串中间放置空白。如果你需要空白，或者你要连接的字串多于两个，你可以使用 join 操作符，在第二十九章，函数，中介绍。更常用的是人们在一个双引号引起的字串里做隐含的字串连接：

```
$fullname = "$firstname $lastname";
```

3.9 移位操作符

按位移位操作符（<< 和 >>）返回左参数向左（<<）或向右（>>）移动由右参数声明位（是 bit）数的值。参数应该是整数。比如：

1. << 4; # 返回 16
2. >> 4; # 返回 2

3.10 命名单目操作符和文件测试操作符

在第二十九章里描述的一些"函数"实际上都是单目操作符。表 3-2 列出所有命名的单目操作符。

表 3-2 命名单目操作符

-X (file tests)	gethostbyname	localtime	return
alarm	getnetbyname	lock	rmdir
caller	getpgrp	log	scalar
chdir	getprotobyname	lstat	sin
chroot	glob	my	sleep
cos	gmtime	oct	sqrt
defined	goto	ord	srand
delete	hex	quotemeta	stat
do	int	rand	uc
eval	lc	readlink	ucfirst
exists	lcfirst	ref	umask
exit	length	require	undef

单目操作符比某些双目操作符的优先级高。比如：

```
sleep 4 | 3;
```

并不是睡 7 秒钟；它先睡 4 秒钟然后把 sleep 的返回值（典型的是零）与 3 进行按位或操作，就好象该操作符带这样的圆括弧：

```
(sleep 4) | 3;
```

与下面相比：

```
print 4 | 3;
```

上面这句先拿 4 和 3 进行或操作，然后再打印之（本例中是 7），就好象是下面这样写的一样：

```
print (4 | 3);
```

这是因为 print 是一个列表操作符，而不是一个简单的单目操作符。一旦你知道了哪个操作符是列表操作符，你再把单目操作符和列表操作符区分开就不再困难了。当你觉得有问题时，你总是可以用圆括弧把一个命名的单目操作符转换成函数。记住：如果看起来象函数，那它就是函数。

有关命名单目操作符的另一个趣事是，它们中的许多在你没有提供参数时，缺省使用 $_。不过，如果你省略了参数，而跟在命名单目操作符后面的记号看起来又象一个参数开头的话，那 Perl 就傻了，因为它期望的是一个项。如果 Perl 的记号是列在表 3-3 中的一个字符，那么该记号会根据自己是期待一个项还是操作符转成不同的记号类型。

表 3-3 模糊字符

字符	操作符	项
+	加法	单目正号
-	减法	单目负号
*	乘法	*类型团
/	除法	/模式/
<	小于号，左移	,<<END
.	连接	.3333
?	?:	?模式?
%	模除	%assoc
&	&, &&	&subroutine（子过程）

所以，典型的错误是：

next if length < 80;

在这里，< 在分析器眼里看着象 <> 输入符号（一个项）的开始，而不是你想要的“小于”（操作符）。我们实在是没办法修补这个问题同时还令 Perl 没有毛病。如果你实在懒得连 $_ 这两个字符都不愿意敲，那么用下面的代替：

```
next if length() <80;

next if (length) < 80;

next if 80 > length;

next unless length >== 80;
```

当（分析器）期望一个项时，一个负号加一个字母总是被解释成一个文件测试操作符。文件测试操作符是接受一个参数的单目操作符，其参数是文件名或者文件句柄，然后测试该相关

的文件，看看某些东西是否为真。如果忽略参数，它测试 $_，但除了 -t 之外，-t 是测试 STDIN。除非另有文档，它测试为真时返回 1，为假时返回 ""，或者如果文件不存在或无法访问时返回未定义。目前已实现的文件测试操作符列在表 3-4。

表 3-4 文件测试操作符

操作符	含义
-r	文件可以被有效的 UID/GID 读取。
-w	文件可以被有效的 UID/GID 写入。
-x	文件可以被有效的 UID/GID 执行。
-o	文件被有效 UID 所有
-R	文件可以被真实的 UID/GID 读取。
-W	文件可以被真实的 UID/GID 写入。
-X	文件可以被真实的 UID/GID 执行。
-O	文件被真实的 UID 所有
-e	文件存在
-z	文件大小为零
-s	文件大小不为零（返回大小）
-f	文件是简单文件
-d	文件是目录
-l	文件是符号连接
-p	文件是命名管道（FIFO）。
-S	文件是套接字
-b	文件是特殊块文件
-c	文件是特殊字符文件
-t	文件句柄为一个 tty 打开了
-u	文件设置了 setuid 位
-g	文件设置了 setgid 位
-k	文件设置了 sticky 位
-T	文件是文本文件
-B	文件是一个二进制文件（与-T 对应）
-M	自从修改以来的文件以天记的年龄（从开始起）
-A	自从上次访问以来的文件以天记的年龄（从开始起）
-C	自从 inode 修改以来的文件以天记的年龄（从开始起）

请注意 -s/a/b/ 并不是做一次反向替换。不过，说 -exp($foo) 仍然会和你预期的那样运行，因为只有跟在负号后面的单个字符才解释成文件测试。

文件权限操作符 -r ，-R，-w，-W，-x 和 -X 的解释各自基于文件和用户的用户 ID 和组 ID。可能还有其他原因让你无法真正读，写或执行该文件，比如 Andrew File System(AFS) 的的访问控制列表（注：不过，你可以用 use filetest 用法覆盖内建的语义。参阅第三十一章，用法模块）。还要注意的是，对于超级用户而言，-r，-R，-w 和 -W 总总是返回 1，并且如果文件模式里设置了执行位，-x 和-X 也返回 1。因此，由超级用户执行的脚本可能需要做一次 stat 来检测文件的真实模式，或者暂时把 UID 设置为其他的什么东西。

其他文件测试操作符不关心你是谁。任何人都可以用这些操作符来测试"普通"文件：

```
while (<>) {
    chomp;
    next unless -f $_;      #忽略“特殊”文件
    ...
}
```

-T 和 -B 开关按照下面描述的方法运转。检查文件的第一块的内容，查找是否有类似控制字符或者设置了第八位的字符（这样看起来就不象 UTF-8）。如果有超过三分之一的字符看起来比较特殊，它就是二进制文件；否则，就是文本文件。而且，任何在第一块里包含 ASCII NUL（\0 ）的文件都会被认为是二进制文件。如果对文件句柄使用 -T 或 -B，则检测当前输入（标准 I/O 或者“stdio”）缓冲区，而不是文件的第一块。-T 和 -B 对空文件都返回真，或者测试一个文件句柄时读到 EOF（文件结束）时也返回真。因为 Perl 需要读文件才能进行 -T 测试，所以你大概不想在某些特殊文件上用 -T 把系统搞得挂起来，或者是发生其他让你痛苦的事情吧。所以，大多数情况下，你会希望先用 -f 测试，比如：

```
next unless -f $file && -T $file;
```

如果给任何文件测试（操作符）（或者是 stat 或 lstat 操作符）的特殊文件句柄只包含单独一个下划线，则使用前一个文件测试的 stat 结构，这样就省了一次系统调用。（对 -t 无效，而且你还要记住 lstat 和 -l 会在 stat 结构里保存符号连接而不是真实文件的数值。类似地，在一个正常的 stat 的后面的 -l _ 总是会为假。）

下面是几个例子：

```
print "Can do.\n" if -r $a || -w _ || -x _;

stat($filename);
print "Readable\n" if -r _;
print "Writable\n" if -w _;
print "Executable\n" if -x _;
print "Setuid\n" if -u _;
print "Setgid\n" if -g _;

print "Sticky\n" if -k _;
print "Text\n" if -T _;
print "Binary\n" if -B _;
```

-M，-A 和 -C 返回脚本开始运行以来一天（包括分数日子）计的文件年龄。这个时间是保存在特殊变量 $^T （$BASETIME） 里面的。因此，如果文件在脚本启动后做了改变，你就会得到一个负数时间。请注意，大多数时间值（概率为 86400 分之 86399）都是分数，所以如果不用 int 函数就拿它和一个整数进行相等性测试，通常都会失败。比如：

```
next unless -M $file > .5;   # 文件长于 12 小时
&newfile if -M $file < 0;    # 文件比进程新
&mailwarning if int(-A) == 90;   # 文件 ($_) 是 90 十天前访问的
```

要把脚本的开始时间重新设置为当前时间，这样：

```
$^T = time;
```

3.11 关系操作符

Perl 有两类关系操作符。一类操作符操作数字值，另一类操作字串值，在表 3-5 中列出。

表 3-5 关系操作符

数字	字串	含义
>	gt	大于
>=	ge	大于或等于
<	lt	小于
<= </td>	le	小于或等于

这些操作符在真时返回 1 而为假时返回 ""。请注意关系操作符不能结合，这就意味着 $a < $b < $c 是语法错误。

如果没有区域声明，字串的比较基于 ASCII/Unicode 的顺序比较，而且和一些计算机语言不同的是，在比较中，尾部的空白也计入比较中。如果有区域声明，比较顺序以所声明区域的字符集顺序为基础。（以区域字符集为基础的比较机制可能可以也可能不能和目前正在开发的 Unicode 比较机制很好地交互。）

3.12 相等操作符

相等操作符在表 3-6 里面列出，它们和关系操作符很象。

表 3-6 相等操作符

数字	字串	含义
==	eq	等于
=	ne	不等于
<=>	cmp	比较，带符号结果

等于和不等于操作符为真时返回 1，为假时返回""（和关系操作符一样）。<=> 和 cmp 操作符在左操作数小于右操作数时返回 -1，相等时返回 0，而大于时返回 1。尽管相等操作符和关系操作符很象，但是它们的优先级比较低，因此 $a < $b <=> $c <$d 语法上是合法的。

因为很多人看过“星球大战”，<=>操作符也被称为“飞船”操作符。

3.13 位操作符

和 C 类似，Perl 也有位操作符 AND，OR，和 XOR（异或）：&，| 和 ^。在本章开始的时候，你辛辛苦苦地检查表格，发现按位 AND （与）操作符比其他的优先级高，但我们那时候是骗你的，在这里我们会一并讨论一下。

这些操作符对数字值和对字串值的处理不同。（这是少数几个 Perl 关心的区别。）如果两个操作数都是数字（或者被当作数字使用），那么两个操作数都被转换成整数然后在两个整数之间进行位操作。我们保证这些整数是至少 32 位长，不过在某些机器上可以是 64 位长。主要是要知道有一个由机器的体系所施加的限制。

如果两个操作数都是字串（而且自从它们被设置以来还没有当作数字使用过），那么该操作符用两个字串里面来的位做位操作。这种情况下，没有任何字长限制，因为字串本身没有尺寸限制。如果一个字串比另一个长，Perl 就认为短的那个在尾部有足够的 0 以弥补区别。

比如，如果你 AND 两个字串：

"123.45" & "234.56"

你得到另外一个字串：

"020.44"

不过，如果你拿一个字串和一个数字 AND：

"123.45" & 234.56

那字串先转换成数字，得到：

1. 45 & 234.56

然后数字转换成整数：

1. & 234

最后得到值为 106。请注意所有位字串都是真（除非它们结果是字串“0”）。这意味着如果你想看看任意字节是否为非零，你不能这么干：

if ("fred" & "\1\2\3\4") { ... }

你得这么干：

if( )"fred" & "\1\2\3\4") =~ /[^\0]/) { ... }

3.14 C 风格的逻辑（短路）操作符

和 C 类似，Perl 提供 &&（逻辑 AND）和 ||（逻辑 OR）操作符。它们从左向右计算（ && 比 || 的优先级稍稍高一点点），测试语句的真假。这些操作符被认为是短路操作

符，因为它们是通过计算尽可能少的操作数来判断语句的真假。例如，如果一个 && 操作符的左操作数是假，那么它永远不会计算右操作数，因为操作符的结果就是假，不管右操作数的值是什么。

例子	名称	结果
$a && $b	And	如果$a 为假则为$a，否则$b
$a | | $b	Or	如果$a 为真则为$a，否则$b

这样的短路不仅节约时间，而且还常常用于控制计算的流向。比如，一个经常出现的 Perl 程序的俗语是：

```
open(FILE, "somefile") || die "Can't open somefile: $!\n";
```

在这个例子里，Perl 首先计算 open 函数，如果值是真（somefile 被成功打开），die 函数的执行就不必要了，因此忽略。你可以这么读这句文本"打开文件，要不然就去死！"。

&& 和 || 操作符和 C 不同的是，它们不返回 0 或 1，而是返回最后计算的值。如果是 ||，这个特性好就好在你可以从一系列标量数值中选出第一个为真的值。所以，一个移植性相当好的寻找用户的家目录的方法可能是：

```
$home = $ENV{HOME}
  || $ENV{LOGDIR}
  || (getpwuid($<)) [7]
  || die "You're homeless!\n";
```

另一方面，因为左参数总是在标量环境里计算，所以你不能把 || 用于在两个集群之间选择其一进行赋值：

```
@a = @b || @c;        # 这样可不对
@a = scalar(@b) || @c;    # 上面那句实际上是这个意思，@a 里只有 @b 最后的元素
@a = @b ? @b : @c;    # 这个是对的
```

Perl 还提供优先级比较低的 and 和 or 操作符，这样程序的可读性更好而且不会强迫你在列表操作符上使用圆括弧。它们也是短路的。参阅表 1-1 获取完整列表。

3.15 范围操作符

范围操作符 .. 根据环境的不同实际上是两种不同的操作符。

在标量环境里，.. 返回一个布尔值。该操作符是双稳定的，类似一个电子开关，并且它仿真 sed，awk，和各种编辑器的行范围（逗号）操作符。每个 .. 操作符都维护自身的状态。只要它的左操作数为假就一直为假。一旦左操作数为真，该范围操作符就保持真的状态直到右操作数为真，右操作数为真之后该范围操作符再次为假。该操作符在下次计算之前不会变成假。它可以测试右操作数并且在右操作数变真后在同一次计算中变成假（awk 的范围操作符的特性），不过它还是会返回一次真。如果你不想拖到下一次计算中才测试右操作数（也是 sed 的范围操作符的工作方式），只需要用三个点（...）代替两个点（..）。对于 .. 和 ...，当操作符处于假状态后就不再测试右操作数，而当操作符处于真状态后就不再测试左操作数。

返回的值要么是代表假的空字串或者是代表真的一个序列数（从 1 开始）。该序列数每次碰到新范围时重置。在一个范围里的最后序列数后面附加了字串"E0"，这个字串不影响它的数字值，只是给你一些东西让你可以搜索，这样你可以把终点排除在外。你也可以通过等待 1 的序列数的方法把启始点排除在外。如果标量 .. 的某个操作数是数字文本，那么该操作数隐含地与 $.变量对比，$. 里包含你的输入文件的当前行号。比如：

```
if(101 .. 200) {print;}     # 打印第二个一百行

next line if( 1.. /^$/);    # 忽略开头行

s/^/> / if (/^$/ .. eof());    # 引起体
```

在列表环境里， .. 返回一列从左值到右值计数（以一）的数值。这样对书写 (1 .. 10) 循环和数组片段的操作很有帮助：

```
for (101 .. 200) {print;}       # 打印 101102 。。。 199200

@foo = @foo[0 .. $#foo];        # 一个昂贵的空操作

@foo = @foo[ -5 .. -1];           # 最后 5 个元素的片段
```

如果左边的值大于右边的值，则返回一个空列表。（要产生一列反序的列表，参阅 reverse 操作符。）

如果操作数是字串，范围操作符利用早先讨论过的自增处理。（注：如果在所声明的终值不是自增处理中产生的序列中的数，那么该序列将继续增加直到下一个值比声明的终值长为止。）因此你可以说：

```
@alphabet = ('A' .. 'Z');
```

以获得所有英文字母，或者：

```
$hexdigit = (0 .. 9, 'a' .. 'f')[$num & 15];
```

获得一个十六进制位，或者：

```
@z2 = ('01' .. '31'); print $z2[$mday];
```

获得带有前导零的日期。你还可以说：

```
@combos = ('aa' .. 'zz');
```

获取所有两个小写字符的组合。不过，用下面的语句要小心：

```
@bigcombos = ('aaaaaa' .. 'zzzzzz');
```

因为这条语句要消耗很多内存。准确地说，它需要存储 308,915,776 个标量的空间。希望你分配了一个非常大的交换分区。可能你会考虑用循环代替它。

3.16 条件操作符

和 C 里一样，?: 是唯一的一个三目操作符。它通常被成为条件操作符，因为它运转起来非常象一个 if-then-else，而且，因为它是一个表达式而不是一个语句，所以它可以非常容易地嵌入其他表达式和函数调用中。作为三目操作符，它的两个部分分隔了三个表达式：

COND ? THEN : ELSE

如果条件 COND 为真，那么只计算 THEN 表达式，并且其值成为整个表达式的值。否则，只计算 ELSE 表达式的值，并且其值成为整个表达式的值。

不管选择了哪个参数，标量或者列表环境都传播到该参数。（第一个参数总是处于标量环境，因为它是一个条件。）

```
$a = $ok ? $b :$c;    # 得到一个标量

@a = $ok ? @b : @c;    # 得到一个数组

$a = $ok ? @b :@C;    # 得到一个数组元素的计数
```

你会经常看到条件操作符嵌入在一列数值中以格式化 printf，因为没人愿意只是为了在两个相关的值之间切换而复制整条语句。

```
printf "I have $d camel$.\n",
   $n,    $n == 1 ? "" : "s";
```

?: 的优先级比逗号高，但是比你可能用到的其他大多数操作符（比如本例中的 ==）都低，因此通常你用不着用圆括弧括任何东西。不过如果你愿意，你可以用圆括弧让语句更清晰。对于嵌套在其他 THEN 部分的其他条件操作符，我们建议你在它们中间放入断行和缩进，就好象它们是普通 if 语句一样：

```
$leapyear =
   $year % 4 == 0
      ? $year % 100 == 0
         ? $year % 400 == 0
            ? 1
            :0
         :1
   :0;
```

类似地，对于嵌套在 ELSE 部分的更早的条件，你也可以这样处理：

```
$leapyear =
   $year % 4
      ? 0
      : $year %100
         ? 1
         : $year % 400
         ? 0
         :1;
```

不过通常最好把所有 COND 和 THEN 部分垂直排列：

```
$leapyear =
   $year % 4?0:
   $year % 100 ? 1:
```

```
    $year % 400 ? 0:1
```

把问号和冒号对齐可以让你在非常混乱的结构中找到感觉：

```
printf "Yes, I like my %s book!\n",
    $i18n eq "french"   ? "chameau"   :
    $i18n eq "german"   ? "Kamel"     :
    $i18n eq "japanese"   ? "\x{99F1}\x{99DD}"   :
                "camel"
```

如果第二个和第三个参数都是合法的左值（也就是说你可以给他们赋值），而且同时为标量或者列表（否则，Perl 就不知道应该给赋值的右边什么环境），那你就可以给它赋值（注：这样无法保证你的程序有很好的可读性。但是这种格式可以用于创建一些很酷的东西。）：

```
($a_or_b ? $a : $b) = $c;       # 把 $a 或 $b 置为 $c 的值
```

请注意，条件操作符比各种各样赋值操作符绑定得更紧。通常这是你所希望的（比如说上面 $leapyear 赋值），但如果你不用圆括弧的话，你就没法让它反过来。不带圆括弧（在条件操作符）中使用嵌入的赋值会给你带来麻烦，并且这时还不会有分析错误，因为条件操作符左值分析。比如说，你可能写下面这些：

```
$a %2 ? $a += 10 : $a += 2       # 错
```

上面这个会分析为：

```
(($a % 2) ? ($a += 10) : $a) += 2
```

3.16 赋值操作符

Perl 可以识别 C 的赋值操作符，还提供了几个自己的。这里是一些：

```
=     **=     +=     *=     &=     <<=     &&=
              -=     /=     |=     >>=     ||=
              .=     %=     ^=
                     x=
```

每个操作符需要一个目标左值（典型值是一个变量或数组元素）在左边以及一个表达式在右边。对于简单赋值操作符：

```
TARGET = EXPR
```

EXPR 的值被存储到 TARGET 指明的变量或者位置里面。对其他操作符而言， Perl 计算表达式：

```
TARGET OP= EXPR
```

就好象它是这么写的一样：

```
TARGET = TARGET OP EXPR
```

这是有利于简便的规则，不过它在两个方面会误导我们。首先，赋值操作符总是以普通赋值的优先级进行分析的，不管 OP 本身的优先级是什么。第二，TARGET 只计算一次。通常这样做没有什么问题，除非存在副作用，比如一个自增：

```
$var[$a++] += $value;          # $a 增加了一次

$var[$a++] = $var[$a++] + $value;    # $a 增加了两次
```

不象 C 里，赋值操作符生成一个有效的左值。更改一个赋值等效于先赋值然后把变量修改为赋的值。这招对修改一些东西的拷贝很管用，象这样：

```
($tmp = $global) += $constant;
```

等效于：

```
$tmp = $global + $constant;
```

类似：

```
($a += 2) *= 3;
```

等效于：

```
$a += 2;

$a *= 3;
```

本招并非绝招，不过下面是你经常看到的习惯用语：

```
($new = $old) =~ s/foo/bar/g;
```

无论什么情况，赋值的值是该变量的新值。因为赋值操作符从右向左结合，所以这一点可以用于给多个变量赋同一个值，比如：

```
$a = $b = $c =0;
```

把 0 赋予 $c，其结果（还是 0）给 $b，其结果（依然为 0）再给 $a。

列表赋值可能只能在列表环境里用简单赋值操作符，=.，赋值。列表赋值返回该列新值，就象标量赋值一样。在标量环境里，列表赋值返回赋值右边可获得的值的数目，就象我们在第二章，集腋成裘，里谈到的一样。我们可以利用这个特性测试一个失败（或者不再成功）时返回空列表的函数，比如：

```
while (($key, $value) = each %gloss) { ... }


next unless ($dev, $ino, $mode) = stat $file;
```

3.18 逗号操作符

双目“,”是逗号操作符。在标量环境里它先在空环境里计算它的左参数，把那个值抛弃，然后在标量环境里计算它的右参数并且返回之。就象 C 的逗号操作符一样。比如：

```
$a = (1,3);
```

把 3 赋予 $a。不要混淆标量环境用法和列表环境用法。在列表环境里，逗号只是列表参数的分隔符，而且把它的两个参数都插入到列表中。并不抛弃任何数值。

比如，如果你把前面例子改为：

```
@a = (1,3);
```

你是在构造一个两元素的列表，而：

atan2(1,3);

是用两个参数调用函数 atan2。

连字符 => 大多数时候就是逗号操作符的同义词。对那些成对出现的文档参数很有用。它还强制把它紧左边的标识符解释为一个字串。

3.19 列表操作符（右向）

列表操作符的右边是所有列表操作符的参数（用逗号分隔的），所以如果你往右看的话，列表操作符的优先级比逗号低，一旦列表操作符开始啃那些逗号分隔的参数，你能让它停下来的唯一办法就是停止整个表达式的记号（比如分号或语句修改器），或者停止当前子表达式的记号（比如右圆括弧或花括弧），或者是我们下面要讲的低优先级的逻辑操作符。

3.20 逻辑与，或，非和异或

作为 &&，|| 和 ！的低优先级候补，Perl 提供了 and，or，和 not 操作符。这些操作符的性质和它们对应的短路操作符是一样的，尤其是 and 和 or，这样它们不仅可以用于逻辑表达式也可以用于流控制。

因为这些操作符的优先级远远比从 C 借来的那些低，所以你可以很放心地把它们用在一个列表操作符后面而不用加圆括弧：

```
unlink "alpha", "beta", "gamma"
    or gripe(), next LINE;
```

而对 C 风格的操作符你就得这么写：

```
unlink("alpha", "beat", "gamma") || (girpe(), next LINE);
```

不过你不能简单地把所有 || 替换为 or。假设你把：

```
$xyz = $x || $y || $z;
```

改成：

```
$xyz = $x or $y or $z;   # 错
```

这两句是完全不同的！赋值的优先级比 or 高但是比 || 低，所以它总是会给 $xyz 赋值 $x，然后再进行 or。要获得和 || 一样的效果，你就得写：

```
$xyz = ( $x or $y or $z );
```

问题的实质是你仍然必须学习优先级（或使用圆括弧），不管你用的是哪种逻辑操作符。

还有一个逻辑 xor 操作符在 C 或 Perl 里面没有很准确的对应，因为另外一个异或操作符（^）按位操作。xor 操作符不能短路，因为两边都必须计算。$a xor $b 的最好的等效可能是 !$a = !$b。当然你也可以写 !$a ^ !$b，甚至可以是 $a ? !$b : !!$b。要点是 $a 和 $b 必须在布尔环境里计算出真或假，而现有的位操作符在没有帮助的前提下并不提供布尔环境。

3.21 Perl 里没有的 C 操作符

下面是 Perl 里没有的 C 操作符：

单目 & 取址操作符。不过，Perl 的 \ 操作符（用于使用引用）填补了这个生态空白：

$ref_to_var = \$var;

不过 Perl 的引用要远比 C 的指针更安全。
单目 * 解取址操作符。因为 Perl 没有地址，所以它不需要解取址。它有引用，因此 Perl 的变量前缀字符用做截取址操作符，并且还标明类型：$，@，%，和 &。不过， 有趣的是实际上有一个 * 解引用操作符，但因为 * 是表示一个类型团的趣味字符， 所以你无法将其用于同一目的。 (类型) 类型转换操作符。没人喜欢类型转换。
to top

第四章 语句和声明

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，而当作字串时会把它当作""（空字串），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\en"; sleep(1);
}


for (reverse 'BOOM', 1 .. 10) {             # 一样的东西
```

```
    print "$_\en"; sleep(1);
}

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

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

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 foreache 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 操作符立即退出当事循环。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，那它将就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 rddo 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_perference) {
    $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：got LABLE，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。你可能会想用一个运行时的 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，说你调用了一个没有定义的子过程。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，就好象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，因为它们在编译是做处理，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'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：

```
{

    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。

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，字串到数字的非法转换，把未定义的值用做合法字串或数字，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 wranings 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。

第五章 模式匹配

Perl 内置的模式匹配让你能够简便高效地搜索大量的数据。不管你是在一个巨型的商业门户站点上用于扫描每日感兴趣的珍闻报道，还是在一个政府组织里用于精确地描述人口统计（或者人类基因组图），或是在一个教育组织里用于在你的 web 站点上生成一些动态信息，Perl 都是你可选的工具。这里的一部分原因是 Perl 的数据库联接能力，但是更重要的原因是 Perl 的模式匹配能力。如果你把"文本"的含义尽可能地扩展，那么可能你做的工作中有 90% 是在处理文本。这个领域实在就是 Perl 的最初专业，而且一直是 Perl 的目的——实际上，它甚至是 Perl 的名字：Practical Extraction and Report Language （实用抽取和报表语言）。Perl 的模式提供了在堆积成山的数据中扫描数据和抽取有用信息的强大工具。

你可以通过创建一个正则表达式（或者叫 regex）来声明一个模式，然后 Perl 的正则表达式引擎(我们在本章余下的部分称之为"引擎")把这个正则表达式拿过并判断模式是否(以及如何）和你的数据相匹配。因为你的数据可能大部分由文本字串组成，所以你没有理由不用正则表达式搜索和替换任意字节序列，甚至有些你认为是"二进制"的数据也可以用正则处理。对于 Perl 而言，字节只不过碰巧是那些数值小于 256 的自然数而已。（更多相关内容见第十五章，Unicode。）

如果你通过别的途径已经知道正则表达式了，那么我们必须先警告你在 Perl 里的正则表达式是有些不同的。首先，理论上来讲，它们并不是完全"正则"的，这意味着 Perl 里的正则可以处理比计算机科学课程里教的正则表达式更多的事情。第二，因为它们在 Perl 里用得实在是太广泛了，所以在这门语言里，它们有属于自己的特殊的变量，操作符，和引用习惯；这些东西都和语言本身紧密地结合在一起。而不象其他语言那样通过库松散地组合在一起。Perl 的程序员新手常常徒劳地寻找地这样的函数：

```
match( $string, $pattern );

subst( $string, $pattern, $replacement );
```

要知道在 Perl 里，匹配和子串都是非常基本的任务，所以它们只是单字符操作符：m/PATTERN/ 和 s/PATTERN/REPLACEMENT/（缩写为 m// 和 s///）。它们不仅语法简单，而且还象双引号字串那样分析，而不只是象普通操作符那样处理；当然它们的操作还是象操作符的，所以我们才叫它们操作符。你在本章通篇都会看到这些操作符用于匹配字串。如果字串的一部分匹配了模式，我们就说是一次成功的模式匹配。特别是在你用 s/// 的时候，成功匹配的部分将被你在 REPLACEMENT 里声明的内容代替。

本章所有的内容都是有关如何制作和使用模式的。Perl 的正则表达式非常有效，把许多含义包含到了一个很小的表达式里。所以如果你想直接理解一个很长的模式，那很有可能被吓着。不过如果你能把长句分解成短句，并且还知道引擎如何解释这些短句，那你就能理解任何正则表达式。一行正则表达式相当于好几百行 C 或者 JAVA 程序并不罕见。正则表达式可能比一个长程序的单一一行要难理解，但是如果从整体来看，正则表达式通常要比很长的程序要好理解。你只需要提前知道这些事情就可以了。

5.1 正则表达式箴言

在我们开始讲述正则表达式之前，让我们先看看一些模式的样子是什么。正则表达式里的大多数字符只是代表自身。如果你把几个字符排在一行，它们必须按顺序匹配，就象你希望的那样。因此如果你写出一个模式匹配：

```
/Frodo/    （译注：记得电影“魔戒”吗？;））
```

你可以确信除非该字串在什么地方包含子字串"Frodo"，否则该模式不会匹配上。（一个子字串只是字串的一部分。）这样的匹配可以发生在字串里的任何位置，只要这五个字符以上面的顺序在什么地方出现就行。

其他字符并不匹配自身，而是从某种角度来说表现得有些"怪异"。我们把这些字符称做元字符。（大多数元字符都是自己淘气，但是有一些坏得把旁边的字符也带"坏"了。）

下面的就是这些字符：

```
\ | ( ) [ { ^ $ * + ? .
```

实际上元字符非常有用，而且在模式里有特殊的含义；我们会一边讲述，一边告诉你所有的那些含义。不过我们还是要告诉你，你仍然可以在任意时刻使用前置反斜杠的方法来匹配这十二个字符本身。比如，反斜杠本身是一个元字符，因此如果你要匹配一个文本的反斜杠，你要在反斜杠前面放一个反斜杠：\\。

要知道，反斜杠就是那种让其他字符"怪异"的字符。事实是如果你让一个怪异元字符再次怪异，它就会正常——双重否定等于肯定。因此反斜杠一个字符能够让它正确反映文本值，但是这条只对标点符号字符有用；反斜杠（平时正常）的字母数字字符作用相反：它把该文本字符变成某些特殊的东西。不论什么时候你看到下面的双字符序列：

```
\b \D \t \3 \s
```

你就应该知道这些序列是一个元符号，它们匹配某些特殊的东西。比如，\b 匹配一个字边界，而 \t 匹配一个普通水平制表字符。请注意一个水平制表符是一个字符宽，而一个字边界是零字符宽，因为它是两个字符之间的位置。所以我们管 \b 叫一个零宽度断言。当然，\t 和 \b 还是相似的，因为他们都断言某些和字串里的某个特殊位置相关的东西。当你在正则表达式里断言某些东西的时候，你的意思只是说，如果要匹配模式，那些东西必须是真的。

一个正则表达式里的大多数部分都是某种断言，包括那些普通字符，只不过它们是断言它们必须匹配自身。准确来说，它们还断言下一个东西将匹配字串里下一个字符，这也是为什么我们说水平制表符是"单字符宽"。有些断言（比如 \t）当匹配的时候就吞掉字串的一部分，而其他的（比如 \b）不会这样。但是通常我们把"断言"这个词保留给零宽度断言用。为避免混淆，我们把这些东西称做宽度为一个原子的东西。（如果你是一个物理学家，你可以把非零宽的原子当作物质，相比之下零宽断言类似无质量的光子。）

你还会看到有些元字符不是断言；而是结构元素（就好象花括弧和分号定义普通 Perl 代码的结构，但是实际上什么也不干）。这些结构元字符在某种程度上来说是最重要的元字符，因为学习阅读正则表达式关键的第一步就是让你的眼睛学会挑出结构元字符。一旦你学会了

挑出结构元字符，阅读正则表达式就是如坐春风（注：当然，有时候风力强劲，但绝对不会把你刮跑。）

有一个结构元字符是竖直条，表示侯选项：

```
/Frodo|Pippin|Merry|Sam/  （译注：电影“魔戒”里 Shire 的四个小矮人）
```

这意味着这些字串的任何一个都会触发匹配；这个内容在本章稍后的“侯选项”节描述。并且我们还会在那节后面的“捕获和集群”节里告诉你如何使用圆括弧，把你的模式各个部分括起来分组：

```
/(Frodo|Drogo|Bilbo) Baggins/  （译注：Bilbo Baggins 是 Frodo 的叔叔，老一辈魔戒承载者。）
```

或者甚至：

```
/(Frod|Drog|Bilb)o Baggins/
```

你看到的其他的东西是我们称之为量词的东西，它表示在一行里面前面匹配的东西应该匹配几个。量词是这些东西：

```
* + ? *? {3} {2,5}
```

不过你永远不会看到它们这样独立地存在。量词只有附着在原子后面才有意义 —— 也就是说，断言那些有宽度的（注：量词有点象第四章，语句和声明，里的语句修饰词，也是只能附着在单个语句后面。给一个零宽度的断言附着量词就象试图给一个声明语句附着一个 while 修饰词一样 —— 两种做法都和你跟药剂师要一斤光子一样无聊。药剂师只卖原子什么的。）量词只附着在前一个原子上，从我们人类的眼光来看，这通常量化为只有一个字符。如果你想匹配一行里的三个“bar”的副本，你得用圆括弧把“bar”的三个独立的字符组合成一个“分子”，象这样：

```
/(bar){3}/
```

这样将和“barbarbar”匹配。如果你用的是 /bar{3}/，那么匹配的是“barrr“ —— 这东西表明你是苏格兰人（译注：爱尔兰人说英文的时候尾音比较长），而不是 barbarbar 人。（话又说回来，也可能不是。我们有些很喜欢的元字符就是爱尔兰人。）有关量词的更多东西，参阅后面的“量词”。

你已经看到了一些继承了正则表达式的野兽，现在一定迫不及待地想驯服它们。不过，在我们开始认真地讨论正则表达式之前，我们需要先向回追溯一些然后再谈谈使用正则表达式的模式匹配操作符。（并且如果你在学习过程中碰巧多遇到几只“野兽”，那么不妨给我们的学习向导留一条不错的技巧。）

5.2 模式匹配操作符

从动物学角度来说，Perl 的模式匹配操作符函数是某种用来关正则表达式的笼子。我们是有意这么设计的；如果我们任由正则怪兽在语言里四处乱逛，Perl 就完全是一个原始丛林了。当然，世界需要丛林——它们是生物种类多样性的引擎，但是，丛林毕竟应该放在它们应该在的位置。一样，尽管也是组合多样化的引擎，正则表达式也应该放在它们应该在的模式匹配操作符里面。那里是另外一个丛林。

因为正则表达式还不够强大，m// 和 s/// 操作符还提供了（同样也是限制）双引号代换的能力。因为模式是按照类似双引号字串那样分析的，所以所有的双引号代换都有效，包括变量代换（除非你用单引号做分隔符）和用反斜杠逃逸标识的特殊字符。（参阅本章后面的"特殊字符"。）在字串被解释成正则表达式之前首先应用这些代换。（也是 Perl 语言里极少数的几个地方之一，在这些地方一个字串要经过多于一次处理。）第一次处理是不那么正常的双引号代换，不正常是因为它知道它应该转换什么和它应该给正则表达式分析器传递什么。因此，任何后面紧跟竖直条，闭圆括弧或者字符串结尾的 $ 都不会被当作变量代换，而是当作典型的正则表达式的行尾断言。所以，如果你说：

```
$foo = "bar";

/$foo$/;
```

双引号代换过程是知道那两个 $ 符作用是不同的。它先做 $foo 的变量代换，然后把剩下的交给正则表达式分析器：

```
/bar$/;
```

这种两回合分析的另一个结果是普通的 Perl 记号分析器首先查找正则表达式的结尾，就好象它在查找一个普通字串的结尾分隔符一样。只有在它找到字串的结尾后（并且完成任意变量代换），该模式才被当作正则表达式对待。这意味着你无法在一个正则构造里面"隐藏"模式的结尾分隔符（比如一个字符表或者一个正则注释，我们还没有提到这些东西）。Perl 总是会在任何地方识别该分隔符并且在该处结束该模式。

你还应该知道在模式里面代换变量会降低模式匹配的速度，因为它会觉得需要检查变量是否曾经变化过，如果变化过，那么它必须重新编译模式（这样更会降低速度）。参阅本章后面的"变量代换"。

tr/// 转换操作符不做变量代换；它甚至连正则表达式都不用！（实际上，它可能并不属于本章，但我们实在想不出更好的地方放它。）不过，它在一个方面还是和 m// 和 s/// 一样的：它用 =~ 和 !~ 操作符与变量绑定。

第三章，单目和双目操作符，里描述的 =~ 和 !~ 操作符把它们左边的标量表达式和在右边的三个引起类操作符之一绑定在一起： m// 用于匹配一个模式，s/// 用于将某个符合模式的子字串代换为某个字串，而 tr/// （或者其同义词，y///）用于将一套字符转换成另一套。（如果把斜杠用做分隔符，你可以把 m// 写成 //，不用写 m。）如果 =~ 或 !~ 的右手边不是上面三个，它仍然当作是 m// 匹配操作，不过此时你已经没有地方放跟在后面的修饰词了（参阅后面的"模式修饰词"），并且你必须操作自己的引号包围：

```
print "matches"  if  $somestring =~ $somepattern;
```

不过，我们实在没道理不明确地说出来：

```
print "matches" if  $somestring =~ m/$somepattern/;
```

当用于匹配操作时，有时候 =~ 和 !~ 分别读做"匹配"和"不匹配"（因为"包含"和"不包含"会让人觉得有点模糊）。

除了在 m// 和 s/// 操作符里使用外，在 Perl 的另外两个地方也使用正则表达式。split 函数的第一个参数是一个特殊的匹配操作符，它声明的是当把字串分解成多个子字串后不返回什么东西。参阅第二十九章，函数，里的关于 split 的描述和例子。qr// （"引起正则表达"）操作符同样也通过正则表达式声明一个模式，但是它不是为了匹配匹配任何东西（和 m// 做的不一样）。相反，编译好的正则表达的形式返回后用于将来的处理。参阅"变量代换"获取更多信息。

你用 m//，s/// 或者 tr/// 操作符和 =~ （从语言学上来说，它不是真正的操作符，只是某种形式的标题符）绑定操作符把某一字串和这些操作符之一绑定起来。下面是一些例子：

```
$haystack =~ m/meedle/          # 匹配一个简单模式

$haystack =~ /needle/           # 一样的东西


$italiano =~ s/butter/olive oil/    # 一个健康的替换


$rotate13 =~ tr/a-zA-Z/n-za-mN-ZA-M/    # 简单的加密
```

如果没有绑定操作符，隐含地用 $_ 做"标题"：

```
/new life/ and              # 搜索 $_ 和（如果找到）

/new civilizations/       # 再次更宽范围地搜索 $_
```

```
s/sugar/aspartame/          # 把一个替换物替换到 $_ 里


tr/ATCG/TAGC              # 修改在 $_ 里表示的 DNA
```

因为 s/// 和 tr/// 修改它们所处理的标量，因此你只能把它们用于有效的左值：

```
"onshore" =~ s/on/off/;          # 错；编译时错误
```

不过，m// 可以应用于任何标量表达式的结果：

```
if (( lc $magic_hat->fetch_contents->as_string) =! /rabbit/) {
   print  "Nyaa, what's up doc?\n";
}
else {
   print "That trick never works!\n";
}
```

但是，在这里你得更小心一些，因为 =~ 和 !~ 的优先级相当高——在前一个例子里，左边的项的圆括弧是必须的（注：如果没有圆括弧，低优先级的 lc 将会应用于整个模式匹配而不只是对 magic hat 对象的方法调用。）。!~ 绑定操作符作用和 =~ 类似，只是把逻辑结果取反：

```
if ($song !~ /words/) {
   print qq/"$song" appears to be a song without words. \n/;
}
```

因为 m//，s///，和 tr/// 都是引号包围操作符，所以你可以选择自己的分隔符。这时其运行方式和引起操作符 q//，qq//，qr//，和 qw//一样（参阅第二章，集腋成裘，中的"选择自己的引号"）。

```
$path =~ s#/tmp#/var/tmp/scratch#;


if ($dir =~ m[/bin]) {
```

```
    print "No binary directories please.\n";
}
```

当你把成对的分隔符和 s/// 或者 tr/// 用在一起的时候，如果第一部分是四种客户化的括弧对之一（尖括弧，圆括弧，方括弧或者花括弧），那么你可以为第二部分选用不同于第一部分的分隔符：

```
s(egg)<larva>;
s{larva}{pupa};
s[pupa]/imago/;
```

也可以在实际使用的分隔符前面加空白字符：

```
s (egg) <larva>;
s {larva}   {pupa};
s [pupa]   /imago/;
```

每次成功匹配了一个模式（包括替换中的模式），操作符都会把变量 $`，$&，和 $' 分别设置为匹配内容左边内容，匹配的内容和匹配内容的右边的文本。这个功能对于把字串分解为组件很有用：

```
"hot cross buns" =~ /cross/;
print "Matched: <$`> $& <$'>\n";    # Matched: <hot > cross < buns>
print "Left:    <$`>\n";           # Left:    <hot >
print "Match:   <$&>\n";           # Match:   <cross>
print "Right:   <$'>\n";           # Right:   < buns>
```

为了有更好的颗粒度和提高效率，你可以用圆括弧捕捉你特别想分离出来的部分。每对圆括弧捕捉与圆括弧内的模式相匹配的子模式。圆括弧由左圆括弧的位置从左到右依次排序；对应那些子模式的子字串在匹配之后可以通过顺序的变量 $1，$2，$3 等等获得：

```
$_ = "Bilbo Baggins's birthday is September 22";
/(.*)'s birthday is (.*)/;
print "Person: $1\n";
```

```
print "Date: $2\n";
```

$`， $&， $' 和排序的变量都是全局变量，它们隐含地局部化为属于此闭合的动态范围。它们的存在直到下一次成功的匹配或者当前范围的结尾，以先到者为准。我们稍后在其它课题里有关于这方面内容里更多介绍。

一旦 Perl 认为你的程序的任意部分需要 $`， $&， 或 $'，它就会为每次模式匹配提供这些东西。这样做会微微减慢你的程序的速度。Perl 同样还利用类似的机制生成 $1，$2 等等，因此你也会为每个包含捕捉圆括弧的模式付出一些代价。（参阅"集群"获取在保留分组的特征的同时避免捕获的开销的方法。）但如果你从不使用 $`，$& 或者 $'，那么不带捕获圆括弧的模式不会有性能损失。因此，如果可能地话，通常你应该避免使用 $`，$& 和 $'，尤其是在库模块里。但是如果你必须至少使用它们一次（而且有些算法的确因此获益非浅），那么你就随便用它们吧，因为你已经为之付出代价了。在最近的 Perl 版本里，$& 比另外两个开销少。

5.2.1 模式修饰词

我们稍后将逐个讨论模式匹配操作符，但首先我们先谈谈另一个这些模式操作符都有的共性：修饰词。

你可以在一个 m//，s///，qr//，或者 tr/// 操作符的最后一个分隔符后面，以任意顺序放一个或多个单字母修饰词。为了保持清晰，修饰词通常写成"/o 修饰词"并且读做"斜杠 o 修饰词"），即使最后的分隔符可能不是一个斜杠也这么叫。（有时候人们把"修饰词"叫做"标志"或者"选项"也可以。）

有些修饰词改变单个操作符的特性，因此我们将在后面仔细讨论它们。其他的修改正则表达式的解释方式，所以我们在这里讨论它们。m//，s/// 和 qr// 操作符（tr/// 操作符并不接受正则表达式，所以这些修饰词并不适用。）的最后一个分隔符后面都接受下列修饰词：

修饰词	含义
/i	忽略字母的大小写（大小写无关）
/s	令 . 匹配换行符并且忽略不建议使用的 $* 变量
/m	令 ^ 和 $ 匹配下一个嵌入的 \n。
/x	忽略（大多数）空白并且允许模式中的注释
/o	只编译模式一次

/i 修饰词是说同时匹配大写或者小写（以及在 Unicode 里的标题）。也是为什么/perl/i 将匹配字串 "PROPERLY" 或 "perlaceous"（几乎是完全不同的东西）。use locale 用法可能也会对被当作相同的东西有影响。（这可能对包含 Unicode 的字串有负面影响。）

/s 和 /m 修饰词并不涉及任何古怪的东西。它们只是影响 Perl 对待那些包含换行符的匹配的态度。不过它们和你的字串是否包含换行符无关；它们关心的是 Perl 是否应该假设你的字串包含单个行（/s）还是多个行（/m），因为有些元字符根据你是否需要让它们工作于面向行的模式而有不同的行为。

通常，元字符 "." 匹配除了换行符以外的任何单个字符，因为它的传统含义是匹配一行内的某个字符。不过，带有 /s 时，"." 元字符也可以匹配一个换行符，因为你已经告诉 Perl 忽略该字串可能包含多个换行符的情况。（/s 修饰词同样还令 Perl 忽略我们已经不鼓励使用的 $* 变量，我们也希望你也忽略。）另一方面，/m 修饰词还修改元字符 ^ 和 $ 的解释 —— 通过令它们匹配字串里的换行符后面的东西，而不仅仅是字串的结尾。参阅本章的"位置"节的例子。

/o 操作符控制模式的重新编译。除非你选用的分隔符是单引号（m'PATTERN'，s'PATTERN'REPLACEMENT'，或者 qr'PATTERN'），否则每次计算模式操作符的时候，任何模式里的变量都会被代换（并且可能会导致模式的重新编译）。如果你希望这样的模式被且只被编译一次；那么就该使用 /o 修饰词。这么做可以避免开销巨大的运行时重新编译；这么做非常有用，尤其是你在转换的值在执行中不会改变的情况下。不过，/o 实际上是让你做出了不会改变模式中的变量的承诺。如果你改变了这些变量，Perl 设置都不会注意到。为了更好地控制重编译，你可以使用 qr// 正则表达式引起操作符。详情请参阅本章后面的"变量代换"节。

/x 是表达修饰词：它允许你利用空白和解释性注释扩展你的模式的易读性，你甚至还可以把模式扩展得超过一行的范围。

也就是说，/x 修改空白字符（还有 # 字符）的含义：它们不再是普通字符那样的自匹配字符，而是转换成元字符，这些元字符的特征类似空白（和注释字符）。因此，/x 允许（在模式里面）将空白，水平制表符和换行符用于格式化，就象普通 Perl 代码一样。它还允许用通常在模式里没有特殊含义的 # 字符引入延伸到当前模式行行尾的注释。（注：请注意不要在注释里包含模式分隔符——因为"先找结尾"的规则，Perl 没办法知道你在该点上并不想结束。）如果你想匹配一个真正的空白字符（或者 # 字符），那你就要把它们放到字符表里，或者用反斜杠逃逸，或者用八进制或者十六进制逃逸的编码。（但是空白通常用一个 \s* 或 \s+ 序列匹配，因此实际中这种情况出现得并不多。）

总结而言，这些特性朝着把传统的正则表达式变成更可读的语言迈进了一大步。从"回字有四种写法"精神出发，现在写一个正则表达式的方法是不止一种了。实际上，我们有不止两种的方法：（译注：TMTOWTDI："There's More Then One Way To Do It", "做事的方法不止一种". Perl 文化口号，见本书尾部的词汇表．）

```
m/\w+:(\s+\w+)\s*\d+/;        # 一个词，冒号，空白，词，空白，数字。
```

```
m/\w+: (\s+ \w+) \s* \d+/x;   # 一个词，冒号，空白，词，空白，数字。


m{
    \w+:          # 匹配一个词和一个冒号。
    (          # 分组开始。
       \s+       # 匹配一个或多个空白。
       \w+       # 匹配另外一个词。
    )          # 分组结束。
    \s*          # 匹配零或更多空白。
    \d+          # 匹配一些数字
}x;
```

我们会在本章稍后描述这些元符号。(本节本来是讲模式修饰词的,但是我们却因为对 /x 过于兴奋而超出了我们的控制。) 下面是一个正则表达式，它找出一个段落里面的重复的词，我们从 Perl Cookbook 里直接把这个例子偷了出来。它使用 /x 和 /i 修饰词，以及后面描述的 /g 修饰词。

```
# 找出段落里面的重复的单词，可能会跨越行界限。
# 将 /x 用于空白和注释，/i 以匹配在"Is is this ok?"里的两个`is'
# 用 /g 找出所有重复。


$/ = "";       # "paragrep" 模式
while( <> ) {
   while ( m{
         \b          # 从字边界开始
         (\w\S+)          # 找出一个字块
         (
```

```
            \s+       # 由一些空白分隔
            \l        # 然后再次分块
        )+            # 重复动作
        \b            # 直到另外一个字边界
    }xig
  )
  {
    print "dup word '$1' at paragraph $.\n";
  }
}
```

当对本章运行这个程序时，它的输出象下面这样：

dup word 'that' at paragraph 100 （译注：只对英文原版有效 :)）

看到这些，我们就知道这个重复是我们有意做的。

5.2.2 m// 操作符（匹配）

```
EXPR =~ m/PATTERN/cgimosx
EXPR =~ /PATTERN/cgimosx
EXPR =~ ?PATERN?cgimosx
m/PATTERN/cgimosx
/PATTERN/cgimosx
?PATTERN?cgimosx
```

m// 操作符搜索标量 EXPR 里面的字串，查找 PATTERN。如果使用 / 或 ? 做分隔符，那么开头的 m 是可选的。? 和 ' 做分隔符时都有特殊含义：前者表示只匹配一次；后者禁止进行变量代换和六种转换逃逸（\U 等，后面描述）。

如果 PATTERN 计算出的结果是空字串，则要么是你用 // 把它声明成空字串或者是因为一个代换过来的变量就是空字串，这时就用没有隐藏在内层块（或者一个 split，grep，或者 map）里的最后执行成功的正则表达式替代。

在标量环境里，该操作符在成功时返回真（1），失败时返回假（""）。这种形式常见于布尔环境：

```
if($shire =~ m/Baggins/) { ...}   # 在 $shire 里找 Baggins，译注：shire 知道哪里么？;)

if($shire =~ /Baggins/)   { ...}   # 在 $shire 里找 Baggins


if(m#Baggins#)      {...}   # 在 $_ 里找

if( /Baggins/ )      {...}    # 在 $_ 里找
```

在列表环境里使用，m// 返回一个子字串的列表，这些子字串匹配模式里的捕获圆括弧（也就是 $1，$2，$3 等等），这些捕获圆括弧将在稍后的"捕获和集群"里描述。当列表返回的时候，这些序列数变量仍然是平滑的。如果在列表环境里匹配失败，则返回一个空列表。如果在列表环境中匹配成功，但是没有使用捕获圆括弧（也没有 /g），则返回则返回一列 (1)。因此它在失败时返回一列空列表，所以这种形式的 m// 仍然能用于布尔环境，但是仅限于通过列表赋值间接参与的情况：

```
if( ($key, $values) = /(\w+): (.*)/) { ... }
```

用于 m//（不管是什么形式）的合法修饰词见表 5-1。

表 5-1。 m// 修饰词

修饰词	含义
/i	或略字母大小写
/m	令 ^ 和 $ 匹配随后嵌入的 \n。
/s	令 . 匹配换行符并且忽略废弃了的 $*。
/x	或略（大多数）空白并且允许在模式里的注释
/o	只编译模式一次
/g	全局地查找所有匹配
/cg	在 /g 匹配失败后允许继续查找

头五个用于正则表达式的修饰词我们前面描述过了。后面两个修改匹配操作本身的特性。/g 修饰词声明一个全局匹配——也就是说，在该字串里匹配尽可能多的次数。它的具体特性取

决于环境。在列表环境里，m//g 返回所有找到的东西的列表。下面的语句找出所有我们提到的 "perl"， "Perl"，"PERL" 的地方：

```
if( @perls = $paragrapth =~ /perl/gi) {
   printf "Perl mentioned %d times.\n", scalar @perls;
}
```

如果在 /g 模式里没有捕获圆括弧，那么返回完整的匹配。如果有捕获圆括弧，那么返回捕获到的字串。想象一下这样的字串：

```
$string = "password=xyzzy verbose=9 score=0";
```

并且假设你想用这个字串初始化下面这样的散列：

```
%hash = (password => "xyzzy", verbose => 9, socre => 0);
```

当然，你有字串但还没有列表。要获取对应的列表，你可以在列表环境里用 m//g 操作符，从字串里捕获所有的键/值对：

```
%hash = $string =~ /(\w+)=(\w+)/g;
```

(\w+) 序列捕获一个字母数字单词。参阅"捕获和集群"节。

在标量环境里使用时，/g 修饰词表明一次渐进地匹配，它令 Perl 从上一次匹配停下来的位置开始一次对同一个变量的新的匹配。\G 断言表示字符串中的那个位置；\G 的描述请参阅本章后面的"位置"一节。如果除了用 /g，你还用了 /c（表示"连续"）修饰词，那么当 /g 运行结束后，失败的匹配不会重置位置指针。

如果分隔符是 ?，就象 ?PATTERN?，那么运行起来和 /PATTERN/ 搜索一样，区别是它在两次 reset 操作符调用之间只匹配一次。如果你只想匹配程序运行中模式出现的第一次，而不是所有的出现，那么这是一个很方便的优化方法。你每次调用此操作符时都会运行搜索，直到它最终匹配了什么东西，然后它就关闭自身，在你明确地用 reset 把它重置之前它一直返回假。Perl 替你跟踪这个匹配状态。

当一个普通模式匹配想找出最后一个匹配而不是第一个，那么 ?? 操作符很好用：

```
open DICT, "/usr/dict/words" or die "Can't open words: $!\n";
while (<DICT>) {
   $first = $1 if ??(^neur.*)?;
```

```
    $last = $1 if /(^neur.*)/;
}
print $first, "\n";       # 打印"neurad"
print $last, "\n";        # 打印 "neurypnology"
```

在调用 reset 操作符时，reset 只重置那些编译进同一个包的 ?? 记录。你说 m?? 的时候等效于说 ??。

5.2.3 s/// 操作符（替换）

```
LVALUE =~ s/PATTERN/REPLACEMENT/egimosx

s/PATTERN/REPLACEMENT/egimosx
```

这个操作符在字串里搜索 PATTERN，如果找到，则用 REPLACEMENT 文本替换匹配的子字符串。（修饰词在本节稍后描述。）

```
$lotr = $hobbit;       # 只是拷贝 Hobbit  译注：影片魔戒里，Hobbit 人住在 Shire :)

$lotr =~ s/Bilbo/Frodo/g;    # 然后用最简单的方法写结局，译注：Frodo 代替 Bilbo 成了魔戒的看护人，又是魔戒
```

一个 s/// 操作符的返回值(在标量和列表环境里都差不多)是它成功的次数(如果与 /g 修饰词一起使用，返回值可能大于一）。如果失败，因为它替换了零次，所以它返回假（""），它等效于数字 0。

```
if( $lotr =~ s/Bilbo/Frodo/) { print "Successfully wrote sequel." }

$change_count = $lotr =~ s/Bilbo/Frodo/g;
```

替换部分被当作双引号包围的字串看待。你可以在替换字串里使用我们前面描述过的任何动态范围的模式变量（$`，$&，$'，$1，$2，等等），以及任何其他你准备使用的双引号包围的小发明。比如下面是一个小例子，用于找出所有字串 "revision"，"version"，或者 "release"，并且用对应的大写字串替换，我们可以用 \u 逃逸处理替换的目标部分：

```
s/revision|version|release/\u$&/g;    # | 用于表示模式中的“或”
```

所有的标量变量都在双引号包围的环境中扩展开，而不仅仅是这些特殊的变量。假设你有一个散列 %Names，把版本号映射为内部的项目名；比如，$Name{"3.0"} 可能是名为 "Isengard" 的代码名。你可以用 s/// 找出版本号并且用它们对应的项目名替换掉：

```
s/version ([0-9.]+)/the $Names{$1} release/g;
```

在在替换字串里，$1 返回第一对（也是唯一的一对）捕获圆括弧。（愿意的话你还可以在模式里用 \1，但是这个用法在替换中已经废弃了，在一个普通的双引号包围的字串里，\1 的意思是 Control-A。）

如果 PATTERN 是一个空字串，则使用上一次成功执行的正则表达式取代。PATTERN 和 RELPACEMENT 都需要经受变量代换，不过每次计算 s/// 操作符的时候都进行代换，而 REPLACEMENT 只是在有匹配的时候才做变量代换。（如果你使用了 /g 修饰词，那么 PATTERN 在一次计算中可能匹配多次。）

和前面一样，表 5-2 中的头五个修饰词修改正则表达式的性质；他们与 m// 和 qr// 中的一样。后面两个修改替换操作符本身。

表 5-2 s/// 修饰词

修饰词	含义
/i	或略字母大小写
/m	令 ^ 和 $ 匹配随后嵌入的 \n。
/s	令 . 匹配换行符并且忽略废弃了的 $*。
/x	或略（大多数）空白并且允许在模式里的注释
/o	只编译模式一次
/g	全局地查找所有匹配
/e	把右边当作一个表达式计算

/g 修饰词用于 s/// 的时候就会把每个匹配 PATTERN 的东西用 REPLACEMENT 值替换，而不仅仅是所找到的第一个。一个 s///g 操作符的作用象一次全局的搜索和替换，令所有修改同时发生，很象 m//g，只不过 m//g 不改变任何东西。（而且 s///g 也和标量 m//g 不一样，它不是递增匹配。）

/e 修饰词把 REPLACEMENT 当作一个 Perl 代码块，而不仅仅是一个替换的字串。执行这段代码后得出的结果当作替换字串使用。比如，s/(0-9)+)/sprintf("%#x", $1)/ge 将把所有数字转换成十六进制，比如，把 2581 变成 0xb23。或者，假设在我们前一个例子里，你不知道是否所有版本都有名称，因此，你希望把这些没有名称的保留不动。可以利用稍微有点创造力的 /x 格式，你可以说：

```
s{
   version
   \s+
   (
      [0-9.]+
   )
}{
   $Names{$1}
      ? "the $Names[$1} release"
      : $&
}xge;
```

你的 s///e 的右边（或者本例中的下半部分）在编译时与你的程序的其他部分一起做语法检查和编译。在编译过程中，任何语法错误都会被检测到，而运行时例外则被忽略。在第一个 /e 后面每多一个 e（象 /ee，/eee 等等）都等效于对生成的代码调用 eval STRING，每个 /e 相当于一次调用。这么做等于计算了代码表达式的结果并且把例外俘获在特殊变量 $@ 里。参阅本章后面的"编程化模式"获取更多信息。

5.2.3.1 顺便修改一下字串

有时候你想要一个新的，修改过的字串，而不是在旧字串上一阵乱改，新字串以旧字串为基础。你不用写：

```
$lotr = $hobbit;
$lotr =~ s/Bilbo/Frodo/g;
```

你可以把这些组合成一个语句。因为优先级关系，必须在赋值周围使用圆括弧，因为它们大多和使用了 =~ 的表达式结合在一起。

```
($lotr = $hobbit ) =~ s/Bilbo/Frodo/g;
```

如果没有赋值语句周围的圆括弧，你只修改了 $hobbit 并且把替换的个数存储在 $lotr 里，那样会得到很傻的结局。

你不能对数组直接使用 s/// 操作符。这时，你需要一个循环。幸运的是，for/foreach 的别名特性加上它把 $_ 当作缺省循环变量，这样就产生了 Perl 标准的用于搜索和替换一个数组里每个元素的俗语：

```
for (@chapters) { s/Bilbo/Frodo/g }       # 一章一章的替换

s/bilbo/Frodo/g for @chapters;          # 一样的东西
```

就象一个简单的标量变量一样，如果你想把初始的值保留在其他地方，你也可以把替换和赋值结合在一起：

```
@oldhues = ('bluebird', 'bluegrass', 'bluefish', 'the blues');

for (@newhues = @oldhues) { s/blue/red/}

print "@newhues\n";       # 打印：redbird redgrass redfish the reds
```

对同一个变量执行重复替换的最经典的方法是用一个单程循环。比如，下面是规范变量里的空白的方法：

```
for ($string) {

    s/^\s+//;    # 丢弃开头的空白

    s/\s+$//;    # 丢弃结尾的空白

    s/\s+/ /g;    # 压缩内部的空白

}
```

这个方法正好和下面的是一样的：

```
$string = join(" ", split " ", $string);
```

你还可以把这样的循环和赋值用在一起，就象我们在数组的例子里做的那样：

```
for( $newshow = $oldshow ) {

    s/Fred/Homer/g;

    s/Wilma/Marge/g;

    s/Pebbles/Lisa/g;

    s/Dino/Bart/g;
```

```
}
```

5.2.3.2 当全局替换不够“全局”地时候

有时候，你用 /g 不能实现全部修改的发生，这时要么是因为替换是从右向左发生的，要么是因为你要求 $` 的长度在不同的匹配之间改变。通常你可以通过反复调用 s/// 做你想做的事情。不过，通常你希望当 s/// 失败的时候循环停下来，因此你必须把它放进条件里，这样又让循环的主体无所事事。因此我们只写一个 1，这也是一件无聊的事情，不过有时候无聊比没希望好。下面是一些例子，它们又用了一些正则表达式怪兽：

```
# 把逗号放在一个整数的合理的位置

1 while s/(\d)(\d\d\d)(?!\d)/$1,$2/;


# 把水平制表符扩展为八列空间

1 while s/\t+/' ' x (length($&)*8 - length($`)%8)/e;


# 删除 (嵌套(甚至深层嵌套(象这样)))的括弧

1 while s/\([^()]*\)//g;


# 删除重复的单词（以及三重的（和四重的。。。））

1 while s/\b(\w+) \1\b/$1/gi;
```

最后一个需要一个循环是因为如果没有循环，它会把：

```
Paris in THE THE THE THE spring.
```

转换成：

```
Paris in THE THE spring。
```

这看起来会让那些懂点法文的人觉得巴黎位于一个喷冰茶的喷泉中间，因为"the"（法文）是法文“tea”的单词。当然，巴黎人从来不会上当。

5.2.4 tr/// 操作符（转换）

```
LVALUE =~ tr/SEARCHLIST/REPLACEMENTLIST/cds

tr/SEARCHLIST/REPLACEMENTLIST/cds
```

对于 sed 的爱好者而言，y/// 就是 tr/// 的同义词。这就是为什么你不能调用名为 y 的函数，同样也不能调用名为 q 或 m 的函数。在所有的其他方面，y/// 都等效于 tr///，并且我们不会再提及它。

把这个操作符放进关于模式匹配的章节看起来其实有点不合适，因为它不使用模式。这个操作符逐字符地扫描一个字串，然后把每个在 SEARCHLIST （不是正则表达式）里出现的字符替换成对应的来自 REPLACEMENTLIST（也不是替换字串）的字符。但是它看上去象 m// 和 s///，你甚至还可以和它一起使用 =~ 和 !~ 绑定操作符，因此我们在这里描述它。（qr// 和 split 都是模式匹配操作符，但是它们不能和绑定操作符一起使用，因此因此在我们本书的别处描述它们。自己找找看。）

转换返回替换或者删除了的字符个数。如果没有通过 =~ 或者 !~ 操作符声明的字串，那么使用 $_ 字串。SEARCHLIST 和 REPLACEMENTLIST 可以用一个划线定义一个顺序字符的范围：

```
$message =~ tr/A-Za-z/N-ZA-Mn-za-m?;    # 旋转 13 加密
```

请注意想 A-Z 这样的范围假设你用的是线性字符集，比如 ASCII。但是不同的字符集的字符排列顺序是不一样的。一个合理的原则是，只使用起始点都是相同大小写的字母序列，如 (a-e,A-E)，或者数字 (0-4)。任何其他的范围都有问题。如果觉得有问题，他们写成你用的整个字符集：ABCDE。

SEARCHLIST 和 REPLACEMENTLIST 都不会象双引号字串那样进行变量代换；不过，你可以使用那些映射为特殊字符的反斜杠序列，比如 \n 或 \015。

表 5-3 是可用于 tr/// 操作符的修饰词。它们和用于 m//，s///，或 qr// 上的完全不同，即使有些看起来有点象。

表 5-3。tr/// 修饰词

修饰词	含义
/c	与 SEARCHLIST 为补
/d	删除找到的但是没有替换的字符
/s	消除重复的字符。

如果声明了 /c 修饰词，那么 SEARCHLIST 里的字符会被求补；也就是说，实际的搜索列表包含所有不在 SEARCHLIST 里的字符。如果是 Unicode，这样可能会代表许多字符，不过因为它们是逻辑存储的，而不是物理存储，所以你不用害怕会用光内存。

/d 修饰词把 tr/// 转换成所谓的"过滤吸收"操作符：任何由 SEARCHLIST 声明的但是没有在 RELPACEMENTLIST 里给出替换的字符将被删除。（这样比一些 tr(1) 程序的性质显得更加灵活，那些程序删除它们在 SERACHLIST 里找到的任何东西。）

如果声明了 /s 修饰词，被转换成相同字符的顺序字符将被压缩成单个字符。

如果使用了 /d 修饰词，那么 RELPACEMENTLIST 总是严格地解释成声明的样子。否则，如果 REPLACEMENTLIST 比 SEARCHLIST 短，则复制 REPLACEMENTLIST 的最后一个字符，直到足够长为止。如果 RELPACEMENTLIST 为空，则复制 SEARCHLIST，这一点虽然奇怪，但很有用，尤其是当你只是想计算字符数，而不是改变它们的时候。也有利于用 /s 压缩字符。

```
tr/aeiou/!/;          # 把所有元音字母转换成!

tr{/\\\r\n\b\f. }{_};      # 把怪字符转成下划线


tr/A-Z/a-z/ for @ARGV;      # 把字符规则化为小写 ASCII


$count = ($para =~ tr/\n//);   # 计算$para 里的换行符

$count = tr/0-9//;       # 计算$_里的位


$word =~ tr/a-zA-Z//s;      # bookkeeper -> bokeper


tr/@$%*//d;          # 删除这里几个字符

tr#A-Za-z0-9+/##cd;       # 删除非 base64 字符


# 顺便修改

($HOST = $host) =~ tr/a-z/A-Z/;
```

```
$pathname =~ tr/a-zA-Z/_/cs;    # 把非 ASCII 字母换成下划线


tr [\200-\377]
   {\000-\177];          # 剥除第八位，字节操作
```

如果在 SEARCHLIST 里同一个字符出现的次数多于一次，那么只有第一个有效。因此：

```
tr/AAA/XYZ/
```

将只会把（$_ 里的）任何单个字符 A 转换成 X。

尽管变量不会代换进入 tr///，但是你还是可以用 eval EXPR 实现同样效果：

```
$count = eval "tr/$oldlist/$newlist/";
die if $@;    # 传播非法 eval 内容的例外
```

最后一条信息：如果你想把你的文本转换为大写或者小写，不要用 tr///。用双引号里的 \U 或者 \L 序列（或者等效的 uc 和 lc 函数），因为它们会关心区域设置或 Unicode 信息，而 tr/a-z/A-Z/ 不关心这些。另外在 Unicode 字串里，\u 序列和它的对应 ucfirst 函数能够识别标题格式，对某些语言来说，比简单地转换成大写更突出。

5.3 元字符和元符号

既然我们尊重这些神奇的笼子，那么我们就可以回过头来看看笼子里的动物了，也就是那些你放在模式里好看的符号。到现在你应该已经看到这样的事实，就是这些符号并不是普通的函数调用或者算术操作符那样的 Perl 代码。正则表达式本身就是嵌入 Perl 的小型语言。（在现实社会里总是有小丛林。）

Perl 里的模式识别所有的 12 个传统的元字符（所谓十二烂人），以及它们的所有潜能和表现力。许多其他正则表达式包里也能看到它们：

```
\ | () [ { ^ $ * + ? .
```

它们中有些曲解规则，令跟在它们后面本来正常的字符变成特殊的。我们不喜欢把长序列叫做“字符”，因此，如果它们组成长序列后，我们叫它们元符号（有时候干脆就叫“符号”）。但是在顶级，这十二个元字符就是你（和 Perl）需要考虑的所有内容。任何东西都是从这里开始的。

有些简单的元字符就代表它们自己，象 . 和 ^ 和 $。它们并不直接影响它们周围的任何东西。有些元字符运行起来象前缀操作符，控制任何跟在后面的东西，象反斜杠 "\" 。其他的则像后缀操作符，控制紧排在它们前面的东西，像 *，+，和 ?。有一个元字符：|，其作用象中缀操作符，站在它控制的操作数中间。甚至还有括弧操作符，作用类似包围操作符，控制一些被它包围的东西，像 (...) 和 [...]。圆括弧尤其重要，因为它们在内部声明 | 的范围，而在外部声明 *，+ 和 ? 的范围。

如果你只学习十二个元字符中的一个，那么选反斜杠。（恩。。。还有圆括弧）这是因为反斜杠令其他元字符失效。如果在一个 Perl 模式里，一个反斜杠放在一个非字母数字字符前，这样就让下一个字符成为一个字面的文本。如果你象在一个模式文本里匹配十二个元字符中的任何一个，你可以在它们前面写一个反斜杠。因此，\. 匹配一个真正的点，\$ 匹配真正的美圆符，\\ 是一个真正的反斜杠等等。这样做被称做"逃逸"元字符，或曰"引号包围之"，或者有时候就叫做"反斜杠某"。（当然，你已经知道反斜杠可以用于禁止在双引号字串里进行变量代换。）

虽然一个反斜杠把一个元字符转换成一个文本字符，它对后继的字母数字字符的作用却是完全另一码事。它把本来普通的东西变特别。也就是说，它们在一起形成元字符。我们在表 5-7 里给出了一个按字母排序的元字符表。

5.3.1 元字符表

符号	原子性	含义
\...	变化	反逃逸下一个非字母数字字符，转意下一个字母数字（可能）
... | ...	否	可选（匹配前者或者后者）。
(...)	是	分组（当作单元对待）。
[...]	是	字符表（匹配一个集合中的一个字符）。
	否	如果在字串开头则为真（或者可能是在任何换行符后面。）
.	是	匹配一个字符（通常除了换行符以外）。
$	否	在字串尾部时为真（或者可能是在任何换行符前面）。

至于量词，我们会在它们自己的节里详细描述。量词表示前导的原子（也就是说，单字符或者分组）应该匹配的次数。它们列在表 5-5 中。

表 5-5。 正则量词

量词	原子性	含义
*	否	匹配 0 或者更多次数（最大）。
+	否	匹配 或者更多次数（最大）。

?	否	匹配 1 或者 0 次（最大）。
{COUNT}	否	匹配 COUNT 次
{MIN,}	否	匹配至少 MIN 次（最大）。
{MIN,MAX}	否	匹配至少 MIN 次但不超过 MAX 次（最大）
*?	否	匹配 0 或者更多次（最小）
+?	否	匹配 1 或者更多次（最小）
??	否	匹配 0 或者 1 次（最小）
{MIN,}?	否	匹配最多 MIN 次（最小）
{MIN,MAX}?	否	匹配至少 MIN 次但不超过 MAX 次（最小）

最小量词会试图匹配在它的许可范围内的尽可能少的次数。最大量词会试图匹配在它的许可范围内的尽可能多的次数。比如， .+ 保证至少匹配字串的一个字符，但是如果有机会，它会匹配所有机会。这里的机会将在稍后的"小引擎的/能与不能/"节里讲。

你还会注意量词决不能量化。

我们想给新类型的元符号一个可以扩展的语法。因为我们只需要使用十二个元字符，所以我们选用原先被认为是非法正则的序列做任意语法扩展。这些元符号的形式都是 (?KEY...)；也就是，一个开圆括弧后面跟着一个问号，然后是 KEY 和模式其余部分。KEY 字符表明它是哪种正则扩展。参阅表 5-6 看看正则扩展的一个列表。它们中大多数性质象列表，因为它们基于圆括弧，不过它们还是有附加含义。同样，只有原子可以量化，因为它们代表真正存在（潜在地）的东西。

表 5-6 扩展的正则序列

扩展	原子性	含义
(?#...)	否	注释，抛弃
(?:...)	是	只集群，不捕获的圆括弧
(?imsx-imsx)	否	打开/关闭模式修饰词
(?imsx-imsx:...)	是	集群圆括弧加修饰词
(?=...)	否	如果前向查找断言成功，返回真
(?!...)	否	如果前向查找断言失败，返回真
(?<=...) </td>	否	如果前向查找断言成功，返回真
(?	否	如果前向查找断言失败，返回真
(?>...)	是	匹配未反向跟踪的子模式
(?{...})	否	执行嵌入的 Perl 代码
(??{...})	是	匹配来自嵌入 Perl 代码。

(?(...)...	...)	是	匹配 if-then-elase 模式
(?(...)...)	是	匹配 if-then 模式	

最后，表 5-7 显示了所有你常用的字母数字元符号。（那些在变量代换回合处理过的符号在原子性列里用一个划线标记，因为引擎永远不会看到它们。）

表 5-7。字母数字正则元符号

符号	原子性	含义
\0	是	匹配空字符（ASCII NUL）。
\NNN	是	匹配给出八进制的字符，最大值为\377。
\n	是	匹配前面第 n 个捕获字串（十进制）。
\a	是	匹配警钟字符（BEL）。
\A	否	如果在字串的开头为真
\b	是	匹配退各字符（BS）。
\b	否	在字边界为真
\B	否	不在字边界时为真
\cX	是	匹配控制字符 Control-x（\cZ，\c[，等）。
\C	是	匹配一个字节（C 字符），甚至在 utf8 中也如此（危险）
\d	是	匹配任何数字字符
\D	是	匹配任何非数字字符
\e	是	匹配逃逸字符（ASCII ESC，不是反斜杠）。
\E	——	结束大小写（\L，\U）或者掩码（\Q）转换
\f	是	匹配进页字符（FF）。
\G	否	如果在前一个 m//g 的匹配结尾位置时为真
\l	——	只把下一个字符变成小写
\L	——	把\E 以前的字母都变成小写
\n	是	匹配换行符字符（通常是 NL，但是在 Mac 上是 CR）。
\N{NAME}	是	匹配命名字符（\N{greek:Sigma}）。
\p{PROP}	是	匹配任何有命名属性的字符
\P{PROP}	是	匹配任何没有命名属性的字符
\Q	——	引起（消元）直到\E 前面的字符
\r	是	匹配返回字符（通常是 CR，但是在 Mac 上是 NL）。
\s	是	匹配任何空白字符。
\S	是	匹配任何非空白字符。

\t	是	匹配水平制表符（HT）。
\u	——	只把下一个字符变成标题首字符
\U	——	大写（不是标题首字符）\E 以前的字符。
\w	是	匹配任何"字"字符（字母数字加"_" ）。
\W	是	匹配任何"非字"字符。
\x{abcd}	是	匹配在十六进制中给出的字符。
\X	是	匹配 Unicode 里的"组合字符序列"字串。
\z	否	只有在字串结尾时为真
\Z	否	在字串结尾或者在可选的换行符之前为真。

如果在 \p 和 \P 里的属性名字是一个字符，那么花括弧是可选的。如果 \x 里的十六进制数为两位或者更少，那么花括弧也是可选的。在 \N 里的花括弧决不能省略。

只有在含义中带"匹配。。。"或者"匹配任何。。。"字样的元符号才能够在字符表（方括弧）里面使用。也就是说，字符表仅限于包含特定的字符集，因此在字符表里面，你只能使用那些描述其他特定字符集的元符号，或者那些描述特定独立字符的元符号。当然，这些元符号也可以和其他非分类元符号一起在字符表外面用，不过，这里请注意 =b 是两只完全不同的怪兽：它在字符表内是退格字符，而在外边是一个字边界断言。

一个模式可以匹配的字符的数量和一个双引号字串可以替换的字符的数量有一些重叠。因为正则要经历两个回合，所以有时候应该由哪个回合处理一个给定的字符会显得有些混乱。如果出现混乱，这种字符的变量代换回合就推迟给正则表达式分析器。

但是只有当变量代换回合知道它正在分析一个正则的时候，它才能把变量代换推迟给正则分析器。你可以把正则表达式声明为普通的双引号包围字串，但这样你就必须遵循普通的双引号包围规则。任何前面碰巧映射为时间字符的元符号仍然生效，即使它们没有被推迟给正则分析器也如此。但是在普通的双引号里你不能使用任何其它的元符号（或者任何类似的构造，比如 `...`，qq(...)，qx(...)，或者等效的"此处"文档）。如果你想你的字串分析成一个正则表达式而不做任何匹配，你应该使用 qr// （引号构造正则）操作符。

请注意大小写和元引号包围转换逃逸（\U 和它的伙伴）必须在变量代换回合处理，因为这些元符号的用途就是影响变量代换。如果你用单引号禁止变量代换，那么你也不能获得转换逃逸。在任何单引号字串里，都不会进行变量或者转换逃逸（\U 等）的扩展，在单号包围的的 m'...' 或者 qr'...' 操作符里也不会。甚至在你做代换的时候，如果这些转换逃逸是一个变量代换的结果，那么它们也会被忽略，因为这个时候他们想要影响变量代换已经太晚了。

尽管字符替换操作符不处理正则表达式，但是我们讨论过的任何匹配单个特定字符的元符号在 tr/// 操作中仍然可用。而其他的用不了（除了反斜杠以外，它继续按照它原来的样子运转。）

5.3.2 特定的字符

如前所述，非特殊的东西在模式里匹配自身。这意味着一个 /a/ 匹配一个 "a"，一个 /=/ 匹配一个 "=" 等等。不过，有些字符可不是那么容易在键盘上敲进去，或者即使你敲进去了，也不会在打印输出中显示出来；最臭名昭著的例子就是控制字符。在正则表达式里，Perl 识别下列双引号包围的字符别名：

逃逸	含义
\0	空字符（ASCII NUL）
\a	警铃（BEL）
\e	逃逸（ESC）
\f	进纸（FF）
\n	换行符 （NL，Mac 里的 CR）
\r	回车（CR，Mac 里的 NL）
\t	水平制表符（HT）

就象在双引号包围字串里一样，Perl 还识别模式里的下面四种元符号：

\cX

一个命名的控制字符，象 \cC 指 Control-C，\cZ 指 Control-Z，\c[指 ESC，而 \c? 表示 DEL。

\NNN

用两位或者三位八进制码声明的字符。除了小于 010 的数值（十进制 8）外，前导的 0 是可选的，因为（和在双引起字串里不同）一位数字的东西总认为是用于在模式里捕获字串的引用。如果你在模式里先捕获了至少 n 个子字串，那多位数字解释成第 n 个引用（这里 n 被认为是一个十进制数）。否则，他们被解释成一个用八进制声明的字符。

\x{LONGHEX}

一个用一到两个十六进制数位（[0-9a-fA-F]）声明的字符，比如 \x1B。一位数字的形式只有在

	后面跟随的字符不是一个十六进制数字才可用。如果使用了花括弧，你想用多少位数字都可以，这时候结果可能是一个 Unicode 字符。比如，\x{262} 匹配一个 Unicode YIN YANG。
\N{NAME}	一个命名字符，如 \N{GREEK SMALL LETTER EPSILON}，\N{Greek:epsilon}，或者 \N{epsilon}。它要求使用第三十一章，用法模块，里描述的 use charnames 用法，它同时还判断你可能使用那些名字中的哪一个（分别是":long"，":full"，":short"，对应上面三个风格。）

你可以在离你最近的 Unicode 标准文档里找到所有 Unicode 字符名字的列表，或者在 PATH_TO_PERLLIB/unicode/Names.txt 里也有。

5.3.3 通配元符号

三个特殊的元符号可以用做通用通配符，它们的每一个都可以匹配"任何"字符（是"任何"中的某些字符）。它们是句点（"."），\c 和 \x。它们都不能在字符表里使用。你不能在字符表里用句点是因为它会匹配（几乎）任何存在的字符，因此它本身就是某种万能字符。如果你想包括或者排除所有东西，也没有什么必要使用一个字符表。特殊通配符 \C 和 \X 有着特殊的结构化含义，而这些特殊含义和选择单个 Unicode 字符的表示法关联得并不好，而该表示法才是字符表运行的层次.

句点元字符匹配除了换行符以外的任何单字符。（如果带着 /s 修饰词，也能匹配换行符。）和十二个特殊字符里的其它字符一样，如果你想匹配一个文本句点，你就必须用一个反斜扛逃逸它。比如，下面的代码检查一个文件名是否以一个句点后面跟着一个单字符扩展名结尾的：

```
if ($pathname =~ /\.(.)\z/s) {
    print "Ends in $1\n";
}
```

第一个句点是逃逸了的，是文本句点，而第二个句点说"匹配任何字符"。\z 说只匹配字串末尾的东西，而 \s 修饰词令点也可以匹配换行符。（的确，用换行符做文件扩展名不怎么漂亮，但并不是说就不能做。）

点元字符经常和量词一起使用。.* 匹配尽可能多的字符，而 .*? 匹配尽可能少的字符。不过有时候它不用量词而是自己解决长度问题： /(..):(..):(..)/ 匹配三个用冒号分隔的域，每个域两个字符长。

如果你在一个 use utf8 用法的词法范围里编译的模式里使用一个点，那么它就匹配任何 Unicode 字符。（你可能不需要用 use utf8，不过偶然还是会发生的，在你阅读到这里的时候你可能不需要这个用法。）

```
use utf8;

use charnames qw/:full/;

%BWV[887] = "G\N{MUSIC SHARP SIGN} minor";

($note, $black, $mode) = $BWV[886] =~ /^([A-G])(.)\s+(\S+)/;

print "That's lookin' sharp!\n" if $black eq chr(9839);
```

元符号 \X 在更广的概念上匹配字符。它实际上是匹配一个由一个或多个 Unicode 字符组成的字串，这个字串就是所谓的"组合字符序列"。这样的序列包括一个基本字符和后面跟着任意个"标志"字符（象变音符和分音符那样的区分标志）一起组成一个逻辑单元。\X 实际上等效于 (?:\PM\pM*)。这样做允许匹配一个逻辑字符，即使这几个字符实际上是由几个独立的字符组成的也行。如果匹配任意组合字符，那么在 /\X/ 里匹配的长度将超过一个字符长。（而且这里的字符长度和字节长度没有什么关系）。

如果你在使用 Unicode 并且真的想获取单字节而不是单字符，那么你可以使用 \C 元字符。它将总是匹配一个字节（具体说，就是一个 C 语言的 char 类型），而不管是否会与你的 Unicode 字符流步调失调。参阅第十五章里关于做这些事情时合适的警告。

5.4 字符表

在模式匹配里，你可以匹配任意字符，不管它们有没有特殊性质。有四种声明字符表的方法（译注：孔乙己？：）。你可以按照传统的方法声明字符集——用方括弧和枚举可能的字符，或者或者你可以使用三种记忆法中的任意一种：经典 Perl 表，新 PerlUnicode 属性，或者标准 POSIX 表。这些缩写均只匹配其字符集中的一个字符。你可以量化它们，使它们可以匹配更多的字符，比如 \d+ 匹配一个或者多个数字。（一个比较容易犯的错误是认为 \w 匹配一个字。用 \w+ 匹配一个字。）

5.4.1 客户化字符表

一个方括弧中的一个枚举字符列表被称为字符表，它匹配列表中的任何一个字符。比如，[aeiouy] 匹配一个英文中的元音字母。（对于威尔士要加 "w"，对于苏格兰加个 "r"。）要匹配一个右方括弧，你可以用反斜杠逃逸之或者把它放在列表开头。

字符范围可以用一个连字符和 a-z 表示法表示。你可以合并多个范围；比如 [0-9a-fA-F] 匹配一个十六进制"位"。你可以用反斜杠避免连字符被解释为一个范围分隔符，或者把它放在表的开头或者结尾（后面的方法虽然不易读，但是比较常用）。

在字符表开头的脱字符（或者说是抑扬符号，或者帽子，或者向上箭头 "^" ）反转该字符表，结果是匹配任何不在此列表中的字符。（要匹配脱字符，要么不要放在开头，或者是用反斜杠逃逸）。比如，[^aeiouy] 匹配任何不是元音的字母。不过，对待字符表反意要小心些，因为字符的领域在不断扩展。比如，那个字符表匹配辅音——而在西利尔语，希腊语和几乎任何语言里还匹配空白，换行符和其他任何东西（包括元音），更不用说中日韩文里的标记了。而且以后还可能有 Cirth，Tengwar，和 Klingon。（当然，还可能有 Linear B 和 Etruscan）所以你最好还是明确声明你的辅音，比如[cbdfghjklmnpqrstvwxyz]，或者简写为 [b-df-hj-p-tv-z]。（这样还解决了"y"需要在两个地方同时出现的问题，排除了一个补集。）

字符表里支持普通字符元符号，（参阅"声明字符"），比如 \n，\t，\cX，\NNN，和 \N{NAME}。另外，你可以在一个字符表里使用 \b 表示一个退格，就象它在双引号字串里显示的那样。通常，在一个模式匹配里，它意味着一个字边界。但是零宽度的断言在字符表里没有任何意义，因此这里的 \b 返回到它在字串里的普通含义。你还可以使用我们本章稍后预定义的任何字符表（经典，Unicode 或 POSIX），但是不要把它们用在一个范围的结尾——那样没有意义，所以 "-" 会被解释成文本。

所有其他的元符号在方括弧中都失去特殊意义。实际上，你不能在这里面使用三个普通通配符中的任何一个："."，\X 或 \C。不允许第一个字符通常令人奇怪，不过把普遍意义的字符表用做有限制的形式的确没有什么意义，而且你常会在一个字符表中要用到文本句点——比如，当你要匹配文件名的时候。而且在字符表里声明量词，断言或者候选都是没有意义的，因为这些字符都是独立解释的。比如，[fee|fie|foe|foo] 和 [feio|] 是一样的。

5.4.2 典型 Perl 字符表缩写

从一开始，Perl 就已经提供了一些字符表缩写。它们在表 5-8 中列出。它们都是反斜杠字母元字符，而且把它们的字母换成大写后，它们的含义就是小写版本的反义。这些元字符的含义并不像你想象的那么固定，其含义可能在新的 Unicode 标准出台后改变，因为新标准会增加新的数字和字母。（为了保持旧的字节含义，你总是可以使用 use bytes。要解

释 utf8 的含义，参阅本章后面的 "Unicode 属性"。不管怎样，utf8 含义都是字节含义的超集。）

表 5-8 典型字符表

符号	含义	做字节	做 utf8
\d	数字	[0-9]	\p{IsDigit}
\D	非数字	[^0-9}	\P{IsDigit}
\s	空白	[\t\n\r\f]	\p{IsSpace}
\S	非空白	[^ \t\n\r\f]	\P{IsSpace}
\w	字	[a-zA-Z0-9_]	\p{IsWord}
\W	非字	[^a-zA-Z0-9_]	\P{IsWord}

（好好好，我们知道大多数字里面没有数字和下划线，\w 的用意是用于匹配典型的编程语言里的记号。就目前而言，是匹配 Perl 里的记号。）

这些元符号在方括弧外面或者里面都可以使用，也就是说不管作为独立存在的符号还是作为一个构造成的字符表的一部分存在都行：

```
if ($var =~ /\D/)    { warn "contains non-digit" }

if ($var =~ /[^\w\s.]/) { warn "contains non-(word, space, dot)"}
```

5.4.3 Unicode 属性

Unicode 属性可以用 \p{PROP} 及其补集 \P{PROP} 获取. 对于那些比较少见的名字里只有一个字符的属性，花括弧是可选的，就象 \pN 表示一个数字字符（不一定是十进制数 — 罗马数字也是数字字符）。这些属性表可以独自使用或者与一个字符表构造一起使用：

```
if ($var =~ /^\p{IsAlpha}+$/)     {print "all alphabetic" }

if ($var =~ s/[\p{Zl}\p{Zp}]/\n/g) {print "fixed newline wannabes"}
```

有些属性是直接由 Unicode 标准定义的，而有些属性是 Perl 基于标准属性组合定义的，Zl 和 Zp 都是标准 Unicode 属性，分别代表行分隔符和段分隔符，而 IsAlpha[2] 是 Perl 而 IsAlpha[2] 是 Perl 定义的，是一个组合了标准属性 Ll, Lu, Lt, 和 Lo （也就是小写，大写，标题或者其它字母）的属性表。在 Perl 5.6.0 里，要想用这些属性，你得用 use utf8。将来会放松这个限制。

还有很多其它属性。我们会列出我们知道的那些，但是这个列表肯定不完善。很可能在新版本的 Unicode 里会定义新的属性，而且你甚至也可以定义自己的属性。稍后我们将更详细地介绍这些。

Unicode 委员会制作了在线资源，这些资源成为 Perl 用于其 Unicode 实现里的各种文件。关于更多这些文件的信息，请参阅第 15 章。你可以在文档 PATH_TO_PERLLIB/unicode/Unicode3.html 里获得很不错的关于 Unicode 的概述性介绍。这里 PATH_TO_PERLLIB 是下面命令的打印输出：

```
perl -MConfig -le 'print $Config{privlib}'
```

大多数 Unicode 的属性形如 \p{IsPROP}。Is 是可选的，因为它太常见了，不过你还是会愿意把它们写出来的，因为可读性好。

5.4.3.1 Perl 的 Unicode 属性

首先，表 5-9 列出了 Perl 的组合属性。它们定义得合理地接近于标准 POSIX 定义的字符表。

表 5-9. 组合 Unicode 属性

属性	等效
IsASCII[2]	[\x00-\x7f]
IsAlnum[2]	[\p{IsLl}\p{IsLu}\p{IsLt}\p{IsLo}\p{IsNd}]
IsAlpha[2]	[\p{IsLl}\p{IsLu}\p{IsLt}\p{IsLo}]
IsCntrl[2]	\p{IsC}
IsDigit[2]	\p{Nd}
IsGraph[2]	[^\pC\p{IsSpace}]
IsLower[2]	\p{IsLl}
IsPrint[2]	\P{IsC}
IsPunct[2]	\p{IsP}
IsSpace[2]	[\t\n\f\r\p{IsZ}]
IsUpper[2]	[\p{IsLu}\p{IsLt}]
IsWord[2]	[_\p{IsLl}\p{IsLu}\p{IsLt}\p{IsLo}\p{IsNd}]
IsXDigit[2]	[0-9a-fA-F]

Perl 还为标准 Unicode 属性（见下节）的每个主要范畴提供了下列组合：

属性	含义	是否规范的

IsC[?]	错误控制代码等	是
IsL[?]	字母	部分
IsM[?]	标志	是
IsN[?]	数字	是
IsP[?]	标点	否
IsS[?]	符号	否
IsZ[?]	分隔符	是

5.4.3.2 标准的 Unicode 属性

表 5-10 列出了大多数基本的标准 Unicode 属性，源自每个字符的类别。没有哪个字符是多于一个类别的成员。有些属性是规范的，而有些只是提示性的。请参阅 Unicode 标准获取那些标准的说辞，看看什么是规范信息，而什么又是提示信息。

表 5-10 标准 Unicode 属性

属性	含义	规范化
IsCc[?]	其他，控制	是
IsCf[?]	其他，格式	是
IsCn[?]	其他，未赋值	是
IsCo[?]	其他，私有使用	是
IsCs[?]	其他，代理	是
IsLl[?]	字母，小写	是
IsLm[?]	字母，修饰词	否
IsLo[?]	字母，其他	否
IsLt[?]	字母，抬头	是
IsLu[?]	字母，大写	是
IsMc[?]	标记，组合	是
IsMe[?]	标记，闭合	是
IsMn[?]	标记，非空白	是
IsNd[?]	数字，十进制数	是
IsNl[?]	数字，字母	是
IsNo[?]	数字，其他	是
IsPc[?]	标点，联接符	否
IsPd[?]	标点，破折号	否

IsPe[?]	标点，关闭	否
IsPf[?]	标点，结束引用	否
IsPi[?]	标点，初始引用	否
IsPo[?]	标点，其他	否
IsPs[?]	标点，打开	否
IsSc[?]	符号，货币	否
IsSk[?]	符号，修饰词	否
IsSm[?]	符号，数学	否
IsSo[?]	符号，其他	否
IsZl[?]	分隔符，行	是
IsZp[?]	分隔符，段落	是
IsZs[?]	分隔符，空白	是

另外一个有用的属性集是关于一个字符是否可以分解为更简单的字符。（规范分解或者兼容分解）。规范分解不会丢失任何格式化信息。兼容分解可能会丢失格式信息，比如一个字符是否上标等。

属性	信息丢失
IsDecoCanon[?]	无
IsDecoCompat[?]	有一些（下列之一）
IsDCcircle[?]	字符周围的圆
IsDCfinal[?]	最终的位置（阿拉伯文）
IsDCfont[?]	字体变体的选择
IsDCfraction[?]	俚语字符片段
IsDCinitial[?]	起始位置选择（阿拉伯文）
IsDCisolated[?]	隔离位置选择（阿拉伯文）
IsDCmedial[?]	中间位置选择（阿拉伯文）
IsDCnarrow[?]	窄字符
IsDCnoBreadk[?]	空白或连字符的非中断选
IsDCsmall[?]	小字符
IsDCsquare[?]	CJK 字符周围的方块
IsDCsub[?]	脚标
IsDCsuper[?]	上标
IsDCvertical[?]	旋转（水平或垂直）
IsDCwide[?]	宽字符

IsDCcompat[?]	标识（杂项）

下面是那些对双向书写的人感兴趣的属性：

属性	含义
IsBidiL[?]	从左向右（阿拉伯语，希伯来语）
IsBidiLRE[?]	从左向右插入
IsBidiLRO[?]	从左向右覆盖
IsBidiR[?]	从右向左
IsBidiAL[?]	阿拉伯语从右向左
IsBidiRLE[?]	从右向左插入
IsBidiRLO[?]	从右向左覆盖
IsBidiPDF[?]	流行方向的格式
IsBidiEN[?]	欧洲数字
IsBidlES[?]	欧洲数字分隔符
IsBidkET[?]	欧洲数字结束符
IsBidiAN[?]	阿拉伯数字
IsBidiCS[?]	普通数字分隔符
IsBidiNSM[?]	非空白标记
IsBidiBN[?]	与边界无色
IsBidiB[?]	段落分隔符
IsBidiS[?]	段分隔符
IsBidiWS[?]	空白
IsBidiON[?]	其他无色字符
IsMirrored[?]	当使用从右向左模式时反向

下面的属性根据元音的发音把它们分类为各种音节：

IsSylA	IsSylE	IsSylO	IsSylWAA	IsSylWII
IsSylAA	IsSylEE	IsSylOO	IsSylWC	IsSylWO
IsSylAAI	IsSylI	IsSylU	IsSylWE	IsSylWOO
IsSylAI	IsSylII	IsSylV	IsSylWEE	IsSylWU
IsSylC	IsSylN	IsSylWA	IsSylWI	IsSylWV

比如，\p{IsSyLA} 将匹配 \N{KATAKANA LETTER KA}，但不匹配 \N{KATAKANA LETTER KU}。

既然你现在基本上已经知道了所有的这些 Unicode 3.0 属性，那我们还要说的是在版本 5.6 的 Perl 里有几个比较秘传的属性还没有实现，因为它们的实现有一部分是基于 Unicode 2.0 的，而且，象那些双向的算法还在我们的制作之中。不过，等到你读到这些的时候，那些缺失的属性可能早就实现了，所以我们还是把它们列出来了。

第六章 子过程

象其他的语言一样，Perl 也支持自定义的子过程．（注：我们也把它们叫做函数，不过函数和子过程在 Perl 里是一样的东西．有时候我们甚至叫它们方法，方法和函数或子过程是同样的方式定义的，只是调用方式不同．）这些子过程可以在主程序中的任何地方定义，也可以用 do，require 或 use 关键字从其他文件中加载．或者直接使用 eval 在运行的时候产生．你甚至可以使用第十章"包"中"自动装载"一节描述的机制在运行时加载它们．你可以间接调用子过程，使用一个包含该子过程名字或包含指向该子过程引用的变量来调用，或者通过对象，让对象决定调用哪个子过程．你可以产生只能通过引用使用的匿名子过程，如果必要，你还可以通过闭合，用匿名子过程克隆几乎相同的函数．我们将在第八章"引用"中的相关小节中讲述．

1.0 语法

声明一个命名子过程，但不定义它，使用下面的形式：

```
sub NAME

sub NAME PROTO

sub NAME       ATTRS

sub NAME PROTO ATTRS
```

声明并且定义一个命名子过程，加上一个 BLOCK：

```
sub NAME              BLOCK
```

```
sub NAME PROTO       BLOCK

sub NAME       ATTRS BLOCK

sub NAME PROTO ATTRS BLOCK
```

创建一个匿名子过程或子句，把 NAME 去掉就可以：

```
sub                  BLOCK

sub      PROTO       BLOCK

sub              ATTRS BLOCK

sub      PROTO ATTRS BLOCK
```

PROTO 和 ATTRS 表示原型和属性，分别将在本章下面的章节中讨论．相对于 NAME 和 BLOCK 它们并不很重要．NAME 和 BLOCK 是基本部分，甚至有时候它们也可以省略．

对于没有 NAME 的形式，你还必须提供调用子过程的方法．因此你必须保存返回值，因为这种形式的 sub 声明方法不但在编译的时候编译，同时也产生一个运行时的返回值，所以我们就可以保证保存它：

```
$subref = sub BLOCK;
```

可以用下面的方法引入在另一个模块中定义的子过程：

use MODULE qw(NAME1 NAME2 NAME2...)

直接调用子过程可以用下面的方法：

```
  NAME(LIST)       # 有圆括弧时 & 是可选的

   NAME LIST        # 如果预声明/输入了子过程，那么圆括弧是选的

   &NAME             # 把当前的 @_ 输出到该子过程

                                #（并且绕开原型）.
```

间接调用子过程(通过名字或引用)，可以使用下面的任何一种方法：

1. &$subref(LIST) # 在间接调用的时候，& 不能忽略
2. $subref->(LIST) # （除非使用中缀表示法）
3. &$subref # 把当前的 @_ 输出到该子过程

正式情况下，一个子过程的名字包括 & 前缀，一个子过程可以使用 & 前缀调用，但通常情况下 & 是可选的，如果预先定义了子过程，那么圆括弧也是可选的．但是，在只使用子过程名字的时候，& 不能省略，例如当子过程名字被用做一个参数来判断是否它已经定义过的时候，或者当你使用 $subref = \&name 来获取一个命名子过程的引用的时候．同样，当你使用 &$subref() 或 &{$subref()} 进行一个间接子过程调用的时候也不能省略 &．不过，如果使用一种更方便的形式 $subref->()，则不需要 &．参看第八章，那里有更多有关子过程引用的内容．

Perl 并不强制子过程名字使用大写风格．但是按惯例由 perl 的运行时系统间接调用的函数都是大写的 (BEGIN， CHECK， INIT， END， AUTOLOAD， DESTORY，和所有第十四章 "捆绑变量"涉及到的函数)．因此你应该避免使用这种大写风格．(但是操作常量值的子过程通常也写成大写的)．

2.0 语意

在你记住所有语法前，你只需要记住下边这种定义子过程的普通方法：

```
sub razzle {
        print "Ok, you've been razzled.\n";
}
```

和调用子过程的正常方法就是：

```
razzle();
```

在上边的写法中，我们省略了输入(参数)和输出(返回值)．但是 Perl 向子过程中传入数据和子过程传出数据的方法非常简单：所有传入的参数被当成单个平面标量列表，类似的多个返回值也被当成单个平面标量列表返回给调用者．当使用任意 LIST 时也一样，任何传入的数组或散列的值都代换到一个平面的列表里面，同时也失去了它们的标识，不过有几种方法可以绕开这个问题，这种自动的列表代换在很多场合非常有用．参数列表和返回值列表都可以根据你的需要包含任意多个标量成员(当然你可以使用原型定义来约束参数的类型)．实际上，Perl 是按照支持可变参函数(可以支持任何数量的参数)概念来设计的．C 则不同，虽然 C 也勉强支持一些变参的函数，例如 printf (3)．

现在，如果你将设计一种可以支持不定数量的任意参数的语言，你最好让你的语言在处理这些任意长的参数列表上容易些．所有传入 Perl 过程的参数都是以 @_ 身份传入的．如果你调用一个有两个参数的函数，它们在函数内部可以作为 @_ 数组的前两个成员访问：$_[0] 和 $_[1]．因为 @_ 只是一个有着奇怪名字的普通数组，所以你可以象处理普通数组一样随意处理它．（注：这个领域是 Perl 和传统的编程语言冲突得最厉害的地方．）数组 @_ 是一个本地数组，但是它的值是实际标量参数的别名(通常称为引用传参)因而如果修改了 @_ 中的成员那么同时也修改了对应的实际参数的值.(通常的语言中很少这么做，但是采用这种方法在 Perl 中可以很容易的返回所需要的值).

子过程(其他的程序块也一样)的返回值是过程最后一个表达式的值. 或者你可以在子过程的任何一个地方明确使用一个 return 语句来返回值并且退出子过程．不管是那种方法，当在一个标量或列表环境中调用子过程时，最后一个表达也将在同样的标量或列表环境中求值.

2.1 参数列表的技巧

Perl 没有命名的正式参数，但是在实际中你可以将 @_ 的值拷贝到一个 my 列表，这样就可以方便使用这些正式参数(不一样的是，这样拷贝就将引用传参的语义变为了传值传参，也许传值传参正是很多用户通常希望参数被处理的方法，即使他们不知道这些计算机术语)，下面是一个典型的例子：

```
sub aysetenv {
        my ($key, $value) = @_;
        $ENV{$key} = $value unless $ENV{$key};
}
```

但是没人要你一定要给你的参数命名，这就是 @_ 数组的全部观点．例如，计算一个最大值，你可以简单直接遍历 @_ 数组：

```
sub max {
        $max = shift(@_);
        for my $item (@_) {
                $max = $item if $max < $item;
        }
```

```
        return $max;
}

$bestday = max($mon, $tue, $wed, $thu, $fri);
```

或者你可以一次将 @_ 填入一个散列：

```
sub configuration {
        my %options = @_;
        print "Maximum verbosity.\n" if $options{VERBOSE} == 9;
}

configuration(PASSWORD => 'xyzzy', VERBOSE => 9, SOCRE => 0);
```

下面是一个例子，这里不命名正式参数，这样你可以修改实际参数的值：

```
upcase_in($v1, $v2);   # 这里改变 $v1 和 $v2
sub upcase_in {
        for (@_) { tr/a-z/A-Z/ }
}
```

但是你不允许用这种方法修改常量，如果一个参数是一个象 "hobbit" 这样的实际标量值或象 $1 这样只读标量，当你试图修改它时，Perl 会抛出一个例外(可能的致命错误或者可能的威胁).例如，下面的例子将不能工作：

```
upcase_in("fredrick");
```

如果将 upcase_in 函数写成返回它的参数的一个拷贝会比直接改变参数安全得多：

```
($v3, $v4) = upcase($v1, $v2);
sub upcase {
        my @parms = @_;
        for (@parms) { tr/a-z/A-Z/ }
```

```
        # 检查我们是否在列表环境中被调用的

        return wantarray ? @parms : $parms[0];

}
```

注意这个函数(没有原型)并不在意传进来的参数是真的标量还是数组．Perl 将所有的参数粉碎成一个又大又长的平面 @_ 数组列表．这是 Perl 简单传参方式闪光的地方之一．甚至我们可以给它象下面这样的参数的时候都不需要修改 upcase 的定义，upcase 将照样工作得呗棒：

```
   @newlist = upcase(@list1, @list2);

   @newlist = upcase( split /:/, $var);
```

但是，如果象下边这样用，就不会得到你想要的的结果：

```
   (@a, @b) = upcase( @list1, @list3);   # 错
```

因为，和 @_ 一样，返回列表同样是一个平面列表．因此所有的返回值将存储在 @a 中，@b 是空的.可以在下面"传递引用"部分看到替代的办法．

2.2 错误指示

如果你希望你的函数能够以特定的方式返回，使调用者能够得知发生了一个错误．在 Perl 中实现这个目的最自然的一种方法就是用一个不带参数的 return 语句．这样当函数在标量环境中使用时，调用者得到一个 undef，如果在列表环境中使用，调用者得到一个空列表．

特殊情况下，你可以选者产生一个例外来指示错误，但是必须谨慎使用这种方法．因为你的程序将被例外处理程序终结．例如，在一个文件操作函数中，打开文件失败几乎不是例外的事件．因此，最好能够忽略这种失败．当你在无效的环境下调用函数时，wantarray 内建函数将返回 undef．因此如果你想忽略它，你可以使用下面的方法：

```
if($something_went_awry) {

        return if defined wantarray;   # 很好，不是空环境

        die "Pay attention to my error, you danglesocket!!!\n";

}
```

2.3 范围问题

因为每次调用都有自己的参数数组，因此子过程可以递归调用，甚至可以调用它自己. 如果使用 & 的形式调用子过程，那么参数列表是可选的. 如果使用了 & 并且省略了参数列表，那么有一些特殊的规则：调用过程中的 @_ 数组将做为被调用子过程的参数. 新用户可能不想使用这种有效的机制.

```
&foo(1,2,3)    # 传递三个参数
foo(1,2,3)     # 和上面一样

foo();             # 传递一个空列表
&foo();          # 和上面一样

&foo;          # foo() 获取当前的参数，和 foo(@_) 一样，但更快!
foo;            # 如果预定义了子过程 foo，那么和 foo() 一样，否则
               # 就是光字 "foo"
```

使用 & 形式调用子过程不仅可以省略掉参数列表，同时对你提供的参数也不进行任何原型检查. 这种做法一部分是因为历史原因形成，另一部分原因是为了在用户清楚自己在干什么的情况下提供一个方便的办法. 你可以参看本章后面的"原型"小节.

在函数中访问一个并没有定义成该函数私有的变量不一定是全局变量；它们遵循第二章 "集腋成裘"中"名字"一节中提到的块作用范围规则，这意味着他们首先在词法作用范围里面决定该变量，然后才扩展到单个包作用范围. 从子过程的角度看来，任何在一个闭合的词法作用域中的 my 变量仍然优先使用.

例如，下面例子中的 bumpx 函数使用了文件作用范围中的 $x 变量，这是因为 my 变量被定义的作用范围 --- 也就是文件本身 --- 并没有在定义子过程之前结束.

```
# 文件顶部
my $x = 10;              # 声明和初始化变量
sub bumpx { $x++ }    # 函数可以看到外层词法变量
```

C 和 C++ 程序员很可能认为 $x 是一个"文件静态"变量．它对其他文件中的函数是私有的，但是在上例中在 my 后面定义的函数可以透视到这个变量．那些由 C 程序员转变而来的 Perl 程序员在 Perl 中找不到他们熟悉的文件或函数的"静态变量"．Perl 程序员避免使用 "static"这个词．因为静态系统过时而且乏味，并且因为在历史使用中这个词被搞得一团糟．

虽然 Perl 语法中没有包括"static"这个词，但是 Perl 程序员同样能够创建函数的私有变量，并且保持跨函数访问．Perl 中没有专门的词来表述他们．利用 Perl 丰富的作用范围规则结合自动内存管理，就可以有很多方式实现"static"关键字的功能．

词法变量并不会只是因为退出了它们的作用范围后就被自动内存垃圾收集回收，它们要等到不再使用后才被回收，这个概念十分重要．为了创建一个在跨函数调用中不被重置的私有变量，你可以将整个函数用花括弧括起来，并将 my 定义和函数定义放入该函数块中．你甚至可以放入多个函数定义，这样该私有变量就可以被这些函数共享访问．

```
{
        my $counter = 0;
        sub next_counter { return ++$counter }
        sub prev_counter { return --$counter }
}
```

通常，对词法变量的访问被限制在同一个词法作用域中．两个函数的名字可以被全局访问(在同一个包内)，并且因为它们是在 $counter 的作用域中定义的，它们仍然可以访问该变量，即使其他函数访问不到也无妨．

如果这个函数是通过 require 或 use 加载的，那么也可以．如果它全在主程序中，那么你就要确保使任何运行时的 my 赋值要足够地早，你可以将整个程序块放在主程序的最前边，也可以使用 BEGIN 或 INIT 程序块来保证它在你的程序之前运行：

```
BEGIN {
        my @scale = ('A' .. 'G');
        my $note = -1;
        sub next_pitch { return $scale[ ($note += 1) %= @scale ] );
        }
```

BEGIN 既不会影响子过程的定义，也不会影响子过程里使用的任意词法的一致性．这里它仅仅保证在子程序被调用之前变量就被初始化．想了解定义私有变量和全局变量更多的内容，请分别参考 29 章"函数"的 my 和 our 的说明，BEGIN 和 INIT 在第十八章"编译"中解释．

3.0 传入引用

如果你想在一个函数中传入或传出不止一个的数组或散列结构，同时你希望它们保持它们的一致性，那么你就需要使用一个更明确的传递引用的机制．在你使用传递引用之前，你需要懂得第八章里有关引用的细节．本小节不着重讲述引用的内容．

这里有几个简单的例子，首先，让我们定义一个函数，这个函数使用数组的引用作为参数．当这个数组非常大时，作为一个引用传递要比传入一长列值要快得多：

```
$total = sum (\@a );


sub sum {
        my ($aref) = @_;
        my ($total) = 0;
        foreach (@$aref) { $total += $_ }
        return $total;
}
```

下面让我们将几个数组传入一个函数，并且使用使用 pop 得到每个数组的最后一个元素，并返回每个数组最后一个元素组成的一个新的数组：

```
@tailings = popmany (\@a, \@b, \@c, \@d );


sub popmany {
        my @retlist = ();
        for my $aref (@_) {
                push @retlist, pop @$aref;
```

```
        }
        return @retlist;
}
```

下面是一个函数，能够返回一个列表，这个列表包含在每个传入的散列结构中都出现的键字.

```
@common = inter (\%foo, \%bar, \%joe );
sub inter {
        my %seen;
        for my $href (@_) {
                while (my $k = each %$href ) {
                        $seen{$k}++;
                }
        }
        return grep { $seen{$_} == @_ } keys %seen;
}
```

这里我们只用了普通的列表返回机制.当你想传送或返回一个散列结构时会发生什么？如果你仅用其中的一个，或者你不在意它们连在一起，那么使用普通的调用方法就行了，如果不是，那么就会稍微复杂一些.

我们已经在前面提到过，人们常会在下面的写法中遇到麻烦：

```
(@a, @b) = func(@c, @d);
```

或这里：

```
(%a, %b) = func(%c, %d);
```

这些表达式将不会正确工作，它只会设置 @a 或%a，而 @b 或 %b 则是空的.另外函数不会得到两个分离的数组和散列结构作为参数：它和往常一样从 @_ 中得到一个长列表.

你也许想在函数的输入和输出中都使用引用. 下面是一个使用两个数组引用作为参数的函数，并且根据数组中包含元数的多少为顺序返回两个数组的引用：

```
($aref, $bref) = func(\@c, \@d);
print "@$aref has more than @$bref\n";
sub func {
        my ($cref, $dref) = @_;
        if (@$cref > @$dref) {
                return ($cref, $dref);
        } else {
                return ($dref, $cref);
        }
}
```

如何向函数传入或传出文件句柄或目录句柄，请参阅第八章的"文件句柄引用"和"符号表句柄"小节.

4.0 函数原型

Perl 可以让你定义你自己的函数，这些函数可以象 Perl 的内建函数一样调用. 例如 push(@array, $item)，它必须接收一个 @array 的引用，而不仅仅是 @array 中的值，这样这个数组才能够被函数改变. 函数原型能够让你声明的子过程能够象很多内建函数一样获得参数，就是获得一定数目和类型的参数. 我们虽然称之为函数原型，但是它们的运转更像调用环境中的自动模板，而不仅仅是 C 或 java 程序员认为的函数原型. 使用这些模板，Perl 能够自动添加隐含的反斜扛或者调用 scalar，或能够使事情能变成符合模板的其他一些操作. 比如，如果你定义：

```
sub mypush (\@@);
```

那么 mypush 就会象 push 一样接受参数. 为了使其运转，函数的定义和调用在编译的时候必须是可见的.函数原型只影响那些不带 & 方式调用的函数. 换句话说，如果你象内建函数一样调用它，它就像内建函数一样工作. 如果你使用老式的方法调用子过程，那么它就象老式子过程那样工作. 调用中的 & 消除所有的原型检查和相关的环境影响.

因为函数原型仅仅在编译的时候起作用，自然它对象 \&foo 这样的子过程引用和象 &{$subref} 和 $subref->() 这样的间接子过程调用的情况不起作用. 同样函数原型在

方法调用中也不起作用. 这是因为被调用的实际函数不是在编译的时候决定的，而是依赖于它的继承，而继承在 Perl 中是动态判断的.

因为本节的重点主要是让你学会定义象内建函数一样工作的子过程，下面使一些函数原型，你可以用来模仿对应的内建函数：

声明为	调用
sub mylink ($$)	mylink $old, $new
sub myreverse (@)	myreverse $a, $b, $c
sub myjoin ($@)	myjoin ":", $a, $b, $c
sub mypop (\@)	mypop @array
sub mysplice(\@$$@)	mysplice @array, @array, 0, @pushme
sub mykeys (\%)	mykeys %($hashref)
sub mypipe (**)	mypipe READHANDLE, WRITEHANDLE
sub myindex ($$;$)	myindex &getstring, "substr"
	myindex &getstring, "substr", $start
sub mysyswrite (*$;$$)	mysyswrite OUTF, $buf
	mysyswrite OUTF, $buf, length($buf)-$off, $off
sub myopen (*;$@)	myopen HANDLE
	myopen HANDLE, $name
	myopen HANDLE, "-	", @cmd
sub mygrep (&@)	mygrep { /foo/ } $a, $b, $c
sub myrand ($)	myrand 42
sub mytime ()	mytime

任何带有反斜扛的原型字符(在上表左列中的圆括弧里)代表一个实际的参数(右列中有示例)必须以以这个字符开头.例如 keys 函数的第一个参数必须以 % 开始，同样 mykeys 的第一个参数也必须以 % 开头.

分号将命令性参数和可选参数分开. (在 @ 或 % 前是多余的，因为列表本身就可以是空的) . 非反斜扛函数原型字符有特殊的含义. 任何不带反斜扛的 @ 或 % 会将实际参数所有剩下的参数都吃光并强制进入列表环境. (等同于语法描述中的 LIST). $ 代表的参数强迫进入标量环境. & 要求一个命名或匿名子过程的引用.

函数原型中的 * 允许子过程在该位置接受任何参数，就像内建的文件句柄那样：可以是一个名字，一个常量，标量表达式，类型团或者类型团的引用. 值将可以当成一个简单的标量

或者类型团（用小写字母的）的引用由子过程使用. 如果你总是希望这样的参数转换成一个类型团的引用，可以使用 Symbol：：qualify_to_ref，象下面这样：

```
use Symblo ’qualify_to_ref’;


sub foo (*) {
        my $fh = qualify_to_ref(shift, caller);
        ...
}
```

注意上面表中的最后三个例子会被分析器特殊对待，mygrep 被分析成一个真的列表操作符，myrand 被分析成一个真的单目操作符就象 rand 一样，同样 mytime 被分析成没有参数，就象 time 一样.

也就是说，如果你使用下面的表达式：

```
   mytime +2;
```

你将会得到 mytime()+2，而不是 mytime(2)，这就是在没有函数原型时和使用了单目函数原型时分析的得到的不同结果.

mygrep 例子同样显示了当 & 是第一个参数的时候是如果处理的. 通常一个 & 函数原型要求一个象 \&foo 或 sub{} 这样参数. 当它是第一个参数时，你可以在你的匿名子过程中省略掉 sub，只在"非直接对象"的位置上传送一个简单的程序块(不带冒号). 所以 & 函数原型的一个重要功能就是你可以用它生成一个新语法，只要 & 是在初始位置：

```
sub try (&$) {
        my ($try, $catch) = @_;
        eval { &$try };
        if ($@) {
                local $_ = $@;
                &$catch;
```

```
        }
}
sub catch (&) { $_[0] }

try {
        die "phooey";
}         # 不是函数调用的结尾!
catch {
        /phooey/ and print "unphooey\n";
};
```

它打印出 "unphooey". 这里发生的事情是这样的，Perl 带两个参数调用了 try，匿名函数 {die "phooey";} 和 catch 函数的返回值，在本例中这个返回值什么都不是，只不过是它自己的参数，而整个块则是另外一个匿名函数. 在 try 里，第一个函数参数是在 eval 里调用的，这样就可以捕获任何错误. 如果真的出了错误，那么调用第二个函数，并且设置 $_ 变量以抛出例外. （注：没错，这里仍然有涉及 @_ 的可视性的问题没有解决. 目前我们忽略那些问题. 但是如果我们将来把 @_ 做成词法范围的东西，就象现在试验的线程化 Perl 版本里已经做的那样，那么那些匿名子过程就可以象闭合的行为一样. ）如果你觉得这些东西听起来象胡说八道，那么你最好看看第二十九章里的 die 和 eval，然后回到第八章里看看匿名函数和闭合. 另外，如果你觉得麻烦，你还可以看看 CPAN 上的 Error 模块，这个模块就是实现了一个用 try，catch，except，otherwise，和 finally 子句的灵活的结构化例外操作机制.

下面是一个 grep 操作符的重新实现(当然内建的实现更为有效)：

```
sub mygrep (&@) {
        my $coderef = shift;
        my @result;
        foreach $_ (@_) {
                push(@result, $_) if &$coderef;
        }
```

```
        return @result;
}
```

一些读者希望能够看到完整的字母数字函数原型. 我们有意把字母数字放在了原型之外，为的是将来我们能够很快地增加命名的，正式的参数. （可能）现在函数原型的主要目的就是让模块作者能够对模块用户作一些编译时的强制参数检查.

4.1 内联常量函数

带有 () 的函数原型表示这个函数没有任何参数，就象内建函数 time 一样. 更有趣的是，编译器将这种函数当作潜在的内联函数的候选函数. 当 Perl 优化和常量值替换回合后，得到结果如果是一个固定值或者是一个没有其他引用的语法作用域标量时，那么这个值就将替换对这个函数的调用. 但是使用 &NAME 方式调用的函数不被"内联化"，然而，只是因为它们不受其他函数原型影响. (参看第三十一章"用法模块"中的 use constant，这是一种定义这种固定值的更简单的方法).

下面的两种计算 ∏ 的函数写法都会被编译器"内联化"：

```
sub pi () { 3.14159 }          # 不准确，但接近
sub PI () { 4 * atan2(1, 1) }      # 和它的一样好
```

实际上，下面所有的函数都能被 Perl "内联化"，因为 Perl 能够在编译的时候就能确定所有的值：

```
sub FLAG_FOO ()     { 1 << 8 }
sub FLAG_BAR ()     { 1 << 9 }
sub FLAG_MASK ()    { FLAG_FOO | FLAG_BAR }

sub OPT_GLARCH ()   { (0x1B58 & FLAG_MASK) == 0 }
sub GLARCH_VAL ()   {
        if (OPT_GLARCH) { return 23 }
        else            { return 42 }
}
```

```
sub N () { int(GLARCH_VAL) / 3 }

BEGIN {                 # compiler runs this block at compile time

        my $prod = 1;       # persistent, private variable

        for (1 .. N) { $prod *= $_ }

        sub NFACT () { $prod }

}
```

最后一个例子中，NFACT 函数也将内联化，因为它有一个空的函数原型并且函数返回的变量并没有被函数修改，而且不能被其他东西改变，因为它在一个语法作用范围里面．因此编译器在编译的时候预先计算它的值，并用这个值替换所有使用 NFACT 的地方．

如果你重新定义已经被内联化的子过程，那么你会收到一个命令性警告(你可以使用这个警告来确认一个子过程是不是已经被内联化了)因为重新定义的子过程会用先前编译产生的值代替，因此这个警告足够的确定这个子过程是否被内联化．如果你需要重新定义子过程，你可以通过删除 () 函数原型(这个更改调用方法)或者重新修改函数的写法来阻挠内联化机制来避免子过程被内联化．例如：

```
sub not_inlined () {

        return 23 if $$;

}
```

参看第十八章学习更多有关程序编译和执行阶段的知识．

4.2 谨慎使用函数原型

最好在新函数中使用函数原型，而不在旧函数中使用函数原型．Perl 中函数原型是环境模板，而不象 ANSI C 中的函数原型，因此你必须十分注意函数原型是否将你的子过程带入了一个新的环境．例如，你想写一个只有一个参数的函数，象下面这个函数：

```
sub func ($) {

        my $n = shift;

        print "you gave my $n\n";
```

```
}
```

这将得到一个单目操作符(象 rand 内建函数)并且改变了编译器确定函数参数的方法. 使用了新的函数原型，该函数就只使用一个标量环境下的参数，而不是在列表环境下的多个参数.如果你在以数组或者列表表达式中调用这个函数,即使这个数组或列表只包含一个元素，你可能会得到完全不同的结果：

```
func @foo;       # 计算 @foo 元素个数

func split /:/;       # 计算返回的域的个数

func "a", "b", "c";   # 只传递 "a"，抛弃 "b" 和 "c"

func("a", "b", "c");   # 马上生成一个编译器错误!
```

你已经隐含地在参数列表前面提供了一个 scalar，这的确令人有点吃惊. 如果 @foo 只包含一个元素，那么传递给函数不是这个元素，而是 1(@foo 的元素个数). 并且在第二个例中，split 在标量环境中被调用，吞没你的整个 @_ 参数列表. 在第三个例子中，因为 func 已经用函数原型定义为一个单目操作符，因此只有 "a" 传递给了 func；然后 func 返回值被丢弃,因为逗号操作符的存在因此继续处理下两个元素并返回 "c".最后一个例子，在编译的时候用户将得到一个语法错误.

如果你想写一个新的代码得到一个只使用一个标量参数的单目操作符,而不是任何旧的标量表达式，你可以使用下面的函数原型使它使用标量引用：

```
sub func (\$) {

        my $nref = shift;

        print "you gave me $$nref\n";

}
```

现在，编译器可以让下面的例子中，参数以 $ 开头的通过：

```
func @foo;       # 编译器错误，看见了 @，但要的是 $

func split/:/;       # 编译器错误，看见了函数，但要的是 $

func $s;          # 这个是对的 -- 获取了真的 $ 符号

func $a[3];       # 这个也对
```

```
func $h{stuff}[-1];    # 这个也对

func 2+5;          # 标量表达式也会导致编译器错误

func ${\(2+5) };         # 对，不过它是不是比病毒还糟糕？
```

如果你不小心，你可能因为使用函数原型遇到很多麻烦. 但如果你非常注意，你可以使用函数原型来作很多漂亮的工作. 函数原型是非常强大的，当然需要谨慎使用才能得到好的结果.

5.0 子过程属性

子过程的定义和声明能够附带一些属性. 如果属性列表存在，它使用空格或者冒号分割，并等同于通过 use attributes 定义的一样. 请阅读三十一章的 use attributes 获得内部细节. 有三个标准的子过程属性：locked， method 和 左值.

5.1 Locked 和 method 属性

```
# 在这个函数里只允许一个线程

sub afunc : locked { ... }


# 在一个特定的对象上之允许一个线程进入这个函数

sub afunc : locked method { ... }
```

只有在子过程或者方法要被多个线程调用的时候，设置 locked 属性才有意义. 当设置一个不是方法的子过程的时候，Perl 确保在进入子过程之前获得一个锁. 当设置一个方法子过程时(具有 method 属性的子过程)，Perl 确保在执行之前锁住它的第一个参数(所属的对象).

method 属性能够被它自己使用：

```
sub afunc : method { ... }
```

现在它只是用来标记子过程，使之不产生 "Ambiguous call resolved as CORE：：%s" 警告. (我们以后可以给它更多的含义).

属性系统是用户可扩展的，Perl 可以让你创建自己的属性名. 这些新的属性必须是简单的标记名字(除了 "_" 字符之外没有任何标点符号). 它们后边可以有一个参数列表用来检查它的花括弧是否匹配正确.

下面是一些正确的语法的例子(即使这些属性是未知的)：

```
sub fnord (&\%) : switch(10, foo(7,3)) : expensive;

sub plugh () : Ugly('\(") :Bad;

sub xyzzy : _5x5 { ... }
```

下面是一些不正确语法的例子：

```
sub fnord : Switch(10, foo());   # ()-字串不平衡

sub snoid : Ugly ('(');      # ()-字串不平衡

sub xyzzy : 5x5;      # "5x5" 不是合法的标识符

sub plugh : Y2::north;      # "Y2::north"不是简单标识符

sub snurt : foo + bar;      # "+" 不是一个冒号或空格
```

属性列表作为一个常量字符串列表传递进子过程相关的代码.它的正确工作方法是高度试验性的.查阅 attributes(3) 获得属性列表的详细信息和操作方法.

5.3 左值属性

除非你定义子过程返回一个 左值，否则你你不能从子过程中返回一个可以修改的标量值：

```
my $val;
sub canmod : 左值 {
        $val;
}
sub nomod {
        $val;
}


canmod() = 5;      # 给 $val 赋值为 5

nomod()  = 5;       # 错误
```

如果你正传递参数到一个有 左值 属性的子过程，你一般会使用圆括弧来防止歧义：

```
canmod $x = 5;      # 先给 $x 赋值 5!
canmod 42 = 5;      # 无法改变常量，编译时错误
canmod($x)= 5;      # 这个是对的
canmod(42)= 5;      # 这个也对
```

如果你想使用省略的写法，你可以在子过程只使用一个参数的情况下省略圆括弧. 使用 ($) 函数原型定义一个函数可以使该函数被解释为一个具有命名的单目操作符优先级的操作符. 因为命名单目操作符优先级高于赋值，所以你不再需要圆括弧(需不需要圆括弧只是一个代码风格的问题).

当一个子过程允许空参数时(使用 () 函数原型)，你可以使用下面的方法而不会引起歧义：

```
canmod = 5;
```

因为没有哪个合法项以 = 开头，因此它能正确工作. 同样，具有左值属性的方法调用在不传送任何参数时也能省略圆括弧：

```
$obj->canmod = 5;
```

我们保证在未来的 Perl 版本中不改变上面的两种方法. 当你希望在方法调用中封装对象属性时，它们是非常简便的方法(因此它们可以象方法调用一样被继承但又象变量一样访问).

左值子过程和子过程的赋值表达式右边部分可以通过使用标量替换子过程的方法，来确定是标量环境还是列表环境.例如：

```
data(2,3) = get_data(3,4);
```

上边两个子过程都在标量环境中调用，而在：

```
(data(2,3)) = get_data(3,4);
```

和：

```
(dat(3), data(3) = get_data(3,4);
```

中，所有的子过程在列表环境中被调用.

在当前的实现中不允许从左值子过程直接返回数组和散列结构.不过你总是可以返回一个引用来解决这个问题.

第七章 格式

Perl 有一个机制帮助你产生简单的报告和图表. 为了实现这个机制，Perl 帮助你格式化你的输出，使它打印出来的时候看起来比较接近于你想要的结果. 它能保持跟踪象一页里面有多少行，当前的页码，以及什么时候打印页头等等的东西. 使用的关键字是从 FORTRAN 里面借来的：format 用来声明而 write 用来执行；参看第二十九章，函数，获取相关内容. 所幸，布局时非常易读的，很象 BASIC 中的 PRINT USING 语句. 也可以将它想象成 nroff(如果你知道 nroff，这也许不象是一个比较).

格式输出，和包和子过程一样，是声明而不是执行，因此它们可以在你的程序中任何地方出现. (通常最好将所有的格式输出放在一起). 它们有他们自己的名字空间，与 Perl 中其它类型的名字空间是区分开来的. 这就是说如果你有一个函数 "Foo"，但它不同于一个名字为 "Foo" 的格式输出. 然而和一个文件句柄相关联的格式输出的缺省名字和该文件句柄的名字相同. 因而，STDOUT 的缺省格式输出的名字是 "STDOUT"，文件句柄 TEM P 的缺省格式输出名字为 "TEMP"，它们看起来是一样的，实际上是不一样的.

输出纪录格式输出象下边一样定义：

```
format NAME =
FORMLIST
.
```

如果省略 NAME，将定义格式输出 STDOUT. FORMLIST 由一些有序的行组成，每一行都是下面三种类型中的一种：

1. 注释，以第一列为 # 来表示.
2. 一个格式行，用来定义一个输出行的格式
3. 参数行，用来向前面的格式行中插入值

格式行除了那些需要被替换的部分外，严格按照它们的声明被输出.（注：而且，甚至那些你放进去维护列完整性的域也如此.在一个格式行中没有任何东西可以导致域的伸缩或者移位. 你看到的列是按照 WYSIWYG 的概念分布的---假设你用的是定宽字体. 就连控制字符都假设是宽度为一的字符.）格式行中每个被替换的部分分别以 @ 或者 ^ 开头. 这些行不作任何形式的变量代换. @ 域(不要同数组符号 @ 相混淆)是普通的域. 另一种域，^ 域用来进行多行文本块填充. 域的长度通过在格式符号 @，^ 后跟随特定长度的 <，>，| 来定义，同时，<，>，| 还分别表示，左对齐，右对齐，居中对齐. 如果变量超出定义的长度，那么它将被截断.

作为右对齐的另外一种方式，你可以使用 #(在 @ 或 ^ 后边)来指定一个数字域. 你可以在这种区域中插入一个 . 来制定小数点的位置. 如果这些区域的值包含一个换行符，那么只输出换行符前面的文本. 最后，特殊的域 @* 可以被用来打印多行不截断的值；这种区域通常在一个格式行中出现.

参数行指定参数的顺序必须跟相应的格式行的域顺序一致.不同参数的表达式需要使用逗号分隔. 参数行被处理之前所有的参数表达式都在列表环境中求值，因此单个列表表达式会产生多个列表元素. 通过使用圆括弧将表达式括起来，可以使表达式扩展到多行 (因此，圆括弧必须是第一行的第一个标志). 这样就可以将值同相应的格式域对应起来方便阅读.

如果一个表达式求出的值是一个有小数部分的数字，并且如果对应的格式域指定了输出的小数部分的格式(除了没有内嵌 . 的多个 # 字符的格式)，用来表示小数点的字符总是由 LC_NUMERIC 区域参数确定. 这就是，如果运行时环境恰好是德国本地化参数，一个逗号总是用来替代句点. 参看 perllocale 手册页获取更多的内容.

在一个表达式中，空白字符 \n，\t，和 \f 总是被解释成单个空格. 因而，你可以认为这样的过滤表达式作用于每个格式中的表达式：

```
$value =~ tr/\n\t\f/ /;
```

余下的空白字符，\r， 如果格式行允许的话，将强制输出一个换行符.

以 ^ 开头的格式域不同于 @ 格式域，它会被特殊对待. 例如一个 # 区域，如果值没有定义，那么这个区域将变为空白. 对于其他的区域类型，^ 会使用一种特殊的填充模式. 提供的值必须是一个包含字符串的标量变量名，而不是一个强制表达式. Perl 在这个区域中放入尽可能多的文本，并且将已经打印过的字符截去，这样当下次引用该变量的时候，就可以打印更多的文本. (这就是说在 write 调用过程中，变量本身将发生变化，原来的值将不

被保留．因此如果你想保持最初的值，你需要使用一个临时变量来代替原来的变量)．通常你应该使用一组垂直对齐的格式区域打印一块文本.你也许会想用文本 "..." 来结束最后的区域，这样当文本太长不能完整地打印的时候，指定的文本将会被打印．你也可以改变变量 $：(当你使用 English 模块，那么就是 $FORMAT_LINE_BREAK_CHARACTERS)来改变用来表示中断的合法字符.

使用 ^ 区域能够产生变长的纪录．如果格式区域太短，你可以重复几次带有 ^ 区域的格式行．如果你用这个方法处理一块较短的数据，那么你将会得到几个空白输出行．为了避免空白行，你可以在格式行中的任意地方放置一个 ~ (波浪号)．(在输出中波浪号本身会被转换成一个空格)．如果你使用第二个 ~ 波浪号，该格式行会被重复直到在该格式行中所有域中的文本被用尽为止．(因为 ^ 区域会吃掉所要打印的字符串，因此前面的格式行能够运行，但是如果你使用和两个波浪号结合的一个 @ 域，你最好每次给这表达式不同的值！使用 shift 或者其他带有副作用的操作符，来用尽所有的值．)

标头的处理缺省使用当前文件句柄名加上 _TOP 后缀的格式来处理.它在每一页的开头被触发．参看二十九章的 write.

```
# a report on the /etc/passwd file

format STDOUT_TOP =

                        Passwd File

Name                Login    Office   Uid   Gid Home

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

.

format STDOUT =

@<<<<<<<<<<<<<<<<<< @|||||||| @<<<<<<@>>>> @>>>> @<<<<<<<<<<<<<<<<<<

$name,              $login,  $office,$uid,$gid, $home

.


# a report from a bug report form

format STDOUT_TOP =

                        Bug Reports
```

```
@<<<<<<<<<<<<<<<<<<<<<<<<     @|||          @>>>>>>>>>>>>>>>>>>>>
$system,                      $%,          $date
------------------------------------------------------------------

.

format STDOUT =
Subject: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
         $subject
Index: @<<<<<<<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
       $index,                        $description
Priority: @<<<<<<<<<< Date: @<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
          $priority,        $date,    $description
From: @<<<<<<<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
      $from,                         $description
Assigned to: @<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
             $programmer,            $description
~                                    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                     $description
~                                    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                     $description
~                                    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                     $description
~                                    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                     $description
~                                    ^<<<<<<<<<<<<<<<<<<<<<<<...
```

```
$description
```

.

除非一个格式是在词法变量的作用范围内定义，否则该词法变量在格式中是不可见的.

在同一个输出通道中将 print 和 write 混合起来是可以的，但是你必须自己操作特殊变量 $- (在 English 模块中是 $FORMAT_LINES_LEFT).

7.1 格式变量

当前的格式名字存储在 $~ 中 ($FORMAT_NAME)，当前的表头格式名字存储在 $^ ($FORMAT_TOP_NAME). 当前输出的页号在 $% ($FORMAT_PAGE_NUMBER)，每页中的行数在 $= ($FORMAT_LINES_PER_PAGE).是否自动刷新输出缓冲区存储在 $| ($FORMAT_AUTOFLUSH). 在每一页(除了第一页)表头之前需要输出的字符串存储在 $^L ($FORMAT_FORMFEED). 这些变量以文件句柄为基础设定，因此你需要 select 与特定格式关联的文件句柄来影响这些格式变量：

```
select((select(OUTF),
    $~ = "My_Other_Format",
    $^ = "My_Top_Format"
    )[0]);
```

是不是很难看？可是这是一个习惯用法，因此当你看见它时不要感到惊讶.你至少可以使用一个临时变量来保持前一个文件句柄：

```
$ofh = select(OUTF);
$~   = "My_Other_Format";
$^   = "My_Top_Format";
select($ofh);
```

通常这是一个更好的方式，因为这不仅仅是增加了可读性， 但是你现在在代码有了一个中间语句，这样你可以在单步调试的时候可以在这里停下来，如果你使用 English 模块，你甚至可以这样读取变量名字：

```
use English;
$ofh = select(OUTF);
```

```
$FORMAT_NAME     = "My_Other_Format";
$FORMAT_TOP_name = "My_Top_Format";
select($ofh);
```

但是你仍然要调用这些 select，如果你想避免使用他们，使用 Perl 集成的 FileHandle[2] 模块．现在你就可以使用小写的方法名来访问这些特殊变量：

```
use FileHandle;
OUTF->format_name("My_Other_Format");
OUTF->format_top_name("My_Top_Format");
```

这样看起来更好！

因为跟在格式行后面的数值行可以包含任意的表达式(提供给 @ 域，而不是 ^ 域)，所以你可以使用一些高级的处理，象 sprintf 或者一个你自己的函数．例如为了在一个数字里面插入一些逗号，你可以使用下面的方法：

```
format Ident =
    @<<<<<<<<<<<<<<<
    commify($n)
.
```

为了在格式区域的实际输出中得到一个真的 @，~ 或者 ^，可以象下面一样：

```
format Ident =
I have an @ here.
         "@"
.
```

将整行文本居中，可以使用下面的方法：

```
format Ident =
@||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

```
"Some text line"
.
```

> *域长度标识保证在域中的文本将是右对齐，但只是在你定义的域内精确地对齐. Perl 中*没有一种内置的方法可以获得 "将这个区域在页右侧对齐，而不管它的宽度" 这样的效果，你必须指定用于对齐的左侧位置. 你可以基于当前列号(不提供)来产生它们自己的格式，来达到上面的目的，然后 eval 它：

```
$format  = "format STDOUT = \n"
         . '^' . '<' x $cols . "\n"
         . '$entry' . "\n"
         . "\t^" . "<" x ($cols-8) . "~~\n"
         . '$entry' . "\n"
         . ".\n";
print $format if $Debugging;
eval $format;
die $@ if $@;
```

上边最重要的行恐怕就是 print. print 将打印出下边这样的格式：

```
format STDOUT =
^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
$entry
    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<~~
$entry
.
```

这里有一个小程序来达到 fmt(1) Unix 工具的功能：

```
format =
```

```
^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ~~
$_

.

$/ = "";
while (<>) {
    s/\s*\n\s*/ /g;
    write;
}
```

7.2 页脚

$^ ($FORMAT_TOP_NAME) 包含了当前表头格式的名称，目前没有对应的机制来自动获得页脚的定义．除非你认为它是一个主要问题，否则你不会知道一个格式将会有多大．它现在已经在我们的 TODO 列表里面．（注：不过这并不能保证我们就一定会做它．格式在 WWW， Unicode，XML，XSLT，和任何它们后来的事物占统治地位的时代显得有些过时了．）

这里由一个策略：如果你有一个给定尺寸的页脚，你可以在每个 write 之前通过检查 $- ($FORMAT_LINES_LEFT) 来获得页脚，然后你自己打印页脚．

也有另一个策略：使用 open(MESELF， "|-") 打开一个指向你自己的管道并且总是 write 到 MESELF 而不是 STDOUT．让你的子过程处理它的 STDIN 来重新安排页头和页脚．这种方法不是很方便，但是的确可以运行．

7.2.1 访问格式的内部

为了对内部格式化机制进行低级访问，你可以使用内建的 formline 操作符和直接访问 $^A ($ACCUMULATOR 变量)．(格式最终编译成为一系列的对 formline 的调用)例如：

```
$str = formline <<'END', 1,2,3;
@<<<  @|||  @>>>
```

```
END


print "Wow, I just stored `$^A' in the accumulator!\n";
```

或者创建一个 swrite 子过程，它对于 write 的作用就像 sprintf 对 printf 的作用，可以象下边的代码一样使用：

```
use Carp;
sub swrite {
        croak "usage: swrite PICTURE ARGS" unless @_;
        my $format = shift;
        $^A = "";
        formline($format, @_);
        return $^A;
}


$string = swrite(<<'END', 1, 2, 3);
Check me out
@<<<  @||| @>>>
END
print $string;
```

如果你在使用 FileHandle[2] 模块，你可以使用象下边代码一样使用 formline，使一块文本在第 72 列处折行.

```
   use FileHandle;
   STDOUT->formline("^" . ("<" x 72 ) . "~~\n", $long_text);
```

第八章 引用

不管是从理论还是实践的角度出发，Perl 都是偏爱平面线性的数据结构的。并且对许多问题来说，这些也就是你所要的东西。

假设你想制作一个简单的表（二维数组），为一组人员显示生命数据用——包括年龄，眼睛颜色，和重量等。你可以通过先给每个独立的成员创建一个数组来实现这个目的。

```
@john = (47, "brown", 186);

@mary = (23, "hazel", 128);

@bill = (35, "blue",  157);
```

然后你就可以构造一个附加的数组，该数组由其他数组的名字组成：

```
@vitals = ('john', 'mary', 'bill');
```

在小镇上过了一夜之后，为了把 John 的眼睛变成"红色"（"red"），我们需要一个仅仅通过使用字串"john"就可以改变数组 @john 的内容的方法。这就是间接的基本问题，而不同的语言是用不同的方法来解决这个问题的。在 C 里，间接的最常见的形式就是指针，它可以让一个变量保存另外一个变量的内存地址。在 Perl 里，间接的最常见的形式是引用。

8.1 什么是引用？

在我们的例子里，$vitals[0] 的值是"john"。也就是说它正好包含另外一个（全局）变量的名字。我们说第一个变量提到了第二个变量，并且这种参考叫符号引用，因为 Perl 必须在一个符号表里找出 @john 来才能找到它。（你可以把符号引用看作类似文件系统中的符号联接的东西）。我们将在本章晚些时候讨论符号引用。

另外一种引用是硬引用，这种引用是大多数 Perl 程序员用来实现它们的间接访问方法（只要不是他们的草率行为）。我们叫它们硬引用并不是因为它们用起来很难（译注："hard reference" 硬引用，在英文中"hard"有"困难，硬"的意思），而是因为它们是现实并且存在的。如果你愿意，那么你可以把硬引用当作真正的引用而把符号引用当作虚假的引用。它们的区别就好象真正的友谊和见面打个招呼一样。如果我们没有声明我们指的是哪种引用，

那么我们说的就是硬引用。图 8-1 描述了一个叫 $bar 的变量引用了一个叫 $foo 的变量的内容，而 $foo 的值是“bot”。

和符号引用不同的是，真实引用所引用的不是另外一个变量的名字（名字只是一个数值的容器），而是实际的数值本身，是一些内部的数据团。我们没有什么好字眼来描述这样的东西，可是我们又不得不描述，于是我们就叫它引用。举例来说，假如你创建了一个指向某个词法范围数组 @array 的硬引用。那么，即使在 @array 超出了范围之后，该引用以及它所引用的参考物也仍然继续存在。一个引用只有在对它的所有引用都消失之后才会被摧毁。

除了指向它的引用之外，引用实际上并没有自己的名字。换句话说，每个 Perl 变量都存储在某个符号表里，保存着一个指向所引用的东西的硬引用（否则就没有名字）。引用物可以很简单，比如一个数字或者字串，也可以很复杂，比如一个数组或散列。不管哪种情况，从变量到数值之间都只有一个引用。你可以创建指向相同引用物的额外的引用，但该变量并不知道（或在乎）这些引用。（注：如果你觉得奇怪，那么你可以用 Devel::Peek 模块计算引用计数，这个包是和 Perl 捆绑发布的。）

符号引用只是一个字串，它的值碰巧和包的符号表里什么东西的名字相同。它和你平时处理的字串没有什么太大的区别。但是硬引用却是完全不同的家伙。它是三种基本的标量类型中的第三种，其他两种是字串和数字。硬引用除了指向某些事物之外并不知道它们的名字，并且这些引用物在一开始的时候并没有名字也是非常正常的事情。这样的未名引用叫做匿名，我们将在下面的“匿名数据”里讨论它们。

在本章的术语里，引用一个数值就是创建一个指向它的硬引用。（我们有一个操作符用于这种创建动作）。这样创建的引用只是一个简单的标量，它和所有其他标量一样在我们熟悉的环境里有着一样的行为。给这个标量解引用（析引用）意味着我们使用这个引用访问引用物。引用和解引用都是只发生在某些明确的机制里，在 Perl 里从来不会出现隐含地引用或解引用的现象。哦，是几乎从来不发生。

一次函数调用可以使用明确的引用传参语意——只要它有这样声明的原型。如果是这样，那么函数的调用者并不明确地传递一个引用，但是你在函数里面还是要明确的对它（参数）进行解引用。参阅第六章，子过程，里的“原型”节。如果从绝对诚实的角度出发，那么你在使用某些类型的文件句柄的时候仍然是有一些隐藏在幕后的解引用发生，但是这么做是为了保持向下兼容，并且对于那些偶然使用到的用户来说是透明的。最后，有两个内建的函数，bless 和 block，它们都接受一个引用作为自己的参数，但是都会隐含地对这些引用进行解引用以实现各自的功能。不过除了我们这里招供的这些以外，基本的概念还是一致的，那就是 Perl 并不想介入你自己的间接层次中。

一个引用可以指向任何数据结构。因为引用是标量，所以你可以把它们保存在数组和散列里，因此我们就可以做出数组的数组，散列的数组，数组的散列，散列和函数的数组等。在第九章，数据结构，里有这些东西的例子。

不过，你要记住的是，Perl 里的数组和散列都是故意作成一维的。也就是说，它们的元素只能保存标量值（字串，数字，和引用）。当我们使用"数组的数组"这样的习语的时候，我们的意思实际上是"一个保存指向一些数组的引用的数组"，就好象我们说"函数散列"的时候，我们实际上是说"一个保存着一些指向子过程的引用的散列"。但因为引用是在 Perl 里实现这些构造的唯一方法，所以我们那些稍微短的并非准确的习语也就并不是完全不对，因此也不应该完全忽视，除非你碰到准确性问题。

8.2 创建引用

创建引用的方法有好多种，我们在讲述它们的时候大多会先描述它们，然后才解释如何使用（解引用）所生成的引用。

8.2.1 反斜杠操作符

你可以用一个反斜杠创建一个指向任何命名变量或者子过程的引用。（你还可以把它用于一个匿名标量值，比如 7 或 "camel"，尽管你通常并不需要这些东西。）乍一看，这个操作符的作用类似 C 里的 &（取址）操作符。

下面是一些例子：

```
$scalarref = \$foo;

$constref  = \186_282.42;

$arrayref  = \@ARGV;

$hashref   = \%ENV;

$coderef   = \&handler;

$globref   = \*STDOUT;
```

反斜杠操作符可以做的事情远远不止生成一个引用。如果你对一个列表使用反斜杠，那么它会生成一整列引用。参阅"你用硬引用可以实现的其他技巧"一节。

8.2.2 匿名数据

在我们刚刚显示的例子里，反斜杠操作符只是简单地复制了一个已经存在于一个命名变量上的引用 ——但有一个例外。186_282.42 不是一个命名变量的引用——它只是一个数值。它是那种我们早先提到过的匿名引用。匿名引用物只能通过引用来访问。我们这个例子碰巧是一个数字，但是你也可以创建匿名数组，散列，和子过程。

8.2.2.1 匿名数组组合器

你可以用方括弧创建一个创建一个指向匿名数组的引用：

```
$arrayref = [1, 2, ['a', 'b', 'c', 'd']];
```

在这里我们组合成了一个三个元素的匿名数组，该数组最后一个元素是一个指向有着四个元素的匿名数组（在图 8-2 里演示）。（我们稍后描述的多维语法可以用于访问这些东西。比如， $arrayref->[2][1] 将具有数值"b"。）

现在我们有一个方法来表示我们本章开头的表：

```
$table = [ [ "john", 47, "brown", 186],
           [ "mary", 23, ""hazel", 128],
           [ "bill", 35, "blue", 157] ];
```

只是当 Perl 分析器在一个表达式里需要项的时候，方括弧才可以这样运做。你可不要把它们和表达式里的方括弧混淆起来——比如 $array[6]——尽管与数组的记忆性关联是有意为之的。在一个引起的字串里，方括弧并不组成匿名数组；相反，它们成为字串里的文本字符。（在字串里方括弧的确仍然可以当作脚标使用，否则你就不能打印象 "VAL=$array[6]\n" 这样的字串。如果要我们绝对诚实，你实际上是可以偷偷地把匿名数组组合器放到字串里，但只能是在它被潜入到一个更大的表达式中，并且该表达式被代换的情况下才可以实现。我们将在本章稍后讲述这个酷酷的特性，因为它既包括引用也包括解引用。）

8.2.2.2 匿名散列组合器

你可以用花括弧创建一个指向匿名散列的引用：

```
$hashref = {
        'Adam' => 'Eve',
        'Clyde' => $bonnie,
```

```
        'Antony' => 'Cleo' . 'patra',
};
```

对于散列的数值（但不是键字），你可以自由地混合其他匿名数组，散列，和子过程，组合成你需要的复杂数据结构。

现在我们有了表示本章开头的表的另外一种方法：

```
$table = {
        "john" => [47, "brown", 186],
        "mary" => [23, "hazel", 128],
        "bill" => [35, "blue", 157],
};
```

这是一个数组散列。选择最好的数据结构是难度很高的工种，下一章专门讲这个。但是为了恶作剧，我们甚至可以将散列的散列用于我们的表：

```
$table = {
        "john" => { age    => 47,
                    eyes   => "brown",
                    weight => 186,
        },
        "mary" => { age    => 23,
                    eyes   => "hazel",
                    weight => 128,
        },
        "bill" => { age    => 35,
                    eyes   => "blue",
                    weight => 157,
        },
```

```
};
```

和方括弧一样，只有在 Perl 分析器在表达式里需要一个项的时候，花括弧才运做。你也不要把它和在表达式里的花括弧混淆了，比如 $hash{key}——尽管与散列的记忆性关联（也）是有意为之的。同样的注意事项也适用于在字串里的花括弧。

但是有另外一个注意事项并不适用于方括弧。因为花括弧还可以用于几种其他的东西（包括块），有时候你可能不得不在语句的开头放上一个 + 或者一个 return 来消除这个位置的花括弧的歧义，这样 Perl 就会意识到这个开花括弧不是引出一个块。比如，如果你需要一个函数生成一个新散列然后返回一个指向它的引用，那么你有下列选择：

```
sub hashem {       { @_ }    }   # 不声不响地错误 -- returns @_

sub hashem {      +{ @_ } }     # 对

sub hashem { return {@_}}       # 对
```

8.2.2.3 匿名子过程组合器

你可以通过用不带子过程名字的 sub 创建一个匿名子过程：

```
$coderef = sub { print "Boink!\n" };   # 现在 &$coderef 打印 "Boink!"
```

请注意分号的位置，在这里要求用分号是为了终止该表达式。（在更常见的声明和定义命名子过程的用法 sub NAME {} 里面不需要这个分号。）一个没有名字的 sub {} 并不象是个声明而更象一个操作符——象 do {} 或 eval {} 那样——只不过在里面的代码并不是立即执行的。它只是生成一个指向那些代码的引用，在我们的例子里是存储在 $coderef 里。不过，不管你执行上面的行多少次，$coderef 都仍然将指向同样的匿名子过程。（注：不过就算只有一个匿名子过程，也有可能有好几份词法变量的拷贝被该子过程使用，具体情况取决于子过程生成的时候。这些东西在稍后的"闭合"（闭包）里讨论）。

8.2.3 对象构造器

子过程也可以返回引用。这句话听起来有些陈腐，但是有时候别人要求你用一个子过程来创建引用而不是由你自己创建引用。特别是那些叫构造器的特殊子过程创建并返回指向对象的引用。对象只是一种特殊的引用，它知道自己是和哪个类关联在一起的,而构造器知道如何创建那种关联关系。这些构造器是通过使用 bless 操作符，将一个普通的引用物转换成一个对象实现的，所以我们可以认为对象是一个赐过福的引用。（译注：bless 在英文中原意是"赐福"，Perl 中使用这样的词作为操作符名称，想必和 Larry 先生是学习语言出身，并且是虔诚的教徒的背景有关系吧，不过这个词的确非常贴切的形容了该操作符的作用，所以，

我就直译为"赐福"，希望不会破坏原味。）这儿可和宗教没什么关系；因为一个类起的作用是用户定义类型，给一个引用赐福只是简简单单地把它变成除了内建类型之外的用户定义类型。构造器通常叫做 new——特别是 C++ 程序员更是如此看待——但在 Perl 里它们可以命名为任何其他名字。

构造器可以用下列任何方法调用：

```
$objref = Doggie::->new(Tail => 'short', Ears => 'long');  #1

$objref = new Doggie:: Tail => 'short', Ears => 'long';       #2

$objref = Doggie->new(Tail => 'short', Ears => 'long');    #3

$objref = new Doggie Tail => 'short', Ears => 'long';         #4
```

第一个和第二个调用方法是一样的。它们都调用了 Doggie 模块提供的一个叫 new 的函数。第三个和第四个调用和头两个是一样的，只不过稍微更加模糊一些：如果你定义了自己的叫 Doggie 的子过程，那么分析器将被你弄糊涂。（这就是为什么人们坚持把小写的名字用于子过程，而把大写的名字用于模块。）如果你定义了自己的 new 子过程，并且偏偏有没有用 require 或者 use 使用 Doggie 模块（这两个都有声明该模块的作用。），那么第四个方法也会引起歧义。如果你想用方法 4，那么最好声明你的模块。（并且还要注意看看有没有 Doggie 子过程。）参阅第十二章，对象，看看有关 Perl 对象的讨论。

8.2.4 句柄引用

你可以通过引用同名的类型团来创建指向文件句柄或者目录句柄：

```
splutter(\*STDOUT);


sub splutter {
        my $fh = shift;
        print $fh = "her um well a hmmm\n";
}


$rec = get_rec(\*STDIN);
sub get_rec {
```

```
    my $fh = shift;
    return scalar <$fh>;
}
```

如果你是在传递文件句柄，你还可以使用光文件句柄来实现：在上面的例子中，你可以使用 *STDOUT 或者 *STDIN，而不是 *STDOUT 和 *STDIN。

尽管通常你可以互换地使用类型团和指向类型团的引用，但还是有少数几个地方是不可以这么用的，简单的类型团不能 bless 成对象，并且类型团引用无法传递出一个局部化了的类型团的范围。

当生成新的文件句柄的时候，老的代码通常做类似下面这样的东西来打开一个文件列表：

```
for $file (@name) {
    local *FH;
    open(*FH, $file) || next;
    $handle($file) = *FH;
}
```

这么做仍然可行，但是现在我们有更简单的方法：就是让一个未定义的变量自动激活一个匿名类型团：

```
for $file (@name) {
    my $fh;
    open($fh, $file) || next;
    $handle($file) = $fh;
}
```

使用间接文件句柄的时候，Perl 并不在意你使用的是类型团，还是指向类型团的引用，或者是更奇异的 I/O 对象中的一个。就大多数目的来说，你几乎可以没有任何区别的使用类型团或者类型团引用之一。但是我们前面也承认过，这两种做法仍然有一些隐含的细小区别。

8.2.5 符号表引用

在特别的环境里，你在开始写程序的时候可能不知道你需要什么样的引用。你可以用一种特殊的语法创建引用，人们常说是 *foo{THING} 语法。*foo{THING} 返回一个指向 *foo 里面 THING 槽位的引用，这个引用就是在符号表里保存 $foo，@foo，%foo，和友元的记录。

```
$scalarref = *foo{SCALAR};       # 和 \$foo 一样

$arrayref  = *ARGV{ARRAY};         # 和 \@ARGV 一样

$hashref   = *ENV{HASH};       # 和 \%ENV 一样

$coderef   = *handler{CODE};         # 和 \&handler 一样

$globref   = *foo{GLOB};           # 和 \*foo 一样

$ioref     = *STDIN{IO};             # ? ...
```

所有这些语句都具有自释性，除了 *STDIN{IO} 之外。它生成该类型团包含的实际的内部 IO::Handle 对象，也就是各种 I/O 函数实际上感兴趣的类型团的部分。为了和早期版本的 Perl 兼容，*foo{FILEHANDLE} 是上面的 *foo{IO} 说法的一个同义词。

理论上来说，你可以在任何你能用 *HANDLE 或者 *HANDLE 的地方使用 *HANDLE{IO}，比如将文件句柄传入或者传出子过程，或者在一个更大的数据结构里存储它们。（实际上，仍然有一些地方不能这么互换着用。）它们的优点是它们只访问你需要的真实的 I/O，而不是整个类型团，因此如果你通过一个类型团赋值剪除了比你预计的要多的东西也不会有什么风险（但如果你总是给一个标量变量赋值而不是给类型团赋值，那么你就 OK 了）。缺点是目前没有办法自动激活这么一个。（注：目前，open my $fh 自动激活一个类型团而不是一个 IO::Handle 对象，不过有朝一日我们总会修正这个问题的，所以你不应该依赖 open 目前自动激活的类型团的特征）。

```
splutter(*STDOUT);

splutter(*STDOUT{IO});


sub splutter {

   my $fh = shift;

   print $fh "her um well a hmmm\n";

}
```

上面对 splutter 的两个调用都打印 "her um well a hmm"。

如果编译器还没有看到特定的 THING，那么 *foo{THING} 这样的构造返回 undef。而且 *foo{SCALAR} 返回一个指向一个匿名标量的引用，即使编译器还没有看到 $foo。（Perl 总是给任何类型团加一个标量，它把这个动作当作一个优化，以便节省其他的什么地方的一些代码。但是在未来的版本中不要指望 Perl 保持这种做法。）

8.2.6 引用的隐含创建

创建引用的最后一种做法就根本算不上什么方法。如果你在一个认为存在引用的左值环境里做解引用的动作，那么引用就会自动出现。这样做是非常有用的，并且它也是你希望的。这个论题我们在本章稍后探讨，那时侯我们将讨论如何把我们到此为止创建的引用给解引用掉。

8.3 使用硬引用

就象我们有无数的方法创建引用一样，我们也有好几种方法使用引用（或者称之为解引用）。使用过程中只有一个最高级的原则：Perl 不会做任何隐含的引用或者解引用动作。（注：我们已经承认这句话是撒了一个小谎。我们不想再说了。）如果一个标量挂在了一个引用上，那么它总是表现出简单标量的行为。它不会突然就成为一个数组或者散列或是子过程，你必须明确地告诉它进行转变，方法就是对它解引用。

8.3.1 把一个变量当作变量名使用

如果你看到一个标量，比如 $foo，你应该把它看成“foo 的标量值。”也就是说，在符号表里有一条 foo 记录，而趣味字符 $ 是一个查看其内部的标量值的方法。如果在里面的是一个引用，那么你可以通过在前面再增加一个趣味字符来查看引用的内容（解引用）。或者用其他方法查看它，你可以把 $foo 里的文本字串 foo 替换成一个指向实际引用物的标量变量。这样做对任何变量类型都是正确的，因此不仅仅 $$foo 是指 $foo 指向的标量值，@$bar 是 $bar 指向的数组值， %$glarch 是 $glarch 指向的散列数值，等等。结果是你可以在任何简单标量前面放上一个额外的趣味字符将它解引用：

```
$foo        = "three humps";

$scalarref    = \$foo;       # $scalarref 现在是一个指向 $foo 的引用

$camel_model     = $$scalarref;    # $camel_model 现在是"three humps"
```

下面是其他的一些解引用方法：

```
$bar =   $$scalarref;


push(@$arrayref, $filename);
$$arrarref[0]   = "January";                    # 设置 @$arrayref 的第一个元
素
@$arrayref[4..6]=qw/May June July/;   # 设置若干个 @$arrayref 的元素


%$hashref = (KEY => "RING", BIRD => "SING");   # 初始化整个散列
$$hashref{KEY} = "VALUE";                       # 设置一个键字/数值对
@$hashref{"KEY1", "KEY2"} = {"VAL1", "VAL2"};          # 再设置两对


&$coderef(1,2,3);


print $handleref "output\n";
```

这种类型的解引用只能使用一个简单的标量变量（没有脚标的那种）。也就是说，解引用在任何数组或者散列查找之前发生（或者说是比数组和散列查找绑定得更紧）。还是让我们用一些花括弧来把我们的意思表示得明确一些：一个象 `$$arrayref[0]` 这样的表达式等于 `${$arrayref}[0]` 并且意思是数组的第一个元素由 `$arrayref` 指向。类似的，`$$hashref{KEY}` 和 `${$hashref}{KEY}` 一样，并且和 `${$hashref{KEY}}` 没什么关系，后者将对一个叫做 `%hashref` 的散列里的记录进行解引用的操作。你在意识到这一点之前可能会非常悲惨。

你可以实现多层引用和解引用，方法是连接合适的趣味字符。下面的程序打印 "howdy"：

```
$refrefref = \\\"howdy";
print $$$$refrefref;
```

你可以认为美圆符号是从右向左操作的。但是整个链条的开头必须是一个简单的，没有脚标的标量变量。不过，还有一种方法变得更神奇，这个方法我们前面已经偷偷用过了，我们会在下一节解释这种方法。

8.3.2 把一个 BLOCK 块当作变量名用

你不仅可以对一个简单的变量名字进行解引用，而且你还可以对一个 BLOCK 的内容进行解引用。在任何你可以放一个字母数字标识符当变量或者子过程名字一部分的地方，你都可以用一个返回指向正确类型的 BLOCK 代替该标识符。换句话说，早先的例子都可以用下面这样的方法明确化：

```
$bar = ${$scalarref};

push(@{$arrayref}, $filename);

${$arrayref}[0] = "January";

@{$arrayref}[4..6] = qw/May June July/;

${$hashref}{"KEY"} = "VALUE";

@{$hashref}{"KEY", "KEY2"} = ("VAL1", "VAL2");

&{$coderef}(1,2,3);
```

更不用说：

```
$refrefref = \\\"howdy";

print ${${${$refrefref}}};
```

当然，在这么简单的情况下使用花括弧的确非常愚蠢，但是 BLOCK 可以包含任意地表达式。特别是，它可以包含带脚标的表达式。

在下面的例子里，我们假设 $dispatch{$index} 包含一个指向某个子过程的引用（有时候我们称之为 "coderef"）。这个例子带着三个参数调用该子过程：

```
&{ $dispatch{$index} } (1, 2, 3);
```

在这里，BLOCK 是必要的。没有这个外层的花括弧对，Perl 将把 $dispatch 当作 coderef 而不是 $dispatch{$index}。

8.3.3 使用箭头操作符

对于指向数组，散列，或者子过程的引用，第三种解引用的方法涉及到使用 -> 中缀操作符。这样做就形成了一种语法糖，这样就让我们可以更容易访问独立的数组或者散列元素，或者间接地调用一个子过程。

解引用的类型是由右操作数决定的，也就是，由直接跟在箭头后面的东西决定。如果箭头后面的东西是一个方括弧或者花括弧，那么左操作数就分别当作一个指向一个数组或者散列的引用，由右边的操作数做下标定位。如果箭头后面的东西是一个左圆括弧，那么左操作数就当作一个指向一个子过程的引用看待，然后用你在圆括弧右边提供的参数进行调用。

下面的东西每三行都是一样的，分别对应我们已经介绍过的三种表示法。（我们插入了一些空白，以便将等效的元素对齐。）

```
$  $arrayref  [2]  = "Dorian";      #1

${ $arrayref }[2]  = "Dorian";      #2

$arrayref->[2]  = "Dorian";        #3


$  $hashref  {KEY} = "F#major";      #1

${ $hashref }{KEY} = "F#major";      #2

     $hashref->{KEY} = "F#major";          #3


&  $coderef  (Presto => 192);      #1

&{ $coderef }(Presto => 192);      #2

$coderef->(Presto => 192);          #3
```

你可以注意到，在每个三行组里，第三种表示法的趣味字符都不见了。这个趣味字符是由 Perl 猜测的，这就是为什么你不能用它对整个数组，整个散列，或者是它们的某个片段进行解引用。不过，只要你坚持使用标量数值，那么你就可以在 -> 左边使用任意表达式，包括另外一个解引用，因为多个箭头操作符是从左向右关联的：

```
print $array[3]->{"English"}->[0];
```

你可以从这个表达式里推论出 @array 的第四个元素是一个散列引用，并且该散列里的 "English" 记录的数值是一个数组的引用。

请注意 $array[3] 和 $array->[3] 是不一样的。第一个东西讲的是 @array 里的第四个元素，而第二个东西讲的是一个保存在 $array 里的数组（可能是匿名的数组）引用的第四个元素。

假设现在 $array[3] 是未定义。那么下面的语句仍然合法：

```
$array[3]->{"English"}->[0] = "January";
```

这个东西就是我们在前面提到过的引用突然出现的例子，这时候，当引用用做左值的时候是自动激活的（也就是说，当给它赋予数值的时候）。如果 $array[3] 是未定义，那么它会被自动定义成一个散列引用，这样我们就可以在它里面给 $array[3]->{"English"} 设置一个数值。一旦这个动作完成，那么 $array[3]->{"English"} 自动定义成一个数组引用，这样我们就可以给那个数组的第一个元素赋一些东西。请注意右值稍微有些不同：print $array[3]->{"English"}->[0] 只定义了 $array[3] 和 $array[3]->{"English"}，没有定义 $array[3]->{"English"}->[0]，因为最后一个元素不是左值。（你可以认为前面两个在右值环境里有定义是一只臭虫。我们将来可能除掉这只虫子。）

在方括弧或花括弧之间，或者在一个闭方括弧或花括弧与圆括弧之间的箭头是可选的。后者表示间接的函数调用。因此你可以把前面的代码缩减成：

```
$dispatch{$index}(1, 2, 3);

$array[3]{"English"}[0] = "January";
```

在普通的数组的情况下，这些东西给你一个多维的数组，就好象 C 的数组：

```
$answer[$x][$y][$z] += 42;
```

当然，并不完全象 C 的数组。其中之一就是 C 的数组不会按照需要增长，而 Perl 的却会。而且，一些在两种语言里相似的构造是用不同的方法分析的。在 Perl 里，下面的两个语句做的事情是一样的：

```
$listref->[2][2] = "hello";      # 相当干净

$$listref[2][2]  = "hello";      # 有点混乱
```

上面第二句话可能会让 C 程序员觉得诧异，因为 C 程序员习惯于使用 *a[i] 表示"a 的第 i 个元素所指向的内容"。但是在 Perl 里，五个元素（$ @ * % &）实际上比花括弧或者方括弧绑定得都更紧密。（注：但不是因为操作符优先级。在 Perl 里的趣味字符不是操作符。Perl 的语法只是简单地禁止任何比一个简单变量或者块更复杂的东西跟在趣味字符后面，个中缘由也是有很多有趣的原因的。）因此，被当作指向一个数组引用的是 $$listref 而不是 $listref[2]。如果你想要 C 的行为，你要么是写成 ${$listref[2]} 以强迫 $listref[2] 先于前面的 $ 解引用操作符计算，要么你就要使用 -> 表示法：

```
$listref[2]->{$greeting} = "hello";
```

8.3.4 使用对象方法

如果一个引用碰巧是一个指向一个对象的引用，那么定义该对象的类可能提供了访问该对象内部的方法，并且如果你只是使用这些类，那么通常应该坚持使用那些方法（与实现这些方法相对）。换句话说就是要友善，并且不要把一个对象当作一个普通引用看待，虽然在你必须这么做的时候 Perl 也允许你这么看。我们不想在这个问题上搞极权。但是我们的确希望有一些礼貌。

有了这种礼貌，你就在对象和数据结构之间获得了正交。在你需要的时候，任何数据结构都可以认为是一个对象。或者是在你不需要的时候都认为不是对象。

8.3.5 伪散列

一个伪散列是一个指向数组的任意引用，它的第一个元素是一个指向散列的引用。你可以把伪散列引用当作一个数组引用（如你所料），也可以把它当作一个散列引用（出乎你的意料）。下面是一个伪散列的例子。

```
$john = [ {age => 1, eyes => 2, weight => 3}, 47, "brown", 186 ];
```

在 $john->[0] 下面的散列定义了随后的数组元素（47, "brown", 186）的名字（"age", "eyes" , "weight"）。现在你可以用散列和数组的表示法来访问一个元素：

```
$john->{weight}          # 把 $john 当作一个 hashref 对待

$john->[3]             # 把 $john 当作一个 arrayref 对待
```

伪散列的魔术并不那么神奇；它只知道一个“技巧”：如何把一个散列解引用转变成一个数组解引用。在向伪散列里增加其他元素的时候，在你使用散列表示法之前，必须明确告诉下层的散列那些元素将放在哪里：

```
$john->[0]{height} = 4;       # 高度是元素 4 的数值

$john->{height} = "tall";       # 或者 $john->[4] = "tall"
```

如果你试图从一个伪散列中删除一个键字，那么 Perl 将抛出一个例外，尽管你总是可以从映射散列中删除键字。如果你试图访问一个不存在的键字，那么 Perl 也会抛出一个例外，这里“存在”的意思是在映射散列里出现：

```
delete $john->[9]{height};       # 只从下层散列中删除

$john->{height};          # 现在抛出一个例外
```

```
$john->[4];             # 仍然打印 "tall"
```

除非你很清楚自己在干什么，否则不要把数组分成片段。如果数组元素的位置移动了，那么映射散列仍然指向原来的元素位置，除非你同时也明确改变它们。伪散列的道行并不深。

要避免不一致性，你可以使用 use fields 用法提供的 fields::phash 函数创建伪散列：

```
use fields;

$ph = fields::phash(age => 47, eyes => "brown", weight => 186);

print $ph->(age);
```

有两个方法可以检查一个键字在伪散列里面是否存在。第一个方法是使用 exists，它检查给出的字段是否已经设置过了。它用这种办法来对应一个真正的散列的行为。比如：

```
use fields;

$ph = fields::phash([qw(age eyes brown)], [47]);

$ph->{eyes} = undef;


print exists $ph->{age};       # 对，'age' 在声明中就设置了

print exists $ph->{weight};      # 错，'weight' 还没有用呢

print exists $ph->{eyes};      # 对，你的 'eyes' 已经修改过了
```

第二种方法是在第一个数组元素里放着的影射散列上使用 exists。这样就检查了给出的键字对该伪散列是否是一个有效的字段：

```
print exists $ph->[0]{age};       # 对，'age' 是一个有效的字段

print exists $ph->[0]{name};    # 错，不能使用 'name'
```

和真正的散列里发生的事情有些不同，在伪散列元素上调用 delete 的时候只删除对应该键字的数组值，而不是映射散列的真正的键字。要删除该键字，你必须明确地从映射散列中删除之。一旦你删除了该键字，那么你就不再能用该名字作为伪散列的脚标：

```
print delete $ph->{age};       # 删除并返回 $ph->[1], 47

print exists $ph->{age};       # 现在是错的

print exists $ph->[0]{age};       # 对，'age' 键字仍然可用
```

```
print delete $ph->[0]{age};       # 现在 'age' 键字没了
print $ph->{age};                 # 运行时例外
```

你可能会想知道是因为什么原因促使人们想出这种伪装在散列的外衣下面的数组的。数组的查找速度比较快并且存储效率也高些，而散列提供了对你的数据命名的方便（而不是编号）；伪散列提供了两种数据结构的优点。但是这些巨大的优点只有在你开始考虑 Perl 的编译阶段的时候才会出现。在一两个用法的帮助下，编译器可以核实对有效的数据域的访问，这样你就可以在程序开始运行之前发现不存在的脚标（可以查找拼写错误）。

伪散列的速度，效率，和编译时访问检查（你甚至可以把这个特性看作一种安全性）的属性令它成为创建高效健壮的类方法的非常便利的工具。参阅第十二章里的 use fields 以及第三十一章，实用模块里的讨论。

伪散列是一种比较新的并且相对试验性的特性；因此，其下的实现在将来很有可能被修改。要保护自己免于受到这样的修改的影响，你应该总是使用 fields 里面有文档记录的 phash 和 new 函数。

8.3.6 硬引用可以用的其他技巧

我们前面提到过，反斜杠操作符通常用于一个引用中生成一个引用，但是并非必须如此。如果和一列引用物一起使用，那么它生成一列对应的引用。下面的例子的第二行和第一行做的事情是一样的，因为反斜杠是自动在整个列表中分布的。

```
@reflist = (\$s, \@a, \%h, \&f);       # 一列四个元素的表
@reflist = \($s, @a, %h, &f);                  # 同样的东西
```

如果一个圆括弧列表只包含一个数组或者散列，那么所有它的数值都被代换，并且返回每个引用：

```
@reflist = \(@x);                    # 代换数组，然后获得引用
@reflist = map (\$_) @x;          # 一样的东西
```

如果你在散列上试验这些代码，那么结果将包含指向数值的引用（如你所料），而且还包含那些键字的拷贝的引用（这可是你始料未及的）。

因为数组和散列片段实际上都是列表，因此你可以用反斜杠逃逸两者中的任意一个获取一个引用的列表。下面三行中的任何一个实际上做的都是完全一样的事情：

```
@envrefs = \@ENV{'HOME', 'TERM'};       # 反斜杠处理一个片段
```

```
@envrefs = \($ENV{HOME}, $ENV{TERM} );   # 反斜杠处理一个列表

@envrefs = ( \$ENV{HOME}, \$ENV{TERM} );   # 一个两个引用的列表
```

因为函数可以返回列表，所以你可以给它们加上反斜杠。如果你有超过一个的函数需要调用，那么首先把每个函数的返回值代换到一个大的列表中，然后在给整个列表加上反斜杠：

```
@reflist = \fx();

@reflist = map { \$_ } fx();      # 一样的东西


@reflist = \( fx(), fy(), fz() );

@reflist = ( \fx(), \fy(), fz() );      # 一样的东西

@reflist = map { \$_ } fx(), fy(), fz();   # 一样的东西
```

反斜杠操作符总是给它的操作数提供一个列表环境，因此那些函数都是在列表环境中调用。如果反斜杠本身是处于标量环境，那么你最终会得到一个指向该函数返回的列表中的最后一个数值的引用：

```
@reflist = \localtime();          # 引用九个时间元素中的每一个

@lastref = \localtime();          # 引用的是它是否为夏时制
```

从这个方面来看，反斜杠的行为类似命名的 Perl 列表操作符，比如 print，reverse 和 sort，它们总是在它们的右边提供一个列表环境，而不管它们左边是什么东西。和命名的列表操作符一样，使用明确的 scalar 强迫跟在后面的进入标量环境：

```
$dateref = \scalar localtime();      # \"Thu Apr 19 22:02:18 2001"
```

你可以使用 ref 操作符来判断一个引用指向的是什么东西。把 ref 想象成一个“typeof”（类型为）操作符，如果它的参数是一个引用那么返回真，否则返回假。返回的数值取决于所引用的东西的类型。内建的类型包括 SCALAR，ARRAY，HASH，CODE，GLOB，REF，LVALUE，IO，IO::Handle，和 Regexp。在下面，我们用它检查子过程参数：

```
sub sum {
    my $arrayref = shift;
    warn "Not an array reference" if ref($arrayref) ne "ARRAY";
    return eval join("+", @$arrayref);
```

```
}
```

如果你在一个字串环境中使用硬引用，那么它将被转换成一个包含类型和地址的字串：SCALAR(0x12fcde)。（反向的转换是不能实现的，因为在字串化过程中，引用计数信息将被丢失 ——而且让程序可以访问一个由一个随机字串命名的地址也太危险了。）

你可以用 bless 操作符把一个引用和一个包函数关联起来作为一个对象类。在你做这些的时候，ref 返回类名字而不是内部的类型。在字串环境里使用的对象引用返回带着内部和外部类型的字串，以及在内存中的地址：MyType=HASH(0x21cda) 或者 IO::Handle=IO(0x186904)。参阅第十二章获取有关对象的更多的细节。

因为你对某些东西解引用的方法总是表示着你在寻找哪种引用物，一个类型团可以用与引用一样的方法来使用，尽管一个类型团包含各种类型的多个引用。因此 ${*main::foo} 和 ${\$main::foo} 访问的都是同一个标量变量，不过后者更高效一些。

下面是一个把子过程调用的返回值代换成一个字串的技巧：

```
print "My sub returned @{[ mysub(1, 2, 3) ]} that time.\n";
```

它的运转过程如下。在编译时，当编译器看到双引号字串里的 @{...} 的时候，它就会被当作一个返回一个引用的块分析。在块里面，方括弧创建一个指向一个匿名数组的引用，该数组来自方括弧里面的东西。因此在运行时，mysub(1,2,3) 会在列表环境中调用，然后其结果装载到一个匿名数组里，然后在块里面返回一个指向该匿名数组的引用。然后该数组引用马上就被周围的 @{...} 解引用，而其数组的值就被代换到双引号字串中，就好象一个普通的数组做的那样。这样的强词夺理式的解释也适用于任意表达式，比如：

```
print "We need @{ [$n + 5] } widgets!\n";
```

不过要小心的是：方括弧给它们的表达式提供了一个列表环境。在本例中它并不在意是否为列表环境，不过前面对 mysub 的调用可能会介意。如果列表环境有影响的时候，你可以使用一个明确的 scalar 以强迫环境为标量：

```
print "Mysub return @{ [scalar mysub(1,2,3)] } now.\n";
```

8.3.7 闭合（闭包）

我们早些时候谈到过用一个没有名字的 sub {} 创建匿名子过程。你可以把那些子过程看作是在运行时定义，这就意味着它们有一个生成的时间和一个定义的地点。在创建子过程的时候，有些变量可能在范围里，而调用子过程的时候，可能有不同的变量在范围里。

先让我们暂时忘掉子过程，先看看一个指向一个词法变量的引用：

```
{
    my $critter = "camel";
    $critterref = \$critter;
}
```

$$critterref 的数值仍将是“camel”，即使在离开闭合的花括弧之后 $critter 消失了也如此。但是 $critterref 也可以指向一个指向了 $critter 的子过程：

```
{
    my $critter = "camel";
    $critterref = sub { return $critter };
}
```

这是一个闭合（闭包），这个词是来自 LISP 和 Scheme 那些机能性（functional）编程世界的。（注：在这样的语言环境里，“functional”（机能性）应该看作是“dysfunctional”（机能紊乱）的反义词）。这就意味着如果你某一时刻在特定的词法范围定义了一个匿名函数，那么它就假装自己是在那个范围里运行的，即使后面它又从该范围之外调用。（一个力求正统的人会说你用不着使用“假装”这个词——它实际上就是运行在该范围里。）

换句话说，Perl 保证你每次都获得同一套词法范围变量的拷贝，即使该词法变量的其他实例在该闭合的实例之前或者自该闭合存在开始又创建其他实例也如此。这样就给你一个方法让你可以在定义子过程的时候设置子过程里的数值，而不仅仅是在调用它们的时候。

你还可以把闭合看作是一个不用 eval 书写子过程模板的方法。这时候词法变量用做填充模板的参数，作用主要是设置很少的一些代码用于稍后运行。在基于事件的编程里面，这种做法常常叫做回调（callback），比如你把一些代码和一次键盘敲击，鼠标点击，窗口露出等等关联起来。如果当作回调使用，闭合做的就是你所预期的，即使你不知道机能性编程的第一件事情也无妨。（请注意这些与闭合相关的事情只适用于 my 变量。全局量还是和往常一样运转，因为它们不是按照词法变量的方式创建和删除的。）

闭合的另外一个用途就是函数生成器，也就是说，创建和返回全新函数的函数。下面是一个用闭合实现的函数生成器的例子：

```
sub make_saying {
        my $salute = shift;
```

```
        my $newfunc = sub {
                my $target = shift;
                print "$salute, $target!\n";
        };
        return $newfunc;        # 返回一个闭合
}

$f = make_saying("Howdy");      # 创建一个闭合
$g = make_saying("Greetings");  # 创建另外一个闭合

# 到时...

$f->("world");
$g->("earthings");
```

它打印出：

```
Howdy, world!
Greetings earthlings!
```

特别要注意 $salute 是如何继续指向实际传递到 make_saying 里的数值的，尽管到该匿名子过程 运行的时候 my $salute 已经超出了范围。这就是用闭合来干的事情。因为 $f 和 $g 保存着指向函数的引用，当调用它们的时候，这些函数仍然需要访问独立的 $salute 版本，因此那些变量版本自动附着在四周。如果你现在覆盖 $f，那么它的版本的 $salute 将自动消失。（Perl 只是在你不再查看的时候才做清理。）

Perl 并不给对象方法（在第十二章描述）提供引用，但是你可以使用闭合获取类似的效果。假设你需要这么一个引用：它不仅仅指向他代表的方法的子过程，而且它在调用的时候，还会在特定的对象上调用该方法。你可以很方便地把对象和方法都看成封装在闭合中的词法变量：

```
sub get_method {
        my ($self, $methodname) = @_;
        my $methref = sub {
                # 下面的 @_ 和上面的那个不一样!
                return $self->$methodname(@_);
        };
        return $methref;
}

my $dog = new Doggie::
Name => "Lucky",
Legs => 3,
Tail => "clipped";

our $wagger = get_method_ref($dog, 'wag');
$wagger->("tail");      # 调用 $dog->wag('tail').
```

现在，你不仅可以让 Lucky 摇动尾巴上的任何东西，就算词法 $dog 变量已经跑出了范围并且现在你看不到 Lucky 了，那么全局的 $wagger 变量仍然会让它摇尾巴——不管它在哪里。

8.3.7.1 用闭合做函数模板

拿闭合做函数模板可以让你生成许多动作类似的函数。比如你需要一套函数来生成各种不同颜色的 HTML 字体变化：

```
print "Be", red("careful"), "with that ", green("light"), "!!!";
```

red 和 green 函数会非常类似。我们已经习惯给我们的函数名字，但是闭合没有名字，因为它们只是带倾向性的匿名子过程。为了绕开这个问题，我们将使用给我们的匿名子过程命

名的非常技巧性的方法。你可以把一个 coderef 绑定到一个现存的名字上，方法是把它赋予一个类型团，该类型团的名字就是你想要的函数。（参阅第十章，包，里的"符号表"一节。在本例中，我们将把它绑定到两个不同的名字上，一个是大写，一个是小写。

```
@colors = qw(red blue green yellow orange purple wiolet);

for my $name (@colors) {

        no strict 'refs';       # 允许符号引用

        *$name = *(uc $name) = sub { "<FONT COLOR='$name'7gt;@_</FONT>" };

}
```

现在你可以调用名字叫 red, RED, blue, BLUE, 等等的函数，并且就会调用合适的函数。这个方法减少了编译时间并且节约了内存，并且还减少了错误的发生，因为语法检查是在编译时进行的。在匿名子过程里任意的变量都必须是词法范围的，这样才能创建闭合。因此上面的例子中使用了 my。

这个例子是极少数给闭合原型有意义的地方。如果你想在这些函数的参数上强制标量环境（在我们的例子上可能不是一个好主意），你可以用下面的方法写：

```
*$name = sub ($) {"$_[0]" };
```

这么做几乎已经足够好了。不过，因为原型检查是发生在编译时间，上面的运行时赋值发生得太晚了，因而没有什么价值。你应该把整个赋值循环放到一个 BEGIN 块里，强迫它在编译时发生。（更好的方法是你把它放到一个模块里，这样你就可以在编译时 use 它了。）这样在编译剩下的时间里，原型就是可见的了。

8.3.7.2 嵌套的子过程

如果你熟悉在子过程里嵌套使用其他子过程（从其他编程语言中学来），每个子过程都有自己的私有变量，你可能不得不稍微地按照 Perl 的习惯来处理它们。命名的子过程并不合适做嵌套，但是匿名子过程却可以（注：更准确地说是全局命名的子过程不能嵌套。糟糕的是，全局命名子过程就是我们所拥有的唯一一种命名子过程。我们还没有实现词法范围的命名子过程（被称为 my subs），但是到我们实现它们的时候，它们应该可以正确嵌套。）不管怎样，我们都可以用闭合模拟嵌套的，词法范围的子过程。下面是一个例子：

```
sub outer {

        my $x = $_[0] + 35;
```

```
        local *inner = sub { return $x * 19};

        return $x + inner();

}
```

因为闭合的临时赋值，现在 inner 只能从 outer 里面调用。但只要你是在 outer 里调用它的，那么它就能从 outer 的范围里对词法变量 $x 的正常访问。

这么做对于创建一个相对另外一个函数是局部函数的时候有一些有趣的效果，这些效果不是 Perl 正常时支持的。因为 local 是动态范围，并且函数名字对于它们的包来说是全局的，那么任何其他 outer 调用的函数也可以调用 inner 的临时版本。要避免这些，你应该需要额外的间接的层次：

```
sub outer {

        my $x = $_[0] + 35;

        my $inner = sub {return $x * 19 };

        return $x + $inner->();

}
```

8.4 符号引用

如果你试图给一个不是硬引用的数值进行解引用会发生什么事情？那么该值就会被当作一个符号引用。也就是说，该引用被解释成一个代表某个全局量的名字的字串。

下面是其运转样例：

```
    $name = "bam";

    $$name = 1;          # 设置 $bam

    $name->[0] = 4;      # 设置 @bam 的第一个元素

    $name->{X} = "Y";      # 设置 %bam 的 X 元素为 Y

    @$name = ();         # 清除 @bam

    keys %$name;         # 生成 %bam 的键字

    &$name;          # 调用 &bam
```

这么用的功能是非常强大的，并且有点危险，因为我们有可能原来想用的是（有着最好的安全性）硬引用，但是却不小心用了一个符号引用。为了防止出现这个问题，你可以说：

```
use strict 'refs';
```

然后在闭合块的剩余部分，你就只被允许使用硬引用。更内层的块可以用下面的语句撤消这样的命令：

```
no strict 'refs';
```

另外，我们还必须理解下面两行程序的区别：

```
${identifier};      # 和 $identifier 一样
${"identifier"};      # 也是 $identifier，不过却是一个符号引用。
```

因为第二种形式是引号引起的，所以它被当作一个符号引用，如果这时候 use strict 'refs' 起作用的话它将会生成一个错误。就算 strict 'refs' 没有起作用，它也只能指向一个包变量。但是第一种形式是和没有花括弧的形式相等的，它甚至可以指向一个词法范围的变量，只要你定义了这么一个。下面的例子显示了这个用法（而且下一节就是讨论这个问题的）。

只有包变量可以通过符号引用访问，因为符号引用总是查找包的符号表。由于词法变量不是在包的符号表里面，因此它们对这种机制是隐形的。比如：

```
our $value = "global";
{
        my $value = "private";
        print "Inside, mine is ${value},";
        print "but ours is ${'value'}.\n";
}
print "Outside, ${value} is again ${'value'}.\n";
```

它打印出：

```
Inside, mine is private, but ours is global.
Outside, global is again global.
```

8.5 花括弧，方括弧和引号

在前面一节里，我们指出了 ${identifier} 是不被当作符号引用看待的。你可能会想知道这个机制是怎么和保留字相交互的，简单的回答是：它们没有相互交互。尽管 push 是一个保留字，下面这两句还是打印 "pop on over"：

```
$push = "pop on ";

print "${push}over";
```

这里的原因是这样的，历史上，Unix shell 是用花括弧来隔离变量名字和后继的字母数字文本的（要不然这些字母数字就成了变量名字的一部分。）这也是许多人预料的变量代换的运转方式，因此我们在 Perl 令之以同样的方法运转。但是就 Perl 而言，这种表示法的使用得到了扩展，它可以用于任意用来生成引用的花括弧，不管它们是否位于引号里面。这就意味着：

```
print ${push} . 'over';
```

或者甚至是下面（因为空格无所谓）：

```
print ${ push } . 'over';
```

都打印出 "pop on over"，尽管花括弧在双引号外边。同样的规则适用于任意给散列做脚标的标识符。因此，我们可以不用写下面这样的句子：

```
$hash{ "aaa" }{ "bbb" }{ "ccc" }
```

你可以只写：

```
$hash{ aaa}{ bbb }{ ccc }
```

或者

```
$hash{aaa}{bbb}{ccc}
```

而不用担心这些脚标是否为保留字。因此：

```
$hash{ shift }
```

被代换成 $hash{"shift"}。你可以迫使代换当作保留字来用，方法是增加任何可以令这个名字不仅仅是一个标识符的东西：

```
$hash{ shift() }

$hash{ +shift }
```

```
$hash{ shift @_ }
```

8.5.1 引用不能当作散列键字用

散列键字在内部都存储成字串。(注：它们在外部也存储成字串，比如在你把它们放到 DBM 文件中的时候。实际上，DBM 文件要求它们的键字（和数值）是字串。）如果你试图把一个引用当作一个散列的键字存储，那么该键字值将被转换成一个字串：

```
$x{ \$a } = $a;

($key, $value) = each %x;

print $$key;         # 错误
```

我们前面已经说过你不能把一个字串转换回硬引用。因此如果你试图解引用 $key，（它里面只保存着一个字串），那么它不会返回一个硬引用，而是一个符号引用——并且因为你可能没有叫 SCALAR(0x1fc0e) 的变量，所以你就无法实现你的目的。你可能要做一些更象：

```
$r = \@a;

$x{ $r } = $r;
```

这样的东西。这样你至少能使用散列值，这个值是一个硬引用，但你不能用键字，它不是硬引用。

尽管你无法把一个引用存储为键字，但是如果你拿一个硬引用在一个字串的环境中使用，(象我们前面的例子）那么 Perl 保证它生成一个唯一的字串，因为该引用的地址被当作字串的一部分包含。这样你实际上就可以把引用当作一个唯一的键字使用。只是你后面就没有办法对它解引用了。

有一种特殊类型的散列，在这种散列里，你可以拿引用当作键字。通过与 Perl 捆绑在一起的 Tie::RefHasn 模块的神奇（这个词是技术术语，如果你翻翻 Perl 源程序目录里的 mg.c 文件就会明白。），你就可以做我们刚才还说不能做的事情：

```
use Tie::RefHandle;

tie my %h, 'Tie::RefHash';

%h = (

["this", "here"]    => "at home",

["that", "there"]    => "elsewhere",
```

```
);


while ( my($keyref, $value) = each %h ) {
        print "@$keyref is $value\n";
}
```

实际上，通过将不同的实现与内建类型捆绑，你可以把标量，散列，和数组制作成可以拥有我们上面说过不行的行为。你看，这些作者就那么傻...

有关捆绑的更多细节，参阅第十四章，捆绑变量。

8.5.2 垃圾收集，循环引用和弱引用

高级语言通常都允许程序员不用担心在用过内存之后的释放问题。这种自动化收割处理就叫做垃圾收集。就大多数情况而言，Perl 使用一种快速并且简单的以引用为基础的垃圾收集器。

如果一个块退出了，那么它的局部范围的变量通常都释放掉，但是你也有可能把你的垃圾藏起来，因此 Perl 的垃圾收集器就找不到它们了。一个严重的问题是如果一片不可访问的内存的引用记数为零的话，那么它将得不到释放。因此，循环引用是一个非常糟糕的主意：

```
{       # 令 $a 和 $b 指向对方
    my ($a, $b);
    $a = \$b;
    $b = \$a;
}
```

或者更简单的就是：

```
{       # 令 $a 指向它自己
    my $a;
    $a = \$a;
}
```

即使 $a 应该在块的结尾释放，但它实际上却没有。在制作递归数据结构的时候，如果你想在程序退出之前回收内存，那么你不得不自己打破（或者弱化，见下文）这样的自引用。（退出后，这些内存将通过一个开销比较大但是完整的标记和打扫垃圾收集。）如果数据结构是一个对象，你可以可以用 DESTROY 方法自动打破引用；参阅第十二章的"用 DESTROY 方法进行垃圾收集"。

一个类似的情况可能出现在缓冲中——缓冲是存放数据的仓库，用于以更快的速度进行检索采用的方法。在缓冲之外，有指向缓冲内部的引用。问题发生在所有这些引用都被删除的时候，但是缓冲数据和它的内部引用仍然存在。任何引用的存在都会阻止 Perl 回收引用物，即使我们希望缓冲数据在我们不再需要的时候尽快消失也是如此。对于循环引用，我们需要一个不影响引用计数的引用，因而就不会推迟垃圾收集了。

弱引用解决了循环引用和缓冲数据造成的问题，它允许你"弱化"任意引用；也就是说，让它不受引用计数的影响。当最后一个指向对象的未弱化引用被删除之后，该对象就被删除并且所有指向改对象的弱引用都被释放。

要使用这个特性，你需要来自 CPAN 的 WeakRef[2] 包，它包含附加的文档。弱引用是试验性特性。不过，有些人就是特别能钻。

第九章 数据结构

Perl 免费提供许多数据结构，这些数据结构在其他编程语言里是需要你自己制作的。比如那些计算机科学的新芽们都需要学习的堆栈和队列在 Perl 里都只是数组。在你 push 和 pop（或者 shift 和 unshift）一个数组的时候，它就是一个堆栈；在你 push 和 shift（或者 unshift 和 pop）一个数组的时候，它就是一个队列。并且世界上有许多树结构的作用只是为了给一些概念上是平面的搜索表文件提供快速动态的访问。当然，散列是内建于 Perl 的，可以给概念上是平面的搜索表提供快速动态的访问，只有对编号不敏感的递归数据结构才会被那些脑袋已经相当程度编了号的人称为美人。

但是有时候你需要嵌套的数据结构，因为这样的数据结构可以更自然地给你要解决的问题建模。因为 Perl 允许你组合和嵌套数组和散列以创建任意复杂的数据结构。经过合理地组合，它们可以用来创建链表，二叉树，堆，B-tree（平衡树），集，图和任何你设计的东西。参阅 Mastering Algorithms with Perl（O'Reilly, 1999），Perl Cookbook（O'Reilly

1998），或者 CPAN——所有这些模块的中心仓库。不过，你需要的所有东西可能就是简单地组合数组和散列，所以我们就在本章介绍这些内容。

9.1 数组的数组

有许多种类型的嵌套数据结构。最容易做的是制作一个数组的数组，也叫做两维数组或者矩阵。（明显的总结是：一个数组的数组的数组就是一个三维数组，对于更高的维数以此类推。）多维数组比较容易理解，并且几乎所有它适用的东西都适用于我们在随后各节里将要讲述的其他更奇特的数据结构。

9.1.1 创建和访问一个两维数组

下面是如何把一个两维数组放在一起的方法：

```
# 给一个数组赋予一个数组引用列表。
@Aoa = (
    ["fred", "barney" ],
    ["george", "jane", "elroy" ],
    ["homer", "marge", "bart" ],
);

print $AoA[2][1];       # 打印 "marge"
```

整个列表都封装在圆括弧里，而不是花括弧里，因为你是在给一个列表赋值而不是给引用赋值。如果你想要一个指向数组的引用，那么你要使用方括弧：

```
# 创建一个指向一个数组的数组的引用。
$ref_to_AoA = [
[ "fred", "barney", "pebbles", "bamm bamm", "dino", ],
[ "homer", "bart", "marge", "maggie", ],
[ "george", "jane", "elroy", "judy", ],
];
```

```
print $ref_to_AoA->[2][3];      # 打印 "judy"
```

请记住在每一对相邻的花括弧或方括弧之间有一个隐含的 ->。因此下面两行：

```
$AoA[2][3]

$ref_to_AoA->[2][3]
```

等效于下面两行：

```
$AoA[2]->[3]

$ref_to_AoA->[2][3]
```

不过，在第一对方括弧前面没有隐含的 ->，这也是为什么 $ref_to_AoA 的解引用要求开头的 ->。还有就是要记住你可以用负数索引从一个数组后面向前面计数，因此：

```
$AoA[0][-2]
```

是第一行的倒数第二个元素。

9.1.2 自行生长

大的列表赋值是创建大的固定数据结构的好方法，但是如果你想运行时计算每个元素，或者是一块一块地制作这些结构的时候该怎么办呢？

让我们从一个文件里读入一个数据结构。我们将假设它是一个简单平面文本文件，它的每一行是结构的一个行，并且每行包含由空白分隔的元素。下面是处理的方法：（注：在这里和其他章节一样，我们忽略那些通常你要放进去的 my 声明。在这个例子里，你通常写 my @tmp = split。）

```
while (<>) {
        @tmp = split;           # 把元素分裂成一个数组
        push @AoA, [ @tmp ];       # 向 @AoA 中增加一个匿名数组引用
}
```

当然，你不必命名那个临时的数组，因此你还可以说：

```
while(<>){
        push @AoA, [ split ];
}
```

如果你想要一个指向一个数组的数组的引用，你可以这么做：

```
while (<>){
        push @ref_to_AoA, [ split ];
}
```

这些例子都向数组的数组增加新的行。那么如何增加新的列呢？如果你只是对付两维数组，通常更简单的方法是使用简单的赋值：（注：和前面的临时赋值一样，我们在这里已经简化了；在这一章的循环在实际的代码中应该写做 my $x。）

```
for $x (0..9){                  # 对每一行...
        for $y (0..9) {             # 对每一列...
                $AoA[$x][$y] = func($x, $y);    # ...设置调用
        }
}


for $x (0..9) {                 # 对每一行...
        $ref_to_AoA->[$x][3] = func2($x);       # ...设置第四行
}
```

至于你给元素赋值的顺序如何则没有什么关系，而且 @AoA 的脚标元素是否存在也没有什么关系；Perl 会很开心地为你创建它们，并且把中间的元素根据需要设置为未定义值。（如果有必要，Perl 甚至会为你在 $ref_to_AoA 中创建最初的引用。）如果你只是想附加一行，你就得做得更奇妙一些：

```
# 向一个已经存在的行中附加一个新列
push @{ $AoA[0] }, "wilma", "betty";
```

请注意下面这些无法运转：

```
push $AoA[0], "wilma", "betty";          # 错误!
```

上面的东西甚至连编译都过不了，因为给 push 的参数必须是一个真正的数组，而不只是一个指向一个数组的引用。因此，第一个参数绝对必须以 @ 字符开头。而跟在 @ 后面的东西则可以忽略一些。

9.1.3 访问和打印

现在把数据结构打印出来。如果你只想要一个元素，下面的就足够了：

```
print $AoA[3][2];
```

但是如果你想打印所有的东西，那你不能这么写：

```
print @AoA;          # 错误!
```

这么做是错误的，因为它会打印出字串化的引用，而不是你的数据。Perl 从来不会自动给你解引用。你必须自己用一两个循环遍历你的数据。下面的代码打印整个结构，循环遍历 @AoA 的元素并且在 print 语句里对每个元素进行解引用：

```
for $row (@AoA) {
    print "@$row\n";
}
```

如果你想追踪脚标，你可以这么做：

```
for $i (0..$#AoA) {
    print "row $i is: @{$AoA[$i]}\n";
}
```

或者甚至可以是下面这样：

```
for $i (0..$#AoA){
        for $j (0..$#{$AoA[$i]}){
                print "element $i $j is $AoA[$i][$j]\n";
        }
```

```
}
```

就象你看到的那样，这里的程序有点复杂。这就是为什么很多时候你用一个临时变量事情会变得更简单：

```
for $i (0..$#AoA){
        $row = $AoA[$i];
        for $j (0..$@{$row}){
                print "element $i $j is $row->[$j]\n";
        }
}
```

9.1.4 片段

如果你想访问一个多维数组的某个片段（一行的一部分），你就是在准备做一些奇特的脚标处理。指针箭头赋予我们一种访问单一变量的极好的方法，但是对于片段而言却没有这么好的便利方法。当然，你总是可以用一个循环把你的片段一个一个地取出来：

```
   @part = ();
   for ($y = 7; $y < 13; $y++) {
      push @part, $AoA[4][$y];
   }
```

这个循环可以用一个数组片段代替：

```
   @part = @{ $AoA[4] } [7..12];
```

如果你想要一个两维的片段，比如 $x 在 4..8 而 $y 是 7..12，下面是实现的一些方法：

```
@newAoA = ();
for ($startx = $x = 4; $x <= 8; $x++) {
        for ($starty = $y=7; $y <= 12; $y++) {
                $newAoA[$x - $startx][$y - $starty] = $AoA[$x][$y];
```

```
        }
}
```

在这个例子里，我们的两维数组 @newAoA 里的每个独立的数值都是一个一个地从一个两维数组 @AoA 中取出来赋值的。另外一个方法是创建一个匿名数组，每个由一个 @AoA 中我们要的子数组组成，然后然后把指向这些匿名数组的引用放到 @newAoA 中。然后我们就可以把引用写到 @newAoA （也是脚标，只是这么说而已），而不用把一个子数组值写到两维数组 @newAoA 中。这个这个方法消除了内层的循环：

```
for ($x = 4; $x <= 9; $x++) {
        push @newAoA, [ @{ $AoA[$x] } [ 7..12] ];
}
```

当然，如果你经常这么做，那么你可能就应该写一个类似 extract_rectangle 这样的子过程。而如果你经常对大的多维数组做这样的处理，那么你可能要使用 PDL（Perl Data Language）模块，你可以在 CPAN 找到。

9.1.5 常见错误

正如我们早先提到过的那样，Perl 数组和散列都是一维的。在 Perl 里，甚至"多维"数组实际上都是一维的，但是该维的数值实际上是其他数组的引用，这样就把许多元素压缩成了一个。

如果你不首先解引用就把这些打印出来，那么你看到的就是字串化的引用而不是你需要的数字。比如，下面两行：

```
@AoA = ([2, 3], [4, 5, 7], [0] );
print "@AoA";
```

结果是象下面这样的东西：

```
ARRAY(0x83c38) ARRAY(0x8b194) ARRAY(0x8b1d0)
```

但是，下面这行显示 7：

```
print $AoA[1][2];
```

在构造一个数组的数组的时候，要记得为子数组构造新的引用。否则，你就你就会只创建了一个包含子数组元素计数的数组，象这样：

```
for $i (1..10) {
        @array = somefunc($i);
        $AoA = @array;          # 错误!
}
```

在这里 @array 是在一个标量环境里访问的，因此生成它的元素的计数，然后这个计数被忠实地赋予 $AoA[$i]。赋予引用的正确方法我们将在稍后介绍。

在产生前面的错误之后，人们认识到他们需要赋一个引用值，因此人们随后很自然会犯的的错误包括把引用不停地放到同一片内存位置：

```
for $i (1..10) {
        @array = somefunc($i);
        $AoA[$i] = \@array;          # 又错了!
}
```

每个 for 循环的第二行生成的引用都是一样的，也就是说，一个指向同一个数组 @array 的引用。的确，这个数组在循环的每个回合中都会变化，但是当所有的话都说完了，所有的事都做完了之后， $AoA 就包含 10 个指向同一数组的引用，这个时候它保存给它的最后一次赋值的数值。 print @{$AoA[1]} 将检索和 print @{$AoA[2]} 一样的数值。

下面是更成功的方法：

```
for $i (1..10) {
        @array = somefunc($i);
        $AoA[$i] = [ @array ];       # 正确!
}
```

在 @array 周围的方括弧创建一个新的匿名数组，@array 里的元素都将拷贝到这里。然后我们就把一个指向它的引用放到这个新的数组里。

一个类似的结果——不过更难读一些——可以是：

```
for $i (1..10) { @array = somefunc($i); @{$AoA[$i]} = @array; }
```

因为 $AoA 必须是一个新引用，所以该引用自动生成。然后前导的 @ 把这个新引用解引用，结果是 @array 的数值赋予了（在列表环境中）$AoA[$i] 引用的数组。出于程序清晰性的考虑，你可以避免这种写法。

但是有一种情况下你可能要用这种构造。假设 $AoA 已经是一个指向数组的引用的数组。也就是说你要做类似下面这样的赋值：

= $AoA[3] = \@original_array;=

然后我们再假设你要修改 @original_array（也就是要修改 $AoA 的第四行）这样它就指向 @array 的元素。那么下面的代码可以用：

= @{$AoA[3]} = @array;=

在这个例子里，引用本身并不变化，但是被引用数组的元素会变化。这样就覆盖了 @original_array 的数值。

最后，下面的这些看起来很危险的代码将跑得很好：

```
for $i (1..10) {
        my @array = somefunc($i);
        $AoA[$i] = \@array;
}
```

这是因为在循环的每个回合中，词法范围的 my @array 都会重新创建。因此即使看起来好象你每次存储的都是相同的变量的引用，但实际上却不是。这里的区别是非常微小的，但是这样的技巧却可以生成更高效的代码，付出的代价是可能有误导稍微差一些的程序员。（更高效是因为它没有最后赋值中的拷贝。）另一方面，如果你必须拷贝这些数值（也就是循环中第一个赋值干的事情），那么你也可以使用方括号造成的隐含拷贝，因而省略临时变量：

```
for $i (1..10) {
        $AoA[$i] = [ somefunc($i)];
}
```

概括来说：

```
$AoA[$i] = [ @array ];      # 最安全，有时候最快
$AoA[$i] = \@array;         # 快速但危险，取决于数组的自有性
```

```
@{ $AoA[$i] } = @array;       # 有点危险
```

一旦你掌握了数组的数组，你就可以处理更复杂的数据结构。如果你需要 C 结构或者 Pascal 记录，那你在 Perl 里找不到任何特殊的保留字为你设置这些东西。Perl 给你的是更灵活的系统。如果你对记录结构的观念比这样的系统的灵活性差，或者你宁愿给你的用户一些更不透明的，更僵化的东西，那么你可以使用在第十二章，对象，里详细描述的面向对象的特性。

Perl 只有两种组织数据的方式：以排序的列表存储在数组里按位置访问，或者以未排序的键字/数值对存储在散列里按照名字访问。在 Perl 里代表一条记录的最好的方法是用一个散列引用，但是你选择的组织这样的记录的方法是可以变化的。你可能想要保存一个有序的记录列表以便按照编号来访问，这种情况下你不得不用一个散列数组来保存记录。或者，你可能希望按照名字来寻找记录，这种情况下你就要维护一个散列的散列。你甚至可以两个同时用，这时候就是伪散列。

在随后的各节里，你会看到详细地讲述如何构造（从零开始），生成（从其他数据源），访问，和显示各种不同的数据结构的代码。我们首先演示三种直截了当的数组和散列的组合，然后跟着一个散列函数和更不规则的数据结构。最后我们以一个如何保存这些数据结构的例子结束。我们在讲这些例子之前假设你已经熟悉了我们在本章中前面已经设置了的解释集。

9.2 数组的散列

如果你想用一个特定的字串找出每个数组，而不是用一个索引数字找出它们来，那么你就要用数组的散列。在我们电视角色的例子里，我们不是用第零个，第一个等等这样的方法查找该名字列表，而是设置成我们可以通过给出角名字找出演员列表的方法。

因为我们外层的数据结构是一个散列，因此我们无法对其内容进行排序，但是我们可以使用 sort 函数声明一个特定的输出顺序。

9.2.1 数组的散列的组成

你可以用下面的方法创建一个匿名数组的散列：

```
# 如果键字是标识符，我们通常省略引号
%HoA = (
flintstones     => [ "fred", "barney" ],
```

```
jetsons   => [ "george", "jane", "elroy" ],
simpsons   => [ "homer", "marge", "bart" ],
);
```

要向散列增加另外一个数组，你可以简单地说：

```
$HoA{teletubbies} = [ "tinky winky", "dipsy", "laa-laa", "po" ];
```

9.2.2 生成数组的散列

下面是填充一个数组的散列的技巧。从下面格式的文件中读取出来：

```
flintsotnes:    fred barney wilma dino
jetsons:   george jane elroy
simpsons:   homer marge bart
```

你可以用下列两个循环之一：

```
while( <> ) {
        next unless s/^(.*?):\S*//;
        $HoA{$1} = [ split ];
}

while ( $line = <> ) {
        ($who, $rest) = split /:\S*/, $line, 2;
        @fields = spilt ' ', $rest;
        $HoA{$who} = [ @fields ];
}
```

如果你有一个子过程叫 get_family，它返回一个数组，那么你可以用下面两个循环之一填充 %HoA：

```
for $group ( "simpsons", "jetsons", "flintstones" ) {
```

```
        $HoA{$group} = [ get_family($group) ];
}

for $group ( "simpsons", "jetsons", "flintstones" ) {
        @members = get_family($group);
        $HoA{$group} = [ @members ];
}
```

你可以用下面的方法向一个已存在的数组追加新成员：

```
   push @{ $HoA{flintstones}}, "wilma", "pebbles";
```

9.2.3 访问和打印数组的散列

你可以用下面的方法设置某一数组的第一个元素：

```
$HoA{flintstones}[0] = "Fred";
```

要让第二个 Simpson 变成大写，那么对合适的数组元素进行一次替换：

```
$HoA{simpsons}[1] =~ s/(\w)/\u$1/;
```

你可以打印所有这些家族，方法是遍历该散列的所有键字：

```
for $family ( keys %HoA ){
        print "$family: @{ $HoA{$family} }\n";
}
```

我们稍微多做一些努力，你就可以一样追加数组索引：

```
for $family ( keys %HoA ) {
        print "$family: ";
        for $i ( 0 .. $#{ $HoA{$family} }) {
                print " $i = $HoA{$family}[$i]";
```

```
        }
        print "\n";
}
```

或者通过以数组拥有的元素个数对它们排序：

```
for $family ( sort { @{$HoA{$b}} <=> @{$HoA{$a}} } keys %HoA ){
        print "$family: @{ $HoA{$family}}\n";
}
```

或者甚至可以是以元素的个数对数组排序然后以元素的 ASCII 码顺序进行排序（准确地说是 utf8 的顺序）：

```
# 打印以成员个数和名字排序的所有内容
for $family ( sort { @{$HoA{$b}} <=> @{$HoA{$a}}} keys %HoA) {
        print "$family: ", join(", ", sort @{$HoA{$family}}), "\n";
}
```

9.3 散列的数组

如果你有一堆记录，你想顺序访问它们，并且每条记录本身包含一个键字/数值对，那么散列的数组就很有用。在本章中，散列的数组比其他结构用得少一些。

9.3.1 组成一个散列的数组

你可以用下面方法创建一个匿名散列的数组：

```
@AoH = (
{
        husband => "barney",
        wife    => "betty",
        son     => "bamm bamm",
},
```

```
{
        husband => "george",
        wife    => "jane",
        son     => "elroy",
},
{
        husband => "homer",
        wife    => "marge",
        son     => "bart",
},
);
```

要向数组中增加另外一个散列，你可以简单地说：

```
push @AoH, { husband => "fred", wife => "wilma", daughter => "pebbles" };
```

9.3.2 生成散列的数组

下面是一些填充散列数组的技巧。要从一个文件中读取下面的格式：

```
   husband=fred friend=barney
```

你可以使用下面两个循环之一：

```
while (<>) {
        $rec = {};
        for $field ( split ) {
                ($key, $value) = split /=/, $field;
                $rec->{$key} = $value;
        }
```

```
        push @AoH, $rec;
}

while (<>) {
        push @AoH, { split /[\s=]+/ };
}
```

如果你有一个子过程 get_next_pair 返回一个键字/数值对，那么你可以用它利用下面两个循环之一来填充 @AoH：

```
while ( @fields = get_next_pari()) {
        push @AoH, {@fields};
}

while (<>) {
        push @AoH, { get_next_pair($_) };
}
```

你可以象下面这样向一个现存的散列附加新的成员：

```
$AoH[0]{pet} = "dino";
$AoH[2]{pet} = "santa's little helper";
```

9.3.3 访问和打印散列的数组

你可以用下面的方法设置一个特定散列的数值/键字对：

```
$AoH[0]{husband} = "fred";
```

要把第二个数组的丈夫（husband）变成大写，用一个替换：

```
$AoH[1]{husband} =~ s/(\w)/\u$1/;
```

你可以用下面的方法打印所有的数据：

```
for $href ( @AoH ) {
        print "{ ";
                for $role ( keys %$href ) {
                        print "$role=$href->{$role} ";
                }
                print "}\n";
}
```

以及带着引用打印：

```
for $i ( 0 .. $#AoH ) {
        print "$i is { ";
                for $role ( keys %{ $AoH[$i] } ) {
                        print "$role=$AoH[$i]{$role} ";
                }
                print "}\n";
}
```

9.4 散列的散列

多维的散列是 Perl 里面最灵活的嵌套结构。它就好象绑定一个记录，该记录本身包含其他记录。在每个层次上，你都用一个字串（必要时引起）做该散列的索引。不过，你要记住散列里的键字/数值对不会以任何特定的顺序出现；你可以使用 sort 函数以你喜欢的任何顺序检索这些配对。

9.4.1 构成一个散列的散列

你可以用下面方法创建一个匿名散列的散列：

```
%HoH = (
flintstones => {
```

```
        husband => "fred",
        pal     => "barney",
},
jetsons => {
        husband => "george",
        wife    => "jane",
        "his boy" => "elroy",       # 键字需要引号
},
simpsons => {
        husband => "homer",
        wife    => "marge",
        kid     => "bart",
},


);
```

要向 %HoH 中增加另外一个匿名散列，你可以简单地说：

```
$HoH{ mash } = {
        captain => "pierce",
        major   => "burns",
        corporal=> "radar",
}
```

9.4.2 生成散列的散列

下面是一些填充一个散列的散列的技巧。要从一个下面格式的文件里读取数据：

flintstones

husband=fred pal=barney wife=wilma
pet=dino

你可以使用下面两个循环之一：

```
while( <> ){
        next unless s/^(.*?):\S*//;
        $who = $1;
        for $field ( split ) {
                ($key, $value) = split /=/, $field;
                $HoH{$who}{$key} = $value;
        }
}


while( <> ){
        next unless s/^(.*?):\S*//;
        $who = $1;
        $rec = {};
        $HoH{$who} = $rec;
        for $field ( split ) {
                ($key, $value) = split /=/, $field;
                $rec->{$key} = $value;
        }
}
```

如果你有一个子过程 get_family 返回一个键字/数值列表对，那么你可以拿下面三种方法的任何一种，用它填充 %HoH：

```
for $group ("simpsons", "jetsons", "flintstones" ) {
```

```
        $HoH{$group} = {get_family($group)};
}

for $group ( "simpsons", "jetsons", "flintstones" ) {
        @members = get_family($group);
        $HoH{$group} = {@menbers};
}

sub hash_families {
        my @ret;
        for $group (@_) {
                push @ret, $group, {get_family($group)};
        }
        return @ret;
}

%HoH = hash_families( "simpsons", "jetsons", "flintstones" );
```

你可以用下面的方法向一个现有的散列附加新的成员：

```
%new_floks = (
wife => "wilma",
pet  => "dino",
);

for $what (keys %new_floks) {
```

```
        $HoH{flintstones}{$what} = $new_floks{$what};
}
```

9.4.3 访问和打印散列的散列

你可以用下面的方法设置键字/数值对：

```
    $HoH{flintstones}{wife} = "wilma";
```

要把某个键字/数值对变成大写，对该元素应用一个替换：

```
    $HoH{jetsons}{'his boy'} =~ s/(\w)/\u$1/;
```

你可以用先后遍历内外层散列键字的方法打印所有家族：

```
for $family ( keys %HoH ) {
        print "$family: ";
        for $role ( keys %{ $HoH{$family} } ){
                print "$role=$person ";
        }
        print "\n";
}
```

在非常大的散列里，可能用 each 同时把键字和数值都检索出来会略微快一些（这样做可以避免排序）：

```
while ( ($family, $roles) = each %HoH ) {
        print "$family: ";
        while ( ($role, $person) = each %$roles ) {
                print "$role=$person";
        }
        print "\n";
}
```

（糟糕的是，需要存储的是那个大的散列，否则你在打印输出里就永远找不到你要的东西.）你可以用下面的方法先对家族排序然后再对脚色排序：

```
for $family ( sort keys %HoH ) {
        print "$family:  ";
        for $role ( sort keys %{ $HoH{$family} } ) {
                print "$role=$HoH{$family}{$role} ";
        }
        print "\n";
}
```

要按照家族的编号排序（而不是 ASCII 码（或者 utf8 码）），你可以在一个标量环境里使用 keys：

```
for $family ( sort { keys %{$HoH{$a} } <=> keys %{$HoH{$b}}} keys %HoH ) {
        print "$family: ";
        for $role ( sort keys %{ $HoH{$family} } ) {
                print "$role=$HoH{$family}{$role};
        }
        print "\n";
}
```

要以某种固定的顺序对一个家族进行排序，你可以给每个成员赋予一个等级来实现：

```
$i = 0;
for ( qw(husband wife son daughter pal pet) ) { $rank{$_} = ++$i }

for $family ( sort { keys %{$HoH{$a} } <=> keys %{$HoH{$b}}} keys %HoH ) {
        print "$family: ";
        for $role ( sort { $rank{$a} <=> $rank{$b} } keys %{ $HoH{$family} }) {
```

```
            print "$role=$HoH{$family}{$role} ";
        }
        print "\n";
}
```

9.5 函数的散列

在使用 Perl 书写一个复杂的应用或者网络服务的时候，你可能需要给你的用户制作一大堆命令供他们使用。这样的程序可能有象下面这样的代码来检查用户的选择，然后采取相应的动作：

```
if    ($cmd =~ /^exit$/i)      { exit }
elsif ($cmd =~ /^help$/i)      { show_help() }
elsif ($cmd =~ /^watch$/i)     { $watch = 1 }
elsif ($cmd =~ /^mail$/i)      { mail_msg($msg) }
elsif ($cmd =~ /^edit$/i)      { $edited++; editmsg($msg); }
elsif ($cmd =~ /^delete$/i)    { confirm_kill() }
else {
    warn "Unknown command: `$cmd'; Try `help' next time\n";
}
```

你还可以在你的数据结构里保存指向函数的引用，就象你可以存储指向数组或者散列的引用一样：

```
%HoF = (                          # Compose a hash of functions
    exit    =>  sub { exit },
    help    =>  \&show_help,
    watch   =>  sub { $watch = 1 },
    mail    =>  sub { mail_msg($msg) },
```

```
    edit    =>  sub { $edited++; editmsg($msg); },
    delete  =>  \&confirm_kill,
);

if   ($HoF{lc $cmd}) { $HoF{lc $cmd}->() }   # Call function
else { warn "Unknown command: `$cmd'; Try `help' next time\n" }
```

在倒数第二行里，我们检查了声明的命令名字（小写）是否在我们的“遣送表”%HoF 里存在。如果是，我们调用响应的命令，方法是把散列值当作一个函数进行解引用并且给该函数传递一个空的参数列表。我们也可以用 &{ $HoF{lc $cmd} }( ) 对散列值进行解引用，或者，在 Perl 5.6 里，可以简单地是 $HoF{lc $cmd} ()。

9.6 更灵活的记录

到目前为止，我们在本章看到的都是简单的，两层的，同质的数据结构：每个元素包含同样类型的引用，同时所有其他元素都在该层。数据结构当然可以不是这样的。任何元素都可以保存任意类型的标量，这就意味着它可以是一个字串，一个数字，或者指向任何东西的引用。这个引用可以是一个数组或者散列引用，或者一个伪散列，或者是一个指向命名或者匿名函数的引用，或者一个对象。你唯一不能干的事情就是向一个标量里填充多个引用物。如果你发现自己在做这种尝试，那就表示着你需要一个数组或者散列引用把多个数值压缩成一个。

在随后的节里，你将看到一些代码的例子，这些代码设计成可以演示许多你想存储在一个记录里的许多可能类型的数据，我们将用散列引用来实现它们。这些键字都是大写字串，这是我们时常使用的一个习惯（有时候也不用这个习惯，但只是偶然不用）——如果该散列被用做一个特定的记录类型。

9.6.1 更灵活的记录的组合，访问和打印

下面是一个带有六种完全不同的域的记录：

```
$rec = {
        TEXT        => $string,
        SEQUENCE    => [ @old_values ],
        LOOKUP      => { %some_table },
```

```
        THATCODE    => sub { $_[0] ** $_[1] },
        HANDLE      => \*STDOUT,
};
```

TEXT 域是一个简单的字串。因此你可以简单的打印它：

```
print $rec->{TEXT};
```

SEQUENCE 和 LOOKUP 都是普通的数组和散列引用：

```
print $rec->{SEQUENCE    }[0];
$last = pop @{ $rec->{SEQUENCE} };

print $rec->{LOOKUP}{"key"};
($first_k, $first_v) = each %{ $rec->{LOOKUP} };
```

THATCODE 是一个命名子过程而 THISCODE 是一个匿名子过程，但是它们的调用是一样的：

```
$that_answer = $rec->{THATCODE}->($arg1, $arg2);
$this_answer = $rec->{THISCODE}->($arg1, $arg2);
```

再加上一对花括弧，你可以把 $rec->{HANDLE} 看作一个间接的对象：

```
print { $rec->{HANDLE} } "a string \n";
```

如果你在使用 FileHandle[2] 模块，你甚至可以把该句柄看作一个普通的对象：

```
use FileHandle;
$rec->{HANDLE}->autoflush(1);
$rec->{HANDLE}->print("a string\n");
```

9.6.2 甚至更灵活的记录的组合，访问和打印

自然，你的数据结构的域本身也可以是任意复杂的数据结构：

```
%TV = (
```

```
flintstones => {
        series   => "flintstones",
        nights   => [ "monday", "thursday", "friday" ],
        members  => [
        { name => "fred",    role => "husband", age  => 36, },
        { name => "wilma",   role => "wife",    age  => 31, },
        { name => "pebbles", role => "kid",     age  =>  4, },
        ],
},



jetsons     => {
        series   => "jetsons",
        nights   => [ "wednesday", "saturday" ],
        members  => [
        { name => "george",  role => "husband", age  => 41, },
        { name => "jane",    role => "wife",    age  => 39, },
        { name => "elroy",   role => "kid",     age  =>  9, },
        ],
},


simpsons    => {
        series   => "simpsons",
        nights   => [ "monday" ],
```

```
        members  => [
        { name => "homer", role => "husband", age => 34, },
        { name => "marge", role => "wife",    age => 37, },
        { name => "bart",  role => "kid",     age => 11, },
        ],
},
);
```

9.6.3 复杂记录散列的生成

因为 Perl 分析复杂数据结构相当不错，因此你可以把你的数据声明作为 Perl 代码放到一个独立的文件里，然后用 do 或者 require 等内建的函数把它们装载进来。另外一种流行的方法是使用 CPAN 模块（比如 XML::Parser）装载那些用其他语言（比如 XML）表示的任意数据结构。

你可以分片地制作数据结构：

```
$rec = {};
$rec->{series} = "flintstones";
$rec->{nights} = [ find_days()];
```

或者从文件里把它们读取进来（在这里,我们假设文件的格式是 field=value 语法）：

```
@members = ();
while (<>) {
        %fields = split /[\s=]+/;
        push @members, {%fields};
}
$rec->{members} = [ @members ];
```

然后以一个子域为键字，把它们堆积到更大的数据结构里：

$TV{ $rec->{series} } = $rec;

你可以使用额外的指针域来避免数据的复制。比如，你可能需要在一个人的记录里包含一个“kids”（孩子）数据域，这个域可能是一个数组，该数组包含着指向这个孩子自己记录的引用。通过把你的数据结构的一部分指向其他的部分，你可以避免因为在一个地方更新数据而没有在其他地方更新数据造成的数据倾斜：

```
for $family (keys %TV) {
        my $rec = $TV{$family};       # 临时指针
        @kids = ();
        for $person ( @{$rec->{members}}) {
                if ($person->{role} =~ /kid|son|daughter/) {
                        push @kids, $person;
                }
        }
        # $rec 和 $TV{$family} 指向相同的数据!
        $rec->{kids} = [@kids];
}
```

这里的 $rec->{kids} = [@kids] 赋值拷贝数组内容——但它们只是简单的引用，而没有拷贝数据。这就意味着如果你给 Bart 赋予下面这样的年龄：

$TV{simpsons}{kids}[0]{age}++; # 增加到 12

那么你就会看到下面的结果，因为 $TV{simpsons}{kids}[0] 和 $TV{simpsons}{members}[2] 都指向相同的下层匿名散列表：

print $TV{simpsons}{members}[2]{age}; # 也打印 12

现在你打印整个 %TV 结构：

```
for $family ( keys %TV ) {
        print "the $family";
```

```
        print " is on ", join (" and ", @{ $TV{$family}{nights} }), "\n";
        print "its members are:\n";
        for $who ( @{ $TV{$family}{members} } ) {
                print " $who->{name} ($who->{role}), age $who->{age}\n";
        }
        print "children: ";
        print join (", ", map { $_->{name} } @{ $TV{$family}{kids} } );
        print "\n\n";
}
```

9.7 保存数据结构

如果你想保存你的数据结构以便以后用于其他程序，那么你有很多方法可以用。最简单的方法就是使用 Perl 的 Data::Dumper 模块，它把一个（可能是自参考的）数据结构变成一个字串，你可以把这个字串保存在程序外部，以后用 eval 或者 do 重新组成：

```
use Data::Dumper;
$Data::Dumper::Purity = 1;        # 因为 %TV 是自参考的
open (FILE, "> tvinfo.perldata")    or die "can't open tvinfo: $!";
print FILE Data::Dumper->Dump([\%TV], ['*TV']);
close FILE          or die "can't close tvinfo: $!";
```

其他的程序（或者同一个程序）可以稍后从文件里把它读回来：

```
open (FILE, "< tvinfo.perldata")   or die "can't open tvinfo: $!";
undef $/;              # 一次把整个文件读取进来
eval ;             # 重新创建 %TV
die "can't recreate tv data from tvinfo.perldata: $@" if $@;
close FILE          or die "can't close tvinfo: $!";
```

```
print $TV{simpsons}{members}[2]{age};
```

或者简单的是：

```
do "tvinfo.perldata"      or die "can't recreate tvinfo: $! $@";
print $TV{simpsons}{members}[2]{age};
```

还有许多其他的解决方法可以用，它们的存储格式的范围从打包的二进制（非常快）到 XML（互换性非常好）。检查一下靠近你的 CPAN 镜象！

第十章 包

在本章里，我们开始有好玩的东西了，因为我们要开始讲有关软件设计的东西。如果我们要聊一些好的软件设计，那么我们就必须先侃侃懒惰，急燥，和傲慢，这几样好的软件设计需要的基本要素。

我们经常落到使用拷贝和粘贴（ICP-I Copy & Paste）的陷阱里，而如果一个循环或者一个子过程就足够了，（注：这是伪懒惰的一种形式）那么这时候我们实际上应该定义一个更高层次的抽象。但是，有些家伙却走向另外一个极端，定义了一层又一层的高层抽象，而这个时候他们应该用拷贝和粘贴。（注：这是伪傲慢的一种形式。）不过，通常来讲，我们大多数人都应该考虑使用更多的抽象。

落在中间的是那些对抽象深度有平衡观念的人，不过他们马上就开始写它们自己的抽象层，而这个时候它们应该重用现有的代码。（注：你也许已经猜到了——这是为急燥。不过，如果你准备推倒重来，那么你至少应该发明一种更好的东西。）

如果你准备做任何这样的事情，那么你都应该坐下来想想，怎样做才能从长远来看对你和你的邻居最有好处。如果你准备把你的创造力引擎作用到一小块代码里，那么为什么不把这个你还要居住的这个世界变得更美好一些呢？（即使你的目的只是为了程序的成功，那你就要确信你的程序能够符合社会生态学的要求。）

朝着生态编程的第一步是：不要在公园里乱丢垃圾（译注：否则砸到小朋友...或者花花草草...：）。当你写一段代码的时候，考虑一下给这些代码自己的名字空间，这样你的变量和函数就不会把别人的变量和函数搞砸了，反之亦然。名字空间有点象你的家，你的家里想

怎么乱都行，只要你保持你的外部界面对其他公民来说是适度文明的就可以了。在 Perl 里，一个名字空间叫一个包。包提供了基本的制作块，在它上面构造更高级的概念，比如模块和类等。

和"家"的说法相似，"包"的说法也有一些模糊。包独立于文件。你可以在一个文件里有许多包，或者是一个包跨越多个文件，就好象你的家可以是在一座大楼里面的小小的顶楼（如果你是一个穷困潦倒的艺术家），或者你的家也可以由好多建筑构成（比如你的名字叫伊丽沙白女王）。但家的常见大小就是一座建筑，而包通常也是一个文件大，Perl 给那些想把一个包放到一个文件里的人们提供了一些特殊的帮助，条件只是你愿意给文件和包相同的名字并且使用一个 .pm 的扩展名，pm 是"perl module" 的缩写。模块（module）是 Perl 里重复使用的最基本的模块。实际上，你使用模块的方法是 use 命令，它是一个编译器指示命令，可以控制从一个模块里输入子过程和变量。到目前为止你看到的每一个 use 的例子都是模块复用的例子。

如果其他人认为你的模块有用，那么你应该把它们放到 CPAN。Perl 的繁荣是和程序员愿意和整个社区分享他们劳动的果实分不开的。自然，CPAN 也是你可以找到那些其他人已经非常仔细地上载上去给别人用的模块的地方。参阅第二十二章， CPAN，以及 www.cpan.org 获取详细信息。

过去 25 年左右的时间里，设计计算机语言的趋势是强调某种偏执。你必须编制每一个模块，就好象它是一个围城的阶段一样。显然有些封建领地式的文化可以使用这样的方法，但并不是所有文化都喜欢这样。比如，在 Perl 文化里，人们让你离它们的房子远一点是因为他们没有邀请你，而不是因为窗户上有窗栅。（注：不过，如果需要， Perl 提供了一些窗栅。参阅第二十三章，安全，里的"处理不安全数据"。）

这本书不是讲面向对象的方法论的，并且我们在这里也不想把你推到面向对象的狂热中去，就算你想进去我们的态度也这样。关于这方面的东西已经有大量书籍了。Perl 对面向对象设计的原则和 Perl 对其他东西的原则是一样的：在面向对象的设计方法有意义的地方就用它，而在没有意义的地方就绕开它。你的选择。

在 OO 的说法中，每个对象都属于一个叫做类的组。在 Perl 里，类和包以及模块之间的关系是如此地密切，以至于许多新手经常认为它们是可以互换的。典型的类是用一个定义了与该类同名的包名字的模块实现的。我们将在随后的几章里解释这些东西。

当你 use 一个模块的时候，你是从软件复用中直接受益。如果你用了类，那么如果一个类通过继承使用了另外一个类，那么你是间接地从软件复用中受益。而且用了类，你就获得了更多的一些东西：一个通往另外一个名字空间的干净的接口。在类里面，所有东西都是间接地访问的，把这个类和外部的世界隔离开。

就象我们在第八章，引用，里提到的一样，在 Perl 里的面向对象的编程是通过引用来实现的，这些引用的引用物知道它们属于哪些类。实际上，如果你知道引用，那么你就知道几乎所有有关对象的困难。剩下的就是"放在你的手指下面"，就象画家会说的那样。当然，你需要做一些练习。

你的基本练习之一就学习如何保护不同的代码片段，避免被其他人的变量不小心篡改。每段代码都属于一个特定的包，这个包决定它里面有哪些变量和代码可以使用。当 Perl 碰到一段代码的时候，这段代码就被编译成我们叫做当前包的东西。最初的当前包叫做"main"，不过你可以用 package 声明在任何时候把当前的包切换成另外一个。当前包决定使用哪个符号表查找你的变量，子过程，I/O 句柄和格式等。

任何没有和 my 关联在一起的变量声明都是和一个包相关联的——甚至是一些看起来无所不在的变量，比如 $_ 和 %SIG。实际上，在 Perl 里实际上没有全局变量这样的东西。（特殊的标识符，比如 _ 和 SIG，只是看上去象全局变量，因为它们缺省时属于 main 包，而不是当前包。）

package 声明的范围从声明本身开始直到闭合范围的结束（块，文件，或者 eval—— 以先到为准）或者直到其他同层次的 package 声明，它会取代前面的那个。（这是个常见的实践。）

所有随后的标识符（包括那些用 our 声明的，但是不包括那些用 my 或者那些用其他包名字修饰的的变量。）都将放到属于当前包的符号表中。（用 my 声明的变量独立于包；它们总是属于包围它们的闭合范围，而且也只属于这个范围，不管有什么包声明。）

通常，一个 package 声明如果是一个文件的第一个语句的话就意味着它将被 require 或者 use 包含。但这只是习惯，你可以在任何可以放一条语句的地方放一个 package 声明。你甚至可以把它放在一个块的结尾，这个时候它将没有任何作用。你可以在多于一个的地方切换到一个包里面；包声明只是为该块剩余的部分选择将要使用的符号表。（这也是一个包实现跨越多个文件的方法。）

你可以引用其他包里的标识符（注：我们说的标识符的意思是用做符号表键字的东西，可以用来访问标量变量，数组变量，子过程，文件或者目录句柄，以及格式等。从语法上来说，标签（Label）也是标识符，但是它们不会放到特定的符号表里；相反，它们直接附着在你的程序里的语句上面。标签不能用包名字修饰。），方法是用包名字和双冒号做前缀（"修饰"）：$Package::Variable。如果包名字是空，那么就假设为 main 包。也就是说，$::sail 等于 $main::sail。（注：为了把另外一点容易混淆的概念理清楚，在变量名 $main::sail 里，我们对 main 和 sail 使用术语 "标识符"，但不把 main::sail 称做标识符。我们叫它一个变量名。因为标识符不能包含冒号。）

老的包分隔符还是一个单引号，因此在老的 Perl 程序里你会看到象 $main'sail 和 $somepack'horse 这样的变量。不过，双冒号是现在的优选的分隔符，部分原因是因为它更具有可读性，另一部分原因是它更容易被 emacs 的宏读取。而且这样表示也令 C++ 程序员觉得明白自己在做什么——相比之下，用单引号的时候就能让 Ada 的程序员知道自己在做什么。因为出于向下兼容的考虑，Perl 仍然支持老风格的语法，所以如果你试图使用象 "This is $owner's house" 这样的字串，那么你实际上就是在访问 $owner::s；也就是说，在包 owner 里的 $s 变量，这可能并不是你想要的。你可以用花括弧来消除歧义，就象 "This is ${owner}'s house"。

双冒号可以用于把包名字里的标识符链接起来：$Red::Blue::Var。这就意味着 $var 属于 Red::Blue 包。Red::Blue 包和任何可能存在的 Red 或者 Blue 包都没有关系。也就是说，在 Red::Blue 和 Red 或者 Blue 之间的关系可能对那些书写或使用这个程序的人有意义，但是它对 Perl 来说没有任何意义。（当然，在当前的实现里，符号表 Red::Blue 碰巧存储在 Red 符号表里。但是 Perl 语言对此没有做任何直接的利用。）

由于这个原因，每个 package 声明都必须声明完整的包名字。任何包名字都没有做任何隐含的"前缀"的假设，甚至（看起来象）在一些其他包声明的范围里声明的那样也如此。

只有标识符（以字母或者一个下划线开头的名字）才存储在包的符号表里。所有其他符号都保存在 main 包里，包括所有非字母变量，比如 $!，$?，和 $_。另外，在没有加以修饰的时候，标识符 STDIN，STDOUT，STDERR，ARGV，ARGVOUT，ENV，INC，和 SIG 都强制在包 main 里，即使你是用做其他目的，而不是用做它们的内建功能也如此。不要把你的包命名为 m，s，tr，q，qq，qr，qw，或者 qx，除非你想自找一大堆麻烦。比如，你不能拿修饰过的标识符形式做文件句柄，因为它将被解释成一个模式匹配，一个替换，或者一个转换。

很久以前，用下划线开头的变量被强制到 main 包里，但是我们发现让包作者使用前导的下划线作为半私有的标识符标记更有用，这样它们就可以表示为只被该包内部使用。（真正私有的变量可以声明为文件范围的词汇，但是只有在包和模块之间有一对一的关系的时候，这样的做法才比较有效，虽然这样的一对一比较普遍，但并不是必须的。）

%SIG 散列（用于捕获信号；参阅第十六章，进程间通讯）也是特殊的。 如果你把一个信号句柄定义为字串，那么 Perl 就假设它引用一个 main 包里的子过程，除非明确地使用了其他包名字。如果你想声明一个特定的包，那么你要使用一个信号句柄的全称，或者完全避免字串的使用：方法是改为赋予一个类型团或者函数引用：

```
$SIG{QUIT} = "Pkg::quit_chatcher";   # 句柄全称

$SIG{QUIT} = "quit_catcher";   # 隐含的"main::quit_catcher"
```

```
$SIG{QUIT} = *quit_catcher;       # 强制为当前包的子过程

$SIG{QUIT} = \&quit_catcher;    # 强制为当前包的子过程

$SIG{QUIT} = sub { print "Caught SIGQUIT\n" };    # 匿名子过程
```

"当前包"的概念既是编译时的概念也是运行时的概念。大多数变量名查找发生在编译时，但是运行时查找发生在符号引用解引用的时候，以及 eval 分析新的代码的时候。实际上，在你 eval 一个字串的时候，Perl 知道该 eval 是在哪个包里调用的并且在计算该字串的时候把那个包名字传播到 eval 里面。（当然，你总是可以在 eval 里面切换到另外一个包，因为一个 eval 字串是当作一个块对待的，就象一个用 do，require，或者 use 装载的块一样。）

另外，如果一个 eval 想找出它在哪个包里，那么特殊的符号 ***PACKAGE*** 包含当前包名字。因为你可以把它当作一个字串看待，所以你可以把它用做一个符号引用来访问一个包变量。但如果你在这么做，那么你很有机会把该变量用 our 声明，作为一个词法变量来访问。

10.1 符号表

一个包的内容总体在一起称做符号表。符号表都存储在一个散列里，这个散列的名字和该包的名字相同，但是后面附加了两个冒号。因此 main 符号表的名字是 %main::。因为 main 碰巧也是缺省的包，Perl 把 %:: 当作 %main:: 的缩写。

类似，Red::Blue 包的符号表名字是 %Red::Blue::。同时 main 符号表还包含所有其他顶层的符号表，包括它本身。因此 %Red::Blue:: 同时也是 %main::Red::Blue::。

当我们说到一个符号表"包含"其他的符号表的时候，我们的意思是它包含一个指向其他符号表的引用。因为 main 是顶层包，它包含一个指向自己的引用，结果是 %main:: 和 %main::main::，和 %main::main::main::，等等是一样的，直到无穷。如果你写的代码包括便历所有的符号表，那么一定要注意检查这个特殊的情况。

在符号表的散列里，每一对键字/数值对都把一个变量名字和它的数值匹配起来。键字是符号标识符，而数值则是对应的类型团。因此如果你使用 *NAME 表示法，那么你实际上只在访问散列里的一个数值，该数值保存当前包的符号表。实际上，下面的东西有（几乎）一样的效果：

```
*sym = *main::variable;

*sym = $main::{"variable"};
```

第一种形式更高效是因为 main 符号表是在编译时被访问的。而且它还会在该名字的类型团不存在的时候创建一个新的，但是第二种则不会。

因为包是散列，因此你可以找出该包的键字然后获取所有包中的变量。因此该散列的数值都是类型团，你可以用好几种方法解引用。比如：

```
foreach $symname (sort keys %main::) {
        local *sym = $main::{$symname};
        print "\$$symname is defined\n" if defined $sym;
        print "\@$symname is nonnull\n" if   @sym;
        print "\%$symname is nonnull\n" if   %sym;
}
```

因为所有包都可以（直接或间接地）通过 main 包访问，因此你可以在你的程序里写出访问每一个包变量的 Perl 代码。当你用 v 命令要求 Perl 调试器倾倒所有你的变量的时候，它干的事情就是这个。请注意，如果你做这些事情，那么你将看不到用 my 声明的变量，因为它们都是独立于包的，不过你看得到用 our 声明的变量。参阅第二十章，Perl 调试器。

早些时候我们说过除了在 main 里，其他的包里只能存储标识符。我们在这里撒了个小慌：你可以在一个符号表散列里使用任何你需要的字串作为键字——只不过如果你企图直接使用一个非标识符的时候它就不是有效的 Perl：

```
$!@#$%      = 0;          # 错，语法错
#{'!@#$%'}   = 1;          # 正确，用的是未修饰的

${'main::!@#$%'}    = 2;       # 可以在字串里修饰。
print ${ $main::{'!@#$%'}}       # 正确，打印 2
```

给一个匿名类型团赋值执行一个别名操作；也就是，

```
*dick    = *richard;
```

导致所有可以通过标识符 richard 访问的变量，子过程，格式，文件和目录句柄也可以通过符号 dick 访问。如果你只需要给一个特定的变量或者子过程取别名，那么使用一个引用：

```
*dick = \$richard;
```

这样就令 $richard 和 $dick 成为同样的变量，但是 @richard 和 @dick 则剩下来是独立的数组。很高明，是吗？

这也是 Exporter 在从一个包向另外一个包输入符号的时候采用的方法。比如：

```
*SomePack::dick = \&OtherPack::richard;
```

从包 OtherPack[2] 输入 &richard 函数到 SomePack[2]，让它可以当作 &dick 函数用。（ Exporter 模块在下一章描述。）如果你用一个 local 放在赋值前面，那么，该别名将只持续到当前动态范围结束。

这种机制可以用于从一个子过程中检索一个引用，令该引用可以用做一个合适的数据类型：

```
*units = populate();       # 把 \%newhash 赋予类型团
print $units{kg};          # 打印 70；而不用解引用！

sub populate {
    my %newhash = (km => 10, kg => 70);
    return \%newhash;
}
```

类似，您还可以把一个引用传递到一个引用传递到一个子过程里并且不加解引用地使用它：

```
%units = (miles => 6, stones => 11);
fillerup( \%units );       # 传递进一个引用
print $units{quarts};      # 打印 4

sub fillerup {
        local *hashsym = shift;   # 把 \%units 赋予该类型团
        $hashsym{quarts} = 4;   # 影响 \%units；不需要解引用！
}
```

上面这些都是廉价传递引用的巧妙方法，用在你不想明确地对它们进行解引用的时候。请注意上面两种技巧都是只能对包变量管用；如果我们用 my 声明了 %units 那么它们不能运行。

另外一个符号表的用法是制作"常量"标量：

```
*PI = \3.14159265358979;
```

现在你不能更改 $PI，不管怎么说这样做可能是好事情。它和常量子过程不一样，常量子过程在编译时优化。常量子过程是一个原型定义为不接受参数并且返回一个常量表达式的子过程；参阅第六章，子过程，的"内联常量函数"获取细节。use constant 用法（参阅第三十一章，用法模块）是一个方便的缩写：

```
use constant PI => 3.14159;
```

在这个钩子下面，它使用 *PI 的子过程插槽，而不是前面用的标量插槽。它等效于更紧凑（不过易读性稍微差些）：

```
*PI   =   sub ()   {3.14159};
```

不过，这是一个很值得知道的俗语——把一个 sub {} 赋予一个类型团是在运行时给匿名子过程赋予一个名字的方法。

把一个类型团引用赋予另外一个类型团（*sym = *oldvar）和赋予整个类型团是一样的。并且如果你把类型团设置为一个简单的字串，那么你就获得了该字串命名的整个类型团，因为 Perl 在当前符号表中寻找该字串。下面的东西互相都是一样的，不过头两个在编译时计算符号表记录，而后面两个是在运行时：

```
*sym =   *oldvar;

*sym = \*oldvar;       # 自动解引用

*sym = *{"oldvar"};      # 明确的符号表查找

*sym = "oldvar";       # 隐含地符号表查找
```

当你执行任意下列的赋值的时候,你实际上只是替换了类型团里的一个引用:

```
*sym = \$frodo;

*sym = \@sam;

*sym = \%merry;
```

```
*sym = \&pippin;
```

如果你从另外一个角度来考虑问题，类型团本身可以看作一种散列，它里面有不同类型的变量记录。在这种情况下，键字是固定的，因为一个类型团正好可以包含一个标量,一个散列，等等。但是你可以取出独立的引用，象这样：

```
*pkg::sym{SCALAR}       # 和 \$pkg::sym 一样
*pkg::sym{ARRAY}       # 和 \@pkg::sym 一样
*pkg::sym{HASH}                # 和 \%pkg::sym 一样
*pkg::sym{CODE}                # 和 \&pkg::sym 一样
*pkg::sym{GLOB}                 # 和 \*pkg::sym 一样
*pkg::sym{IO}                # 内部的文件/目录句柄，没有直接的等价物
*pkg::sym{NAME}                # “sym”（不是引用）
*pkg::sym{PACKAGE}          # “pkg”（不是引用）
```

你可以通过说 *foo{PACKAGE} 和 *foo{NAME} 找出 *foo 符号表记录来自哪个名字和包。这个功能对那些传递类型团做参数的子过程里很有用：

```
sub identify_typeglob {
        my $glob = shift;
        print 'You gave me ', *{glob}{PACKAGE}, '::', *{$globa}{NAME}, "\n";
}

identify_typeglob(*foo);
identify_typeglob(*bar::glarch);
```

它打印出：

```
You gave me main::foo
You gave me bar::glarch
```

*foo{THING} 表示法可以用于获取指向 *foo 的独立元素的引用。参阅第八章的"符号表引用"一节获取细节。

这种语法主要用于获取内部文件句柄或者目录句柄引用，因为其他内部引用已经都可以用其他方法访问。（老的 *foo{FILEHANDLE} 形式仍然受支持，表示 *foo{IO}，但是不要让这个名字把你涮了，它可不能把文件句柄和目录句柄区别开。）但是我们认为应该概括它，因为它看起来相当漂亮。当然，你可能根本不用记住这些东西，除非你想再写一个 Perl 调试器。

10.2 自动装载

通常，你不能调用一个没有定义的子过程。不过，如果在未定义的子过程的包（如果是在对象方法的情况下，在任何该对象的基类的包里）里有一个子过程叫做 AUTOLOAD，那么就会调用 AUTOLOAD 子过程，同时还传递给它原本传递给最初子过程的同样参数。你可以定义 AUTOLOAD 子过程返回象普通子过程那样的数值，或者你可以让它定义还不存在的子过程然后再调用它，就好象该子过程一直存在一样。

最初的子过程的全称名会神奇地出现在包全局变量 $AUTOLOAD 里，该包和 AUTOLOAD 所在的包是同一个包。下面是一个简单的例子，它会礼貌地警告你关于未定义的子过程调用，而不是退出：

```
sub AUTOLOAD {
        our $AUTOLOAD;
        warn "Attempt to call $AUTOLOAD failed.\n";
}


blarg(10);      # 我们的 $AUTOLOAD 将会设置为 main::blarg
print "Still alive!\n";
```

或者你可以代表该未定义的子过程返回一个数值：

```
sub AUTOLOAD {
        our $AUTOLOAD;
        return "I see $AUTOLOAD(@_)\n";
```

```
}

print blarg(20);        # 打印：I see main::blarg(20)
```

你的 AUTOLOAD 子过程可以用 eval 或者 require 为该未定义的子过程装载一个定义，或者是用我们前面讨论过的类型团赋值的技巧，然后使用特殊形式的 goto 执行该子过程，这种 goto 可以不留痕迹地抹去 AUTOLOAD 过程的堆栈桢。下面我们通过给类型团赋予一个闭合来定义该子过程：

```
sub AUTOLOAD {
        my $name = our $AUTOLOAD;
        *$AUTOLOAD = sub { print "I see $name(@_)\n"};
        goto &$AUTOLOAD;        # 重起这个新过程。
}

blarg(30);              # 打印：I see main::blarg(30)
blarg(40);              # 打印：I see main::blarg(40)
blarg(50);              # 打印：I see main::blarg(50)
```

标准的 AutoSplit[2] 模块被模块作者用于把他们的模块分裂成独立的文件（用以 .al 结尾的文件），每个保存一个过程。该文件放在你的系统的 Perl 库的 auto/ 目录里，在那之后该文件可以根据标准的 AutoLoader[2] 模块的需要自动装载。

一种类似的方法被 SelfLoader[2] 模块使用，只不过它从该文件自己的 DATA 区自动装载函数，从某种角度来看，它的效率要差一些，但是从其他角度来看，它的效率又比较高。用 AutoLoader[2] 和 SelfLoader[2] 自动装载 Perl 函数是对通过 DynaLoader[2] 动态装载编译好的 C 函数的模拟，只不过自动装载是以函数调用的粒度进行实现的，而动态装载是以整个模块的粒度进行装载的，并且通常会一次链接进入若干个 C 或 C++ 函数。（请注意许多 Perl 程序员不用 AutoSplit[2]，AutoLoader，SelfLoader，或者 DynaLoader[2] 模块过得也很好。你只需要知道它们的存在，以防哪天你不用它还真解决不了问题。）

我们可以在把 AUTOLOAD 过程当作其他接口的封装器中获取许多乐趣。比如，让我们假设任何没有定义的函数应该就是哪它的参数调用 system。你要做的就是：

```
sub AUTOLOAD {
        my $program = our $AUTOLOAD;
        $program =~ s/.*::/;        # 截去包名字
        system($program, @_);
}
```

（恭喜，你刚刚实现了和 Perl 一起发布的 Shell 模块的一种冗余的形式。）你可以象下面这样调用你的自动装载器（在 Unix 里）：

```
date();
who('am', 'i');
is('-l');
echo("Abadugabuadaredd...");
```

实际上，如果你预先按照这种方法声明你想要调用的函数，那么你就可以认为它们是内建的函数并且在调用的时候忽略圆括弧：

```
sub date (;$$);          # 允许零到两个参数。
sub who (;$$$$);         # 允许零到四个参数
sub ls;                  # 允许任意数量的参数
sub echo ($@);             # 允许至少一个参数

date;
who "am", "i";
ls "-l";
echo "That's all, folks!";
```

第十一章 模块

模块是 Perl 里重复使用的基本单元。在它的外皮下面，它只不过是定义在一个同名文件（以.pm 结尾）里面的包。本章里，我们将探究如何使用别人的模块以及创建你自己的模块。

Perl 是和一大堆模块捆绑在一起安装的，你可以在你用的 Perl 版本的 lib 目录里找到它们。那里面的许多模块将在第三十二章，标准模块，和第三十一章，用法模块里描述。所有标准模块都还有大量的在线文档，很可能比这本书更新。如果你的 man 命令里没有更丰富的东西，那么请试着使用 perldoc 命令。

综合 Perl 库网络（CPAN）是包含全世界的 Perl 社区所贡献的 Perl 模块的仓库，我们将在第二十二章，CPAN 里介绍它。同样也请参阅 http://www.cpan.org。

11.1 使用模块

模块有两种风格：传统的和面向对象的。传统模块为调用者的输入和使用定义了子过程和变量。面向对象的模块的运转类似类声明并且是通过方法调用来访问的，在第十二章，对象，里描述。有些模块有上面两种类型的东西。

Perl 模块通常用下面的语句包含入你的程序：

```
use MODULE LIST;
```

或者只是：

```
use MODULE;
```

MODULE 必须是一个命名模块的包和文件的标识符。（这里描述的语法只是建议性的；use 语句的详细描述在第二十九章，函数，里。）

use 语句在编译的时候对 MODULE 进行一次预装载，然后把你需要的符号输入进来，这样剩下的编译过程就可以使用这些符号了。如果你没提供你想要的符号的 LIST （列表），那么就使用在模块的内部 @EXPORT 数组里命名的符号——假设你在用 Exporter 模块，有关 Exporter 的内容在本章稍后的"模块私有和输出器"里介绍。（如果你没有提供

LIST，那么所有你的符号都必须在模块的 @EXPORT 或者 @EXPORT_OK 数组里提及，否则否则就会发生一个错误。）

因为模块使用 Exporter 把符号输入到当前包里，所以你可以不加包限制词地使用来自该模块的符号：

```
use Fred;       # 如果 Fred.pm 有 @EXPORT = qw(flintstone)

flintstone();       # ...这里调用 Fred::flintstone()。
```

所有 Perl 的模块文件都有 .pm 的扩展名。use 和 require 都做这种假定（和引起），因此你不用说 "MODULE.pm"。只使用描述符可以帮助我们把新模块和老版本的 Perl 中使用用的 .pl 和 .ph 库区别开。它还把 MODULE 当作一个正式模块名，这样可以在某些有歧义的场合帮助分析器。在模块名字中的任何双冒号都被解释成你的系统的目录分隔符，因此如果你的模块的名字是 Red::Blue::Green，Perl 就会把它看作 Red/Blue/Green.pm。

Perl 将在 @INC 数组里面列出的每一个目录里面查找模块。因为 use 在编译的时候装载模块，所以任何对 @INC 的修改都需要在编译时发生。你可以使用第三十一章里描述的 lib 用法或者一个 BEGIN 块来实现这个目的。一旦包含了一个模块，那么就会向 %INC 哈希表里增加一个键字/数值对。这里的键字将是模块的文件名（在我们的例子中是 Red/Blue/Green.pm）而数值将是全路径名。如果是在一个 windows 系统上合理安装的模块，这个路径可能是 C:/perl/site/lib/Red/Blue/Green.pm。

除非模块起用法的作用，否则它们的名字应该首字母大写。用法是有效的编译器指示器（给编译器的提示），因此我们把小写的用法名字留给将来使用。

当你 use 一个模块的时候，在模块里的所有代码都得到执行，就好象 require 里的通常情况一样。如果你不在乎模块是在编译的时候还是在运行的时候引入的，你可以只说：

```
require MODULE;
```

不过，通常我们更愿意用 use 而不是 require ，因为它在编译的时候就查找模块，因此你可以更早知道有没有问题。

下面的两个语句做几乎完全一样的事情：

```
require  MODULE;

require "MODULE.pm";
```

不过，它们在两个方面不太一样。在第一个语句里，require 把模块名字里的任何双冒号转换成你的系统的目录分隔符，就象 use 那样。第二种情况不做转换，强制你在文本上声明你的模块的路径名，这样移植性比较差。另外一个区别是第一个 require 告诉编译器说，带有关于 "MODULE" 的间接对象符号的表达式（比如 $ob = purgre MODULE）都是模块调用，而不是函数调用。（如果在你自己的模块里有冲突的 purge 定义，那么这里就有区别了。）

因为 use 声明和相关的 no 声明都隐含有一个 BEGIN 块，编译器就会一看到这个声明就装载这个模块（并且运行里面的任何可执行初始化代码），然后才编译剩下的文件。这就是用法如何改变编译器的性质的方法，以及为什么模块可以声明一些子过程，这些子过程可以作为列表操作符用于剩下的编译过程。如果你用 require 代替 use，这些事情就不会发生。使用 require 的唯一原因就是你有两个模块，这两个模块都需要来自对方的函数。（我们不知道这是不是个好理由。）

Perl 模块总是装载一个 .pm 文件，但是这个文件随后可以装载相关的文件，比如动态链接的 C 或 C++库或者自动装载的 Perl 子过程定义。如果是这样，那么附加的东西对模块用户而言是完全透明的。装载（或者安排自动）任何附加的函数或功能的责任在 .pm 文件。正巧是 POSIX 模块动态装载和自动装载两种方法都要用，不过用户可以只说：

```
use POSIX;
```

就可以获取所有输出了的函数和变量。

11.2 创建模块

我们前面说过，一个模块可以有两个方法把它的接口提供给你的程序使用：把符号输出或者允许方法调用。我们在这里先给你演示一个第一种风格的例子；第二种风格用于面向对象的模块，我们将在下一章里描述。（面向对象的模块应该不输出任何东西，因为方法最重要的概念就是 Perl 以该对象的类型为基础自动帮你找到方法自身。）

构造一个叫 Bestiary 的模块，创建一个看着象下面这样的叫 Bestiary.pm 的文件：

```
package   Bestiary;

require      Exporter;


our @ISA    =qw(Exporter);

our @EXPORT     =qw(camel);       # 缺省输出的符号
```

```
our @EXPORT_OK    =qw($weight);       # 按要求输出的符号
our $VERSION    = 1.00;           # 版本号

### 在这里包含你的变量和函数

sub    camel { print "One-hump dromedary" }

$weight = 1024;

1;
```

一个程序现在可以说 use Bestiary 就能访问 camel 函数（但是不能访问 $weight 变量），或者 use Bestiary qw(camel, $weight) 可以访问函数和变量。

你还可以创建动态装载 C 写的代码的模块。参阅第二十一章，内部和外部，获取细节。

11.2.1 模块私有和输出器

Perl 不会自动在它的模块的私有/公有边界上进行检查——和 C++，JAVA 和 Ada 这样的语言不同，Perl 不会被强加的私有性质搞糊涂。Perl 希望你呆在她的起居室外面是因为你没有收到邀请，而不是因为你拿着一把手枪。

Perl 模块和其用户之间有一种约定，有一部分是常见规则而另外一部分是单写的。常见规则部分约定是说禁止一个模块修改任何没有允许它修改的名字空间。为模块单写的约定（也就是文档）可以有其他约束。不过，如果你读过约定以后，我们就假设你知道自己说 use ReadfineTheWorld[2] 的时候就是在重定义世界，并且你愿意面对其后果。重定义世界的最常用的方法是使用 Exporter 模块。我们稍后在本章中就能看到，你甚至可以用这个模块重定义内建的东西。

当你 use 一个模块，通常是这个模块提供了你的程序可以使用的几个函数或者变量，或者更准确地说，为你的程序的当前包提供了函数和变量。这种从模块输出符号（并且把它们输入到你的程序里）的动作有时候被称为污染你的名字空间。大多数模块使用 Exporter 来做这些事情；这就是为什么在接近顶端的地方说这些东西：

```
require Exporter;
our @ISA = ("Exporter");
```

这两行令该模块从 Exporter 类中继承下来。我们在下一章讲继承，但在这里你要知道的所有东西就是我们的 Bestiary 模块现在可以用类似下面的行把符号输出到其他包里：

```
our @EXPORT     =qw($camel %wolf ram);       # 缺省输出
our @EXPORT     =qw(leopard @llama $emu);    # 请求时输出
our %EXPORT_TAGS = (
camelids => [qw($camel @llama)],
critters => [qw(ram $camel %wolf)],
);
```

从输出模块的角度出发，@EXPORT 数组包含缺省时要输出的变量和函数的名字：当你的程序说 use Bestary 的时候得到的东西。在 @EXPORT_OK 里的变量和函数只有当程序在 use 语句里面特别要求它们的时候才输出。最后， %EXPORT_TAGS 里的键字/数值对允许程序包含那些在 @EXPORT 和 @EXPORT_OK 里面列出的特定的符号组。

从输入包的角度出发，use 语句声明了一列可以输入的符号，一组在 %EXPORT_TAGS 里面的名字，一个符号的模式或者什么也没有，这时在 @EXPORT 里的符号将从模块里输入到你的程序里。

你可以包含任意的这些语句，从 Bestiary 模块里输入符号：

```
use Bestiary;          # 输入@EXPORT 符号
use Bestiary();          # 什么也不输入
use Bestiary qw(ram @llama);    # 输入 ram 函数 和@llama 数组
use Bestiary qw(:camelids);       # 输入$camel 和@llama
use Bestiary qw(:DEFAULT);       # 输入@EXPORT 符号
use Bestiary qw(/am/);             # 输入$camle，@llama，和 ram
use Bestiary qw(/^\$/);               # 输入所有标量
use Bestiary wq(:critters !ram);    # 输入 citters 但是把 ram 排除
```

```
use Bestiary wq(:critters !:camelids);

# 输入 critters，但是不包括 camelids
```

把一个符号排除在输出列表之外（或者用感叹号明确地从输入列表里删除）并不会让使用模块的程序无法访问它。该程序总是可以通过带完整修饰词的包名来访问模块的包的内容，比如 %Bestiary::gecko。（因为词法变量不属于包，所以私有属性仍然可实现：参阅下一章的"私有方法"。）

你可以说 BEGIN {$Exporter::Verbose=1 }，这样就可以看到声明是如何处理的，以及实际上有什么东西输入到你的包里。

Exporter 本身是一个 Perl 模块，如果你觉得奇怪，你可以看看类型团巧妙地使用它把符号从一个包输出的另一个包，在 Export 模块里，起关键作用的函数叫 import，它做一些必要的别名工作，把一个包里的符号体现在另外一个包里。实际上，一个 use Bestiary LIST 语句和下面的语句完全一样：

```
BEGIN {
    require Bestiary;
    import Bestiary LIST;
}
```

这意味着你的模块并不一定要使用 Exporter。当你使用一个模块的时候，它可以做任何它喜欢干的事情，因为 use 只是为那个模块调用普通的 import 方法，而你可以把这个方法定义为处理任何你想干的事情。

11.2.1.1 不用 Exporter 的输入方法进行输出

Export 定义一个叫 export_to_level 的方法，用于你（因为某些原因）不能直接调用 Exporter 的 import 方法的情况下。export_to_level 方法用下面的方法调用：

MODULE->export_to_level($where_to_export, @what_to_export);

这里的 $where_to_export 是一个整数，标识调用模块的堆栈把你的符号输出了多远，而 @what_to_export 是是一个数组，里面列出要输出的所有符号（通常是 @_）。

比如，假设我们的 Bestiary 有一个自己的 import 函数：

package Bestiary; @ISA = qw(Exporter); @EXPORT_OK = qw($zoo);

```
sub import { $Bestiary::zoo = "menagerie"; }
```

这个 import 函数的存在抑制了对 Exporter 的 import 函数的继承。如果你希望 Bestiary 的 import 函数在设置了 $Bestiary::zoo 之后的的性质和 Exporter 的 import 函数一样，那么你应该象下面那样定义它：

```
sub import { $Bestiary::zoo = "menagerie";
Bestiary->export_to_level(1,@_); }
```

这样就把符号符号从当前包输出到"上面"一层的包里。也就是说，输出到使用 Bestiary 的程序或者模块里。

11.2.1.2 版本检查

如果你的模块定义了一个 $VERSION 变量，使用你的模块的程序可以识别该模块足够新。比如：

```
use Bestiary 3.14;       # Bestiary 必须是版本 3.14 或者更新

use Bestiary v1.0.4;       # Bestiary 必须是版本 1.0.4 或者更新
```

这些东西都转换成对 Bestiary->require_version 的调用，然后你的模块就继承了它们。

11.2.1.3 管理未知符号

有时候，你可能希望避免某些符号的输出。通常这样的情况出现在你的模块里有一些函数或者约束对某些系统而言没有什么用的时候。你可以通过把它们放在 @EXPORT_FAIL 数组里面避免把这些符号输出。

如果一个程序想输入这些符号中的任何一个， Exporter 在生成一个错误之前给模块一个处理这种情况的机会。它通过带着一个失败符号列表调用 export_fail 的方法来实现这个目的，你可以这样定义 export_fail（假设你的模块使用 Carp 模块）：

```
sub export_fail {

        my $class = shift;

        carp "Sorry, these symblos are unavailable: @_";

        return @_;

}
```

Exporter 提供缺省的 export_fail 方法，它只是简单地不加改变地返回该列表并且令 use 失败，同时给每个符号产生一个例外。如果 export_fail 返回一个空列表，那么就不会记录任何错误并且输出所有请求的符号。

11.2.1.4 标签绑定工具函数

因为在 %EXPORT_TAGS 里列出的符号必须同时在 @EXPORT 或者 @EXPORT_OK 里面出现，所以 Exporter 提供了两个函数让你可以增加这些标签或者符号：

```
%EXPORTER_TAGS = (foo => [qw(aa bb cc)], bar => [qw(aa cc dd)]);


Exporter::export_tags('foo');           # 把 aa，bb 和 cc 加到@EXPORT
Exporter::export_ok_tags('bar');    # 把 aa，cc 和 dd 加到@EXPORT_OK
```

声明非标签名字是错误的。

11.3 覆盖内建的函数

许多内建的函数都可以覆盖，尽管（就象在你的墙里面打洞一样）你应该只是偶然才做这些事情并且只有必要时才这么做。通常，那些试图在一个非 Unix 系统上仿真一些 Unix 系统的功能的包要这种用法。（不要把覆盖和重载两个概念混淆了，重载给内建的操作符增加了面向对象的含义，但并不覆盖什么东西。参阅第十三章里的重载模块的讨论，重载，获取更多信息。）

我们可以通过从一个模块里输入名字来实现重载——预定义的不够好。更准确地说，触发覆盖的是对一个指向类型团的代码引用的赋值动作，就象 *open = \&myopen 里一样。另外，赋值必须出现在其他的包里；这样就不大可能通过故意的类型团别名导致偶然的覆盖。不过，如果你真的是希望做你自己的覆盖，那也别失望，因为 subs 用法令你通过输入语法预定义子过程，这样，这些名字就覆盖了内建的名字：

```
use subs qw(chdir chroot chmod chown);

chdir $somewhere;

sub chdir {...}
```

通常，模块不应该把 open 或 chdir 这样的内建的名字放在缺省的 @EXPORT 列表里输出，因为这些名字可能会不知不觉地跑到别人的名字空间里，并且在人们不知情的情况下把

语意改变了。如果模块包含的是在 @EXPORT_OK 列表里的名字，那么输入者就需要明确地请求那些要覆盖的内建的名字，这样才能保证每个人都是可信的。

内建的函数的最早的版本总是可以通过伪包 CORE 来访问。因此，CORE::chdir 将总是最初编译进 Perl 里的版本，即使 chdir 关键字已经被覆盖了。

不过，覆盖内建函数的机制总是被有意地限制在那些要求这样输入的包中。不过有一个更有覆盖性的机制可以让你在任何地方覆盖一个内建的函数，而不用考虑名字空间的限制。这是通过在 CORE:GLOBAL 伪包里定义该函数来实现的。下面是是用一个可以理解正则表达式的东西替换 glob 操作符的例子。（请注意，这个例子没有实现干净地覆盖 Perl 的内建 glob 的所有的东西，glob 在不同的标量或者列表环境里的行为是不一致的. 实际上，许多 Perl 内建都有这种环境敏感的行为，而一个写得好的覆盖应该充分支持这些行为。有关全功能的 glob 覆盖的例子，你可以学习和 Perl 绑定在一起的 File::Glob 模块。）总之，下面的是一个不全面的例子：

```
*CORE::GLOBAL::glob = sub {
        my $pat = shift;
        my @got;
        local *D;
        if (opendir D, '.') {
                $got = grep /$pat/, readdir D;
                closedir D;
        }
        return @got;
}

package Whatever;

print <^[a-z]+\.pm\$>;      # 显示当前目录里的所有用法
```

通过全局覆盖 glob，这样的抢占强制在任何名字空间里使用一个新的（并且是破坏性的）glob 操作符，而不需要拥有该名字空间模块的认可和协助。自然，这么做必须非常小心——如果必须这么做的话。但很可能不是必须的。

我们对覆盖的态度是：变得比较重要很好，但是更重要的是变得更好。

第十二章 对象（上）

12.1 简单复习一下面向对象的语言

对象是一个数据结构，带有一些行为。我们通常把这些行为称为对象的直接动作，有时候可以把这些对象拟人化。比如，我们可能会说一个长方形"知道"如何在屏幕上显示自己，或者说它"知道"如何计算它自己的区域。

作为一个类的实例，对象从中获益，取得其行为。类定义方法：就是那些应用于类和它的事例的性质。如果需要区分上面两种情况，那么我们就把适用于某一特定对象的方法叫做实例方法，而把那些适用于整个类的方法叫做类方法。不过这样做只是为了方便——对于 Perl 而言，方法就是方法，只是由其第一个参数的类型来区分。

你可以把实例方法看作一个由特定对象执行的某种动作，比如说打印自己啦，拷贝自己啦，或者是更改自己的一个或者多个属性（"把剑命名为 Anduril"）。类方法可以在许多共同的对象上执行操作（"显示所有的剑"）或者提供其他不依赖于任何特定对象的操作（"从现在开始，一旦新铸了一把剑，就在数据库里注册它的主人"）。类里面那些生成对象实例的方法叫构造方法（"铸一把镶满宝石并且带有秘密题字的剑"）。这些通常是类方法（"给我造把新剑"），但也有可能是实例方法（"造一把和我的剑一样的剑"）。

一个类可以从父类中继承方法，父类也叫基类或者超级类。如果类是这样生成的，那么它叫派生类或者子类。（让我们把稀泥和得更糊一些：有些文章里把"基类"定义为"最上层"的超级类。我们这里不是这个意思。）继承令新类的行为和现存的类很象，但是又允许它修改或者增加它的父类没有的性质。如果你调用了一个方法，而在当前的类中没有找到这个方法，Perl 会自动询问父表，以找出定义。比如，剑类可能从一个通用的刀锋类中继承 attack（攻击）方法。父类自己也可以有父类，而 Perl 会根据需要也在那些类中进行搜索。刀锋类可能自己又是从更通用的武器类中继承了它的 attack（攻击）方法。

当一个对象调用 attack（攻击）方法时，产生的效果可能取决于该对象是一把剑还是一支箭。可能这两者之间并没有什么区别——如果剑和箭都是从通用的武器类中继承他们的攻击（attack）性质的话。但是如果两者的行为不同，那么方法分选机制总是会选择最适合该对象需要的那个 attack 方法。总是为某一类型的特定对象选择最合适的行为的特性叫做多样性。它是"不在意"的一个重要形式。

当你在实现一个类的时候，你必须关注那些对象的"内脏"，但是当你使用一个类的时候，你应该把这个对象当作一个黑盒子。你不用知道内部有什么，你也不需要知道它是怎么工作的，而且你和这个黑盒子的交互只用它的方式进行：通过类提供的方法。即使你知道这些方法对对象做些什么处理，你也应该抑制住自己干的冲动。就好象遥控电视一样，即使即使你知道电视里面是怎样运转的，如果没有特别充分的理由，也不应该在电视里上窜下跳地摆弄它。

Perl 可以让你在需要的时候从类的外部观察对象的内部。但是这样做就打破了它的封装的原则——所有对对象的访问都应该只通过方法。封装断开了接口的公开（对象应该如何使用）和实现（它实际上是如何运转的）之间的联系。Perl 除了一个未写明的设计者和用户之间的契约之外没有任何其他明确的接口。双方都依靠常识和公德来运做：用户依靠归档的接口，而设计者不会随便打破该接口。

Perl 并不强制你使用某种编程风格，并且它也不会有一些其他的面向对象的语言里的私有性的困惑。不过，Perl 的确会在自由上令人迷惑，并且，作为一个 Perl 程序员，你所拥有的一个自由就是根据自己的喜好选择或多或少的私有性。实际上，Perl 可以在它的类里面有比 C++ 更强的私有性。也就是说，Perl 不限制你做任何事情，而且实际上它也不限制你自己约束自己——假如你必须这么做的话。本章稍后的"私有方法"和"用做对象的闭合"节演示了你如何才能增大自律的剂量。

不过我们承认，对象的内涵比我们这里说的多得多，而且有很多方法找出面向对象的设计的更多内容。但是那些不是我们这里的目标。所以，我们接着走。

12.2 Perl 的对象系统

Perl 没有为定义对象，类或者方法提供任何特殊的语法。相反，它使用现有的构造来实现这三种概念。（注：现在有了一个软件重用的例子了！）

下面是一些简单的定义，可以让你安心些：

一个对象只不过是一个引用...恩，就是引用。

因为引用令独立的标量代表大量的数据，所以我们不应该对把引用用于所有对象 感到奇怪。从技术上来讲，对象不太适合用引用表示，实际上引用指向的是 引用物。不过这个区别很快就被 Perl 的程序员淡化了，而且因为我们觉得这是 一个很好的转喻，如果这么用合适

的话，我们将永远用下去（注：我们更喜欢语言 的活力，而不是数学的严密。不管你同意与否。） （译注：Larry Wall 是学语言的，不是学数学的，当然这么说．:)）

一个类只是一个包

一个包当作一个类——通过使用包的子过程来执行类的方法，以及通过使用包的 变量来保存类的全局数据。通常，使用一个模块来保存一个或者更多个类。

一个方法只是一个子过程

你只需要在你当作类来使用的包中声明子过程；它们就会被当作该类的方法来 使用。方法调用是调用子过程的一种新的方法，它传递额外的参数：用于调用方法 所使用的对象或包。

12.3 方法调用

如果你把面向对象的编程方法凝缩成一个精华的词，那就是抽象。你会发现这个词是所有那些 OO 的鼓吹者所传播的言辞背后的真正主题，那些言辞包括多型性啦，继承啦还有封装啦。我们相信这些有趣的字眼，但是我们还是会从实际的角度出发来理解它们——它们在方法调用中的作用是什么。方法是对象系统的核心，因为它们为实现所有魔术提供了抽象层。你不是直接访问对象里的一块数据区，而是调用一个实例方法，你不是直接调用某个包里面的子过程，而是调用一个类方法。通过在类的使用和实现之间放置这个间接层，程序设计人员仍然可以自由修改复杂的类的内部机制，因而要冒一点儿破坏使用它的程序的风险。

Perl 支持调用方法的两种不同的语意。一种是你已经在 Perl 别的地方看惯了的风格，而第二种是你可以在其他编程语言中看到的。不管使用哪种方法调用的形式，Perl 总是会给构成这个方法的子过程传递一个额外的初始化参数。如果用一个类调用该方法，那个参数将会是类的名字。如果用一个对象调用方法，那个参数就是对象的引用。不管是什么，我们都叫它方法调用者。对于类方法而言，调用者是包的名字。对于一个实例方法，调用者是调用者是一个声明对象的引用。

换句话说，调用者就是调用方法的那个东西。有些 OO 文章把它叫做代理或演员。从文法上看，调用者既不是动作的对象也不是动作的承担者。它更象一个间接的对象，是代表动作执行后受益人的东西，就向在命令"给我铸把剑！"里的"我"一样。从语意上来看，你既可以把调用者看作动作的施动者，也可以把它看作动作的受动者——更象哪个取决于你的智力取向。我们可不打算告诉你怎样看待它们。

大多数方法是明确调用的，但是也可以隐含调用——由对象析构器，重载的操作符或者捆绑的变量触发的时候。准确地说，这些都不是正常的子过程调用，而是代表对象的 Perl 自动触发的方法调用。析构器在本章后面描述，重载在第十三章，重载，描述；而捆绑在第十四章，捆绑变量。

方法和普通子过程之间的一个区别是，它们的包在什么时候被解析——也就是说，Perl 什么时候决定应该执行该方法或者子过程的哪些代码。子过程的包是在你的程序开始运行之前，在编译的时候解析的。（注：更准确地说，子过程调用解析成一个特定的类型团，它是一个填充到编译好的操作码树上的引用。这个类型团的含义甚至在运行时也是可以协商的——这就是为什么 AUTOLOAD 可以为你自动装载一个子过程。不过，类型团的含义通常在编译的时候也被解析——由一个命名恰当的子过程定义解析。）相比之下，一个方法包直到实际调用的时候才解析。（编译的时候检查原型，这也是为什么编译时可以使用普通子过程而却不能使用方法的原因。）

方法包不能早些解析的原因是相当简单的：包是由调用的类决定的，而在方法实际被调用之前，调用者是谁并不清楚。OO 的核心是下面这样简单的逻辑：一旦得知调用者，则可以知道调用者的类，而一旦知道了类，就知道了类的继承关系，一旦知道了类的继承关系，那么就知道实际调用的子过程了。

抽象的逻辑是要花代价的。因为方法比较迟才解析，所以 Perl 里面向对象的解决方法要比相应的非 OO 解决方法慢。而对我们稍后要介绍的几种更加神奇的技巧而言，它可能慢很多。不过，解决许多常见的问题的原因并不是做得快，而是做得聪明。那就是 OO 的闪光点。

12.3.1 使用箭头操作符的方法调用

我们说过有两种风格的方法调用。第一种调用方法的风格看起来象下面这样：

```
INVOCANT->METHOD(LIST)

INVOCANT->METHOD
```

这种方法通常被称做箭头调用（原因显而易见）。（请不要把->和=>混淆，“双管”箭头起神奇逗号的作用。）如果有任何参数，那么就需要使用圆括弧，而当 INVOCANT 是一个包的名字的时候，我们把那个被调用的 METHOD 看作类方法。实际上两者之间并没有区别，只不过和类的对象相比，包名字与类本身有着更明显的关联。还有一条你必须记住：就是对象同样也知道它们的类。我们会告诉你一些如何把对象和类名字关联起来的信息，但是你可以在不知道这些信息的情况下使用对象。

比如，使用类方法 summon 的构造一个类，然后在生成的对象上调用实例方法 speak，你可以这么说：

```
$mage = Wizard->summon("Gandalf");    # 类方法

$mage->speak("friend");                          # 实例方法
```

summon 和 speak 方法都是由 Wizard 类定义的——或者是从一个它继承来的类定义的。不过你用不着担心这个。用不着管 Wizard 的闲事。

因为箭头操作符是左关联的（参阅第三章，单目和双目操作符），你甚至可以把这两个语句合并成一条：

```
Wizard->summon("Gandalf")->speak("friend");
```

有时候你想调用一个方法而不想先知道它的名字。你可以使用方法调用的箭头形式，并且把方法名用一个简单的标量变量代替：

```
$method = "summon";

$mage = Wizard->$method("Gandalf");    # 调用 Wizard->summon


$travel = $companion eq "Shadowfax" ? "ride" : "walk";

$mage->$travel("seven leagues");    # 调用 $mage->ride 或者 $mage->walk
```

虽然你间接地使用方法名调用了方法，这个用法并不会被 use strict 'refs' 禁止，因为所有方法调用实际上都是在它们被解析的时候以符号查找的形式进行的。

在我们的例子里，我们把一个子过程的名字存储在 $travel 里，不过你也可以存储一个子过程引用。但这样就忽略了方法查找算法，不过有时候你就是想这样处理。参阅"私有方法"节和在"UNIVERSAL：最终的祖先类"节里面的 can 方法的讨论。要创建一个指向某方法在特定实例上的调用的引用，参阅第八章的"闭合"节。

12.3.2 使用间接对象的方法调用

第二种风格的方法调用看起来象这样：

```
METHOD INVOCANT (LIST)

METHOD INVOCANT LIST

METHOD INVOCANT
```

LIST 周围的圆括弧是可选的；如果忽略了圆括弧，就把方法当作一个列表操作符。因此你可以有下面这样的语句，它们用的都是这种风格的方法调用：

```
$mage = summon Wizard "gandalf";
```

```
$nemesis = summon Balrog home => "Moria", weapon => "whip";

move $namesis "bridge";

speak $mage "You cannot pass";

break $staff;          # 更安全的用法： break $staff();
```

你应该很熟悉列表操作符的语法；它是用于给 print 或者 printf 传递文件句柄的相同的风格：

```
print STDERR "help!!!\n";
```

它还和 "Give Gollum the Preciousss" 这样的英语句子类似，所以我们称他为间接对象形式。Perl 认为调用者位于间接对象槽位中。当你看到传递一个内建的函数，象 system system 或 exec 什么的到它的"间接对象槽位中"时，你的实际意思是在同一个位置提供这个额外的，没有逗号的参数（列表），这个位置和你用间接对象语法调用方法时的位置一样。

间接对象形式甚至允许你把 INVOCANT 声明为一个 BLOCK，该块计算出一个对象（引用）或者类（包）。这样你就可以用下面的方法把那两种调用组合成一条语句：

```
speak {summon Wizard "Gandalf" } "friend";
```

12.3.3 间接对象的句法障碍

一种语法总是比另外一种更易读。间接对象语法比较少混乱，但是容易导致几种语法含糊的情况。首先就是间接对象调用的 LIST 部分和其他列表操作符一样分析。因此，下面的圆括弧：

```
enchant $sword ($pips + 2) * $cost;
```

是假设括在所有参数的周围的，而不管先看到的是什么。那么，它就等效于下面这样的：

```
($sword->enchant($pips + 2)) * $cost;
```

这样可不象你想要的：调用 enchant 时只给了 $pips + 2，然后方法返回的值被 $cost 乘。和其他列表操作符一样，你还必须仔细对待 && 和 || 与 and 和 or 之间的优先级。

比如：

```
name $sword $oldname || "Glamdring";    # 在这不能用"or"
```

变成：

```
$sword->name($oldname || "Glamdring");
```

而：

```
speak $mage "friend" && enter();   # 这儿应该用"and"
```

变成奇怪的：

```
$mage->speak("friend" && enter());
```

这些可以通过把它们写成下面的等效形式消除错误：

```
enter() if $mage->speak("friend");

$mage->speak("friend") && enter();

speak $mage "friend" and enter();
```

第二种语法不适用于间接对象形式，因为它的 INVOCANT 局限于一个名字，一个未代换的标量值或者一个块。（注：仔细的读者应该还记得，这些语法项是和允许出现在趣味字符后面的列表是一样的，那些语法项标识一个变量的解引用——比如 @ary，@$aryref，或者 {$aryref}。当分析器看到这些内容之一时，她就有自己的 INVOCANT 了，因此她开始查找她的 LIST。所以下面这些调用：

```
move $party->{LEADER};      # 可能错了！

move $riders[$i];              # 可能错了！
```

实际分析成这样：

```
$party->move->{LEADER};

$riders->move([i]);
```

但是你想要的可能是：

```
$party->{LEADER}->move;

$riders[$i]->move;
```

分析器只是为一个间接对象查找一个调用时稍稍向前看了一点点，甚至看的深度都不如为单目操作符那样深远。如果你使用第一种表示法是就不会发生这件怪事，因此你可能会选择箭头作为你的"武器"。

甚至英语在这方面也有类似的问题。看看下面的句子：“Throw your cat out the window a toy mouse to play with.”如果你分析这句话速度太快，你最后就会把猫仍出去，而不是耗子（除非你意识到猫已经在窗户外边了）。类似 Perl，英语也有两种不同的方法来表达这个代理：“Throw your cat the mouse”和“ Throw the mouse to your cat.”有时候长一点的形式比较清晰并且更自然，但是有时侯短的好。至少在 Perl 里，我们要求你在任何编译为间接对象的周围放上花括弧。

12.3.4 引用包的类

间接对象风格的方法调用最后还有一种可能的混淆，那就是它可能完全不会被当作一个方法调用而分析，因为当前包可能有一个和方法同名的子过程。当用一个类方法和一个文本包名字一起做调用者用的时候，有一个方法可以解析这样的混淆，而同时仍然保持间接对象的语法：通过在包后面附加两个冒号引用类名。

```
$obj = method CLASS::;       # 强制为 "CLASS"->method
```

这个方法很重要，因为经常看到下面的表示法：

```
$obj = new CLASS;       # 不会分析为方法
```

如果当前包有一个子过程叫 new 或者 CLASS 时，将不能保证总是表现得正确。即使你很仔细地使用箭头形式而不是间接对象形式调用方法，也有极小可能会有问题。虽然引入了额外标点的杂音，但 CLASS:: 表示法却能保证 Perl 正确分析你的方法调用。下面例子中前面两个不总是分析成一样的东西，但后面两个可以：

```
$obj = new ElvenRing;       # 可以是 new("ElvenRing")
                                # 甚至是 new(ElvenRing())
$obj = ElvenRing->new;          # 可以是 ElvenRing()->new()


$obj = new ElvenRing::;       # 总是 "ElvenRing"->new()
$obj = ElvenRing::->new;    #  总是 "ElvenRing"->new()
```

包引用表示法可以用一些富有创造性的对齐写得更好看：

```
$obj = new ElvenRing::
       name  => "Narya",
```

```
    owner => "Gandalf",
    domain => "fire",
    stone => "ruby";
```

当然，当你看到双冒号的时候可能还是会说，“真难看！”，所以我们还要告诉你，你几乎总是可以只使用光光的类名字，只要两件事为真。首先，没有和类同名的子过程名。（如果你遵循命名传统：过程名，比如 new 以小写开头，而类名字，比如 ElvenRing[2] 以大写开头，那么就永远不会有这个问题。）第二，类是用下面的语句之一装载的：

```
use ElvenRing;
require ElvenRing;
```

这两种方法都令 Perl 意识到 ElvenRing[2] 是一个模块名字，它强制任何在类名 ElvenRing[2] 前面的光板名字，比如 new，解释为一个方法调用，即使你碰巧在你的当前包里定义了一个自己的 new 子过程，也不会错误解释成子过程。我们通常不会在使用间接对象中碰到问题，除非你在一个文件里填满多个类，这个时候，Perl 就可能不知道一个特定的包名字就是一个类名字。而且那些把子过程的名字命名为类似 ModuleNames[2] 这样的人最终也会陷入痛苦。

12.4 构造对象

所有对象都是引用，但不是所有引用都是对象。一个引用不会作为对象运转，除非引用它的东西有特殊标记告诉 Perl 它属于哪个包。把一个引用和一个包名字标记起来（因此也和包中的类标记起来了，因为一个类就是一个包）的动作被称作赐福（blessing），你可以把赐福（bless）看作把一个引用转换成一个对象，尽管更准确地说是它把该引用转换成一个对象引用。

bless 函数接收一个或者两个参数。第一个参数是一个引用，而第二个是要把引用赐福（bless）成的包。如果忽略第二个参数，则使用当前包。

```
$obj = { };              # 把引用放到一个匿名散列
bless($obj);          # Bless 散列到当前包
bless($obj, "Critter");     # Bless 散列到类 Critter。
```

这里我们使用了一个指向匿名散列的引用，也是人们通常拿来做他们的对象的数据结构的东西。毕竟，散列极为灵活。不过请允许我们提醒你的是，你可以赐福（bless）一个引用为任何你在 Perl 里可以用作引用的东西，包括标量，数组，子过程和类型团。你甚至可以把

一个引用赐福（bless）成一个包的符号表散列——只要你有充分的理由。（甚至没理由都行。）Perl 里的面向对象的特性与数据结构完全不同。

一旦赐福（bless）了指示物，对它的引用调用内建的 ref 函数会返回赐福了的类名字，而不是内建的类型，比如 HASH。如果你需要内建的类型，使用来自 attributes 模块的 reftype。参阅第三十一章，实用模块，里的 use attributes。

这就是如何制作对象。只需要使用某事的引用，通过把他赐福（bless）到一个包里给他赋一个类，仅此而已。如果你在设计一个最小的类，所有要做的事情就是这个。如果你在使用一个类，你要做的甚至更少，因为类的作者会把 bless 隐藏在一个叫构造器的方法里，它创建和返回类的实例。因为 bless 返回其第一个参数，一个典型的构造器可以就是：

```
package Critter;

sub spawn { bless {}; }
```

或者略微更明确地拼写：

```
package Critter;

sub spawn {
        my $self = {};       # 指向一个空的匿名散列

        bless $self, "Critter";   # 把那个散列作成一个 Critter 对象

        return $self;       # 返回新生成的 Critter
}
```

有了那个定义，下面就是我们如何创建一个 Critter 对象了：

```
$pet = Critter->spawn;
```

12.4.1 可继承构造器

和所有方法一样，构造器只是一个子过程，但是我们不把它看作一个子过程。在这个例子里，我们总是把它当作一个方法来调用——一个类方法，因为调用者是一个包名字。方法调用和普通的子过程调用有两个区别。首先，它们获取我们前面讨论过的额外的参数。其次，他们遵守继承的规则，允许一个类使用另外一个类的方法。

我们将在下一章更严格地描述继承下层的机制，而现在，通过几个简单的例子，你就应该可以理解他们的效果，因此可以帮助你设计构造器。比如，假设我们有一个 Sppider 类从

Spider 类继承了方法。特别是，假设 Spider 类没有自己的 spawn 方法。则有下面对应的现象：

方法调用	结果子过程调用
Critter->spawn()	Citter::spawn("Critter")
Spider->spawn()	Critter::spawn("Spider")

两种情况里调用的子过程都是一样的，但是参数不一样。请注意我们上面的 spawn 构造器完全忽略了它的参数，这就意味着我们的 Spider 对象被错误地赐福（bless）成了 Critter 类。一个更好的构造器将提供包名字（以第一个参数传递进来）给 bless：

```
sub spawn {
        my $class = shift;      # 存储包名字
        my $self = { };
        bless( $self, $class);      # 把赐福该包为引用
        return $self;
}
```

现在你可以为两种情况都使用同一个子过程：

```
$vermin = Critter->spawn;
$shelob = Spider->spawn;
```

并且每个对象都将是正确的类。甚至是间接运转的，就象：

```
$type = "Spider";
$shelob = $type->spawn;      # 和 "Spider"->spawn 一样
```

这些仍然是类方法，不是实例方法，因为它的调用者持有的是字串而不是一个引用。

如果 $type 是一个对象而不是一个类名字，前一个构造器的定义将不会运行，因为 bless 需要一个类名字。但是对许多类而言，只有拿一个现有的对象当模板去创建另外一个对象的时候它才有意义。在这些情况下，你可以设计你的构造器，这样他们就可以与对象或者类名字一起运转了：

```
sub spawn {
        my $invocant = shift;
```

```
        my $class = ref($invocant) || $invocant;   # 对象或者类名字
        my  $self = { };
        bless ($self, $class);
        return $self;
}
```

12.4.2 初始器

大多数对象维护的信息是由对象的方法间接操作的。到目前为止我们的所有构造器都创建了空散列，但是我们没有理由让它们这么空着。比如，我们可以让构造器接受额外的参数，并且把它们当作键字/数值对。有关 OO 的文章常把这样的数据称为"所有"，"属性"，"访问者"，"成员数据"，"实例数据"或者"实例变量"等。本章稍后的"实例变量"节详细地讨论这些属性。

假设一个 Horse 类有一些实例属性，比如 "name" 和 "color"：

```
   $steed = Horse->new(name => "shadowfax", color => "white");
```

如果该对象是用散列引用实现的，那么一旦调用者被从参数列表里删除，那么键字/数值对就可以直接代换进散列：

```
sub new {
        my $invocant = shift;
        my $class = ref($invocant) || $invocant;
        my $self = { @_ };          # 剩下的参数变成属性
        bless($self, $class);          # 给予对象性质
        return $self;
}
```

这回我们用一个名字叫 new 的方法做该类的构造器，这样就可以把那些 C++ 程序员哄得相信这些都是正常的。不过 Perl 可不认为"new"有任何特殊的地方；你可以把你的构造器命名为任意的东西。任何碰巧创建和返回一个对象的方法都是实际上的构造器。通常，我们建议你把你的构造器命名为任何在你解决的问题的环境中有意义的东西。比如，在 Tk 模块中的构造器命名为它们创建的窗口构件。在 DBI 模块里，一个叫 connect 的构造器返

回一个数据库句柄对象，而另外一个叫 prepare 的构造器是当作一个实例方法调用的，并且返回一个语句句柄对象。不过如果没有很好的适合环境的构造器名字，那么 new 也不算是一个太坏的选择。而且，随便挑一个名字，这样强制人们在使用构造器之前去读接口文档（也就是类的文档）也不是太坏的事情。

更灵活一些，你可以用缺省键字/数值对设置你的构造器，这些参数可以由用户在使用的时候通过提供参数而覆盖掉：

```
sub new {
        my $invocant = shift;
        my $class = ref($invocant) || $invocant;
        my $self = {
                color => "bay",
                legs => 4,
                owner => undef,
                @_          # 覆盖以前的属性
        };

        return bless $self, $class;
}

$ed     = Horse->new;                       # 四腿湾马
$stallion = Horse->new(color => "black"); # 四腿黑马
```

当把这个 Horse 构造器当作实例方法使用的时候，它忽略它的调用者现有的属性。你可以设计第二个构造器，把它当作实例方法来调用，如果你设计得合理，那你就可以使用来自调用对象的数值作为新生成的对象的缺省值：

```
$steed = Horse->new(color => "dun");
$foal = $steed->clone(owner => "EquuGen Guild, Ltd.");
```

```
sub clone {
        my $model = shift;
        my $self = $model->new(%$model, @_);
        return $self;          # 前面被 ->new 赐福过了
}
```

（你可以把这个功能直接放进 new 里，但是这样的话名字就不是那么适合这个函数了。）

请注意我们即使是在 clone 构造器里，我们也没有硬编码 Horse 类的名字。我们让最初的那个对象调用它自己的 new 方法，不管是什么。如果我们把它写成 Horse->new 而不是 $model->new，那么该类不能帮助实现 Zebra（斑马） 或 Unicorn （独角兽）类。你应该不会想克隆一匹飞马但是却突然发现你得到是一匹颜色不同的马。

不过，有时候你碰到的是相反的问题：你不是想在不同的类里共享一个构造器，而是想多个构造器共享一个类对象。当一个构造器想调用一个基类的构造器作为构造工作的一部分的时候就会出现这种问题。Perl 不会帮你做继承构造。也就是说，Perl 不会为任何基类或者任何其他所需要的类自动调用构造器（或者析构器），所以你的构造器将不得不自己做这些事情然后增加衍生的类所需要的附加的任何属性。因此情况不象 clone 过程那样，你不能把一个现有的对象拷贝到新对象里，而是先调用你的基类的构造器，然后把新的基类对象变形为新的衍生对象。

12.5 类继承

对 Perl 的对象系统剩下的内容而言，从一个类继承另外一个类并不需要给这门语言增加特殊的语法。当你调用一个方法的时候，如果 Perl 在调用者的包里找不到这个子过程，那么它就检查 @ISA 数组（注：发音为 "is a"，象 "A horse is a critter。"里哪样）。 Perl 是这样实现继承的：一个包的 @ISA 数组里的每个元素都保存另外一个包的名字，当缺失方法的时候就搜索这些包。比如，下面的代码把 Horse 类变成 Critter 类的字类。（我们用 our 声明 @ISA，因为它必须是一个打包的变量，而不是用 my 声明的词。）

```
package Horse;
our @ISA = "Critter";
```

你现在应该可以在原先 Critter 使用的任何地方使用 Horse 类或者对象了。如果你的新类通过了这样的空字类测试，那么你就可以认为 Critter 是一个正确的基类，可以用于继承。

假设你在 $steed 里有一个 Horse 对象，并且在他上面调用了一个 move：

```
$steed->move(10);
```

因为 $steed 是一个 Horse，Perl 对该方法的第一个选择是 Horse::move 子过程。如果没有，Perl 先询问 @Horse::ISA 的第一个元素，而不是生成一个运行时错误，这样将导致查询到 Critter 包里，并找到 Critter::move。如果也没有找到这个子过程，而且 Critter 有自己的 @Critter::ISA 数组，那么继续查询那里面的父类，看看有没有一个 move 方法，如此类推直到上升到继承级别里面一个没有 @ISA 的包。

我们刚刚描述的情况是单继承的情况，这时每个类只有一个父类。这样的继承类似一个相关包的链表。Perl 还支持多继承；只不过是向该类的 @ISA 里增加更多的包。这种继承的运做更象一个树状结构，因为每个包可以有多于一个的直接父类。很多人认为这样更带劲。

当你调用了调用者的一个类型为 classname 的方法 methname，Perl 将尝试六种不同的方法来找出所用的子过程（译注：又是孔乙己？：）：

1. 首先，Perl 在调用者自己的包里查找一个叫 classname::methname 的子过程。如果失败，则进入继承，并且进入步骤 2。
2. 第二步，Perl 通过检查 @classname::ISA 里列出的所有父包，检查从基类继承过来的方法，看看有没有 parent::methname 子过程。这种搜索是从左向右，递归的，由浅入深进行的。递归保证祖父类，曾祖父类，太祖父类等等类都进入搜索。
3. 如果仍然失败，Perl 就搜索一个叫 UNIVERSAL::methname 的子过程。
4. 这时，Perl 放弃 methname 然后开始查找 AUTOLOAD。首先，它检查叫做 classmane::AUTOLOAD 的子过程。
5. 如果上面的失败，Perl 则搜索所有在 @classname::ISA 里列出的 parent 包，寻找任何 parent::AUTOLOAD 子过程。这样的搜索仍然是从左向右，递归的，由浅入深进行的。
6. 最后，Perl 寻找一个叫 UNIVERSAL::AUTOLOAD 的子过程。

Perl 会在找到的第一个子过程处停止并调用该子过程。如果没有找到子过程，则产生一个例外，也是你经常看到的：

```
Can't locate object method "methname" via package "classnaem"
```

如果你给你的 C 编译器提供了 -DDEBUGGING 选项，做了一个调试版本的 Perl ，那么如果你给 Perl 一个 -Do 开关，你就能看到它一边解析方法调用一边走过这些步骤。

我们将随着我们的继续介绍更详细地讨论继承机制。

第十二章 对象（下）

12.5.1 通过 @ISA 继承

如果 @ISA 包含多于一个包的名字，包的搜索都是从左向右的顺序进行的。这些搜索是由浅入深的，因此，如果你有一个 Mule 类有象下面这样的继承关系：

```
package Mule;

our @ISA= ("Horse", "Donkey");
```

Perl 将首先在 Horse 里（和他的任何前辈类里，比如 Critter）查找任何在 Mule 里找不到的方法，找不到以后才继续在 Donkey 和其父类里进行查找。

如果缺失的方法在一个基类里发现，Perl 内部把该位置缓存在当前类里，依次提高效率，这样要查找该方法的时候，它不用再跑老远。对 @ISA 的修改或者定义新的方法就会令该缓存失效，因此导致 Perl 再次执行查找。

当 Perl 搜索一个方法的时候，它确信你没有创建一个闭环的继承级别。如果两个类互相继承则可能出现这个问题，甚至是间接地通过其他类这样继承也如此。试图做你自己的祖父即使对 Perl 而言也是荒谬的，因此这样的企图导致抛出一个例外。不过，如果从多于一个类继承下来，而且这些类共享同样的祖宗，Perl 不认为是错误，这种情况类似近亲结婚。你的继承级别看起来不再象一棵树，而是象一个油脂分子图。不过这样不会为难 Perl—— 只要这个图形是真的油脂分子。

当你设置 @ISA 的时候，赋值通常发生在运行时，因此除非你加以预防，否则在 BEGIN，CHECK，或者 INIT 块里的代码不能在继承级别里使用。预防之一是 use base 用法，它令你 require 类并且在编译时把它加入 @ISA 里。下面是你使用它们的方法：

```
packge Mule;
```

```
use base ("Horse", "donkey");      # 声明一个超类
```

它是下面东西的缩写：

```
package Mule;

BEGIN {

        our @ISA = ("Horse", "Donkey");

        require Horse;

        require Donkey;

}
```

只是 use base 还算入所有 use fields 声明中。

有时候人们很奇怪的是，在 @ISA 中包含一个类并没有为你 require （请求）合适的模块。那是因为 Perl 的类系统很大程度上是与它的模块相冲突的。一个文件可以保存许多类（因为他们只是包），而一个包可以在许多文件中提及。但是在最常见的情况下，一个包、一个类、一个模块、一个文件这样总是相当具有可换性——只要你足够倾斜，use base 用法提供了一个声明性的语法，用这个语法可以建立继承，装载模块文件和并且提供任意声明了的基类域。它也是我们不停提到的最便利的对角线。参阅第三十一章的 use base 和 use fields 获取细节。

12.5.2 访问被覆盖的方法

当一个类定义一个方法，那么该子过程覆盖任意基类中同名的方法。想象一下你有一个 Mule（骡）对象（它是从 Horse（马）类和 Donkey（驴）类衍生而来的），而且你想调用你的对象的 breed （种）方法。尽管父类有它们自己的 breed（种） 方法，Mule （骡）类的设计者通过给 Mule（骡）类写自己的 breed 方法覆盖了那些父类的 breed 方法。这这意味着下面的交叉引用可能不能正确工作：

```
$stallion  = Horse->new(gender => "male");

$molly = Mule->new(gender => "female");

$colt = $molly->breed($stallion);
```

现在假设基因工程的魔术治好了骡子臭名昭著的不育症，因此你想忽略无法生活的 Mule::breed 方法。你可以象平常那样调用你的子过程，但是一定要确保明确传递调用者：

```
$colt = Horse::breed($molly, $stallion);
```

不过，这是回避继承关系，实际上总是错误的处理方法。我们很容易想象实际上没有这么个 Horse::breed 子过程，因为 Horse 和 Donkeys 都是从一个公共的叫 Equine （马）的父类继承来的那个秉性。从另一方面来讲，如果你希望表明 Perl 应该从一个特定类开始搜索某方法，那么只要使用普通的方法调用方式，只不过用方法名修饰类名字就可以了：

```
$colt = $molly->Horse::breed($stallion);
```

有时候，你希望一个衍生类的方法表现得象基类中的某些方法的封装器。实际上衍生类的方法本身就可以调用基类的方法，在调用前或者调用后增加自己的动作。你可以把这种表示法只用于演示声明从哪个类开始搜索。但是在使用被覆盖方法的大多数情况下，你并不希望自己要知道或者声明该执行哪个父类被覆盖了的方法。

这就是 SUPER 伪类提供便利的地方。它令你能够调用一个覆盖了的基类方法，而不用声明是哪个类定义了该方法。（注：不要把这个和第十一章的覆盖 Perl 的内建函数的机制混淆了，那个不是对象方法并且不会被继承覆盖。你调用内建函数的覆盖是通过 CORE 伪包，而不是 SUPER 伪包。）下面的子过程在当前包的 @ISA 里查找，而不会要求你声明特定类：

```
package Mule;

our @ISA = qw(Horse Donkey);

sub kick {

        my $self = shift;

        print "The mule kicks!\n";

        $self->SUPER::kick(@_);

}
```

SUPER 伪包只有在一个方法里使用才有意义。尽管一个类的实现器可以在它们自己的代码里面使用 SUPER，但那些使用一个类的对象的却不能。

当出现多重继承的时候，SUPER 不总是按照你想象的那样运行。你大概也可以猜到，它也象 @ISA 那样遵循普通的继承机制的规则：从左向右，递归，由浅入深。如果 Horse 和 Donkey 都有一个 speak 方法，而你需要使用 Donkey 的方法，你将不得不明确命名该父类：

```
sub speak {
```

```
        my $self = shift;
        print "The mule speaks!\n";
        $self->Donkey::speak(@_);
}
```

用于多重继承的情况的更灵活的方法可以用 UNIVERSAL::can 方法进行雕琢，该方法下节介绍。或者你可以从 CPAN 抓 Class::Multimethods 方法下来，它提供了更灵活的解决方法，包括搜索最接近的，而不是最左端的。

Perl 里的每一段代码都知道自己现在在哪个包里，就象最后的 package 语句说的那样。只有在调用 SUPER 的包编译过以后，SUPER 才询问@ISA。它不关心调用者的类，也不关心调用的子过程所属的包。不过，如果你想在另外一个类中定义方法，而且只是修改方法名，那么就有可能出问题：

```
package Bird;
use Dragonfly;
sub Dragonfly::divebomb { shift->SUPER::divebomb(@_) }
```

不幸的是，这样会调用 Bird 的超类，而不是 Dragonfly 的。要想按照你的意愿做事，你还得为 SUPER 的编译明确地切换到合适的包：

```
package Bird;
use Dragonfly;
{
        package Dragonfly;
        sub divebomb { shift->SUPER::divebomb(@_) }
}
```

如上例所示，你用不着只是为了给某个现有类增加一个方法去编辑一个模块。因为类就是一个包，而方法就是一个子过程，你所要做的就是在那个包里定义一个函数，就象我们上面做的那样，然后该类就一下子有了一个新方法。没有要求继承。只需要考虑包，而因为包是全局的，程序的任意位置都可以访问任意包。（小意思！湿湿碎！）

12.5.3 UNIVERSAL：最终的祖先类

如果对调用者的类和所有他的祖先类递归搜索后，还没有发现有正确名字的方法定义，那么会在一个预定义的叫 UNIVERSAL 的类中最后再搜索该方法一次。这个包从来不会在 @ISA 中出现，但如果查找 @ISA 失败总是要查找它。你可以把 UNIVERSAL 看作最终的祖先，所有类都隐含地从它衍生而来。

在 UNIVERSAL 类里面有下面的预定义的方法可以使用，因此所有类中都可以用它们。而且不管它们是被当作类方法还是对象方法调用的都能运行。

INVOCANT->isa(CLASS)

如果 INVOCANT 的类是 CLASS 或者任何从 CLASS 继承来的，isa 方法返回真。 除了包名字以外，CLASS 还可以是一个内建的类型，比如 "HASH" 或者 "ARRAY"。 （准确地检查某种类型在封装和多态性机制中并不能很好地工作。你应该依赖重 载分检给你正确的方法。）

```
use FileHandle;

if (FileHandle->isa("Exporter")) {

        print "FileHandle is an Exporter.\n";

}


$fh = FileHandle->new();

if ($fh->isa("IO::Handle")) {

        print "\$fh is some sort of IOish object.\n"

}

if ($fh->isa("GLOB")) {

        print "\$fh is really a GLOB reference.\n";

}
```

INVOCANT->can(METHOD)

如果 INVOCANT 中有 METHOD，那么 can 方法就返回一个可以调用的该子过程的 引用。如果没有定义这样的子过程，can 返回 undef。

```
    if ($invocant->can("copy")) {
        print "Our invocant can copy.\n";
    }
```

我们可以用这个方法实现条件调用——只有方法存在才调用：

```
    $obj->snarl if $obj->can("snarl");
```

在多重继承情况下，这个方法允许调用所有覆盖掉的基类的方法，而不仅仅是最 左边的那个：

```
sub snarl {
        my $self = shift;
        print "Snarling: @_\n";
        my %seen;
        for my $parend (@ISA) {
                if (my $code = $parent->can("snarl")) {
                        $self->$code(@_) unless $seen{$code}++;
                }
        }
}
```

我们用 %seen 散列跟踪那些我们已经调用的子过程，这样我们才能避免多次调用 同一个子过程。这种情况在多个父类共享一个共同的祖先的时候可能发生。

会触发一个 AUTOLOAD（在下一节描述）的方法将不会被准确地汇报，除非该包已 经声明（但没有定义）它需要自动装载的子过程了。

INVOCANT-VERSION(NEED)

VERSION 方法返回 INVOCANT 的类的版本号，就是存贮在包的 $VERSION 变量里的那只。如果提供了 NEED 参数，它表示当前版本至少不能小于 NEED，而如果真的 小于就会抛出一个例外。这是 use 用以检查一个模块是否足够新所调用的方法。

```
    use Thread 1.0;      # 调用 Thread->VERSION(1.0)
```

```
print "Running versino ", Thread->VERSION, " of Thread.\n";
```

你可以提供自己的 VERSION 方法覆盖掉 UNIVERSAL 的。不过那样会令任何从 你的类衍生的类也使用哪个覆盖类。如果你不想发生这样的事情，你应该把你的 方法设计成把其他类的版本请求返回给 UNIVERSAL。

在 UNIVERSAL 里的方法是内建的 Perl 子过程，如果你使用全称并且传递两个参数，你就可以调用它们，比如 UNIVERSAL::isa($formobj, "HASH")。（但是我们不推荐这么用，因为通常而言 can 包含你真正在找的答案。）

你可以自由地给 UNIVERSAL 增加你自己的方法。（当然，你必须小心；否则你可能真的把事情搞砸，比如有些东西是假设找不到你正在定义的方法名的，这样它们就可以从其他地方自动装载进来。）下面我们创建了一个 copy 方法，所有类的对象都可以使用——只要这些对象没有定义自己的。（我们忘了给调用一个对象做解析。）

```
use Data::Dumper;

use Carp;

sub UNIVERSAL::copy {

        my $self = shift;

        if (ref $self) {

                return eval Dumper($self);    # 没有 CORE 引用

        }else{

                confess "UNIVERSAL::copy can't copy class $self";

        }

}
```

如果该对象包含任意到子过程的引用，这个 Data::Dumper 的策略就无法运转，因为它们不能正确地复现。即使能够拿到源程序，词法绑定仍然会丢失。

12.5.4 方法自动装载

通常，当你调用某个包里面未定义子过程，而该包定义了一个 AUTOLOAD 子过程，则调用该 AUTOLOAD 子过程并且抛出一个例外（参阅第十章，“自动装载 Autoloading”）。方法的运做略有不同。如果普通的方法查找（通过类，它的祖先以及最终的 UNIVERSAL）

没能找到匹配，则再按同样的顺序运行一便，这次是查找一个 AUTOLOAD 子过程。如果找到，则把这个子过程当作一个方法来调用，同时把包的 $AUTOLOAD 变量设置为该子过程的全名（就是代表 AUTOLOAD 调用的那个子过程。）

当自动装载方法的时候，你得小心一些。首先，如果 AUTOLOAD 的子过程代表一个叫 DESTROY 的方法调用，那么它应该立即返回，除非你的目的是仿真 DESTROY，那样的话对 Perl 有特殊含义，我们将在本章后面的"实例析构器"里描述。

```
sub AUTOLOAD {
    return if our $AUTOLOAD =~ /::DESTROY$/;
    ...
}
```

第二，如果该类提供一个 AUTOLOAD 安全网，那么你就不能对一个方法名使用 UNIVERSAL::can 来检查调用该方法是否安全。你必须独立地检查 AUTOLOAD：

```
if ($obj->can("methname") || $obj->can("AUTOLAOD")) {
    $obj->methname();
}
```

最后，在多重继承的情况下，如果一个类从两个或者更多类继承过来，而每个类都有一个 AUTOLOAD，那么只有最左边的会被触发，因为 Perl 在找到第一个 AUTOLOAD 以后就停下来了。

后两个要求可以很容易地通过声明包里的子过程来绕开，该包的 AUTOLOAD 就是准备管理这些方法的。你可以用独立的声明实现这些：

```
package Goblin;
sub kick;
sub bite;
sub scratch;
```

或者用 use subs 用法，如果你有许多方法要声明，这样会更方便：

```
package Goblin;
use subs qw(kick bite scratch);
```

甚至你只是声明了这些子过程而并没有定义它们，系统也会认为它是真实的。它们在一个 UNIVERSAL::can 检查里出现，而且更重要的是，它们在搜索方法的第二步出现，这样就永远不会进行第三步，更不用说第四步了。

“不过，”你可能会说，“它们调用了 AUTOLOAD，不是吗？”的确，它们最终调用了 AUTOLOAD，但是机制是不一样的。一旦通过第二步找到了方法存根(stub)，Perl 就会试图调用它。当最后发现该方法不是想要的方法时，则再次进行 AUTOLOAD 搜索，不过这回它从包含存根的类开始搜索，这样就把方法的搜索限制在该类和该类的祖先（以及 UNIVSRSAL）中。这就是 Perl 如何查找正确的 AUTOLOAD 来运行和如何忽略来自最初的继承树中错误的 AUTOLOAD 部分的方法。

12.5.5 私有方法

有一个调用方法的手段可以完全令 Perl 忽略继承。如果用的不是一个文本方法名，而是一个简单的标量变量，该变量包含一个指向一个子过程的引用，则立即调用该子过程。在前一节的 UNIVERSAL->can 的描述中，最后一个例子使用子过程的引用而不是其名字调用所有被覆盖了的方法。

这个特性的一个非常诱人的方面是他可以用于实现私有方法调用。如果把你的类放在一个模块里，你可以利用文件的词法范围为私有性服务。首先，把一个匿名子过程存放在一个文件范围的词法里：

```
# 声明私有方法
my $secret_door = sub {
    my $self = shift;
    ...
};
```

然后在这个文件里，你可以把那个变量当作保存有一个方法名这样来使用。这个闭合将会被直接调用，而不用考虑继承。和任何其他方法一样，调用者作为一个额外的参数传递进去。

```
sub knock {
        my $self = shift;
        if ($self->{knocked}++ > 5) {
                $self->$secret_door();
```

```
        }
}
```

这样就可以让该文件自己的子过程（类方法）调用一个代码超出该词法范围（因而无法访问）的方法。

12.6 实例析构器

和 Perl 里任何其他引用一样，当一个对象的最后一个引用消失以后，该对象的存储器隐含地循环使用。对于一个对象而言，你还有机会在这些事情发生的时候（对象内存循环使用）捕获控制，方法是在类的包里定义 DESTROY 子过程。这个方法在合适的时候自动被触发，而将要循环使用的对象是它的唯一的参数。

在 Perl 里很少需要析构器，因为存贮器管理是自动进行的。不过有些对象可能有一个位于存储器系统之外的状态（比如文件句柄或数据库联接），而且你还想控制它们，所以析构器还是有用的。

```
package MailNotify;

sub DESTROY {
        my $self = shift;
        my $fh = $self->{mailhandle};
        my $id = $self->{name};
        print $fh "\n$id is signing off at " . localtime( ) . "\n";
        close $fh;   # 关闭 mailer 的管道
}
```

因为 Perl 只使用一个方法来构造一个对象，即使该构造器的类是从一个或者多个其他类继承过来的也这样，Perl 也只是每个对象使用一个 DESTROY 方法来删除对象，也不管继承关系。换而言之，Perl 并不为你做分级析构。如果你的类覆盖了一个父类的析构器，那么你的 DESTROY 方法可能需要调用任意适用的基类的 DESTROY 方法：

```
sub DESTROY {
        my $self = shift;
        # 检查看看有没有覆盖了的析构器
```

```
    $self->SUPER::DESTROY if $self->can("SUPER::DESTROY");
    # 现在干你自己的事情
}
```

这个方法只适用于继承的类；一个对象只是简单地包含在当前对象里——比如，一个大的散列表里的一个数值——会被自动释放和删除。这也是为什么一个简单地通过聚集（有时候叫"有 xx"关系）实现的包含器要更干净，并且比继承（一个"是 xx"关系）更干净。换句话说，通常你实际上只需要把一个对象直接保存在另外一个对象里面而不用通过继承，因为继承会增加不必要的复杂性。有时候当你诉诸多重继承的时候，实际上单继承就足够用了。

你有可能明确地调用 DESTROY，但实际上很少需要这么做。这么做甚至是有害的，因为对同一个对象多次运行析构器可能会有让你不快的后果。

12.6.1 用 DESTROY 方法进行垃圾收集

正如第八章的"垃圾收集，循环引用和弱引用"节里介绍的那样，一个引用自身的变量（或者多个变量间接的相互引用）会一直到程序（或者嵌入的解释器）快要退出的时候才释放。如果你想早一些重新利用这些存储器，你通常是不得不使用 CPAN 上的 WeakRef[2] 方法来明确地打破或者弱化该引用。

对于对象，一个候补的解决方法是创建一个容器类，该容器类保存一个指向这个自引用数据结构的指针。为该被包含对象的类定义一个 DESTROY 方法，该方法手工打破自引用结构的循环性。你可以在 Perl Cookbook 这本书的第十三章里找到关于这些的例子，该章的名称是，"Coping with Circular Data Structures"（对付循环数据结构）。

当一个解释器退出的时候，它的所有对象都删除掉，这一点对多线程或者嵌入式的 Perl 应用非常重要。对象总是在普通引用被删除之前在一个独立的回合里被删除。这样就避免了 DESTROY 方法处理那些本身已经被删除的引用。（也是因为简单引用只有在嵌入的解释器中才会被当作垃圾收集，因为退出一个进程是回收引用的非常迅速的方法。但是退出并不运行对象的析构器，因此 Perl 先做那件事。）

12.7 管理实例数据

大多数类创建的对象实际上都是有几个内部数据域（实例数据）和几个操作数据域的方法的数据结构。

Perl 类继承方法，而不是数据，不过由于所有对对象的访问都是通过方法调用进行的，所以这样运行得很好。如果你想继承数据，那么你必须通过方法继承来实现。不过，Perl 在

多数情况下是不需要这么做的，因为大多数类都把它们的对象的属性保存在一个匿名散列表里。对象的实例数据保存在这个散列表里，这个散列表也是该对象自己的小名字空间，用以划分哪个类对该对象进行了哪些操作。比如，如果你希望一个叫 $city 的对象有一个数据域名字叫 elevation，你可以简单地 $city->{elevation} 这样访问它。可以不用声明。方法的封装会为你做这些。

假设你想实现一个 Person 对象。你决定它有一个叫"name"的数据域，因为某种奇怪的一致性原因，你将把它按照键字 name 保存在该匿名散列表里，该散列表就是为这个对象服务的。不过你不希望用户直接修改数据。要想获得封装的优点，用户需要一些方法来访问该实例变量，而又不用揭开抽象的面纱。

比如，你可能写这样的一对访问方法：

```
sub get_name {
        my $self = shift;
        return $self ->{name};
}

sub set_name {
        my $self = shift;
        $self->{name} = shift;
}
```

它们会导致下面的代码的形成：

```
   $him = Person->new();
   $him->set_name("Laser");
   $him->set_name( ucfirst($him->get_name) );
```

你甚至可以把两个方法组合成一个：

```
sub name {
        my $self = shift;
```

```
        if (@_) { $self->{name} = shift }
        return $self->{name};
}
```

这样会形成下面的代码：

```
$him = Person->new();
$him->name("BitBIRD");
$him->name( ucfirst($him->name) );
```

给每个实例变量（对于我们的 Person 类而言可能是 name，age，height 等等）写一个独立的函数的优点是直接，明显和灵活。缺点是每当你需要一个新的类，你最终都要对每个实例变量定义一个或两个几乎相同的方法。对于开头的少数几个类而言，这么做不算太坏，而且如果你喜欢这么干的话我们也欢迎你这么干。但是如果便利比灵活更重要，那么你可能就会采用后面描述的那种技巧。

请注意我们会变化实现，而不是接口。如果你的类的用户尊重封装，那么你就可以透明地从一种实现切换到另外一种实现，而不会让你的用户发现。（如果你的继承树里的家庭成员把你的类用于子类或者父类，那可能不能这么宽容了，毕竟它们对年你的认识比陌生人要深刻得多。）如果你的用户曾经深入地刺探过你的类中的私有部分，那么所导致的不可避免的损害就是他们自己的问题而不是你的。你所能做的一切就是通过维护好你的接口来快乐地过日子。试图避免这个世界里的每一个人做出一些有些恶意的事情会消耗掉你的所有时间和精力，并且最终你会发现还是徒劳的。

对付家成员更富挑战性。如果一个子类覆盖了一个父类的属性的指示器，那么它是应该访问散列表中的同一个域呢还是不应该？根据该属性的性质，不管那种做法都会产生一些争论。从通常的安全性角度出发，每个指示器都可以用它自己的类名字作为散列域名字的前缀，这样子类和父类就都可以有自己的版本。下面有几个使用这种子类安全策略的例子，其中包括标准的 Struct::Class 模块。你会看到指示器是这样组成的：

```
sub name {
        my $self =shift;
        my $field = __PACKAGE__ . "::name";
        if (@_) { $self->{$field} = shift }
        return $self->{field};
```

```
}
```

在随后的每个例子里，我们都创建一个简单的 Person 类，它有 name，race，和 aliases 域，每种类都有一个相同的接口，但是有完全不同的实现。我们不准备告诉你我们最喜欢哪种实现，因为根据实际使用的环境，我们几乎都喜欢。有些人喜欢弯曲的实现，有些人喜欢直接的实现。

12.7.1 用 use fields 定义的域

对象不一定要用匿名散列来实现。任何引用都可以。比如，如果你使用一个匿名数组，你可以这样设置一个构造器：

```
sub new {
        my $invocant = shift;
        my $class = ref($invocant) || $invocant;
        return bless [], $class;
}
```

以及象下面这样的指示器：

```
sub name {
        my $self = shift;
        if (@_) { $self->[0] = shift }
        return $self->[0];
}


sub race {
        my $self = shift;
        if (@_) { $self->[1] = shift }
        return $self->[1];
}
```

```
sub aliases {
        my $self = shift;
        if (@_) { $self->[2] = shift }
        return $self->[2];
}
```

数组访问比散列快一些，而且占的内存少一些，不过用起来不象散列那样方便。你不得不跟踪所有下标数字（不仅仅在你自己的类里面，而且还得在你的父类等等里面），这些下标用于指示你的类正在使用的数组的部分。这样你才能重复使用这些空间。

use fields 用法可以对付所有这些问题：

```
   package Person;
   use fields qw(name race aliases);
```

这个用法不会为你创建指示器方法，但是它的确是基于一些内建的方法之上（我们叫它伪散列）做一些类似的事情的。（不过你可能会希望对这些域用指示器进行封装，就象我们处理下面的例子一样。）伪散列是数组引用，你可以把它们当作散列那样来用，因为他们有一个相关联的键字映射表。use fields 用法为你设置这些键字映射，等效于声明了哪些域对 Person 对象是有效的；以此令 Perl 的编译器意识到它们的存在。如果你声明了你的对象变量的类型（就象在下个例子里的 my Person $self 一样），编译器也会聪明得把对该域的访问优化成直接的数组访问。这样做更重要的原因可能是它令域名字在编译时是类型安全的（实际上是敲键安全）。（参阅第八章里的"伪散列"。）

一个构造器和例子指示器看起来可能是这个样子的：

```
package Person;
use fields qw(naem race aliases);
sub new {
        my $type = shift;
        my Person $self = fields::new(ref $type || $type);
        $self->{name} = "unnamed";
```

```
        $self->{race} = "unknown";
        $self->{aliases} = [];
        return $self;
}
sub name {
        my Person $self = shift;
        $self->{name} = shift if @_;
        return $self->{name};
}
sub race {
        my Person $self = shift;
        $self->{race} = shift if @_;
        return $self->{race};
}
sub aliases {
        my Person $self = shift;
        $self->{aliases} = shift if @_;
        return $self->{aliases};
}
1;
```

如果你不小心拼错了一个用于访问伪散列的文本键字，你用不着等到运行时才发现这些问题。编译器知道对象 $self 想要引用的数据类型（因为你告诉它了），因此它就可以那些只访问 Person 对象实际拥有的数据域的代码。如果你走神了，并且想访问一个不存在的数据域（比如 $self->{mane}），那么编译器可以马上标出这个错误并且绝对不会让有错误的程序跑到解释器那里运行。

这种方法在声明获取实例变量的方法的时候仍然有些重复，所以你可能仍然喜欢使用下面介绍的技巧之一，这些技巧实现了简单指示器方法的自动创建。不过，因为所有的这些技巧都使用某种类型的间接引用，所以如果你使用了这些技巧，那么你就会失去上面的编译时词法类型散列访问的拼写检查好处。当然，你还是能保留一点点的时间和空间的优势。

如果你决定使用一个伪散列来实现你的类，那么任何从这个类继承的类都必须知晓下面的类的伪散列实现。如果一个对象是用伪散列实现的，那么所有继承分级中的成员都必须使用 use base 和 use fields 声明。比如：

```
package Wizard;

use base "Person";

user fields qw(staff color sphere);
```

这么干就把 Wizard 模块标为 Person 的子类，并且装载 Person.pm 文件。而且除了来自 Person 的数据域外，还在这个类中注册了三个新的数据域。这样，当你写：

```
my Wizard $mage = fields::new("Wizard");
```

的时候，你就能得到一个可以访问两个类的数据域的伪散列：

```
$mage->name("Gandalf");

$mage->color("Grey");
```

因为所有子类都必须知道他们用的是一种伪散列的实现，所以，从效率和拼写安全角度出发，它们应该使用直接伪散列句法：

```
$mage->{name} = "Gandalf";

$mage->{color} = "Grey";
```

不过，如果你希望保持你的实现的可互换性，那么你的类以外的用户必须使用指示器方法。

尽管 use base 只支持单继承，但也算不上非常严重的限制。参阅第三十一章的 use base 和 use fields 的描述。

12.7.2 用 Class::Struct 生成类

标准的 Class::Struct 模块输出一个叫 struct 的函数。它创建了你开始构造一个完整的类所需要的所有机关。它生成一个叫 new 的构造器，为每个该结构里命名的数据域增加一个指示器方法（实例变量）。

比如，如果你把下面结构放在一个 Person.pm 文件里：

```
package Person;

use Class::Struct;

struct Person => {   # 创建一个"Person"的定义

        name => '$',    # name 域是一个标量

        race => '$',    # race 域也是一个标量

        aliases => '@',    # 但 aliases 域是一个数组引用

};

1;
```

然后你就可以用下面的方法使用这个模块：

```
   use Person;

   my $mage = Person->new();

   $mage->name("Gandalf");

   $mage->race("Istar");

   $mage->aliases(  ["Mithrandir","Olorin", "Incanus"] );
```

Class::Struct 模块为你创建上面的所有四种方法。因为它遵守子类安全原则，总是在域名字前面前缀类名字，所以它还允许一个继承类可以拥有它自己独立的与基类同名的域，而又不用担心会发生冲突。这就意味着你在用于这个实例变量的时候，必须用 "Person::name" 而不能用 "name" 当作散列键字来访问散列表。

在结构声明里的数据域可以不是 Perl 的基本类型。它们也可以声明其他的类，但是和 struct 一起创建的类并非运行得最好，因为那些对类的特性做出假设的函数并不是对所有的类都能够明察秋毫。比如，对于合适的类而言，会调用 new 方法来初始化它们，但是很多类有其他名字的构造器。

参阅第三十二章，标准模块，以及它的联机文档里关于 Class::Struct 的描述。许多标准模块使用 Class::Struct 来实现它们的类，包括 User::pwent 和 Net::hostent。阅读它们的代码会很有收获。

12.7.3 使用 Autoloading（自动装载）生成指示器

正如我们早先提到过的，当你调用一个不存在的方法的时候，Perl 有两种不同的手段搜索一个 AUTOLOAD 方法，使用哪种方法取决于你是否声明了一个存根方法。你可以利用这个特性提供访问对象的实例数据的方法，而又不用为每个实例书写独立的函数。在 AUTOLOAD 过程内部，实际被调用的方法的名字可以从 $AUTOLOAD 变量中检索出来。让我们看看下面下面的代码：

```
user Person;

$him = Person->new;

$him->name("Weiping");

$him->race("Man");

$him->aliases( ["Laser", "BitBIRD", "chemi"] );

printf "%s is of the race of %s. \n", $him->name, $him->race;

printf "His aliases are: ", join(", ", @{$him->aliases}), ".\n";
```

和以前一样，这个版本的 Person 类实现了一个有三个域的数据结构：name，race，和 aliases：

```
package Person;

use Carp;


my %Fields = (

"Person::name" => "unnamed",

"Person::race" => "unknown",

"Person::aliases" => [],

);

#下一个声明保证我们拿到自己的 autoloader（自动装载器）。

use subs qw(name race aliases);
```

```
sub new {
        my $invocant = shift;
        my $class = ref($invocant) || $invocant;
        my $self = { %Fields, @_};   # 类似 Class::Struct 的克隆
        bless $self, $class;
        return $self;
}

sub AUTOLOAD {
        my $self = shift;
        # 只处理实例方法，而不处理类方法
        croak "$self not an object" unless ref($invocant);
        my $name = our $AUTOLOAD;
        return if $name =~ /::DESTROY$/;
        unless (exist $self->{name}) {
                croak "Can't access `$name' field in $self";
        }
        if (@_) {return $self->{$name} = shift }
        else { return $self->{$name} }
}
```

如你所见，这里你可找不到叫 name，race，或者 aliases 的方法。AUTOLOAD 过程为你照看那些事情。当某人使用 $him->name("Aragorn") 的时候，那么 Perl 就调用 AUTOLOAD，同时把 $AUTOLOAD 设置为 "Person::name"。出于方便考虑，我们用了全名，这是访问保存在对象散列里的数据域的最正确的方法。这样你可以把这个类用做一个更大的继承级中的一部分，而又不用担心会和其他类中使用的同名数据域冲突。

12.7.4 用闭合域生成指示器

大多数指示器方法实际上干的是一样的事情：它们只是简简单单地从实例变量中把数值抓过来并保存起来．在 Perl 里，创建一个近乎相同的函数族的最自然的方法就是在一个闭合区域里循环．但是闭合域是匿名函数，它们缺少名字，而方法必须是类所在包的符号表的命名子过程，这样它们才能通过名字来调用．不过这不算什么问题——只要把那个闭合域赋值给一个名字合适的类型团就可以了．

```
package Person;


sub new {
        my $invocant = shift;
        my $self = bless( {}, ref $invocant || $invocant);
        $self->init();
        return $self;
}


sub init {
        my $self = shift;
        $self->name("unnamed");
        $self->race("unknown");
        $self->aliases([]);
}


for my $field (qw(name race aliases)) {
        my $slot = __PACKAGE__ . "::$field";
        no strict "refs";    # 这样指向类型团的符号引用就可以用了
```

```
        *$field = sub {
                my $self = shift;
                $self->{$slot} = shift if @_;
                return $self->{$slot};
        };
}
```

闭合域是为你的实例数据创建一个多用途指示器的最干净的操作方法.而且不论对计算机还是你而言,它都很有效.不仅所有指示器都共享同一段代码(它们只需要它们自己的词法本),而且以后你要增加其他属性也方便，你要做的修改是最少的：只需要给 for 循环增加一条或更多条单词，以及在 init 方法里加上几句就可以了.

12.7.5 将闭合域用于私有对象

到目前为止，这些管理实例数据的技巧还没有提供"避免"外部对数据的访问的机制. 任何类以外的对象都可以打开对象的黑盒子然后查看内部--只要它们不怕质保书失效. 增强私有性又好象阻碍了人们完成任务. Perl 的哲学是最好把一个人的数据用下面的标记封装起来：

```
IN CASE OF FIRE
    BREAK GLASS
```

如果可能，你应该尊重这样的封装，但你在紧急情况（比如调试）下仍然可以很容易地访问其内容.

但是如果你确实想强调私有性， Perl 不会给你设置障碍. Perl 提供低层次的制作块，你可以用这些制作块围绕在你的类和其对象周围形成无法进入的私有保护网--实际上，它甚至比许多流行的面向对象的语言的保护还要强.它们内部的词法范围和词法变量是这个东西的关键组件，而闭合域起到了关键的作用.

在"私有方法"节里，我们看到了一个类如何才能使用闭合域来实现那种模块文件外部不可见的方法. 稍后我们将看看指示器方法，它们把类数据归执得连类的其他部分都无法进行不受限制的访问。那些仍然是闭合域相当传统的用法。真正让我们感兴趣的东西是把闭合域用做一个对象。该对象的实例变量被锁在该对象内部--也就是说，闭合域，也只有闭合域才能自由访问。这是非常强的封装形式；这种方法不仅可以防止外部对对象内部的操作，甚至连同一个类里面的其他方法也必须使用恰当的访问方法来获取对象的实例数据。

下面是一个解释如何实现这些的例子。我们将把闭合域同时用于生成对象本身和生成指示器：

```
package Person;

sub new {

        my $invocant = shift;

        my $class = ref($invocant) || $invocant;

        my $data = {

                NAME => "unnamed",

                RACE => "unknown",

                ALIASES =>[],

        };


        my $self = sub {

                my $field = shift;

                ################################

                ### 在这里进行访问检查       ###

                ################################

                if (@_) { $data->{$field} = shift }

                return $data->{$field};

        };

        bless ($self, $class);

        return $self;

};
```

```
# 生成方法名


for my $field (qw(name race aliases)) {
        no strict "refs";    # 为了访问符号表
        *$field = sub {
                my $self = shift;
                return $self->(uc $field, @_);
        };
}
```

new 方法创建和返回的对象不再是一个散列，因为它在我们刚才看到的其他的构造器里。实际上是一个闭合域访问存储在散列里的属性数据，该闭合域是唯一可以访问该属性数据的类成员，而存储数据的散列是用 $data 引用的。一旦构造器调用完成，访问 $data （里面的属性）的唯一方法就是通过闭合域。

在一个类似 $him->name("Bombadil") 这样的调用中，在 $self 里存储的调用对象是那个闭合域，这个闭合域已经由构造器赐福（bless）并返回了。对于一个闭合而言，我们除了能调用它以外，干不了太多什么事，因此我们只是做 $self->(uc $field, @_)。请不要被箭头糊弄了，这条语句只是一个正规的间接函数调用，而不是一个方法调用。初始参数是字串 "name"，而其他的参数就是那些传进来的。（注：当然，双函数调用比较慢，但是如果你想快些，你还会首先选用对象吗？）一旦我们在闭合域内部执行，那么在 $data 里的散列就又可以访问得到了。这样闭合域就可以自由地给任何它愿意的对象访问权限，而封杀任何它讨厌的对象的访问。

在闭合域外面没有任何对象可以不经中介地访问这些非常私有的实例数据，甚至该类里面的其他方法都不能。它们可以按照 for 循环生成的方法来调用闭合域，可能是设置一个该类从来没有听说过的实例变量。但是我们很容易通过在构造器里放上几段代码来阻止这样的方法使用，放那些代码的地方就是你看到的上面的访问检查注释的地方。首先，我们需要一个通用的导言：

```
    use Carp;
    local $Carp::CarpLevel = 1;    # 保持牢骚消息短小
    my ($cpack, $cfile) = caller();
```

然后我们进行各个检查。第一个要确保声明的属性名存在：

```
croak "No valid field '$field' in object"
    unless exists $data->{$field};
```

下面这条语句只允许来自同一个文件的调用：

```
carp "Unmediated access denied to foreign file"
    unless $cfiled eq __FILE__;
```

下面这条语句只允许来自同一个包的调用：

```
carp "Unmediated access denied to foreign package ${cpack}::"
    unless $cpack eq __PACKAGE__;
```

所有这些代码都只检查未经中介的访问。那些有礼貌地使用该类指定的方法访问的用户不会受到这些约束。Perl 会给你一些工具，让你想多挑剔就有多挑剔。幸运的是，不是所有人都这样。

不过有些人应该挑剔。当你写飞行控制软件的时候，严格些就是正确的了。如果你想成为或者要成为这些人员，而且你喜欢使用能干活的代码而不是自己重新发明所有东西，那么请看看 CPAN 上 Damian Conway 的 Tie::SecureHash 模块。它实现了严格的散列，支持公有，保护和私有约束。它还对付我们前面的例子中忽略掉的继承性问题。Damian 甚至还写了一个更雄心勃勃的模块，Class::Contract，在 Perl 灵活的对象系统上强加了一层正式的软件工程层。这个模块的特性列表看起来就象一本计算机科学教授的软件工程课本的目录，（注：你知道 Damian 是干什么的吗？顺便说一句，我们非常建议你看看他的书，Object Oriented Perl（面向对象的 Perl）(Manning Publications, 1999)）。包括强制封装，静态继承和用于面向对象的 Perl 的按需设计条件检查，以及一些用于对象和类层次的属性，方法，构造器和析构器定义的的可声明的语法，以及前提，后记和类固定。天！

12.7.6 新技巧

到了 Perl 5.6，你还可以声明一个方法并指出它是返回左值的。这些是通过做值子过程属性实现的（不要和对象方法混淆了）。这个实验性的特性允许你把该方法当作某些可以在一个等号左边出现的东西：

```
package Critter;
```

```
sub new {
        my $class = shift;
        my $self = { pups => 0, @_ };       # 覆盖缺省。
        bless $self, $class;
}

sub pups : lvalue {             # 我们稍后给 pups()赋值
        my $self = shift;
        $self->{pups};
}

package main;
$varmint = Critter->new(pups => 4);
$varmint->pups *= 2;             # 赋给 $varmint->pups!
$varmint->pups =~ s/(.)/$1$1/;          # 现场修改 $varmint->pups!
print $varmint->pups;             # 现在我们有 88 个 pups。
```

这么做让你以为 $varminit->pups 仍然是一个遵守封装的变量。参阅第六章，子过程，的“左值属性”。

如果你运行的是一个线程化的 Perl，并且你想确保只有一个线程可以调用一个对象的某个方法，你可以使用 locked 和 method 属性实现这些功能：

```
   sub pups : locked method {
      ...
   }
```

当任意线程调用一个对象上的 pups 方法的时候，Perl 在执行前锁住对象，阻止其他线程做同样的事情。参阅第六章里的“locked 和 method 属性”。

12.8 管理类数据

我们已经看到了按对象访问对象数据的几种不同方法。不过，有时候你希望有些通用的状态在一个类里的所有对象之间共享。不管你使用哪个类实例（对象）来访问他们，这些变量是整个类的全局量，而不只是该类的一个实例，（C++ 程序员会认为这些是静态成员数据。）下面是一些类变量能帮助你的情况：

- 保存一个曾经创建的所有对象的计数，或者是仍然存活的数量。
- 保存一个你可以叙述的所有对象的列表
- 保存一个全类范围调试方法使用的日志文件的文件名或者文件描述符。
- 保存收集性数据，比如想一个网段里的所有 ATM 某一天支取的现金的总额。
- 跟踪类创建的最后一个或者最近访问过的对象。
- 保存一个内存里的对象的缓存，这些对象是从永久内存中重新构建的。
- 提供一个反转的查找表，这样你就可以找到一个基于其属性之一的数值的对象。

然后问题就到了哪里去存储这些共享属性上面。Perl 没有特殊的语法机制用于声明类属性，用于实例属性的也多不了什么。Perl 给开发者提供了一套广泛强大而且灵活的特性，这些特性可以根据不同情况分别雕琢成适合特定的需要。然后你就可以根据某种情况选择最有效的机制，而不是被迫屈就于别人的设计决定。另外，你也可以选择别人的设计决定 ——那些已经打包并且放到 CPAN 去的东西。同样，"回字有四种写法"。

和任何与类相关的东西一样，类数据不能被直接访问，尤其是从类实现的外部。封装的理论没有太多关于为实例变量设置严格受控的指示器方法的内容，但是却发明了 public 来直接欺骗你的类变量，就好象设置 $SomeClass::Debug = 1。要建立接口和实现之间干净的防火墙，你可以创建类似你用于实例数据的指示器方法来操作类数据。

假设我们想跟踪 Critter 对象的全部数量。我们将把这个数量存储在一个包变量里，但是提供一个方法调用 population，这样此类的用户就不用知道这个实现：

```
Critter->population()    # 通过类名字访问

$gollum->population()    # 通过实例访问
```

因为在 Perl 里，类只是一个包，存储一个类的最自然的位置是在一个包变量里。下面就是这样的一个类的简单实现。population 方法忽略它的调用者并且只返回该包变量的当前值 $Population。（有些程序喜欢给它们的全局量大写开头。）

```
package Critter;
```

```
our $population = 0;

sub pupulation { return $Population; }

sub DESTROY {$Population --}

sub spawn {

        my $invocant = shift;

        my $class = ref($invocant) || $invocant;

        $Population++;

        return bless { name => shift || "anon" }, $class;

}

sub name {

        my $self = shift;

        $self->{name} = shift if @_;

        return $self->{name};

}
```

如果你想把类数据方法做得想实例数据的指示器那样，这么做：

```
our $Debugging = 0;   # 类数据

sub debug {

        shift;      # 有意忽略调用者

        $Debugging = shift if @_;

        return $Debugging;

}
```

现在你可以为给该类或者它的任何实例设置全局调试级别。

因为它是一个包变量，所以 $Debugging 是可以全局访问的。但是如果你把 our 变量改成 my，那么就只有该文件里后面的代码可以看到它。你还可以走得再远一些——你可以把对类属性的访问限制在该类本身其余部分里。把该变量声明裹在一个块范围里：

```
{
        my $Debugging = 0;      # 词法范围的类数据
        sub debug {
                shift;          # 有意忽略调用者
                $Debugging = shift if @_;
                return $Debugging;
        }
}
```

现在没有人可以不通过使用指示器方法来读写该类属性，因为只有那个子过程和变量在同一个范围因而可以访问它。

如果一个生成的类继承了这些类指示器，那么它们仍然访问最初的数据，不管变量是用 our 还是用 my 定义的。数据是包无关的。当方法在它们最初定义的地方执行的时候，你可以看到它们，但是在调用它的类里面可不一定看得到。

对于某些类数据，这个方法运行得很好，但对于其他的而言，就不一定了。假设我们创建了一个 Critter 的 Warg 子类。如果我们想分离我们的两个数量， Warg 就不能继承 Critter 的 population 方法，因为那个方法总是返回 $Critter::Poplation 的值。

你可能会不得不根据实际情况决定类属性与包相关是否有用。如果你想要包相关的属性，可以使用调用者的类来定位保存着类数据的包：

```
sub debug {
        my $invocant = shift;
        my $class = ref($invocant) || $invocant;
        my $varname = $class . "::Debugging";
        no strict "refs";       # 符号访问包数据
        $$varname = shift if @_;
        return $$varname;
}
```

我们暂时废除严格的引用，因为不这样的话我们就不能把符号名全名用于包的全局量。这是绝对有道理的：因为所有定义的包变量都存活在一个包里，通过该包的符号表访问它们是没什么错的。

另外一个方法是令对象需要的所有东西——甚至它的全局类数据——都可以由该对象访问（或者可以当作参数传递）。要实现这些功能，你通常不得不为每个类都做一个精制的构造器，或者至少要做一个构造器可以调用的精制的初始化过程。在构造器或者初始化器里，你把对任何类数据的引用直接保存在该对象本身里面，这样就没有什么东西需要查看它们了。访问器方法使用该对象来查找到数据的引用。

不要把定位类数据的复杂性放到每个方法里，只要让对象告诉方法数据在哪里就可以了。这个办法只有在类数据指示器方法被当作实例方法调用的时候才好用，因为类数据可能在一个你用包名字无法访问到的词法范围里。

不管你是怎么看待它，与包相关的类数据总是有点难用。继承一个类数据的指示器方法的确更清晰一些，你同样有效地继承了它能够访问的状态数据。参阅 perltootc 手册页获取管理类数据的更多更灵活的方法。

12.9 总结

除了其他东西以外，大概就这么多东西了。现在你只需要走出去买本关于面向对象的设计方法学的书，然后再花 N 个月的时间来学习它就行了。

第十三章 重载

对象非常酷，但有时候它有点太酷了。有时候你会希望它表现得少象一点对象而更象普通的数据类型一点。但是实现这个却有问题：对象是用引用代表的引用，而引用除了当引用以外没什么别的用途。你不能在引用上做加法，也不能打印它们，甚至也不能给它们使用许多 Perl 的内建操作符。你能做的唯一一件事就是对它们解引用。因此你会发现自己在写许多明确的方法调用，象：

```
print  $object->as_string;

$new_object = $subject->add($object);
```

象这样的明确的解引用通常都是好事；你决不能把你的引用和指示物混淆，除非你想混淆它们。下面可能就是你想混淆的情况之一。如果你把你的类设计成使用重载，你可以装做看不到引用而只是说：

```
print $object;

$new_object = $subject + $object;
```

当你重载某个 Perl 的内建操作符的时候，你实际上定义了把它应用于某特定类型的对象时的特性。有很多 Perl 的模块利用了重载，比如 Math::BigInt，它让你可以创建 Math::BigInt 对象，这些对象的性质和普通整数一样，但是没有尺寸限制。你可以用 + 把它们相加，用 / 把它们相除，用 <=> 比较它们，以及用 print 打印它们。

请注意重载和自动装载（autoload）是不一样的，自动装载是根据需要装载一个缺失的函数或方法。重载和覆盖（overriding）也是不一样的，覆盖是一个函数或方法覆盖了另外一个。重载什么东西也不隐藏；它给一个操作添加了新含义，否则在区区引用上进行该操作就是无聊的举动。

13.1 overload 用法

use overload 用法实现操作符重载。你给它提供一个操作符和对应的性质的键字/数值列表：

```
package MyClass;


use overload    '+' => \&myadd,          # 代码引用

'<' => "less_then";                      # 命名方法

'abs' => sub { return @_ },              # 匿名子过程
```

现在，如果你想相加两个 MyClass[2] 类，则调用 myadd 子过程来计算结果。

当你试图用 < 操作符比较两个 MyClass[2] 对象，Perl 注意到该性质被声明为一个字串，然后就把该字串当作一个方法名字而不仅仅是一个子过程名字。在上面的例子里，less_then 方法可以由 MyClass[2] 包本身提供或者从 MyClass[2] 的基类中继承过来，但是 myadd子过程必须由当前包提供。匿名子过程 abs 自己提供得甚至更直接。不过这些过程你提供的，我们叫它们句柄（handler）。

对于单目操作符（那些只有一个操作数的东西，比如 abs），当该操作符应用于该类的一个对象的时候则调用为该类声明的句柄。

对于双目操作符而言，比如 + 或 <，当第一个操作数是该类的一个对象或当第二个操作数是该类的对象而且第一个第一个操作数没有重载性质的时候，则调用该句柄。因此你可以用下面两种句法：

```
$object + 6
```

或：

```
6 + $object
```

而不用担心操作数的顺序。（在第二个例子里，在传递给句柄的时候操作数会对换）。如果我们的表达式是：

```
$animal + $vegetable
```

并且 $animal 和 $vegetable 是不同的类的对象，两个都使用了重载技术，那么 $animal 的重载性质将被触发。（我们希望该动物喜欢蔬菜。）

在 Perl 里只有一个三目操作符，?:，而且幸运的是你不能重载它。

*重载句柄

在操作一个重载了的操作符的时候，其对应的句柄是带着三个参数调用的。前两个参数是两个操作数。如果该操作符只使用一个操作数，第二个参数是 undef。

第三个参数标明前两个参数是否交换。即使是在普通算术的规则里，有些操作也不怎么在乎它们的参数的顺序，比如加法和乘法；不过其他的东西，比如减法和除法则关心。（注：（注：当然，我们并不要求你重载的对象遵循普通算术，不过最好不要让人们吃惊。很奇怪的是，许多语言错误地用字串连接功能重载了 +，它是不能交换的，而且只是暧昧的相加。要找一个不同的解决方法，请参阅 Perl。）看看下面两个的区别：

$object - 6

和：

1. - $object

如果给一个句柄的头两个参数已经被交换过了，第三个参数将是真。否则，第三个参数是假，这种情况下还有一个更好的区别：如果该句柄被另外一个参与赋值的句柄触发（就象在 +=

里用 + 表示如何相加），那么第三个参数就不仅仅是假，而是 undef。这个区别可以应用一些优化。

举个例子，这里是一个类，它让你操作一个有范围限制的数字。它重载了 + 和 -， 这样对象相加或相减的范围局限在 0 和 255 之间：

```
package ClipByte;


use overload '+' => \&clip_add,

'-' => \&clip_sub;


sub new {

        my $class = shift;

        my $value = shift;

        return bless \$value => $class;

}


sub clip_add {

        my ($x, $y) = @_;

        my ($value) = ref($x) ? $$x : $x;

        $value += ref($y) ? $$y :$y;

        $value = 255 if $value > 255;

        $value = 0 if $value < 0;

        return bless \$value => ref($x);

}


sub clip_sub {
```

```
        my ($x, $y, $swap) = @_;
        my ($value) = (ref $x) ? $$x : $x;
        $value     -= (ref $y) ? $$y : $y;
        if ($swap) { $value = -$value }
        $value = 255 if $value > 255;
        $value =   0 if $value < 0;
        return bless \$value => ref($x);
}

package main;

$byte1 = ClipByte->new(200);
$byte2 = ClipByte->new(100);

$byte3 = $byte1 + $byte2;    # 255
$byte4 = $byte1 - $byte2;    # 100
$byte5 = 150 - $byte2;       # 50
```

你可以注意到这里的每一个函数实际上都是一个构造器，所以每一个都使用 bless 把它的新的对象赐福回给当前类--不管是什么；我们假设我们的类可以被继承。我们还假设如果 $y 是一个引用，它是指向一个我们自己类型的对象的引用。除了测试 ref($y) 以外，如果我们想更彻底一些（也慢一些）我们也可以调用 $y->isa("ClipByte")。

13.3 可重载操作符

你只能重载一部分操作符，它们在表 13-1 列出。当你用 use overload 时，操作符也在 %overload::ops 散列列出供你使用，不过其内容和这里的有一点区别。

表 13-1。重载操作符

范畴	操作符
转换	"" 0+ bool
算术	+ - * / % ** x . neg
逻辑	!
位操作	& | ~ ^ ! << >>
赋值	+= -= *= /= %= **= x= .= <<= >>= ++ --
比较	= < < > >= = <=> lt le gt ge eq ne cmp
数学	atan2 cos sin exp abs log sqrt
文本	<> </td>
解引用	${} @{} %{} &{} *{}
伪	nomethod fallback =>

请注意 neg，bool，nomethod，和 fallback 实际上不是 Perl 的操作符。五种解引用，""，和 0+ 可能看起来也不象操作符。不过，它们都是你给 use overload 提供的参数列表的有效键字。这不是什么问题。我们会告诉你一个小秘密：说 overload 用法重载了操作符是一个小花招。它重载了下层的操作符，不管是通过他们的"正式"操作符明确调用的，还是通过一些相关的操作符隐含调用的。（我们提到的伪操作符只能隐含地调用。）换句话说，重载不是在语句级发生的，而是在语义级。原因是我们不是为了好看而是为了正确。请随意进行概括。

请注意 = 并不象你预料的那样重载 Perl 的赋值操作符。那样做是错的，稍后详细介绍这个。

我们将从转换操作符开始讨论，但并不是因为它们最显眼（它们可不抢眼），而是因为它们是最有用的。许多类除了重载用 "" 键字（没错，就是一行上的两个双引号。）声明的字串化以外不会重载任何东西。

转换操作符： ""，0+，bool 这三个键字让你给 Perl 提供分别自动转换成字串，数字和布尔值的性质。

当一个非字串变量当作字串使用的时候，我们就说是发生了字串化。当你通过打印， 替换，连接，或者是把它用做一个散列键字等方法把一个变量转换成字串时就发生 这个动作。字串化也是当你试图 print 一个对象时看到象 SCALAR（0xba5fe0) 这样的东西的原因。

我们说当一个非数字值在任意数字环境下转换成一个数字时发生的事情叫数字化， 这些数字环境可以是任意数学表达式，数组下标，或者是 ... 范围操作符的操作 数。

最后，尽管咱们这里没有谁急于把它称做布尔化，你还是可以通过创建一个 bool 句柄来定义一个对象在布尔环境里应该如何解释（比如 if， unless，while， for，and，or，&&，||，?:，或者 grep 表达式的语句块）。

如果你已经有了它们中的任意一个，你就可以自动生成这三个转换操作符的任何 一个（我们稍后解释自动生成）。你的句柄可以返回你喜欢的任何值。请注意如果 触发转换的操作也被重载，则该重载将在后面立即发生。

这里是一个 "" 的例子,它在字串化时调用一个对象的 as_string 句柄。别忘记 引起引号：

```
package Person;

use overload q("") => \&as_string;

sub new { my $class = shift; return bless {@_} => $class; }

sub as_string {

my $self = shift; my ($key, $value, $result); while (( $key, $value) = each %$self) { $result .= "$key => $value\n"; } return $result; }

$obj = Person->new(height => 72, weight => 165, eyes => "vrown"); print $obj;
```

这里会打印下面的内容（一散列顺序），而不是什么 Person=HASH(0xba1350)之类 的东西：

```
weight => 165 ...
```

（我们真诚的希望此人不是用公斤和厘米做单位的。）

算术操作符：+，-，*，/，%，**，x，.，neg

除了 neg 以外这些应该都很熟悉，neg 是一个用于单目负号（-123 里的 -）的特殊 重载键字。neg 和 - 键字之间的区别允许你给单目负号和双目负号（更常见的 叫法是减号）声明不同的性质。

如果你重载了 - 而没有重载 neg，然后试图使用一个单目负号，Perl 会为你模拟 一个 neg 句柄。这就是所谓的自动生成，就是说某些操作符可以合理地从其他 操作符中归纳出来（以"该重载操作符将和该普通操作符有一样的关系"为假设） 。因为单目符号可以表示成双目符号的一个函数（也就是，-123 等于 0-123）， 在 - 够用的时候，Perl 并不强

制你重载 neg。（当然，如果你已经独裁地定义了 双目负号用来把第二个参数和第一个参数分开，那么单目操作符会是一个很好的 抛出被零除例外的好方法。）

由 . 操作符进行的连接可以通过字串化句柄（见上面的 "" ）自动生成。

逻辑操作符：!

如果没有声明用于 ! 的句柄，那么它可以用 bool，""，或者 0+ 句柄自动生成。 如果你重载了 ! 操作符，那么当表现你的请求的时候，同样还会触发 not 操作 符。（还记得我们的小秘密吗？）你可能觉得奇怪：其他逻辑操作符哪里去了？ 但是大多数逻辑操作符不能重载，因为它们是短路的。他们实际上是控制流 操作符，这样它们就可以推迟对它们的一些参数的计算。那也是 ?: 操作符不能 重载的原因。

位运算操作符：&，|，~，^，<<，>> ~ 操作符是一个单目操作符；所有其他的都是双目。下面是我们如何重载 >>， 使之能做类似 chop 的工作：

```
package ShiftString[2];

use overload '>>' => \&right_shift, '"""' => sub { ${ $_[0] } };

sub new { my $class = shift; my $value = shift; return bless \$value =>
$class; }

sub right_shift { my ($x, $y) = @_; my $value = $$x; substr($value, -$y) = "";
return bless \$value => ref($x); }

$camel = ShiftString[2]->new("Camel"); $ram = $camel >>2; print $ram; #
Cam
```

赋值操作符：+=，-=，*=，/=，%=，**=，x=，.=，<<=，>>=，++，-- 这些赋值操作符可以改变它们的参数的值或者就那样把它的参数放着。结果是只有 新值和旧值不同时才把结果赋值给左手边的操作数。这样就允许同一个句柄同时 用于重载 += 和 +。尽管这么做是可以的，不过我们并不建议你这么做，因为根据 我们稍后将在"当缺失重载句柄的时候（nomethod 和 fallback）"里描述的 语义，Perl 将为 + 激活该句柄，同时假设 += 并没有直接重载。

连接（.=）可以用字串化后面跟着普通的字串连接自动生成。++ 和 -- 操作符 可以从 + 和 -（或者 += 和 -=）自动生成。

实现 ++ 和 -- 的句柄可能会变更（更改）他们的参数。如果你希望自增也能对 字母有效，你可以用类似下面的句柄实现：

```
package MagicDec;

use overload
q(--) => \&decrement,
q("") => sub { ${ $_[0] } };

sub new {
        my $class = shift;
        my $value = shift;
        bless \$value => $class;
}

sub decrement {
        my @string = reverse split (//, ${ $_[0] } );
        my $i;
        for ( $i = 0; $i < @string; $i++ ) {
                last unless $string[$i] =~ /a/i;
                $string[$i] = chr( ord($string[$i]) + 25 );
        }
        $string[$i] = chr( ord($string[$i]) - 1);
        my $result = join(' ', reverse @string);
        $_[0] = bless \$result => ref($_[0]);
}
```

```
package main;


for $normal (qw/perl NZ pa/) {
        $magic = MagicDec->new($normal);
        $magic --;
        print "$normal goes to $magic\n";
}
```

打印出：

```
perl goes to perk
NZ goes to NY
Pa goes to Oz
```

很准确地对 Perl 神奇的字串自增操作符做了反向工程。

++$a 操作符可以用 $a += 1 或 $a = $a +1 自动生成，而 $a-- 使用 $a -= 1 或 $a = $a - 1。不过，这样并不会触发真正的 ++ 操作符会触发的拷贝性质。参阅本章稍后的“拷贝构造器”。

比较操作符：==，<，<=，>，>=，!=，<=>，lt，le，gt，ge，eq，ne，cmp 如果重载了 <=>，那么它可以用于自动生成 <, <=, >, >=, = 和 的性质。 类似地，如果重载了 cmp，那么它可以用于自动生成 lt, le, gt, ge, eq 和 ne 的性质。

请注意重载 cmp 不会让你的排序变成你想象的那么简单，因为将要比较的是字串 化的版本的对象，而不是对象本身。如果你的目的是对象本身那么你还会希望重载 ""。

数学函数：atan2，cos，sin，exp，abs，log，sqrt 如果没有 abs ，那它可以从 < 或 <=> 与单目负号或者减号组合中自动生成。

重载可以用于为单目负号或者为 abs 函数自动生成缺失的句柄，而它们本身也 可以独立的被重载。（没错，我们知道 abs 看起来象个函数，而单目负号象个 操作符，但是从 Perl 的角度而言它们的差别没这么大。）

反复（迭代）操作符：<> 使用 readline（在它从一个文件句柄读入数据的时候，象 while () 里的） 或者 glob （当它用于文件聚团的时候，比如在 @files = <*.*> 里）。

```
package LuckyDraw;

use overload
'<>' => sub {
        my $self = shift;
        return splice @$self, rand @$self, 1;
};

sub new {
        my $class = shift;
        return bless [@_] => $class;
}

package main;

$lotto = new LuckyDraw 1 .. 51;

for (qw(1st 2nd 3rd 4th 5th 6th)) {
        $lucky_number = <$lotto>;
        print "The $_ lucky number is: $lucky_number.\n";
}

$lucky_number = <$lotto>;
print "\nAnd the bonus number is: $lucky_number.\n";
```

在加州，这些打印出：

```
The 1st lucky number is: 18

The 2nd lucky number is: 11

The 3rd lucky number is: 40


The 4th lucky number is: 7

The 5th lucky number is: 51

The 6th lucky number is: 33


And the bonus number is: 5
```

解引用操作符：${}，@{}，%{}，&{}，*{} 对标量，数组，散列，子过程，和团的解引用可以通过重载这五个符号来截获。

Perl 的 overload 的在线文档演示了你如何才能使用这些操作符来仿真你自己的 伪散列。下面是一个更简单的例子，它实现了一个有匿名数组的对象，但是使用 散列引用。别试图把他当作真的散列用；你不能从该对象删除键字/数值对。如果 你想合并数组和散列符号，使用一个真的伪散列。

```
package PsychoHash;


use overload '%{}' => \&as_hash;


sub as_hash {
        my ($x) = shift;
        return { @$x };
}
```

```
sub new {
        my $class = shift;
        return bless [ @_ ] => $class;
}

$critter = new PsychoHash( height => 72, weight => 365, type => "camel" );

print $critter->{weight};       # 打印 365
```

又见第十四章，捆绑变量，那里有一种机制你可以用于重新定义在散列，数组， 和标量上的操作。

当重载一个操作符的时候，千万不要试图创建一个带有指向自己的引用的操作符。比如，

```
use overload '+' => sub { bless [ \$_[0], \$_[1] ] };
```

这么干是自找苦吃，因为如果你说 $animal += $vegetable，结果将令 $animal 是一个引用一个赐福了的数组引用，而该数组引用的第一个元素是 $animal。这是循环引用，意味着即使你删除了 $animal，它的内存仍然不会释放，直到你的进程（或者解释器）终止。参阅第八章，引用，里面的"垃圾收集，循环引用，和弱引用"。

13.4 拷贝构造器（=）

尽管 = 看起来想一个普通的操作符，它做为一个重载的键字有点特殊的含义。它并不重载 Perl 的赋值操作符。它不能那样重载，因为赋值操作符必须保留起来用于赋引用的值，否则所有的东西就都完了。

= 句柄在下面的情况下使用：一个修改器（比如 ++，--，或者任何赋值操作符）施用于一个引用，而该引用与其他引用共享其对象。= 句柄令你截获这些修改器并且让你自己拷贝该对象，这样拷贝本身也经过修改了。否则，你会把原来的对象给改了（*）。

```
$copy = $original;       # 只拷贝引用
++$copy;                 # 修改下边的共享对象
```

现在，从这开始。假设 $original 是一个对象的引用。要让 ++$copy 只修改 $copy 而不是 $original，先复制一份 $copy 的拷贝，然后把 $copy 赋给一个指向这个新对象的引用。直到 ++$copy 执行之后才执行这个动作，所以在自增之前 $copy 和 $original 是一致的——但是自增后就不一样了。换句话说，是 ++ 识别出拷贝的需要，并且调用你的拷贝构造器。

只有象 ++ 或 +=，或者 nomethod 这样的修改器才能知晓是否需要拷贝，我们稍后描述 nomethod。如果此操作是通过 + 自动生成的，象：

```
$copy = $original;

$copy = $copy +1;
```

这样，那么不会发生拷贝，因为 + 不知道它正被当作修改器使用。

如果在某些修改器的运行中需要拷贝构造器，但是没有给 = 声明句柄，那么只要该对象是一个纯标量而不是什么更神奇的东西，就可以自动生成 = 的句柄。

比如，下面的实际代码序列：

```
$copy  = $original;

...

++$copy;
```

可能最终会变成象下面这样的东西；

```
$copy = $original;

...

$copy = $copy->clone(undef, "");

$copy->incr(undef, "");
```

这里假设 $original 指向一个重载的对象，++ 是用 \&incr 重载的，而 = 是用 \&clone 重载的。

类似的行为也会在 $copy = $original++ 里触发，它解释成 $copy = $original; ++$original.

13.5 当缺失重载句柄的时候（nomethod 和 fallback）

如果你在一个对象上使用一个未重载的操作符，Perl 首先用我们早先描述过的规则，尝试从其他重载的操作符里自动生成一个行为。如果这样失败了，那么 Perl 找一个重载 nomethod 得到的行为，如果可行，则用之。这个句柄之于操作符就好象 AUTOLOAD 子过程之于子过程：它是你走投无路的时候干的事情。

如果用了 nomethod，那么 nomethod 键字应该后面跟着一个引用，该引用指向一个接受四个参数的句柄。前面三个参数和任何其他句柄里的没有区别；第四个是一个字串，对应缺失句柄的那个操作符。它起的作用和 AUTOLOAD 子过程里的 $AUTOLOAD 变量一样。

如果 Perl 不得不找一个 nomethod 句柄，但是却找不到，则抛出一个例外。

如果你想避免发生自动生成，或者你想要一个失效的自动生成，这样你就可以得到一个完全没有重载的结果，那么你可以定义特殊的 fallback 重载键字。它有三个有用的状态：

- undef :

如果没有设置 fallback ，或者你把它明确地设为 undef，那么不会影响重载事件序列：先找一个句柄，然后尝试自动生成，最后调用 nomethod。如果都失败则抛出例外。

- false:

如果把 fallback 设置为一个预定义的，但是为假的值（比如 0），那么永远不会进行自动生成。如果存在 nomethod 句柄，那么 Perl 将调用之，否则则抛出例外。

- true:

和 undef 有几乎一样的性质，但是如果不能通过自动生成合成一个合适的句柄， 那么也不会抛出例外。相反，Perl 回复到该操作符没有重载的性质，就好象根本没有在那个类里面使用 use overload 一样。

13.6 重载常量

你可以用 overload::constant 改变 Perl 解释常量的方法，这个用法放在一个包的 import 方法里很有用。（如果这么做了，你应该在该包的 unimport 方法里正确地调用 overload::remove_constant，这样当你需要的时候，包可以自动清理自身。）

overload::constant 和 overload::remove_constant 都需要一个键字/数值对的列表。这些键字应该是 integer，float，binary，q 和 qr 中的任何东西，而且每个数值都应该是一个子过程的名字，一个匿名子过程，或者一个将操作这些常量的代码引用。

```
sub import { overload::constant ( integer => \&integer_handler,
        float => \&float_handler,
        binary => \&base_handler,
        q => \&string_handler,
        qr => \&regex_handler)}
```

当 Perl 记号查找器碰到一个常量数字的时候，就会调用你提供的任何用于 integer 和 float 的句柄。这个功能是和 use constant 用法独立的；象下面这样的简单语句：

```
$year = cube(12) + 1;    # 整数
$pi = 3.14159265358979;    # 浮点
```

会触发你要求的任何句柄。

binary 键字令你截获二进制，八进制和十六进制常量。q 处理单引号字串（包括用 q 引入的字串）和 qq- 里面的子字串，还有 qx 引起的字串和"此处"文档。最后，qr 处理在正则表达式里的常量片段，就好象在第五章，模式匹配，结尾处描述的那样。

Perl 会给句柄传递三个参数。第一个参数是原来的常量，形式是它原来提供给 Perl 的形式。第二个参数是 Perl 实际上如何解释该常量；比如，123_456 将显示为 123456。

第三个参数只为那些由 q 和 qr 处理字串的句柄定义，而且是 qq，q，s，或者 tr 之一，具体是哪个取决于字串的使用方式。qq 意味着该字串是从一个代换过的环境来的，比如双引号，反斜杠，m// 匹配或者一个 s/// 模式的替换。q 意味着该字串来自一个未代换的 s 意味着此常量是一个在 s/// 替换中来的替换字串，而 tr 意味着它是一个 tr/// 或者 y/// 表达式的元件。

该句柄应该返回一个标量，该标量将用于该常量所处的位置。通常，该标量会是一个重载了的对象的引用，不过没什么东西可以让你做些更卑鄙的事情：

```
package DigitDoubler;       # 一个将放在 DigitDoubler.pm 里的模块
use overload;

sub import { overload::constant ( integer => \&handler,
        float => \&handler ) }
```

```
sub handler {
        my ($orig, $intergp, $context) = @_;
        return $interp * 2;       # 所有常量翻番
}

1;
```

请注意该 handler （句柄）由两个键字共享，在这个例子里运行得不错。现在当你说：

```
use DigitDoubler;

$trouble = 123;       # trouble 现在是 246
$jeopardy = 3.21;    # jeopardy 现在是 6.42
```

你实际上把所有东西都改变了。

如果你截获字串常量，我们建议你同时提供一个连接操作符（"."），因为所有代换过的表达式，象 "ab$cd!!"，仅仅只是一个更长的 'ab'.$cd.'!!' 的缩写。类似的，负数被类似的，负数被认为是正数常量的负值，因此，如果你截获整数或者浮点数，你应该给 neg 提供一个句柄。（我们不需要更早做这些，因为我们是返回真实数字，而不是重载的对象引用。）

请注意 overload::constant 并不传播进 eval 的运行时编译，这一点可以说是臭虫也可以说是特性——看你怎么看了。

13.7 公共重载函数

对 Perl 版本 5.6 而言，use overload 用法为公共使用提供了下面的函数：

1. overload::StrVal(OBJ):
   这个函数返回缺少字串化重载（""）时 OBJ 本应该有的字串值。
2. overload::Overloaded(OBJ):
   如果 OBJ 经受了任何操作符重载，此函数返回真，否则返回假。

3. overload::Method(OBJ,OPERATOR):

这个函数返回 OPERATOR 操作 OBJ 的时候重载 OPERATOR 的代码，或者 在不存在重载时返回 undef。

13.8 继承和重载

继承和重载以两种方式相互交互。第一种发生在当一个句柄命名为一个字串而不是以一个代码引用或者匿名子过程的方式提供的时候。如果命名为字串，那么该句柄被解释成为一种方法，并且因此可以从父类中继承。

第二种重载和继承的交互是任何从一个重载的类中衍生出来的类本身也经受该重载。换句话说，重载本身是继承的。一个类里面的句柄集是该类的所有父类的句柄的联合（递归地）。如果在几个不同的祖先里都有某个句柄，该句柄的使用是由方法继承的通用规则控制的。比如，如果类 Alpha 按照类 Beta，类 Gamma 的顺序继承，并且类 Beta 用 \&Beta::plus_sub 重载 +，而 Gamma 用字串 "plus_meth" 重载 +，那么如果你对一个 Alpha 对象用 + 的时候，将调用 Beta::plus。

因为键字 fallback 的值不是一个句柄，它的继承不是由上面给出的规则控制的。在当前的实现里，使用来自第一个重载祖先的 fallback 值，不过这么做是无意的，并且我们可能会在不加说明的情况下改变（当然，是不加太多说明）。

13.9 运行时重载

因为 use 语句是在编译时重载的，在运行时改变重载的唯一方法是：

```
eval "use overload '+' =>\&my_add";
```

你还可以说：

```
eval "no voerload '+', '--', '<=' ";
```

当然，在运行时使用这些构造器是很有问题的。

13.10 重载诊断

如果你的 Perl 是带着 -DDEBUGGING 编译的，你就可以用 -Do 开关或者等价物在运行时查看重载的诊断信息。你还可以用 Perl 内置的调试器的 m 命令推导是重载了哪个操作。

如果你现在觉得"重载"了，下一章可能会把这些东西给你约束回来。

第十三章 捆绑（tie）变量上

有些人类的工作需要伪装起来。有时候伪装的目的是欺骗，但更多的时候，伪装的目的是为了在更深层次做一些真实的通讯。比如，许多面试官希望你能穿西服打领带以表示你对工作是认真的，即使你们俩都知道你可能在工作的时候永远不会打领带。你思考这件事的时候可能会觉得很奇怪：在你脖子上系一块布会神奇地帮你找到工作。在 Perl 文化里， tie 操作符起到类似的作用的角色：它让你创建一个看起来象普通变量的变量，但是在变量的伪装后面，它实际上是一个羽翼丰满的 Perl 对象，而且此对象有着自己有趣的个性。它只是一个让人有点奇怪的小魔术，就好象从一个帽子里弹出一个邦尼兔那样。（译注：英文 tie 做动词有"捆绑"之意，而做名词有"领带"之意．）用另外一个方法来看，在变量名前面的趣味字符 $，@，%，或者 * 告诉 Perl 和它的程序许多事情——他们每个都暗示了一个特殊范畴的原形特性。你可以利用 tie 用各种方法封装那些特性，方法是用一个实现一套新性质的类与该变量关联起来。比如，你可以创建一个普通的 Perl 散列，然后把它 tie（绑）到到一个类上，这个类把这个散列放到一个数据库里，所以当你从散列读取数据的时候，Perl 魔术般地从一个外部数据库文件抓取数据，而当你设置散列中的数值的时候，Perl 又神奇地把数据存储到外部数据库文件里。这里的"魔术"，"神奇"指的是"透明地处理一些非常复杂的任务"。你应该听过那些老话：再先进的技术也和 Perl 脚本没什么区别。（严肃地说，那些在 Perl 内部工作的人们把魔术（magic）一词当作一个技术术语，特指任何附加在变量上的额外的语义，比如 %ENV 或者 %SIG。捆绑变量只是其中一种扩展。）

Perl 已经有内建的 dbmopen 和 dbmclose 函数，它们可以完成把散列变量和数据库系在一起的魔术，不过那些函数的实现是早在 Perl 没有 tie 的时候。现在 tie 提供了更通用的机制。实际上， Perl 本身就是以 tie 的机制来实现 dbmopen 和 dbmclose 的。

你可以把一个标量，数组，散列或者文件句柄（通过它的类型团）系到任意一个类上，这个类提供合适的命名方法以截获和模拟对这些对象的正常访问。那些方法的第一个是在进行 tie 动作本身时调用的：使用一个变量总是调用一个构造器，如果这个构造器成功运行，则返回一个对象，而 Perl 把这个对象藏在一个你看不见的地方——在"普通"变量的深层内部。你总是可以稍后用 tied 函数在该普通变量上检索该对象：

```
tie VARIABLE, CLASSNAME, LIST;   # 把 VARIABLE 绑定到 CLASSNAME
```

```
$object = tied VARIABLE;
```

上面两行等效于：

```
$object = tie VARIABLE, CLASSNAME, LIST;
```

一旦该变量被捆绑，你就可以按照平时那样对待该普通变量，不过每次访问都自动调用下层对象的方法；所有该类的复杂性都隐藏在那些方法调用的背后。如果稍后你想打破变量和类之间的关联，你可以 untie （松绑）那个变量：

```
untie VARIABLE;
```

你几乎完全可以把 tie 看作一种有趣的 bless 类型，只不过它是给一个光秃秃的变量赐福，而不是给一个对象引用赐福。它同样还可以接收额外的参数，就象构造器那样——这个恐怕也不新鲜了，因为它实际上就是在内部调用一个构造器，该构造器的名字取决于你尝试的变量类型：是 TIESCALAR，TIEARRAY，TIEHASH，或者 TIEHANDLE。（注：因为这些构造器是独立的名字，你甚至可以提供一个独立的类来实现它们。那样，你就可以把标量，数组，散列，和文件句柄统统绑定到同一个类上，不过通常不是这么干的，因为它会令其他的魔术方法比较难写。）调用这些构造器的时候，它们用所声明的 CLASSNAME 为它们的调用者作为类方法调用，另外把你放在 LIST 里的任何东西作为附加的参数。（VARIABLE 并不传递给构造器。）

这四种构造器每种都返回一个普通风格的对象。它们并不在乎它们是否从 tie 里调用的，类里的其他方法也不在意，因为如果你喜欢的话你总是可以直接调用它们。从某种意义来说，所有魔术都是在 tie 里，而不是在实现 tie 的类里。该类只是一个有着有趣的方法名的普通类。（实际上，有些捆绑的模块提供了额外的一些方法，这些方法是不能通过捆绑的变量看到的；你必须明确调用这些方法，就象你对待其他对象方法一样。这样的额外方法可以提供类似文件锁，事务保护，或者任何其他实例方法可以做的东西。）

因此这些构造器就象其他构造器那样 bless（赐福）并且返回一个对象引用。该引用不需要指向和被捆绑的变量相同类型的变量；它只是必须被赐福，所以该绑定的变量可以很容易在你的类中找到支持。比如，我们的长例子 TIEARRAY 就会用一个基于散列的对象，这样它就可以比较容易地保存它在模拟的数组的附加信息。

tie 函数不会为你 use 或者 require 一个模块——如果必要的话，在调用 tie前你必须自己明确地做那件事。（另外，为了保持向下兼容，dbmopen 函数会尝试 use 一个或者某个 DBM 实现。但你可以用一个明确的 use 修改它的选择优先级——只要你 use 的模块是在 dbmopen 的模块列表中的一个。参阅 AnyDBM[2]_File 模块的在线文档获取更完善的解释。）

一个捆绑了的变量调用的方法有一个类似 FETCH 和 STORE 这样的预定义好了的名字，因为它们是在 Perl 内部隐含调用的（也就是说，由特定事件触发）。这些名字都在 ALLCAPS，内部隐含调用的（也就是说，有特定事件触发）。这些名字都是全部大写，这是我们遵循的一个称呼这些隐含调用过程的习惯。（其他遵循这种传统的习惯有 BEGIN，CHECK，INIT， END，DESTROY， 和 AUTOLOAD，更不用说 UNIVERSAL->VERSION。实际上，几乎所有 Perl Perl 预定义的变量和文件句柄都是大写的：STDIN，SUUPER，CORE，CORE::GLOBAL，DATA， @EXPORT，@INC，@ISA，@ARGV 和 %ENV。当然，内建操作符和用法是另外一个极端，它们没有用大写的。）

我们首先要介绍的内容很简单：如何捆绑一个标量变量。

14.1 捆绑标量

要实现一个捆绑的标量，一个类必须定义下面的方法：TIESCALAR，FETCH，和 STORE（以及可能还有 DESTROY）。当你 tie 一个标量变量的时候，Perl 调用 TIESCALAR。如果你读取这个捆绑的变量，它调用 FETCH，并且当你给一个变量赋值的时候，它调用 STORE。如果你保存了最初的 tie （或者你稍后用 tied 检索它们），你就可以自己访问下层的对象——这样并不触发它的 FETCH 或者 STORE 方法。作为一个对象，这一点毫不神奇，而是相当客观的。

如果存在一个 DESTROY 方法，那么当指向被捆绑对象的最后一个引用消失时，Perl 就会调用这个 DESTROY 方法，就好象对其他对象那样。当你的程序结束或者你调用 untie 的时候就会发生这些事情，这样就删除了捆绑使用的引用。不过，untie 并不删除任何你放在其他地方的引用；DESTROY 也会推迟到那些引用都删除以后。

在标准的 Tie::Scalar 模块里有 Tie::Scalar 和 Tie::StdScalar 包，如果你不想自己定义所有这些方法，那么它们定义了一些简单的基类。Tie::Scalar 提供了只能做很有限的工作的基本方法，而 Tie::StdScalar 提供了一些方法令一个捆绑了的标量表现得象一个普通的 Perl 标量。（好象没什么用，不过有时候你只是想简单地给普通标量语义上加一些封装，比如，计算某个变量设置的次数。）

在我们给你显示我们精心设计的例子和对所有机制进行完整地描述之前，我们先给你来点开胃的东西——并且给你显示一下这些东西是多简单。下面是一个完整的程序：

```
#! /usr/bin/perl

package Centsible;

sub TIESCALAR { bless \my $self, shift }
```

```
sub STORE { ${ $_[0] } = $_[1] }       # 做缺省的事情
sub  FETCH { sprintf "%0.2f", ${ my $self = shift } }     # 圆整值


package main;
tie $bucks, "Centsible";
$bucks = 45.00;
$bucks *= 1.0715;    # 税
$bucks *= 1.0715;    # 和双倍的税!
print "That will be $bucks, please.\n";
```

运行的时候，这个程序生成：

```
That will be 51.67, please.
```

把 tie 调用注释掉以后，你可以看到区别：

```
That will be 51.66505125, please.
```

当然，这样要比你平时做圆整所做的工作要多。

14.1.1 标量捆绑方法

既然你已经看到我们将要讲的东西，那就让我们开发一个更灵活的标量捆绑类吧。我们不会使用任何封装好了的包做基类（特别是因为标量实在是简单），相反我们会轮流照看一下这四种方法，构造一个名字为 ScalarFile[2] 的例子类。捆绑到这个类上的标量包含普通字串，并且每个这样的变量都隐含地和一个文件关联，此文件就是字串存贮的地方。（你可以通过给变量命名的方法来记忆你引用的是哪个文件。）变量用下面的方法绑到类上：

```
use ScalarFile;      # 装载 ScalarFile.pm
tie $camel, "ScalarFiel", "/tmp/camel.lot";
```

变量一旦捆绑，它以前的内容就被取代，并且变量和其对象内部的联系覆盖了此变量平常的语义。当你请求 $camel 的值时，它现在读取 /tmp/camel.lot 的内容，而当你给 $camel 赋值的时候，它把新的内容写到 /tmp/camel.lot 里，删除任何原来的东西。

捆绑是对变量进行的，而不是数值，因此一个变量的捆绑属性不会随着赋值一起传递。比如，假设你拷贝一个已经捆绑了的变量：

```
$dromedary = $camel;
```

Perl 不是象平常那样从 $camel 标量里读取变量，而是在相关的下层对象上调用 FETCH 方法。就好象你写的是这样的东西：

```
$dromedary = (tied $camel)->FETCH();
```

或者如果你还记得 tie 返回的对象，你可以直接使用那个引用，就象在下面的例子代码里一样：

```
$clot = tie $camel, "ScalarFile", "/tmp/camle.lot";

$dromedary = $camle;       # 通过隐含的接口

$dromedary = $clot->FETCH();   # 一样的东西，不过是明确的方法而已
```

如果除了 TIESCALAR，FETCH，STORE，和 DESTROY 以外，该类还提供其他方法，你也可以使用 $clot 手工调用它们。不过，大家应该做好自己的事情而不要去管下层对象，这也是为什么你看到来自 tie 的返回值常被忽略。如果稍后你又需要该对象（比如，如果该类碰巧记载了任何你需要的额外方法的文档），那么你仍然可以通过 tie 获取该对象。忽略所返回的对象同样也消除了某些类型的错误，这一点我们稍后介绍。

下面是我们的类所需要的东西，我们将把它们放到 ScalarFile[2].pm：

```
package ScalarFile;

use Carp;       # 很好地传播错误消息。

use strict;      # 给我们自己制定一些纪律。

use warnings;      # 打开词法范围警告。

use warnings::register;   # 允许拥护说"use warnings 'ScalarFile'"。

my $count = 0;       # 捆绑了的 ScalarFile[2] 的内部计数。
```

这个标准的 Carp 模块输出 carp，croak，和 confess 子过程，我们将在本节稍后的代码中使用它们。和往常一样，参阅第 32 章，标准模块，或者在线文档获取 Carp 的更多介绍。

下面的方法是该类定义的。

CLASSNAME->TIESCALAR(LIST)

每当你 tie 一个标量变量，都会触发该类的 TIESCALAR 方法。可选的 LIST 包含 任意正确初始化该对象所需要的参数。（在我们的例子里，只有一个参数： 该文件的名字。）这个方法应该返回一个对象，不过这个对象不必是一个标量的 引用。不过在我们的例子里是标量的引用。

```
sub TIESCALAR {      # 在 ScalarFile.pm
        my $class = shift;
        my $filename = shift;
        $count++;   # 一个文件范围的词法，是类的私有部分
        return bless \$filename, $class;
}
```

因为匿名数组和散列构造器（[]和{}）没有标量等价物，我们只是赐福一个词法 范围的引用物，这样，只要这个名字超出范围，它就变成一个匿名。这样做运转得 很好（你可以对数组和散列干一样的事情）——只要这个变量真的是词法范围。 如果你在一个全局量上使用这个技巧，你可能会以为你成功处理了这个全局量， 但直到你创建另外一个 camel.lot 的时候才能意识到这是错的。不要试图写下面 这样的东西：

```
sub TIESCALAR {bless \$_[1], $_[0] }   # 错，可以引用全局量。
```

一个写得更强壮一些的构造器可能会检查该文件名是否可联合。我们首先检查， 看看这个文件是否可读，因为我们不想毁坏现有的数值。（换句话说，我们不应该 假设用户准备先写。他们可能很珍惜此程序以前运行留下来的旧的 Camel Lot 文件。）如果我们不能打开或者创建所声明的文件名，我们将通过返回一个 undef 礼貌地指明该错误，并且还可以通过 carp 打印一个警告。（我们还可以只用 croak —— 这是口味的问题，取决于你喜欢鱼还是牛蛙。）我们将使用 warnings 用法来判断这个用户是否对我们的警告感兴趣：

```
sub TIESCALAR {   # 在 ScalarFile.pm
        my $class = shift;
        my $filename = shift;
        my $fh;
        if (open $fh, "<", $filename or
```

```
        open $fh, ">", $filename)
        {
                close $fh;
                $count++;
                return bless \$filename, $class;
        }
        carp "Can't tie $filename: $!" if warnings::enabled();
        return;
}
```

有了这样的构造器，我们现在就可以把标量 $string 和文件 camel.lot 关联在 一起了：

```
tie ($string, "ScalarFile", "camel.lot") or die;
```

（我们仍然做了一些不应该做的假设。在一个正式版本里，我们可能打开该文件 句柄一次并且在捆绑的过程中记住该文件句柄和文件名，保证此句柄在所有时间 里都是用 flock 排他锁住的。否则我们就会面对冲突条件——参阅第二十三章， 安全，里的"处理计时缝隙"。）

- SELF->FETCH

当你访问这个捆绑变量的时候就会调用这个方法（也就是说，读取其值）． 除了与变量捆绑的对象以外，它没有其他的参数．在我们的例子里，那个对象 包含文件名．

```
sub     FETCH {
        my $self = shift;
        confess "I am not a class method" unless ref $self;
        return unless open my $fh, $$self;
        read($fh, my $value, -s $fh);    # NB: 不要在管道上使用 -s
        return $value;
}
```

这回我们决定：如果 FETCH 拿到了不是引用的东西，那么就摧毁（抛出一个 例外）.（它要么是被当做类方法调用，要么是什么东西不小心把它当成 一个子过程调用了.）我们没有其它返回错误的方法，所以这么做可能是 对的. 实际上，只要我们试图析引用 $self，Perl 都会立刻抛出一个例外； 我们只是做得礼貌一些并且用 confess 把完整的堆栈追踪输出到用户的屏幕 上.（如果这个动作可以认为是礼貌的话.）

如果我们说下面这些话就会看到 camel.lot 的内容：

```
tie($string, "ScalarFile", "camel.lot");

print $string;
```

- SELF->STORE(VALUE)

当设置（赋值）捆绑的变量的时候会运行这个方法. 第一个参数 SELF 与往常 一样是与变量关联的对象；VALUE 是给变量赋的值.（我们这里的"赋值"的 含义比较宽松--任何修改变量的动作都可以叫做 STORE.）

```
sub STORE {

    my($self, $value) = @_;

    ref $self      or confess "not a class method";

    open my $fh, ">", $$self or croak "can't clobber $$self:$!";

    syswrite($fh, $value) == length $value

    or croak "can't write to $$self: $!";

    close $fh      or croak "can't close $$self:$!";

    return $value;

}
```

在给它"赋值"以后，我们返回新值--因为这也是赋值做的事情. 如果赋值失败， 我们把错误 croak 出来. 可能的原因是我们没有写该关联文件的权限，或者 磁盘满，或者磁盘控制器坏了. 有时候是你控制这些局势，有时候是局势控制你.

现在如果我们说下面的话，我们就可以写入 camel.lot 了.

```
tie($string, "ScalarFile", "camel.lot");
```

```
$string = "Here is the first line of camel.lot\n";
$string .= "And here is another line, automatically appended.\n";
```

- SELF->DESTORY

当与捆绑变量相关联的对象即将收集为垃圾时会触发这个方法，尤其是在做一些 特殊处理以清理自身的情况下。和其他类一样，这样的方法很少是必须的，因为 Perl 自动为你清除垂死的对象的内存。在这里，我们会定义一个 DESTROY 方法 用来递减我们的捆绑文件的计数：

```
sub DESTROY {
    my $self = shift;
    confess "wrong type" unless ref $self;
    $count--;
}
```

我们还可以提供一个额外的类方法用于检索当前的计数。实际上，把它称做类方法 还是对象方法并不要紧，但是你在 DESTROY 之后就不再拥有一个对象了，对吧？

```
sub count {
    # my $invocant = shift;
    $count;
}
```

你可以在任何时候把下面这个称做一个类方法：

```
if (ScalarFile->count) {
    warn "Still some tied ScalarFiles sitting around somewhere...\n";
}
```

这就是所要的东西。实际上，比所要的东西还要多，因为我们在这里为完整性， 坚固性和普遍美学做了几件相当漂亮的事情。当然，更简单的 TIESCALAR 类 也是可能的。

14.1.2 魔术计数变量

这里是一个简单的 Tie::Counter 类，灵感来自 CPAN 中同名的模块。捆绑到这个类上的变量每次自增 1。比如：

```
tie my $counter, "Tie::Counter", 100;

@array = qw /Red Green Blue/;

for my $color (@array) {      # 打印：

        print " $counter $color\n";   # 100 Red

}               # 101 Green

# 102 Blue
```

构造器把一个可选的额外参数当作计数器的初始值，缺省时为零。给这个计数器赋值将设置一个新值。下面是类：

```
package Tie::Counter;

sub FETCH    { ++ ${ $_[0] }}

sub STORE    { ${ $_[0] } = $_[1] }

sub TIESCALAR {

        my ($class, $value) = @_;

        $value = 0 unless defined $value;

        bless \$value => $class;

}

1;    # 如果在模块里需要这个
```

多小！看到了吗？要写这么一个类并不需要多少代码。

14.1.3 神奇地消除 $_

这个让人好奇的外部的捆绑类用于防止非局部的 $_ 的使用。它不是用 use 把方法拉进来，而是调用该类的 import 方法，装载这个模块应该用 no，以便调用很少用的 unimport 方法。用户说：

```
   no Underscore;
```

然后所有把 $_ 当作一个非局部的全局变量使用就都会产生一个例外。

下面是一个用这个模块的小测试程序：

```
#!/usr/bin/perl

no Underscore;

@tests = (

"Assignment"  => sub { $_ = "Bad" },

"Reading"     => sub { print },

"Matching"    => sub { $x = /badness/ },

"Chop"        => sub { chop },

"Filetest"    => sub { -x },

"Nesting"     => sub { for (1..3) { print } },

);


while ( ($name, $code) = splice(@tests, 0, 2) ) {

        print "Testing $name: ";

        eval { &$code };

        print $@ ? "detected" : " missed!";

        print "\n";

}
```

这个程序打印出下面的东西：

```
Testing Assignment: detected

Testing Reading: detected

Testing Matching: detected

Testing Chop: detected
```

```
Testing Filetest: detected

Testing Nesting: 123 missed!
```

“丢失”了最后一个是因为它由 for 循环正确地局部化了，并且因此可以安全地访问。

下面是让人感兴趣的外部 Underscore 模块本身。（我们说过它是让人感兴趣的外部吗？）它能运转是因为捆绑的神奇变量被一个 local 有效地隐藏起来了。该模块在它自己的初始化代码里做了一个 tie ，所以 require 也能运转。

```
package Underscore;

use Carp;

sub TIESCALAR { bless \my $dummy => shift }

sub FETCH { croak 'Read access to $_ forbidden' }

sub STORE { croak 'Write access to $_ forbidden' }

sub unimport { tie($_, __PACKAGE__) }

sub import   { untie $_ }

tie($_, __PACKAGE__) unless tied $_;

1;
```

在你的程序里对这个类混合调用 use 和 no 几乎没有任何用处，因为它们都在编译时发生，而不是运行时。你可以直接调用 Underscore->import 和 Underscore->unimport，就象 use 和 no 那样。通常，如果你想反悔并且让自己可以使用 $_，你就要对它使用 local，这也是所有要点所在。

14.2 捆绑数组

一个实现捆绑数组的类至少要定义方法 TIEARRAY，FETCH，和 STORE。此外还有许多可选方法：普遍存在的 DESTROY 方法，还有用于提供 $#array 和 scalar(@array) 访问的 STORESIZE 和 FETCHSIZE 方法。另外，当 Perl 需要清空该数组时会触发 CLEAR ，而当 Perl 在一个真正的数组上需要预先扩充（空间）分配的时候需要 EXTEND。

如果你想让相应的函数在捆绑的数组上也能够运行，你还可以定义 POP，PUSH，SHIFT，UNSHIFT，SPLICE，DELETE，和 EXISTS 方法。Tie::Array 类可以作为一个基类，用于利用 FETCH 和 STORE 实现前五个函数 。（Tie::Array 的 DELETE 和 EXISTS

的缺省实现只是简单地调用 croak。）只要你定义了 FETCH 和 STORE，那么你的对象包含什么样的数据结构就无所谓了。

另外， Tie::StdArray 类（在标准的 Tie::Array 模块中定义）提供了一个基类，这个基类的缺省方法假设此对象包含一个正常的数组。下面是一个利用这个类的简单的数组捆绑类。因为它使用了 Tie::StdArray 做它的基类，所以它只需要定义那些应该以非标准方法对待的方法。

```
#! /usr/bin/perl

package ClockArray;

use Tie::Array;

our @ISA = 'Tie::StdArray';

sub FETCH {

        my ($self, $place) = @_;

        $self->[$place % 12 ];

}


sub STORE {

        my($self, $place, $value ) = @_;

        $self->[$place % 12 ] = $value;

}


package main;

tie my @array, 'ClockArray';

@array = ( "a" ... "z" );

print "@array\n";
```

运行的时候，这个程序打印出 "y z o p q r s t u v w x"。这个类提供一个只有一打位置的数组，类似一个时钟的小时数，编号为 0 到 11。如果你请求第十五个元素，你实际上获得第三个。把它想象成一个旅游助手，用于帮助那些还没有学会如何读 24 小时时钟的人。

14.2.1 数组捆绑方法

前面的是简单的方法。现在让我们看看真正的细节。为了做演示，我们将实现一个数组，这个数组的范围在创建的时候是固定的。如果你试图访问任何超出该界限的东西，则抛出一个例外。比如：

```
use BoundedArray;

tie @array, "BoundedArray", 2;


$array[0] = "fine";

$array[1] = "good";

$array[2] = "great";

$array[3] = "whoa";    # 禁止，显示一个错误信息。
```

这个类的预定义的代码如下：

```
   package BoundedArray;

   use Carp;

   use strict;
```

为了避免稍后定义 SPLICE，我们将从 Tie::Array 类中继承：

```
   use Tie::Array;

   our @ISA = ("Tie::Array");
```

CLASSNAME->TIEARRAY(LIST)

是该类的构造器，TIEARRAY 应该返回一个赐福了的引用，通过该引用模拟这个 捆绑了的数组。

在下一个例子里，为了告诉你并非一定要返回一个数组的引用，我们选择了一个 散列引用来代表我们的对象。散列很适合做通用记录类型：散列的"BOUND"键字 将存储最大允许的范围，而其"DATA"值将保存实际的数据。如果有个类以外的 解引用返回的对象（就是一个数组引用，不用怀疑），则抛出一个例外。

```
sub TIEARRAY {
        my $class = shift;
        my $bound = shift;
        confess "usage: tie( \@ary, 'BoundedArray', max_subscript)"
        if @_ || $bound =~ /\D/;
        return bless { BOUND => $bouind, DATA => [] }, $class;
}
```

现在我们可以说：

```
tie( @array, "BoundedArray", 3);   # 允许的最大索引是 3
```

以确保该数组永远不会有多于四个元素。当对数组的一个独立的元素进行访问或者 存储的时候，将调用 FETCH 和 STORE，就好象是处理标量一样，不过有一个 额外的索引参数。

SELF->FETCH(INDEX)

当访问捆绑的数组里的一个独立的元素的时候调用这个方法。它接受对象后面的 一个参数：我们试图抓取的数值的索引。

```
sub FETCH {
        my ($self, $index ) = @_;
        if ($index > $self->{BOUND} ) {
                confess "Array OOB: $index > $self->{BOUND}";
        }
        return $self->{DATA}[$index];
}
```

SELF->STORE(INDEX, VALUE)

当设置捆绑了的数组里的一个元素的时候调用这个方法。它接受对象后面的两个 参数：我们试图存储的东西的索引和我们准备放在那里的数值。比如：

```
sub STORE {
        my($self, $index, $value) = @_;
        if ($index > $self->{BOUND} ) {
                confess "Array OOB: $index > $self->{BOUND}";
        }
        return $self->{DATA}[$index] = $value;
}
```

SELF->DESTROY

当需要删除捆绑变量和回收它的内存的时候调用这个方法。对于一门有垃圾回收 功能的语言来说，这个东西几乎用不上，所以本例中我们忽略它。

SELF->FETCHSIZE

FETCHSIZE 方法应该返回与 SELF 关联的捆绑数组的条目的总数。它等效于 scalar(@array)，通常等于 $#array + 1。

```
sub FETCHSIZE {
        my $self  = shift;
        return scalar @{$self->{DATA}};
}
```

SELF->STORESIZE(COUNT)

这个方法把与 SELF 关联的捆绑数组的条目总数设置为 COUNT。如果此数组收缩， 那么你应该删除超出 COUNT 范围的记录。如果数组增长，你应该确保新的空位置 是未定义的。对于我们的 BoundedArray[2] 类，我们还要确保该数组不会增长得超出 初始化时设置的限制。

```
sub STORESIZE {
```

```
        my ($self, $count ) = @_;
        if ($count > $self->{BOUND} ) {
                confess "Array OOB: $count > $self->{BOUND}";
        }
        $#{$self->{DATA}} = $count;
}
```

SELF->EXTEND(COUNT)

Perl 使用 EXTEND 方法来表示一个数组将要扩展成保存 COUNT 条记录。这样你就 可以一次分配足够大的内存，而不是以后的多次调用。因为我们的 BoundedArrays[2] 已经固定了上限，所以我们不用定义这个方法。

SELF->EXISTS(INDEX)

这个方法验证在 INDEX 位置的元素存在于捆绑数组中。对于我们的 BoundedArray[2]，我们只需要在核实了查找企图没有超过我们的固定上限，然后就 可以使用 Perl内建的 exists 来核实。

```
sub EXISTS {
        my ($self, $index) = @_;
        if ( $index > $self->{BOUND}) {
                confess "array OOB: $index > $self->{BOUND}";
        }
        exists $self->{DATA}[$index];
}
```

SELF->DELETE(INDEX)

DELETE 方法从捆绑数组 SELF 中删除在 INDEX 位置的元素。对于我们的 BoundedArray[2] 类，这个方法看起来几乎和 EXISTS 完全一样，不过这可不是 标准。

```
sub DELETE {
        my ($self, $index) = @_;
```

```
        print STDERR "deleting!\n";
        if ($index > $self->{BOUND} ) {

                confess "Array OOB: $index > $self->{BOUND}";
        }
        delete $self->{DATA}[$index];
}
```

SELF->CLEAR

当这个数组需要清空的时候调用这个方法。当该数组设置为一列新值（或者一列 空值）的时候发生这个动作，不过不会在提供给 undef 函数的时候发生这个 动作。因为一个清空了的 boundedArray 总是满足上限，所以我们在这里不需要 检查任何东西：

```
sub CLEAR {
        my $self = shift;

        $self->{DATA} = [];
}
```

如果你把数组设置为一个列表，这个动作会触发 CLEAR，但是看不到列表数值。 因此如果你象下面这样违反上界：

```
tie(@array, "BoundedArray", 2);
@array = (1,2,3,4);
```

CLEAR 方法仍将返回成功。而只有在随后的 STORE 中才会产生例外。这样的赋值 触发一个 CLEAR 和四个 STORES。

SELF->PUSH(LIST)

这个方法把 LIST 的元素附加到数组上。下面是它在我们 BoundedArray[2] 类里的 运行方法：

```
sub PUSH {
```

```
        my $self = shift;
        if (@_ + $#{$self->{DATA}} > $self->{BOUND} ) {
                confess "Attempt to push too many elements";
        }
        push @{$self->{DATA}}, @_;
}
```

SELF->UNSHIFT(LIST)

这个方法预先把 LIST 的元素放到数组中。对于我们的 BoundedArray[2] 类而言， 这个子过程类似 PUSH。

SELF->POP

POP 方法从数组中删除最后一个元素并返回之。对于 boundedArray，它只有 一行：sub POP { my $self = shift; pop @{$self->{DATA}}}

SELF->SHIFT

SHIFT 方法删除数组的第一个元素并返回之。对于 boundedArray，它类似 POP。

SELF->SPLICE(OFFSET, LENGTH, LIST)

这个方法让你接合 SELF 数组。为了模拟 Perl 内建的 splice，OFFSET 应该是 可选项并且缺省为零，而负值是从数组后面向前数。LENGTH 也应该是可选项， 缺省为数组剩下的长度。LIST 可以为空。如果正确模拟内建函数，那么它应该 返回原数组从 OFFSET 开始的 LENGTH 长个元素（要被 LIST 代替的元素列表）。

因为接合是有些复杂的操作，我们就不定义它了；我们只需要使用来自 Tie::Array 模块的 SPLICE 子过程，这个子过程是继承 Tie::Array 时免费 时免费拿到的。这样我们用其他 BoundedArray[2] 方法定义了SPLICE，因此范围检查 仍然进行。

上面就是 BoundedArray[2] 类的全部。它只是对数组的语义做了一点点封装。不过我们可以干得更好，而且用的空间更少。

14.2.2 大家方便

变量的一个好特性是它可以代换。函数的一样不太好的特点是它不能代换。你可以使用捆绑的数组把一个函数做成可以代换。假设你想代换一个字串里的任意整数。你可以就说：

```
#! /usr/bin/perl

package RandInterp;

sub TIEARRAY { bless   \my $self};

sub FETCH { int rand $_[1] };


package main;

tie @rand, "RandInterp";


for  (1,10,100,1000) {

        print "A random integer less then $_ would be $rand[$_]\n";

}

$rand[32] = 5;   # 这么干会重新格式化我们的系统表了么？
```

当运行的时候，它打印出下面的内容：

```
A random integer less than 1 would be 0

A random integer less than 10 would be 3

A random integer less than 100 would be 46

A random integer less than 1000 would be 755

Can't locate object method "STORE" via package "RandInterp" at foo line 10.
```

如你所见，我们甚至还没能实现 STORE，不过这不算什么。我们只是和往常一样把它去掉了。

第十四章 捆绑（tie）变量下

14.3 捆绑散列

一个实现捆绑散列的类应该定义八个方法。TIEHASH 构造一个新对象。FETCH 和 STORE 访问键字/数值对。EXISTS 报告某键字是否存在于散列中，而 DELETE 删除一个键字和它关联的数值（注：请注意在 Perl 里，散列里不存在一个键字与存在一个键字但是其对应数值为 undef 是不同的两种情况。这两种情况可以分别用 exists 和 defined 测试。） CLEAR 通过删除所有键字/数值对清空散列。FIRSTKEY 和 NEXTKEY 在你调用 keys， values，或 each 的时候逐一遍历键字/数值对。还有就是和往常一样，如果你想在对象删除的时候执行某种特定的动作，那么你可能还要定义 DESTORY 方法。（如果你觉得方法太多，那么你肯定没有认真阅读上一节关于数组的内容。不管怎样，你都可以自由地从标准的 Tie::Hash 模块继承缺省的方法，只用重新定义你感兴趣的方法。同样，Tie::StdHash 假设此实现同样也是一个散列。）

比如，假定你想创建这么一个散列：每次你给一个键字赋值的时候，它不是覆盖掉原来的内容，而是把新数值附加到一个数值数组上。这样当你说：

```
$h{$k} = "one";

$h{$k} = "two";
```

它实际上干的是：

```
push @{ $h{$k} }, "one";

push @{ $h{$k} }, "two";
```

这个玩叶儿不算什么复杂的主意，所以你应该能用一个挺简单的模块实现。把 Tie::StdHash 用做基类，下面就是干这事的 Tie::AppendHash：

```
package Tie::AppendHash;

use Tie::Hash;

our @ISA = ("Tie::StdHash");

sub STORE {
```

```
        my ($self, $key, $value) = @_;
        push @{$self->{key}}, $value;
}
1;
```

14.3.1 散列捆绑方法

这儿是一个很有趣的捆绑散列类的例子：它给你一个散列，这个散列代表用户特定的点文件（也就是说，文件名开头是一个句点的文件，这样的文件是 Unix 初始化文件的命名传统。）你用文件名做索引（除去开头的句点）把文件名放进散列，而拿出来的时候是点文件的内容。比如：

```
use DotFiles;
tie %dot, "DotFiles";
if ( $dot{profile}  =~ /MANPATH/ or
$dot{login}    =~ /MANPATH/ or
$dot{cshrc}  =~ /MANPATH/ ) {
        print "you seem to set your MANPATH\n";
}
```

下面是使用我们捆绑类的另外一个方法：

```
# 第三个参数是用户名，我们准备把他的点文件捆绑上去。
tie %him,  "DotFiels", "daemon";
foreach $f (keys %him) {
        printf "daemon dot file %s is size %d\n", $f, length $him{$f};
}
```

在我们的 DotFiles[2] 例子里，我们把这个对象当作一个包含几个重要数据域的普通散列来实现，这几个数据域里只有 {CONTENTS} 域会保存一般用户当作散列的东西。下面是此对象的实际数据域：

数据域	内容
USER	这个对象代表谁的点文件
HOME	那些点文件在哪里
CLOBBER	我们是否允许修改或者删除这些点文件
CONTENTS	点文件名字和内容映射的散列

下面是 DotFiles[2].pm 的开头：

```
package DotFiles;

use Carp;

sub whowasi   { (caller(1))[3]. "()" }

my $DEBUG = 0;

sub debug { $DEBUGA = @_ ? shift: 1}
```

对于我们的例子而言，我们希望打开调试输出以便于在开发中的跟踪，因此我们为此设置了$DEBUG。我们还写了一个便利函数放在内部以便于打印警告：whowasi 返回调用了当前函数的那个函数的名字（whowasi 是"祖父辈"的函数）。

下面就是 DotFiles[2] 捆绑散列的方法：

CLASSNAME->TIEHASH(LIST)

这里是 DotFiles[2] 构造器：

```
sub TIEHASH {
        my $self = shift;
        my $user = shift || $>;
        my $dotdir = shift || "";

        croak "usage: @{[ &whowasi ] }  [ USER [DOTDIR]]" if @_;

        $user = getpwuid($user) if $user =~ /^\d+$/;
```

```
        my $dir = (getpwname($user))[7]
        or corak "@{ [&whowasi] }: no user $user";
        $dir  .= "/$dotdir" if $dotdir;

        my $node = {
                USER => $user,
                HOME => $dir,
                CONTENTS => {},
                CLOBBER => 0,
        };

        opendir DIR, $dir
        or croak "@{[&whowasi]}: can't opendir $dir: $!";
        for my $dot ( grep /^\./ && -f "$dir/$_", readdir(DIR)) {
                $dot =~ s/^\.//;
                $node->{CONTENTS}{$dot} = undef;
        }
        closedir DIR;

        return bless $node, $self;
}
```

值得一提的是，如果你准备用文件来测试上面的 readdir 的返回值，你最好预先 准备好有问题的目录（象我们一样）。否则，因为我们没有用 chdir，所以你很 有可能测试的是错误的文件。

SELF->FETCH(KEY)

这个方法实现的是从这个捆绑的散列里读取元素。它在对象后面还有一个参数： 你想抓取的散列元素的键字。这个键字是一个字串，因而你可以对它做你想做的 任何处理（和字串一致）。下面是我们 DotFiles[7] 例子的抓取：

```
sub FETCH {
        carp &whowasi if $DEBUG;
        my $self = shift;
        my $dot = shift;
        my $dir = $self->{HOME};
        my $file = "$dir/.$dot";

        unless (exists $self->{CONTENTS}->{$dot} || -f $file ) {
                carp "@{[&whowasi]}: no $dot file"  if $DEBUG;
                return undef;
        }

        # 实现一个缓冲
        if (defined $self->{CONTENTS}->{$dot} ) {
                return $self->{CONTENTS}->{$dot};
        } else {
                return $self->{CONTENTS}->{$dot} = `cat $dir/.$dot`;
        }
}
```

我们在这里做了一些手脚：我们用的是 Unix 的 cat (1) 命令，不过这样打开 文件的移植性更好（而且更高效）。而且，因为点文件是一个 Unix 式的概念， 所以我们不用太担心。或者是不应该太担心。或者...

SELF->STORE(KEY, VALUE)

当捆绑散列里的元素被设置（或者写入）的时候，这个方法做那些脏活累活。 它在对象后面还有两个参数：我们存贮新值的键字，以及新值本身。

就我们的 DotFiles[2] 例子而言，我们首先要在 tie 返回的最初的对象上调用方法 clobber 以后才能允许拥护覆盖一个文件：

```
sub STORE {

        carp &whowasi if $DEBUG;

        my $self = shift;

        my $dot = shift;

        my $value = shift;

        my $file = $self->{HOME} . "/.$dot";


        croak "@{[&whowasi]}: $file not clobberable"

        unless $self->{CLOBBER};


        open(F, "> $file") or croak "cna't open $file: $!";

        print F $vale;

        close(F);

}
```

如果有谁想删除什么东西，他们可以说：

```
$ob = tie %daemon_dots, "daemon";

$ob->clobber(1);

$daemon_dot{signature} = "A true daemon\n";
```

不过，它们可以用 tied 设置 {CLOBBER}：

```
tie %daemon_dots, "Dotfiles", "daemon";
tied(%daemon_dots)->clobber(1);
```

或者用一条语句：

```
(tie %daemon_dots, "DotFiles", "daemon")->clobber(1);
```

clobber 方法只是简单的几行：

```
sub clobber{
        my $self  = shift;
        $self->{CLOBBER} = @_ ? shift : 1;
}
```

SELF->DELETE(KEY)

这个方法处理从散列中删除一个元素的请求。如果你模拟的散列在某些地方用了 真的 散列，那么你可以只调用真的 delete。同样，我们将仔细检查用户是否 真的想删除文件：

```
sub DELETE {
        carp &whowasi if $DEBUG;
        my $self = shift;
        my $dot = shift;
        my $file = $self->{HOME} . "./$dot";
        croak "@{[&whowasi]}: won't remove file $file"
        unless $self->{CLOBBER};
```

```
        delete $self->{CONTENTS}->{$dot};
        unlink $file or carp "@{[&whowasi]}: can't unlink $file: $!";
}
```

SELF->CLEAR

当需要清理整个散列的时候运行这个方法，通常是给散列赋一个空列表。在我们的 例子里，这可以要删除用户的所有点文件的！这可真是一个危险的方法，所以我们 要求在进行清理之前要把 CLOBBER 设置为大于 1：

```
sub CLEAR {
        carp &whowasi if $DEBUG;
        my $self = shift;
        croak "@{[&whowasi]}: won't remove all dotfiles for $self->{USER}"
        unless $self->{CLOBBER} > 1;
        for my $dot (key % {$self->{COTENTS}}) {
                $self->DELETE($dot);
        }
}
```

SELF->EXISTS(KEY)

当用户在某个散列上调用 exists 函数的时候运行这个方法。在我们的例子里， 我们会查找 {CONTENTS} 散列元素来找出结果：

```
sub EXISTS {
        carp &whowasi if $DEBUG;
        my $self = shift;
        my $dot = sfhit;
        return exists $self->{CONTENTS}->{$dot};
}
```

SELF->FIRSTKEY

当用户开始遍历散列，比如说用一个 keys，或者 values，或者一个 each 调用的 时候需要这个方法。我们通过在标量环境中调用 keys，重置其（keys 的）内部 状态以确保后面 retrun 语句里的 each 将拿到第一个键字。

```
sub FIRSTKEY {


        carp &whowasi if $DEBUG;

        my $self = shift;

        my $temp = keys %{$self->{CONTENTS}};

        return scalar each %{$self->{CONTENTS}};

}
```

SELF->NEXTKEY(PREVKEY)

这个方法是 keys，values 或者 each 函数的叙述器。PREVKEY 是上次访问的 键字，Perl 知道该提供什么。如果 NEXTKEY 方法需要知道它的前面的状态来 计算下一个状态时这个变量很有用。

就我们的例子，我们正在使用一个真正的散列来代表捆绑了的散列的数据，不同的 只是这个散列保存在散列的 CONTENTS 数据域而不是在散列本身。因此我们只需要 依赖 Perl 的 each 叙述器就行了：

```
sub NEXTKEY {

        carp &whowasi if $DEBUG;

        my $self = shift;

        return scalar each %{ $self->{CONTENTS}}

}
```

SELF->DESTORY

当你准备删除这个捆绑的散列对象的时候触发这个方法。你实际上并不需要这个 东西，除非是用做调试和额外的清理。下面是一个非常简单的版本：

```
sub DESTROY{
        carp &whowasi if $DEBUG;
}
```

请注意我们已经给出了所有的方法，你的作业就是退回去把我们代换 @{[&whowasi]} 的地方找出来然后把他们替换成名字为 $whowasi 的简单捆绑 标量，并且要实现相同的功能。

14.4 捆绑文件句柄

一个实现捆绑文件句柄的类应该定义下面的方法：TIEHANDLE 和至少 PRINT，PRINTF，WRITE，READLINE，GETC 和 READ 之一。该类还可以提供一个 DESTROY 方法，以及 BINMODE，OPEN，CLOSE，EOF，FILENO，SEEK，TELL，READ 和 WRITE 方法以便于相应的 Perl 内建函数用这个捆绑的文件句柄。（当然，这也不是绝对正确：WRITE 对应 syswrite 而与 Perl 内建的 write 函数没有任何关系，write 是和 format 声明一起用于打印的。）

当 Perl 内嵌于其他程序中（比如 Apache 和 vi ）时以及当向 STDOUT 和 STDERR 的输出需要以某种特殊的方式重新定向的时候捆绑的文件句柄就特别有用了。

不过捆绑的文件句柄实际上完全不必与文件捆绑。你可以用输出语句制作一个存在于内存的数据结构以及用输入语句把它们读取回来。下面是一个反转 print 和 printf 语句的打印顺序但却不用颠倒相关行顺序的简单方法：

```
package ReversePrint;
use strict;
sub TIEHANDLE {
        my $class = shift;
        bless [], $class;
}
sub PRINT {
        my $self = shift;
        push @$self, join ' ', @_
```

```
}
sub PRINTF {
        my $self = shift;
        my $fmt = shift;
        push @$self, sprintf $fmt, @_;
}
sub READLINE {
        my $self = shift;
        pop @$self;
}

package main;
my $m = "--MORE--\n";
tie *REV, "ReversePrint";

# 做一些 print 和 printf。
print REV "The fox is now dead. $m";
printf REV <<"END", int rand 10000000;
The quick brown fox jumps over
over the lazy dog %d times!
END

print REV <<"END";
The quick brown fox jumps
```

```
over the lazy dog.

END


# 现在从同一个句柄中读回来

print while <REV>;
```

打印出：

```
The quick brown fox jumps

over the lazy dog.

The quick brown fox jumps over

over the lazy dog 3179357 times!

The fox is now dead.--MORE--
```

14.4.1 文件句柄捆绑方法

对于我们扩展的例子而言，我们将创建一个文件句柄，并且向之打印大写字串。为了明确，我们会在把这个文件句柄打开的时候向它打印 ，而当结束关闭的时候打印 。这个方法是我们从格式优良的 XML 中借来的。

下面是我们将实现这个类的 shout.pm 文件的开头：

```
package Shout;

use Carp;       # 这样我们就可以把我们的错误汇报出来
```

然后我们把在 shout.pm 里定义的方法列出来：

CLASSNAME->TIEHANDLE(LIST)

这是该类的构造器，和往常一样，应该返回一个赐福了的引用。

```
sub TIEHANDLE {
        my $class = shift;
        my $form = shift;
```

```
        open my $self, $form, @_ or croak "can't open $form@_: $!";
        if ($form =~ />/) {
                print $self "<SHOUT>\n";
                $$self->{WRITING} = 1;    # 记得写结束标记
        }
        return  bless $self, $class;    # $self 是一个全局引用
}
```

在这里，我们根据传递给 tie 操作符的模式和文件名打开一个新的文件句柄， 向文件中写入 ，然后返回一个指向它的赐福了的引用。在 open 语句里 有一大堆东西，不过我们将只会指出一点，除了通常的 "open or die" 惯用法 以外，my $self 给 open 提供了一个未定义的标量，open 知道自动把那个标量 转成一个类型团。这个变量是类型团这一点非常重要，因为这个类型团不仅包含 文件真实的 I/O 对象，而且还包含各种各样其他可以自由获取的数据结构，比如 一个标量（$$$self），一个数组（@$$self），和一个散列（%$$self）。（我们 不会提到子过程，&$$self。）

$form 是文件名或者模式参数。如果它是一个文件名，@_ 就是空的，所以它的 性质就象一个两个参数的 open。否则，$form 就是剩余参数的模式。

open 之后，我们检测一下看看我们是否应该写入表示开始的标记。如果是，我们 就写。然后我们马上使用那些我们谈到的团数据结构。那个 @@self->{WRITING} 是一个使用团存储有趣信息的一个例子。在这个例子里，我们记住是否写过起始 标记，这样我们才知道我们是否应该做相应的结束标记。我们正在使用 %$$self 散列，所以我们可以给那个数据域一个象样的名字。我们本可以用象 $$$self 这样的标量，但是那样不能自说明。（或者它只能自说明——取决于你如何看它。）

SELF->PRINT(LIST)

这个方法实现了一个向捆绑的句柄 print。LIST 是传递给 print 的东西。我们 下面的方法把 LIST 的每个元素都转换成大写：

```
sub PRINT {
    my $self = shift;
    print $self map {uc} @_;
```

```
}
```

SELF->READLINE

当用尖角操作符（）或者 readline 读句柄的时候，用这个方法提供数据。 当没有更多数据可读的时候，这个方法应该返回 undef。

```
sub READLINE {
   my $self = shift;
   return <$self>;
}
```

在这里，我们只是简单地 return <$self>，这样，根据标量环境还是列表环境， 这个方法就能做出正确的反映。

SELF->GETC

当在捆绑的文件句柄上使用 getc 的时候就会运行这个方法。

```
sub GETC {
   my $self = shift;
   return getc($self);
}
```

和我们 Shout 类的几个方法类似，GETC 只是简单地调用相应的 Perl 内建的函数 然后返回结果。

SELF->OPEN(LIST)

我们的 TIEHANDLE 方法本身就打开一个文件，但是一个使用 Shout 类的程序在 那之后调用 open 将触发这个方法。

```
sub OPEN {
   my $self = shift;
   my $form = shift;
   my $name = "$form@_";
```

```
        $self->CLOSE;
        open($self, $form, @_)    or croak "can't reopen $name: $!";
        if ($form =~ />/) {
                print $self "<SHOU>\n" or croak "can't start print: $!";
                $$self->{WRITING} = 1;       # 记得写结束标记
        }
        else {
                $$self->{WRITING} = 0;       # 记得不要写结束标记
        }
        return 1;
}
```

我们激活了我们自己的 CLOSE 方法明确地关闭文件，以免用户不愿意自己做。 然后我们打开一个新文件，文件名是在 open 里声明的，然后再向里面写东西。

SELF->CLOSE

这个方法处理关闭句柄的请求。在这里，我们搜索到文件的结尾，如果成功， 则打印 ，然后调用 Perl 内建的 close。

```
sub CLOSE {
        my $self = shift;
        if ($$self->{WRITING}) {
                $self->SEEK(0,2)    or return;
                $self->PRINT("</SHOUT>\n")    or return;
        }
        return close $self;
}
```

SELF->SEEK(LIST)

当你对一个捆绑的文件句柄进行 seek 的时候调用 SEEK 方法。

```
sub SEEK {
        my $self = shift;
        my ($offset, $whence) = @_;
        return seek($self, $offset, $whence);
}
```

SELF->TELL

当你对一个捆绑的文件句柄调用 tell 的时候调用这个方法。

```
sub TELL {
        my $self = shift;
        return tell $self;
}
```

SELF->PRINTF(LIST)

当在捆绑的句柄上面使用 printf 的时候运行这个方法。LIST 将包含格式和需要 打印的条目。

```
sub PRINTF {
        my $self = shift;
        my $template = shift;
        return $self->PRINT(sprintf $template, @_);
}
```

在这里，我们用 sprintf 生成格式化字串然后把它传递给 PRINT 转成大写。 不过这里也没有让你一定要用内建的 sprintf 函数的原因。你可以截获百分号 逃逸以满足你自己的目的。

SELF->READ(LIST)

当用 read 或者 sysread 对句柄做读操作时，该这个方法会做出响应。请注意 我们"现场"修改 LIST 的第一个参数，模拟 read 的能力：它填充作为它的 第二个参数传递进来的标量。

```
sub READ {
        my ($self, undef, $length, $offset) = @_;
        my $bufref = \$_[1];
        return read($self, $$bufref, $length, $offset);
}
```

SELF->WRITE(LIST)

当用 syswrite 对该句柄写入的时候调用这个方法。在这里，我们把待写字串变成 大写。

```
sub WRITE {
        my $self = shift;
        my $string = uc(shift);
        my $length = shift || length $string;
        my $offset = shift || 0;
        return syswrite $self, $string, $length, $offset;
}
```

SELF->EOF

如果用 eof 对一个与 Shout 类捆绑的文件句柄进行测试的时候，这个方法返回 一个布尔值。

```
sub EOF {
        my $self = shift;
        return eof $self;
}
```

SELF->BINMODE(DISC)

这个方法声明将要用于这个文件句柄的 I/O 规则。如果没有声明，它把这个捆绑 了的文件句柄置于二进制模式（:raw 规则），用于那些区分文本和二进制文件的 文件系统。

```
sub BINMODE {
        my $self = shift;
        my $disc = shift || ":raw";
        return binmode $self, $disc;
}
```

就这么写，但实际上这个方法在我们的类中没有什么用，因为 open 已经向句柄 中写入数据了。所以在我们的例子里我们可能可以做的更简单些：

```
sub BINMODE { croak("Too late to use binmode") }
```

SELF->FILENO

这个方法应该返回与捆绑的文件句柄相关联的操作系统文件描述符（fileno）。

```
sub FILENO {
        my $self = shift;
        return fileno $self;
}
```

SELF->DESTROY

和其他类型的捆绑一样，当对象将要删除的时候触发这个方法。对象清理自己的 时候很有用。在这里，我们确保文件关闭了，以免程序忘记调用 close。我们可以 只说 close $self，不过更好的方法是调用该类的 CLOSE 方法。这样的话，如果 类的设计者决定修改文件关闭的方法的话，这个 DESTROY 方法就不用改了。

```
sub DESTROY {
        my $self = shift;
        $self->CLOSE;       # 用 Shout 的 CLOSE 方法关闭文件
}
```

下面是我们的 Shout 类的一个演示：

```
#! /usr/bin/perl

use Shout;

tie(*FOO, Shout::, ">filename");

print FOO "hello\n";       # 打印 HELLO。

seek FOO, 0, 0;           # 退回到开头

@lines = <FOO>;           # 调用 READLINE 方法。

close FOO;           # 明确关闭文件。

open(FOO, "+<", "filename");   # 重新打开 FOO，调用 OPEN。

seek(FOO, 8, 0);       # 忽略 "<SHOUT>\n"。

sysread(FOO, $inbuf, 5)       # 从 FOO 读取 5 个字节到 $inbuf。

print "found $inbuf\n";       # 应该打印 "hello"。

seek(FOO, -5, 1);       # 退回 "hello"之前。

syswrite(FOO, "ciao!\n", 6);   # 写 6 个字节到 FOO。

untie(*FOO);          # 明确调用 CLOSE 方法
```

在运行完这些以后，这个文件包含：

```
   CIAO!
```

下面是对付那个内部团的一些更怪异而又神奇的东西。我们和往常一样使用相同的散列，但是有新的键字 PATHNAME 和 DEBUG。首先我们安装一个字串化的重载，这样，如果打印我们的一个对象的时候就会打印出路径名（参阅第十三章，重载）：

```
# 这就是所有酷玩叶儿

use overload q(" ") => sub {$_[0]->pathname };
```

```
# 这里是放你想跟踪的函数的存根。
sub trace {
        my $self = shift;
        local $Carp::CarpLevel = 1;
        Carp::cluck("\ntrace magical method") if $self->debug;
}

# 重载句柄以打印出我们的路径
sub pathname {
        my $self = shift;
        confess "i am not a class method" unless ref $self;
        $$self->{PATHNAME} = shift if @_;
        return $$self->{PATHNAME};
}

# 双重模式
sub debug {
        my $self = shift;
        my $var = ref $self ? \$$self->{DEBUG} : \our $Debug;
        $$var = shift if @_;
        return ref $self ? $$self->{DEBUG} || $Debug : $Debug;
}
```

然后象下面这样在所有你的普通方法的入口处调用 trace：

```
sub GETC { $_[0]->trace;    # 新的
```

```
        my($self) = @_;
        getc($self);
}
```

并且在 TIEHANDLE 和 OPEN 里设置路径名：

```
sub TIEHANDLE {
        my $class    = shift;
        my $form = shift;
        my $name = "$form@_";          # NEW
        open my $self, $form, @_   or croak "can't open $name: $!";
        if ($form =~ />/) {
                print $self  "<SHOUT>\n";
                $$self->{WRITING} = 1;     # Remember to do end tag
        }
        bless $self, $class;           # $fh is a glob ref
        $self->pathname($name);        # NEW
        return $self;
}


sub OPEN { $_[0]->trace;           # NEW
        my $self = shift;
        my $form = shift;
        my $name = "$form@_";
        $self->CLOSE;
        open($self, $form, @_)      or croak "can't reopen $name: $!";
```

```
        $self->pathname($name);         # NEW
        if ($form =~ />/) {
                print $self "<SHOUT>\n" or croak "can't start print: $!";
                $$self->{WRITING} = 1;      # Remember to do end tag
        }
        else {
                $$self->{WRITING} = 0;      # Remember not to do end tag
        }
        return 1;
}
```

有些地方你还需要调用 $self->debug(1) 打开调试。如果你这么做，那么你的所有 Carp::cluck 调用都将会生成有意义的信息。下面是我们做上面的 reopen 的时候得到的信息。它给我们显示了三个深藏的方法，当时我们正在关闭旧文件并且准备打开新文件：

```
trace magical method at foo line 87
    Shout::SEEK('>filename', '>filename', 0, 2) called at foo line 81
    Shout::CLOSE('>filename') called at foo line 65
    Shout::OPEN('>filename', '+<', 'filename') called at foo line 141
```

14.4.2 创建文件句柄

你可以把同一个文件句柄同时 tie （捆绑）到一个两头的管道的输入和输出端。假设你想象下面这样运行 bc （一个任意精度的计算器）程序：

```
use Tie::Open2;

tie *CALC, 'Tie::Open2', "bc -l";
$sum = 2;
for (1 .. 7) {
```

```
        print CALC "$sum * $sum\n";
        $sum = <CALC>;
        print "$_: $sum";
        chomp $sum;
}
close CALC;
```

我们可以看到打印出下面的东西：

```
1: 4
2: 16
3: 256
4: 65536
5: 4294967296
6: 18446744073709551616
7: 340282366920938463463374607431768211456
```

如果你的机器里有 bc 而且还有象下面这样定义的 Tie::Open2，那么你就能看到上面预期的输出。这次我们给我们的内部对象用了一个赐福了的数组。它包含我们的两个真正的文件句柄用于读和写。（打开一个双头管道的脏活由 IPC::Open2 干；我们只做有意思的部分。）

```
package Tie::Open2;
use strict;
use Carp;
use Tie::Handle;      # 不要从这里继承
use IPC::Open2;

sub TIEHANDLE {
```

```
        my ($class, @cmd) = @_;
        no warnings 'once';
        my @fhpair = \do { local(*RDR, *WTR) };
        bless $_, 'Tie::StdHandle' for @fhpair;
        bless(\@fhpair => $class)->OPEN(@cmd) || die;
        return \@fhpair;
}

sub OPEN {
        my ($self, @cmd) = @_;
        $self->CLOSE if grep {defined} @{ $self->FILENO };
        open2(@$self, @cmd);
}

sub FILENO {
        my $self = shift;
        [ map { fileno $self->[$_] } 0, 1];
}

for my $outmeth (qw(PRINT PRINTF WRITE) ) {
        no strict 'refs';
        *$outmeth = sub {
                my $self = shift;
                $self->[1]->$outmeth(@_);
```

```
        };
}

for my $inmeth (qw(READ READLINE GETC) ) {
        no strict 'refs';
        *$inmeth = sub {
                my $self = shift;
                $self->[0]->$inmeth(@_);
        };
}

for my $doppelmeth (qw(BINMODE CLOSE EOF)) {
        no strict 'refs';
        *$doppelmeth = sub {
                my $self = shift;
                $self->[0]->$doppelmeth(@_)  && $self->[1]->$doppelmeth(@_);
        };
}

for my $deadmeth (qw(SEEK TELL)) {
        no strict 'refs';
        *$deadmeth = sub {
                craok("can't $deadmeth a pipe");
        };
```

```
}

1;
```

最后四行以我们的观点来说是非常时髦的。为了解释这里在做什么，请回过头看一眼第八章，引用，里面的"作为函数模板的闭合"。

下面是一套更怪异的类。它的包名字应该给你一些关于它是干什么的线索。

```
use strict;

package Tie::DevNull


sub TIEHANDLE {

        my $class = shift;

        my $fh = local *FH;

        bless \$fh, $class;

}

for (qw(READ READLINE GETC PRINT PRINTF WRITE)) {

        no strict 'refs';

        *$_ = sub { return };

}


package Tie::DevRandom;


sub READLINE { rand() . "\n"; }

sub TIEHANDLE {

        my $class = shift;

        my $fh = local *FH;
```

```
        bless \$fh, $class;
}
sub FETCH { rand() }
sub TIESCALAR {
        my $class = shift;
        bless \my $self, $class;
}

package Tie::Tee;

sub TIEHANDLE {
        my $class = shift;
        my @handles;
        for my $path (@_) {
                open(my $fh, ">$path") || die "can't write $path";
                push @handles, $fh;
        }
        bless \@handles, $class;
}

sub PRINT {
        my $self = shift;
        my $ok = 0;
        for my $fh (@$self) {
```

```
                $ok += print $fh @_;
        }
        return $ok = @$self;
}
```

Tie::Tee 类模拟标准的 Unix tee (1) 程序，它把一个输出流发送到多个不同的目的。Tie::DevNull 类模拟空设备，Unix 系统里的 /dev/null 。而 Tie::DevRandom 类生成可以用做句柄或标量的随机数，具体做什么取决于你调用的是 TIEHANDLE 还是 TIESCALAR！下面是你调用它们的方法：

```
package main;


tie *SCATTER,     "Tie::Tee", qw(tmp1 - tmp2 >tmp3 tmp4);
tie *RANDOM,    "Tie::DevRandom";
tie *NULL,    "Tie::DevNull";
tie  my $randy, "Tie::DevRandom";


for my $i (1..10) {
        my $line = <RANDOM>;
        chomp $line;
        for my $fh ( *NULL, *SCATTER) {
                print $fh "$i: $line $randy\n";
        }
}
```

这个程序在你的屏幕上输出类似下面的东西：

```
1: 0.124115571686165 0.20872819474074
2: 0.156618299751194 0.678171662366353
```

```
3: 0.799749050426126 0.300184963960792

4: 0.599474551447884 0.213935286029916

5: 0.700232143543861 0.800773751296671

6: 0.201203608274334 0.0654303290639575

7: 0.605381294683365 0.718162304090487

8: 0.452976481105495 0.574026269121667

9: 0.736819876983848 0.391737610662044

10: 0.518606540417331 0.381805078272308
```

不过事还没完！它向你的屏幕输出是因为上面的 *SCATTER tie 里的 -。而且那一行还命令它创建文件 tmp1，tmp2，和 tmp4，同时还附加到文件 tmp3 上。（我们在循环里还向 *NULL 输出了，当然那不会在任何有趣的地方显示任何东西，除非你对黑洞感兴趣。）

14.5 一个精细的松绑陷阱

如果你试图使用从 tie 或者 tied 返回的对象，而且该类定义了一个析构器，那么你就得小心一个精细的陷阱。看看下面这个（故意设计的）例子类，它使用一个文件来记录赋予一个标量的所有值：

```
package Remember;


sub TIESCALAR {
        my $class = shift;
        my $filename = shift;
        open (my $handle, ">", $filename)
        or die "Cannot open $filename: $!\n";
        print $handle "The Start\n";
        bless {FH => $handle, VALUE  => 0}, $class;
}
```

```
sub FETCH {
        my $self = shift;
        return $self->{VALUE};
}

sub STORE{
        my $self = shift;
        my $value = shift;
        my $handle = $self->{FH};
        print $handle "$value\n";
        $self->{VALUE} = $value;
}

sub DESTROY {
        my $self = shift;
        my $handle = $self->{FH};
        print $handle "The End\n";
        close $handle;
}

1;
```

然后是一个利用我们 Remember 类的例子：

```
use strict;
```

```
use Remember;


my $fred;
$x = tie $fred, "Remember", "camel.log";
$fred = 1;
$fred = 4;
$fred = 5;
untie $fred;
system "cat camel.log";
```

执行的输出是：

```
The Start
1
4
5
The End
```

到目前为止还不错。现在让我们增加一个额外的方法到 Remember 类，这样允许在文件中有注释，也就是象下面这样的东西：

```
sub comment {
        my $self = shift;
        my $message = shift;
        print {$self->{FH}} $handle $message, "\n";
}
```

下面是前面那个例子，修改以后利用 comment 方法：

```
use strict;
```

```
use Remember;


my ($fred, $x);

$x = tie $fred, "Remember", "camel.log";

$fred = 1;

$fred = 4;

comment $x "changing...";

$fred = 5;

untie $fred;

system "cat camel.log";
```

现在这个文件会是空的，而这样的结果可能不是你想要的。让我们解释一下为什么。捆绑一个变量实际上是把它和构造器返回的对象关联起来。这个对象通常只有一个引用：那个藏在捆绑变量身后的。调用“untie”打破了这个关联并且消除了该引用。因为没有余下什么指向该对象的引用，那么就会出发 DESTROY 方法。

不过，在上面的例子中，我们存贮了第二个指向捆绑到 $x 上的对象。那就意味着在 untie 之后，我们还有一个有效的指向该对象的引用。因此 DESTROY 就不会触发，于是该文件就得不到输出冲刷并且关闭。这就是没有输出的原因：文件句柄的缓冲区仍然在内存里。它在程序退出之前不会存储到磁盘上。

要想侦测到这些东西，你可以用 -w 命令行标志，或者在当前的词法范围里包含 use warnings "untie" 用法。这两种技巧都等效于在仍然存在有捆绑的对象的一个 untie 调用。如果这么处理，Perl 打印下面的警告：

```
untie attempted while 1 inner references still exist
```

要想让程序能够运行而且看不见这些警告，那么就要在调用 untie 之前删除任何多余的指向捆绑对象的引用。你可以用下面的方法明确地处理：

```
undef $x;

untie $fred;
```

不过，通常你可以通过让变量在合适的时刻跑出范围来解决问题。

14.6 CPAN 里的模块

在你开始鼓足干劲写你自己的捆绑模块之前，你应该检查一下是不是已经有人做出来了。在 CPAN 里面有许多捆绑模块，而且每天都在增加。（哦，应该是每个月。）表 14-1 列出了其中的一部分。

表 14-1。 CPAN 的捆绑模块

模块	描述
GnuPG[2]::Tie::Encrypt	把一个文件句柄和 GNU Privacy Guard 加密捆绑在一起
IO::WrapTie	把捆绑对象和一个 IO:Handle 接口封装在一起。
MLDBM	在一个 DBM 文件里透明地存储复杂的数据值，而不仅仅是简单的字串。
Net::NISplusTied	把散列和 NIS+ 表捆绑在一起
Tie::Cache::LRU	实现一个最近最少使用缓冲
Tie::Const	提供常数标量和散列
Tie::Counter	让一个标量变量每接受一次访问就加一
Tie::CPHash	实现一个保留大小写但是又大小写无关的散列
Tie::DB_FileLock	提供对 Berkeley DB 1.x 的锁定访问
Tie::DBI	把散列和 DBI 关系数据库捆绑在一起
Tie::DB_Lock	用共享和排它锁把散列和数据库捆绑在一起
Tie::Dict	把一个散列和一个 RPC 字典服务器捆绑在一起
Tie::Dir	把一个散列捆绑为读取目录
Tie::DirHandle	捆绑目录句柄
Tie::FileLURCache	实现一个轻量级的，基于文件系统的永久的 LRU 缓冲
Tie::FlipFlop	实现一个在两个值之间接换的捆绑
Tie::HashDefaults	令散列有缺省值
Tie::HashHistory	跟踪对散列修改的所有历史
Tie::IxHash	为 Perl 提供有序的关联数组
Tie::LDAP	实现一个 LDAP 数据库的接口
Tie::Persistent	通过 tie 实现一个永久数据结构
Tie::Pick	从一个集合中随机选取（和删除）一个元素
Tie::RDBM	把散列和关系数据库捆绑
Tie::SecureHash	支持基于名字空间的封装
Tie::STDERR	把你的 STDERR 的输出发给另外一个进程，比如一个邮件服务

	器
Tie::Syslog	把一个文件句柄自动捆绑到 syslog 作为其输出
Tie::TextDir	捆绑一个文件目录
Tie::TransactHash	在事务中编辑一个散列，而又不改变事务的顺序
Tie::VecArray	给一个位矢量提供一个数组接口
Tie::Watch	在 Perl 变量中提供观察点
Win32::TieRegistry	提供有效且简单的操作 Microsoft Windows 注册表的方法。

第十五章　UNICODE

如果你还不知道什么是 Unicode，那么你很快就会知道了——即使你略过本章，你也会知道——因为利用 Unicode 工作日益成为必须。(有些人认为它是一个必须有的恶魔，但实际上它更象是必须存在的好人。不管怎样，它都是必须有的痛苦。)

从历史来看，人们制定字符集来反映在他们自己的文化环境里所需要处理的事物。因为所有不同文化的人群都是懒惰的，他们只想包含那些他们需要的符号，而排除他们不需要的符号。如果我们只和自己文化内部的其他人进行交流，那么这些符号是够用的，但是既然我们开始使用互联网进行跨文化的交流，那么我们就会因为原先排除的符号而头疼。在一个美国键盘上键入音调字符已经够难的了。那么应该怎样才能写一个多语言的网页能？

Unicode 就是答案，或者至少是一部分答案（又见 XML）。Unicode 是一种包容性的字符集，而不是排斥性的字符集。尽管人们仍然会就 Unicode 的各种细节争论不休（而且的确还有许多可以争吵的细节），但是 Unicode 的总体目标是让每个人在使用 Unicode 的时候都足够高兴（注：或者在一些场合，并非彻底失望。），这样大家就都愿意把 Unicode 当作交换文本数据的国际媒介。没有人强迫你使用 Unicode，就好象没有人强迫你阅读本章一样（我们希望如此）。我们仍将允许人们在他们自己的文化里使用他们原来的排它的字符集。但是那样的话（如我们所说），移植性就会有问题。

痛苦守恒定律告诉我们，如果我们在一个地方减少痛苦，那么在另外的地方肯定会增加痛苦。在 Unicode 的例子里，我们必须经历从字节语义到字符语义的迁移的痛苦。因为由于历史的偶然性，Perl 是美国人发明的，而 Perl 在历史上就混淆了字节和字符的概念。为了向 Unicode 迁移，Perl 必须在某种程度上分清这两个概念。

荒谬的是，我们让 Perl 分清楚了字符和字节的关系，而却又允许 Perl 程序员混淆它们的概念，而依靠 Perl 来保持界限，就好象我们允许程序员混淆数字和字串而是依靠 Perl 做必要的转换。要扩展可能性，Perl 处理 Unicode 的方法和处理其他东西的方法是一样的：做正确的事情。通常，我们要实现下面四个目标：

目标 1：

旧的面向字节的程序不能同时破坏它们原来用来运行的旧式面向字节的数据。

目标 2：

旧的面向字节的程序在合适的条件下应该能神奇地开始在新的面向字符的数据 上运行。

目标 3：

程序在新的面向字符模式下应该和原来的面向字节的模式运行得一样快。

目标 4：

Perl 应该仍然是一种语言，而不是分裂成一种面向字节的 Perl 和一种面向字符 的 Perl。

把它们合在一起，这些目标实际上是不可能实现的。不过我们已经非常接近了。或者，换句话说，我们仍然在向非常接近目标努力，因为这是一项正在进行的工作。随着 Unicode 的不断发展，Perl 也在不断发展。但是我们的主要目标是提供一条安全的迁移路径，这样我们在迁移的过程中只需要最少的注意就可以了。我们是如何做的是下一节的内容。

15.1 制作字符

在 Perl 5.6 以前的版本，所有字串都被看成一个字节序列（注：你可能愿意把它们称做“八位字节”；那样挺好，不过我们认为如此这两个词几乎同义，所以我们仍然使用那个蓝领阶层的叫法）不过，Perl 5.6 以后的版本里，一个字串可能包含一些比一个字节宽的字符。我们现在不把字串看作一个字节序列，而是一个数字序列，这些数字是处于 0 .. 2**32-1（或者在 64 位机上是 0 .. 2**64-1）之间。这些数字代表抽象的字符，而且在某种意义上数字越大，字符越宽；不过和许多语言不同的是，Perl 没有捆绑在任何特定宽度的字符形式上。Perl 使用一种变长编码（基于 UTF-8），所以这些抽象的数据可能能够，也可能不能够每字节封装一个数字。显然，字符数字 18,446,744,073,709,551,615 （也就是"\x{ffff_ffff_ffff_ffff}"）肯定不能放在一个字节里（实际上，它占了十三个字节），但是如果你的字串里的所有字符都在十进制 0..127 的范围里，那么它们肯定可以包在一个字节的范围里，因为 UTF-8 在低七位空间里和 ASCII 是一样的。

Perl 只有在它认为有益的时候才使用 UTF-8，所以如果你的字串里的所有字符都落在 0..255 的范围里，那么很有可能这些字符都封装在一个字节里——但是如果缺乏其他知识，你也无法确定这一点，因为 Perl 在内部根据需要在定长的 8 位字符和变长 UTF-8 字符之间进行转换。关键是你大多数时间不需要关心这些内容，因为这里的字符语意是一种不考虑表现的抽象含义。

不论什么情况下，如果你的字串包含任意大于十进制 255 的数字，那么该字串肯定是以 UTF-8 的形式存储的。更准确地说，它是以 Perl 扩展了的 UTF-8 版本存储的，我们管它叫 utf8，一方面是尊重那个名称的用法，另一方面也是为了敲击简单。（并且因为"真正"的 UTF-8 只允许包含 Unicode 联盟确认的字符数字。Perl 的 utf8 允许你包含任何你需要的字符数字。Perl 并不在乎你的字符数字是官方认可的还是你认可的。）

我们说过你在大部分时候都不用关心这些，但是人们还是愿意关心。假设你使用一个 v 字串代表一个 IPv4 地址：

```
$locaddr = v127.0.0.1;        # 当然是按照字节存储

$oreilly = v204.148.40.9;     # 可能以字节或者以 utf8 方式存储

$badaddr = v2004.148.40.9     # 当然是以 utf8 方式存储
```

每个人都明白 $badaddr 不是一个 IP 地址。所以我们很容易认为如果 O'Reilly 的网络地址被强制表示成 UTF-8 方式，那么它就无法运行了。但是在字串里的字符都是抽象数字，而不是字节。任何使用 IPv4 地址的东西，比如 gethostbyaddr 函数，都自动把抽象字符数字转换成一个字节形式（并且对 $badaddr 会失效）。

Perl 和真实世界之间的接口需要处理表现形式的细节。现有的接口在最大的可能下都可以不用你告诉它们如何处理而做出正确的动作。但是的确有偶尔的机会需要你给一些接口以某种指导（比如 open 函数），而且如果你写你自己的与现实世界的接口，那么这个接口要么是聪明得能够自己分辨事物，要么至少是能够在非缺省特性的时候能够遵循指导。（注：在一些系统上可能存在一次性切换你的所有接口的方法。如果使用了 -C 命令行开关，（或者全局的 ${@WIDE_SYSTEM_CALLS} 变量设置为 1，那么所有系统调用都会使用对应的宽字符 API。（这个特性目前只在 Microsoft Windows 上实现。） Linux 社区目前的计划是如果 $ENV{LC_CTYPE} 设置为"UTF-8"，那么所有接口都切换到 UTF-8 模式。其他的社区可能采纳其他方法。我们的进展也可能变化。）

因为 Perl 要关心在它自己内部维持透明的字符语意，所以你需要关心字节与字符语意的区别的唯一的地方就是你的接口。缺省时，你所有旧的连接外部世界的 Perl 接口都是面向字节的，因此它们生成和处理面向字节的数据。也就是说，在这个抽象层，你的所有字串都是范围 0..255 的数字序列，所以如果在程序里没有什么东西强迫它们表示成 utf8，你的旧

的程序就继续以面向字节的数据为处理对象，就象它们原来的样子。因此可以在上面的目标 1 上画个勾。

如果你希望你的旧程序可以处理新的面向字符的数据，那么你必须设法标记你的面向字符的接口，这样 Perl 就明白在那些接口上准备处理面向字符的数据。一旦你完成这些工作，Perl 就会自动做任何必须的转换工作以保持字符的抽象性。唯一的区别是你已经引入了一些字串到你的程序里面，这些字串标记为可能包含超过 255 的字符，因此，如果你在字串和 utf8 字串之间进行操作，那么 Perl 将在内部先把字节字串转换成 utf8 字串，然后再执行操作。通常，只有你又把它们发送回给一个字节接口的时候，utf8 字串才转换回字节字串，这个时候，如果字串包含大于 255 的字符，那么你就会有一个问题，这个问题可以用多种不同的方法来处理——取决于那个有问题的接口。因此你可以在目标 2 上也画一个勾。

有时候你想把理解字符语意的编码和那些必须以字节语意运行的编码，比如读取或者写出固定大小的块的 I/O 编码，混合起来使用。这种情况下，你可以在面向字节的编码周围放一个 use bytes 声明以强制它使用字节语意——甚至是在那些标记为 utf8 的字串上。然后你就要对任何必须的转换负责。不过这是一个强化目标 1 的更严格的本地读取，代价放松目标 2 的全局读取。

目标 3 大部分都实现了，原因部分是通过做字节和 utf8 表示形式之间的懒惰的转换另一部分是因为我们在实现 Unicode 里的那些比较慢的特性（比如大表里的属性查找）的时候做得比较隐蔽。

我们通过牺牲一小部分实现其他目标的接口的兼容性实现了目标 4。从某个角度来看，我们没有把 Perl 分裂成不同的两种 Perl；但是以另外一种方法来看，版本 5.6 的 Perl 是一种分裂了的版本，只是它仍然尊重以前的版本，而且我们认为人们在确信新版本能够处理他们的事物之前不会从更早的版本切换到新的版本。不过新版本总是这个样子的，所以我们允许自己在目标 4 上也打个勾。

15.2 字符语意的效果

字符语意的结果是一个典型的内建操作符将以字符为操作对象，除非它位于 use bytes 用法里。不过，即使在 use bytes 用法外面，如果该操作符的所有操作数都以 8 位字符的方式存储（也就是说，没有操作符以 utf8 方式存储），那么字符语意就和字节语意没有区别，并且操作符的结果将以 8 位的方式存储在内部。这样，只要你的程序不使用比 Latin-1 更宽的字符，那么这个程序就保留的向后的兼容性。

utf8 用法主要是一个兼容性设备，它打开分析器对 UTF-8 文本和标识的识别。它还可以用于打开一些更是处于实验阶段的 Unicode 支持特性。我们的远期目标是把 utf8 用法变成透明层（no-op）。

use bytes 用法可能永远不会成为透明层。它不仅对面向字节的编码是必要的，而且在一个函数上定义面向字节的封装在 use bytes 范围外面使用还会有副作用。在我们写这些的时候，唯一定义了的封装是 length，不过随着时间的推移，会有更多封装出现的。要使用这样的封装，你可以说：

use bytes(); # 不加输入 byte 语意地装载封装器 ... $charlen = length("\x{ffff_ffff}"); # 返回 1 $bytelen = byte::length("\x{ffff_ffff}"); # 返回 7

在 use bytes 声明外边，Perl 版本 5.6 的运行特性（或者至少其期望的运行特性）是这样的：

- 字串和模式现在可以包含值大于 255 的字符；

```
use utf8;

$convergence = ″ ″;
```

假设你有一个可以编辑 Unicode 的编辑器编辑你的程序，这样的字符通常会在 文本字串中直接以 UTF-8 字符的方式出现。从现在开始，你必须在你的程序开头 开头声明一个 use utf8 以便允许文本中使用 UTF-8。

如果你没有 Unicode 编辑器，那么你还是可以用 \x 表示法声明特定的 ASCII 码扩展。Latin-1 范围的字符可以以 \x{ab} 的形式或者 \xab 的形式写，但是 如果数字超过两位十六进制数字，那你就必须使用花括号。你可以用 \x 后面跟着 花括号括起来的十六进制编码来表示 Unicode。比如一个 Unicode 笑脸符号是 \x{263A}。在 Perl 里没有什么语法构造假设 Unicode 字符正好就是 16 位，所以你不能用其他语言那样的 \u263A 来表示；\x{263A} 是最相近的等价物。

- 在 Perl 脚本里的标识符可以包含 Unicode 字母数字字符，包括象形文字：

```
use utf8;

$人++       # 又生了一个孩子
```

同样，你需要 use utf8 （至少目前）才能识别你的脚本中的 UTF-8。目前，如果 需要使用规范化的字符形式，你得自己决定——Perl （还）不会试图帮助你 规范化变量名。我

们建议你把你的程序规范化为正常 C 模式（Normalization Form C），因为这种形式是有朝一日可能是 Perl 的缺省规范化形式。参阅 www.unicode.org 获取最新的有关规范化的技术报告。

- 正则表达式现在匹配字符，而不是字节。比如，点匹配一个字符而不是一个字节。 如果 Unicode 协会准备批准 Tengwar 语言，那么（尽管这样的字符在 UTF-8 里 用四个字节表示），但下面的东西是匹配的：

```
"\N{TENGWAR LETTER SILME NUQUERNA}" =~ /^.$/
```

\C 模式用于强制一次匹配是对一个字节的（C 里的“char”，因此是 \C）。用 \C 的时候要小心，因为它会令你和你的字串的字符边界不同步，而且你可能会收到 “Malformed UTF-8 character” 错误。你不能在方括号里使用 \C，因为它不代表任何特定的字符或者字符集。

- 在正则表达式里的字符表匹配字符而不是字节 ，并且匹配那些在 Unicode 属性数据库里声明的字符属性。因此可以把 \w 用于匹配一个象形文字：

```
"人" =~ /\w/
```

- 可以用新的 \p （匹配属性）和 \P（不匹配属性）构造，把命名 Unicode 属性和块范围用做字符表。比如，\p{Lu} 匹配任何有 Unicode 大写字符属性的字符，而 \p{M} 匹配任何标记字符。但字母属性可以忽略花括号，因此标记字符也可以用 \pM 匹配。还有许多预定义的字符表可以用，比如 \p{IsMirrored} 和 \p{InTibetan}：

```
"\N{greek:Iota}" =~ /\p{Lu}/
```

你还可以在放括号字符表里使用 \p 和 \P。（在版本 5.6 的 Perl 里，你需要 使用 use utf8 才能令字符属性正确工作。不过这个限制在将来会消失。）参阅 第五章，模式匹配，获取匹配 Unicode 属性的细节。

- 特殊的模式 \X 匹配任何扩展的 Unicode 序列（Unicode 标准中的“组合字符序列”），这时候，第一个字符是基础字符，而随后的字符是标记字符，这些标记字符附加在基础字符上。它等效于 (?:\PM\pM*)：

```
"o\N{COMBING TILDE BELOW}" =~ /\X/
```

你不能在方括号里使用 \X，因为它可以匹配多个字符，而且它不匹配任何特定的 字符或者字符集。

- r/// 操作符转换字符，而不是转换字节。要把所有 Latin-1 范围以外的字符变成一个问号，你可以说：

```
tr/\0-\x{10ffff}/\0-\xff?/; # utf8 到 latin1 字符
```

- 如果有字符输入，那么大小写转换操作使用 Unicode 的大小写转换表。请注意 uc 转换成大写，而 ucfirst 转换成抬头体（对那些区分这些的语言而言）。通常对应的反斜杠序列有着相同的语意：

```
$x =  "\u$word";   # 把 $word 的第一个字母改成抬头体

$x = "\U$word";      # 大写 $word

$x = "\l$word";      # 小写 $word 的第一个字母

$x = "L$word";      # 小写 $word
```

需要小心的是，Unicode 的大小写转换表并不准备给每种实例都提供循环映射， 尤其是那些大写或者抬头体的字符数和对应小写的字符数不同的语言。正如 Unicode 协会的人所说，尽管大小写属性本身是标准的，大小写映射却只是 报告性的。

- 大多数处理字串内的位置或者长度的操作符将自动切换成使用字符位置，包括 chop，substr，pos，index，rindex，sprintf，write，和 length。有意不做切换的操作符包括 vec，pack，和 unpack。不在意这些东西的操作符包括 chomp，以及任何其他的把字串当作一堆二进制位的操作，比如缺省的 sort 和处理文件名的操作符。

```
use bytes;

$bytelen = length("I do 合 气 道.");      # 15 字节

no bytes;

$charlen = length("I do 合 气 道.");      # 只有 9 字符
```

- pack/unpack 字符“c”和“C”不改变，因为它们常用于面向字节的格式。（同样，类似 C 语言里的“char”。）不过，现在有了一个新的“U”修饰词可以在 UTF-8 字符和整数之间做转换：

```
pack("U*", 1, 20, 300, 4000) eq v1.20.300.4000
```

- chr 和 ord 函数处理字符：

```
chr(1).chr(20).chr(300).chr(400) eq v1.20.300.4000
```

换句话说，chr 和 ord 类似 pack("U") 和 unpack("U")，而不是 pack("C") 和 unpack("C")。实际上，后面两个语句就是你懒得不想详 use bytes 的时候模拟 字节的 chr 和 ord 的方法。

- 最后，scalar reverse 倒转的是字符，而不是字节： 。。。（略）

如果你看看目录 PATH——TOPERLLIB/unicode，你就会找到许多定义上面语意需要的文件。Unicode 协会规定的 Unicode 属性数据库放在文件 Unicode.300（用于 Unicode 3.0）。这个文件已经用 mktables.PL 处理成同目录下的许多小 .pl 文件了（以及子目录 Is/， In/，和 To/），这些文件或目录中的一部分会被 Perl 自动装载用以实现诸如\p（参阅 Is/ 和 In/ 目录）和 uc （参阅 To/ 目录）这样的东西。其他的文件由模块装载，比如 use charname 用法（参阅 Name.pl）。不过到我们写这些为止，还有一些文件只是放在那里，等着你给它们写一个访问模块：

```
ArabLink.pl

ArabLnkGrp.pl

Bidirectional.pl

Block.pl

Category.pl

CombiningClass.pl

Decomposition.pl

JamoShort.pl

Number.pl

To/Digit.pl
```

一个可读性更好的 Unicode 的概述以及许多超级链接都放在 PATH_TO_PERLLIB/unicode/Unicode3.html。

请注意，如果 Unicode 协会制定了新的版本 ，那么这些文件中的一部分的文件名可能会变化，因此你就必须四处刺探。你可以用下面的"咒语"找出 PATH_TO_PERLLIB：

```
%perl -MConfig -le 'print $config{Privlib}'
```

如果想找到现有所有的关于 Unicode 东西，你应该看看 Unicode 标准，版本 3.0（ISBN 0-201-61633-5）。

- 请注意，"人（Unicode）"可以用了在我们写到这些的时候（也就是说，对于版本 5.6 的 Perl），使用 Unicode 上仍然有一些注意事项。（请检查你的在线文档获取最新信息。）

- 目前的正则表达式编译器不生成多形的操作码。这就意味着在编译模式的时候就要判断某个模式是否匹配 Unicode 字符（基于该模式是否包含 Unicode 字符）而不是在匹配该模式的运行的时候。这方面需要改进成只有待匹配的字串是 Unicode 才相应需要匹配 Unicode。

- 目前没有很简单的方法标记从一个文件或者其他外部数据源读取的数据是 utf8。这方面将是近期注意的主要方面，并且在你读取这些的时候可能已经搞定了。

- 我们没有办法把输入和输出转换成除 UTF-8 以外的编码方式。不过我们准备在最近做这些事情，请检查你的在线文档。

- 把本地化设置和 utf8 一起使用会导致奇怪的结果。目前，我们准备把 8 位的区域信息用于范围在 0..255 的字符，不过我们完全可以证明这样做对那些使用超过上面范围的本地化设置是不正确的（当映射成 Unicode 的时候）。而且这样做还会运行得慢一些。我们强烈建议避免区域设置。

Unicode 很好玩——但是你得正确地定义好玩的东西。

第十六章，进程间通讯

计算机进程之间几乎有和人与人之间一样多的交流。我们不应低估进程间通讯的难度。如果你的朋友只使用形体语言，那么你光注意语言暗示对你是一点用都没有。同样，两个进程之间只有达成了通讯的方法以及建筑在该方法之上的习惯的共识以后才能通讯。和任何通讯一样，这些需要达成共识的习惯的范围从词法到实际用法：几乎是从用什么方言到说话的顺序的一切东西。这些习惯是非常重要的，因为我们都知道如果光有语义而没有环境（上下文），通讯起来是非常困难的。

在我们的方言里，进程间通讯通常念做 IPC。Perl 的 IPC 设施的范围从极为简单到极为复杂。你需要用哪种设施取决于你要交流的信息的复杂度。最简单的信息几乎就是没有信息：只是对某个时间点发生了某个事件的知晓。在 Perl 里，这样的事件是通过模拟 Unix 信号系统的信号机制实现的。

在另外一个极端，Perl 的套接字设施允许你与在互联网上的另外一个进程以任何你们同时都支持的协议进行通讯。自然，自由是有代价的：你必须通过许多步骤来设置连接并且还要确保你和那头的进程用的是同样的语言。这样做的结果就是要求你需要坚持许多其他奇怪的习惯。更准确地说，甚至还要求你用象 XML，Java，或 Perl 这样的语言讲话。很恐怖。

上面两个极端的中间的东西是一些主要用于在同一台机器上的进程之间进行通讯的设施。包括老派的文件，管道，FIFO，和各种 Sys V IPC 系统调用。对这些设施的支持因平台的不同而有所差异；现代的 Unix 系统（包括苹果的 Mac OS X）支持上面的所有设施，但是，除信号和 Sys V IPC 以外，Microsoft 操作系统支持剩下的所有的，包括管道，进程分裂，文件锁和套接字。（注：除了 AF_UNIX 套接字）。

关于移植的更多的一般性信息可以在标准的 Perl 文档集中找到 （不管你的系统里是什么格式），他们在 perlport 里。与 Microsoft 相关的信息可以在 perlwin32 和 perlfork 里找到，即使在非 Microsoft 的系统里都安装了它们。相关的书籍，我们介绍下面的：

1. The Perl Cookbook, Tom Christiansen 和 Nathan Torkington （O'Reilly and Associates,1998），第十六到十八章。
2. Advanced Programming in the UNIX Environment, W. Richard Stevens （Addison-Wesley,1992）
3. TCP/IP Illustrated, W. Richard Stevens, 卷 I-III （Addison-Wesley, 1992-1996）

16.1 信号

Perl 使用一种简单的信号处理模型：在 %SIG 散列里包含指向用户定义信号句柄的引用（符号或者硬引用）。某些事件促使操作系统发送一个信号给相关的进程。这时候对应该事件的句柄就会被调用，给该句柄的参数中就有一个包含触发它的信号名字。要想给另外一个进程发送一个信号，你可以用 kill 函数。把这个过程想象成是一个给其他进程发送一个二进制位信息的动作。（注：实际上，更有可能是五或者六个二进制位，取决于你的 OS 定义的信号数目以及其他进程是否利用了你不发送别的信号的这个情况。）。如果另外一个进程安装了一个处理该信号的信号句柄，那么如果收到该信号，它就能够执行代码。不过发送进程没有任何办法获取任何形式的返回，它只能知道该信号已经合法发送出去了。发送者也接收不到任何接收进程对该信号做的处理的信息。

我们把这个设施归类为 IPC 的一种，但实际上信号可以来自许多源头，而不仅仅是其他进程。一个信号也可能来自你自己的进程，或者是用户在键盘上敲入了某种特定键盘序列，比如 Control-C 或者 Control-Z 造成的，也可能是内核在处理某些特殊事件的时候产生的，比如子进程退出或者你的进程用光堆栈或者达到了文件尺寸或内存的极限等。不过你自己的进程可以很容易区别这些场合。信号就好象一个送到你家门口的没有返回地址的神秘包裹。你打开的时候最好小心一点。

因为在 %SIG 里的记录可能是硬链接，所以通常把匿名函数用做信号句柄：

```
$SIG{INT} = sub {die "\nOutta here1\n"};

$SIG{ALRM} = sub { die "Your alarm clock went off" };
```

或者你可以创建一个命名函数，并且把它的名字或者引用放在散列里的合适的槽位里。比如，要截获中断和退出信号（通常和你的键盘的 Control-C 和 Control-\ 绑在一起），你可以这样设置句柄：

```
sub catch_zap {
        my $signame = shift;
        our $shucks++;
        die "Somebody sent me a SIG$signame!";
}

$shucks = 0;

$SIG{INT} = 'catch_zap';      # 意思总是 &main::catch_zap

$SIG{INT} = \&catch_zap;   # 最好的方法
```

```
$SIG{QUIT} = \&catch_zap;      # 把另外一个信号也捕获上
```

注意，我们在信号句柄里做的所有事情就是设置一个全局变量然后用 die 抛出一个例外。如果可能，请力争避免处理比这更复杂的东西，因为在大多数系统上，C 库都是不可再入的。信号是异步传送的（注：与 Perl 层操作码同步的信号传递安排在以后的版本发布，那样应该能解决信号和核心转储的问题。），所以，如果信号传递后你已经在一个相关的 C 库过程里面了，那么调用任何 print 函数（或者只是任何需要 malloc(3) 分配更多内存的函数）在理论上都可能触发内存错误并导致内核转储。（甚至可能 die 过程也有点有点不安全——除非该进程是在一个 eval 里执行的，因为那样会消除来自 die 的 I/O，于是就让它无法调用 C 库。）

一个更简单的捕获信号的方法是使用 sigtrap 用法安装简单的缺省信号句柄：

```
use sigtrap qw(die INT QUIT);

use sigtrap qw(die untrapped normal-signals stack-trace any error-signals);
```

如果你嫌写自己的句柄麻烦，那就可以用这个用法，不过你仍然会希望捕获危险的信号并且执行一个正常的关闭动作。缺省时，这些信号中的一部分对你的进程是致命的，当的程序收到这样的信号时只能停止。糟糕的是，这也意味着不会调用任何用做退出控制的 END 函数和用于对象终止的 DESTROY 方法。但是它们在正常的 Perl 例外中的确是被调用的（比如你调用 die 的时候），所以，你可以用这个用法无痛地把信号转换成例外。甚至在你没有自己处理这些信号的情况下，你的程序仍然能够表现正确。参阅第三十一章，用法模块，里 use sigtrap 的描述获取这个用法的更详细的特性。

你还可以把 %SIG 句柄设置为字串"IGNORE"或者"DEFAULT"，这样，Perl 就会试图丢弃该信号或者允许用缺省动作处理该信号（不过有些信号既不能捕获，也不能忽略，比如 KILL 和 STOP 信号；如果手边有资料，你可以参阅 signal(3)，看看你的系统可以用的信号列表和它们的缺省行为。）

操作系统认为信号是一个数字，而不是一个名字，但是 Perl 和大多数人一样，喜好符号名字，而讨厌神秘的数字。如果想找出信号的名字，你可以把 %SIG 散列里的键字都列出来，或者如果你的系统里有 kill 命令，你可以用 kill -l 把它们列出来。你还可以使用 Perl 标准的 Config 模块来检查你的操作系统的信号名字和信号数字之间的映射。参阅 Config(3) 获取例子。

因为 %SIG 是一个全局的散列，所以给它赋值将影响你的整个程序。如果你把信号捕获局限于某个范围，可能对你的程序的其他部分更有好处。实现这个目的的方法是用一个 local 信号句柄赋值，这样，一旦退出了闭合的语句块，那么该句柄就失去作用了。（但是要记住，local 变量对那些语句块中调用的函数是可见的。）

```
{
    local $SIG{INT} = 'IGNORE';
    ...    # 处理你自己的业务，忽略所有的信号
    fn();    # 在 fn() 里也忽略信号!
    ...    # 这里也忽略。
}       # 语句块退出后恢复原来的 $SIG{INT} 值。

fn();       # 在（假设的） fn() 里没有忽略 SIGINT
```

16.1.1 给进程组发信号

（至少在 Unix 里，）进程是组织成进程组的，一起对应一个完整的任务。比如，如果你运行了单个 shell 命令，这条命令是有一系列过滤器命令组成，相互之间用管道传递数据，这些进程（以及它们的子进程）都属于同一个进程组。该进程组有一个数字对应这个进程组的领头进程的进程号。如果你给一个正数的进程号发送信号，该信号只发送给该进程，而如果你给一个负数进程号发送信号，那么该信号将发送给对应的进程组的所有进程，该进程组的进程组号就是这个负数的绝对值，也就是该进程组领头进程的进程号。（为了方便进程组领头进程，进程组 ID 就是 $$。）

假设你的程序想给由它直接启动的所有子进程（以及由那些子进程启动的孙子进程和曾孙进程等）发送一个挂起信号。实现这个目的的方法是：你的程序首先调用 setpgrp(0,0)，使自己成为新的进程组的领头进程，这样任何它创建的进程都将成为新进程组的一部分。不管那些进程是通过 fork 手工启动的还是通过管道 open 打开的或是用 system("cmd &") 启动的后台进程。即使那些进程有自己的子进程也无所谓，只要你给你的整个进程组发送挂起信号，那么就会把它们都找出来（除了那些设置了自己的进程组或者改变了自己的 UID 的进程——它们对你的信号有外交豁免权。）

```
{
    local $SIG{HUP} = 'IGNORE';    # 排除自己
    kill(HUP, -$$);            # 通知自己的进程组
}
```

另外一个有趣的信号是信号数 0。它实际上不影响目标进程，只是检查一下，看看那个进程是否还活着或者是是否改变了 UID。也就是说，它判断给目标进程发送信号是否合法，而实际上并不真正发送信号。

```
unless ( kill 0 => $kid_pid ) {
   warn "something wicked happened to $kid_pid";
}
```

信号 0 是唯一的一个在 Unix 上和 Windows 上的 Perl 移植作用一样的信号。在 Microsoft 系统里，kill 实际上并不发送信号。相反，它强迫目标进程退出，而退出状态由信号数标明。这些东西以后都会修改。但是，神奇的 0 信号将依然如故，表现出非破坏性的特性。

16.1.2 收割僵死进程

当一个进程退出的时候，内核向它的父进程发送一个 CHLD 信号然后该进程就成为一个僵死进程（zombie，注：这是一个技术术语），直到父进程调用 wait 或者 waitpid。如果你在 Perl 里面启动新进程用的不是 fork，那么 Perl 就会替你收割这些僵死进程，但是如果你用的是一个 fork，那么就得自己做清理工作。在许多（但不是全部）内核上，自动收割的最简单办法就是把 $SIG{CHLD} 设置为 'IGNORE'。另一个更简单（但也更乏味）的方法是你自己收割它们。因为在你开始处理的时候，可能有不止一个子进程已经完蛋了，所以，你必须在一个循环里收割你的子进程直到没有更多为止：

```
use POSIX ":sys_wait_h";
sub REAPER { 1 until waitpid(-1, WNOHANG) == -1) }
```

想根据需要运行这些代码，你要么可以给它设置 CHLD 信号：

```
$SIG{CHLD} =\&REAPER;
```

或者如果你的程序是在一个循环里运行，那你只需要循环调用收割器就行了。这个方法最好，因为它避免了那些信号可能触发的在 C 库里偶然的核心转储。但是，如果在一个很快速的循环里调用，这样做的开销是巨大的，所以一种合理的折衷是用一种混合的方法：你在句柄里尽可能少做处理，把风险降到最低，同时在外部循环中等待收割僵死进程：

```
our $zombies = 0;
$SIG{CHLD} = sub { $zombies++};
```

```
sub reaper {
        my $zombie;
        our %Kid_Status;    # 存储每个退出状态
        $zombies = 0;
        while (($zombie = waitpid( -1, WNOHANG)) != -1) {
                $Kid_Status{$zombie} = $?;
        }
}
while(1) {
        reaper() if $zombies;
        ...
}
```

这段代码假设你的内核支持可靠信号。老的 Sys V 风格的信号是不可靠的，那样的话，想写正确的信号句柄几乎是不可能的。甚至早在 Perl 版本 5.003，只要可能，我们就开始使用 sigaction(2)系统调用了，因为它更可靠些。这意味着除非你在一个古老的操作系统上运行或者跑的是一个古老的 Perl ，你用不着重新安装你的句柄，也不会冒丢失信号的危险。幸运的是，所有带 BSD 风格的系统（包括 Linux，Solaris，和 Mac OS X）以及所有 POSIX 兼容的系统都提供可靠的信号，所以那些老旧的 Sys V 问题更多是历史遗留问题，而不是目前我们要关心的问题。

在新内核上，还有许多其他东西也会运行得更好些。比如，"慢的"系统调用（那种可以阻塞的，就象 read，wait，和 accept）如果被一个信号中断后将自动重新启动。在那些灰暗的旧社会里，用户代码必须记得明确地检查每个慢的系统调用是否带着 $! ($ERRNO) 为 EINTR 失败的，而且如果是这样，那么重起。而且这样的情况不光对 INT 信号，而且对有些无辜的信号，比如 TSTP（来自 Control-Z）或者 CONT （来自把任务放到前台）也会退出系统调用。如果操作系统允许，现在 Perl 自动为你重新启动系统调用。我们通常认为这是一个特性。

你可以检查一下，看看你的系统是否有更严格的 POSIX 风格的信号，方法是装载 Config 模块然后检查 $Config{d_sigaction} 是否为真。要检查慢的系统调用是否可以可以重起，检查你的系统的文档：sigaction(2) 或者 sigvec(3)，或者在你的 C sys/signal.h 里

查找 SV_INTERRUPT 或者 SA_RESTART。如果找到两个或者其中之一，你可能就拥有可重起的系统调用。

16.1.3 给慢速操作调速

信号的一个用途就是给长时间运行的操作设置一个时间限制。如果你用的是一种 Unix 系统（或者任何 POSIX 兼容的支持 ALRM 信号的系统），你就可以让内核在未来的某时刻给你进程发送一个 ALRM 信号：

```
use Fcntl ':flock';

eval {
        local $SIG{ALRM} = sub { die "alarm clock restart" };
        alarm 10;       # 安排 10 秒后报警
        eval {
                flock(FH, LOCK_EX)    # 一个阻塞的，排它锁
                or die "can't flock:$!";
        };
        alarm 0;          # 取消报警
};
alarm 0;              # 避免冲突条件
die if $@ && $@ !~ /alarm clock restart/;    # 重新启动
```

如果你在等待锁的时候报警，你只是把信号缓冲起来然后返回，你会直接回到 flock，因为 Perl 在可能的情况下会自动重起系统调用。跳出去的唯一方法是用 die 抛出一个例外并且让 eval 捕获之。（这样做是可行的，因为例外会退回到调用库的 longjmp(3) 函数，而 longjmp(3) 是真正把你带出重起系统调用的东西。）

我们使用了嵌套的例外陷阱是因为如果 flock 在你的平台上没有实现的话，那么调用 flock 会抛出一个例外，因此你必须确保清理了警告信号。第二个 alarm 0 用于处理这样的情况：信号到达时是在调用 flock 之后但是在到达第一个 alarm 0 之前。没有第二个 alarm，你可能会面对一个很小的冲突条件——不过冲突条件可不会管你的冲突条件是大是小；它们是黑白分明的：要么有，要么无。而我们更希望没有。

16.1.4 阻塞信号

有时候，你可能想在一些关键的代码段里推迟接收信号。你并不想简单地忽略这些信号，只是你做的事情太关键了，因而不能中断。Perl 的 %SIG 散列并不实现信号阻塞，但是 POSIX 模块通过它的调用 sigprocmask(2) 系统调用的接口实现了信号阻塞：

```
use POSIX qw(:signal_h);

$sigset = POSXI::SigSet->new;

$blockset = POSIX::SigSet->new(SIGINT, SIGQUIT, SIGCHLD);

sigprocmask(SIG_BLOCK,  $blockset, $sigset)

or die "Could not block INT, QUIT, CHLD signals: $! \n";
```

一旦上面三个信号都被阻塞了，你就可以毫不担心地执行你的任务了。在你处理完你的关键业务以后，用恢复旧的信号掩码的方法取消信号的阻塞：

```
   sigprocmask( SIG_SETMASK, $sigset)

      or die "Could not restore INT, QUIT, CHLD signals: $!\n";
```

如果阻塞的时候有三个信号中的任何信号到达，那么这时它们会被立即发送。如果有两种或者更多的不同信号在等待，那么它们的发送顺序并没有定义。另外，在阻塞过程中，收到某个信号一次和收到多次是没有区别的。（注：通常是这样。根据最新的规范，可计数信号可能在一些实时系统上有实现，但是我们还没有看到那些系统。）比如，如果你在阻塞 CHLD 信号期间有九个子进程退出，那么你的信号句柄（如果存在）在退出阻塞后仍然只会被调用一次。这就是为什么当你在收割僵死进程的时候，你应该循环到所有的僵死进程都消失。

16.2 文件

你以前可能从未把文件当作一种 IPC 机制，但是它们却占据了进程间通讯的很大一部分份额——远比其他方法的总和份额要大。当一个进程把它的关键数据存放在文件里，而且以后另外一个进程又检索那些数据，那么这就是两个进程在通讯。文件提供了一些这里提到的其他的 IPC 机制所没有的特点：就象纸张埋在地下几千年一样，文件可以比它的作者有更长的保存期。（注：假设我们有人的保存期）。再加上相对而言使用的简单，文件至今仍然流行就一点都不奇怪了。

使用文件在已经消亡的过去和不知何时的未来之间传递信息并不让人奇怪。你在一些永久介质上（比如磁盘）写文件就行了。仅此而已。（如果它包含 HTML，你可能还要告诉一台 web

服务器在那里能找的到。）有趣的问题是如果所有当事人都健在并且试图相互通讯时该怎么办。如果对各自说话的顺序没有一些规定的话，就根本不可能有可靠的交流；这样的规定可以通过文件锁来实现，我们将在下一节介绍。在其后一节里，我们将讨论父进程和其子进程之间存在的特殊关系，这些关系可以让相关的当事人通过对相同文件继承的访问交换信息。

文件当然有其缺点，比如远程访问，同步，可靠性和会话管理等。本章其他节介绍那些着眼于解决这些问题的不同 IPC 机制。

16.2.1 文件锁定

在一个多任务环境里，你需要很小心地避免与其他试图使用你正在用的文件的进程冲突。如果所有进程都只读取文件内容，那么大家相安无事，但是如果有哪怕只有一个进程需要写该文件，那么随后就会发生混乱——除非使用某种排序机制充当交通警察的角色。

绝对不要只是使用文件是否存在（也就是 -e $file）当作文件锁的标志，因为在测试文件名是否存在和你计划的处理（比如创建，打开，或者删除它）之间存在冲突条件。参阅第二十三章，安全，中的"处理冲突条件"，获取更多相关信息。

Perl 的可移植锁定接口是 flock(HANDLE,FLAGS) 函数，在第二十九章，函数，里描述。Perl 只采用那些在最广范围的平台上都能找到的最简单的锁定机制，因此获得了最大的可移植性。这些语意简单得可以在绝大部分系统上使用，包括那些不支持这些传统系统调用的平台，比如 System V 或 Windows NT。（如果你运行的 Microsoft 的系统是早于 NT 的平台，那么你很可能没有这些系统调用支持，就好象你运行 Mac OS X 以前的苹果系统一样。）

锁有两种变体，共享（LOCK_SH 标志）和排它（LOCK_EX 标志）。尽管听着有"排它"的意思，但是进程并不需要服从对文件的锁。也就是说，flock 只是实现了劝告性的锁定，劝告性的锁定，也就意味着锁定一个文件并不阻止其他的进程读取甚至是写入该文件。进程请求一个排它锁只是让操作系统推迟它对文件的处理，直到所有当前的锁持有者，不管是共享锁还是排它锁，都完成操作以后才进行。类似地，如果一个进程请求一个共享锁，它只是推迟处理直到没有排它锁存在。只有所有当事人都使用文件锁机制的时候，你才能安全地访问一个有内容的文件。

因此，flock 缺省时是一个阻塞操作。也就是说，如果你不能立即获取你需要的锁，操作系统会推迟你的处理，直到你能够获得锁为止。下面是如何获取阻塞的共享锁的方法，通常用于读取文件：

```
use Fcntl qw(:DEFAULT :flock);

open(FH, "< filename") or die "can't open filename: $!";
```

```
flock(FH, LOCK_SH)    or die "can't lock filename: $!";

# 现在从 FH 里读取
```

你可以试图请求一个非阻塞的锁，只需要在 flock 请求里加入 LOCK_NB 标志就可以了。如果你不能马上获得锁，那么该函数失败并且马上返回假。下面是例子：

```
   flock(FH, LOCK_FH | LOCK_NB)

      or die "can't lock filename: $!";
```

你除了象我们这样抛出一个例外之外可能还想做点别的事情，但是你肯定不敢对该文件进行任何 I/O 操作。如果你的锁申请被拒绝，你就不应该访问该文件直到你能够拿到锁。谁知道那个文件处于什么样的混乱状态？非阻塞模式的主要目的是让你离开并且在等待期间做些其他的事情。而且它也可以用于生成更友好的交互，比如警告用户说他可能要一段时间才能获取锁，这样用户就不会觉得被抛弃：

```
use Fcntl qw(:DEFAULT :flock);

open(FH, "< filename") or die "can't open filename: $!";

unless (flock(FH, LOCK_SH | LOCK_NB)) {

        local $| = 1;

        print "Waiting for lock on filename...";

        flock(FH, LOCK_SH) or die "can't lock filename: $!";

        print "got it.\n";

}

# 现在从 FH 读数
```

有些人会试图把非阻塞锁放到一个循环里去。非阻塞锁的主要问题是，当你回过头来再次检查的时候，可能其他人已经把锁拿走了，因为你放弃了在队伍里的位置。有时候你不得不排队并且等待。如果你走运的话，可能可以先看看杂志什么的。

锁是针对文件句柄的，而不是文件名。（注：实际上，锁不是针对文件句柄的——他们是针对与文件句柄关联的文件描述符的，因为操作系统并不知道文件句柄。这就意味着我们的所有关于对某文件名没能拿到锁的 die 消息从技术上来讲都是不准确的。不过下面这样的错误信息："I can't get a lock on the file represented by the file descriptor associated with the filehandle originally opened to the path filename, although by now

filename may represend a different file entrely than our handle does" 只能让用户糊涂（更不用说读者了）。当你关闭一个文件，锁自动消除，不管你是通过调用 close 明确关闭该文件还是通过重新打开该句柄隐含的关闭还是退出你的进程。

如果需要获取排它锁（通常用于写），你就得更小心。你不能用普通的 open 来实现这些；如果你用 < 的打开模式，如果文件不存在那么它会失败，如果你用 >，那么它会先删除它处理的任何文件。你应该使用 sysopen 打开文件，这样该文件就可以在被覆盖之前先被锁住。一旦你安全地打开了用于写入的文件（但还没有写），那么先成功获取排它锁，只有这时候文件才被截去。现在你可以用新数据覆盖它。

```
use Fcntl qw(:DEFALUT :flock);

sysopen(FH, "filename", O_WRONLY | O_CREAT)

or die "can't open filename:$!";

flock(FH, LOCK_EX)

or die "can't lock filename:$!";

truncate(FH,0)

or die "can't truncate filename:$!";

# 现在写 FH
```

如果想现场修改文件的内容，那么再次使用 sysopen。这次你请求读写权限，如果必要就创建文件。一旦打开了文件，但是还没有开始读写的时候，先获取排它锁，然后在你的整个事务过程中都使用它。释放锁的最好方法是关闭文件，因为那样保证了在释放锁之前所有缓冲区都写入文件。

一次更新包括读进旧值和写出新值。你必须在单个排它锁里面做两个操作，减少其他进程在你处理之后（甚至之前）读取（马上就不正确了的）数值。（我们将在本章稍后的共享内存的部分再次介绍这个情况。）

```
use Fcntl qw(:DEFAULT :flock);


sysopen(FH, "counterfile", O_RDWR | O_CREAT)

or die "can't open counterfile: $!";

flock(FH, LOCK_EX);
```

```
or die "can't write-lock counterfile: $!";

$counter = <FH> || 0;    # 首先应该 undef

seek(FH, 0, 0)

or die "can't rewind counterfile :$!";

print FH $counter+1, "\n"

or die "can't write counterfile: $!";


# 下一行在这个程序里从技术上是肤浅的，但是一个一般情况下的好主意

truncate(FH, tell(FH))

or die "can't truncate counterfile: $!";

close(FH)

or die "can't close counterfile: $!";
```

你不能锁住一个你还没打开的文件，而且你无法拥有一个施加于多个文件的锁。你能做的是用一个完全独立的文件充当某种信号灯（象交通灯），通过在这个信号灯文件上使用普通的共享和排它锁来提供可控制的对其他东西（文件）的访问。这个方法有几个优点。你可以用一个文件来控制对多个文件的访问，从而避免那种一个进程试图以一种顺序锁住那些文件而另外一个进程试图以其他顺序锁住那些文件导致的死锁。你可以用信号灯文件锁住整个目录里的文件。你甚至可以控制对那些就不在文件系统上的东西的访问，比如一个共享内存对象或者是一个若干个预先分裂出来的服务器准备调用 accept 的套接字。

如果你有一个 DBM 文件，而且这个 DBM 文件没有明确的锁定机制，那么用一个附属的锁文件就是控制多个客户并发访问的最好的方法。否则，你的 DBM 库的内部缓冲就可能与磁盘上的文件之间丢失同步。在调用 dbmopen 或者 tie 之前，先打开并锁住信号灯文件。如果你用 O_RDONLY 打开数据库，那你会愿意使用 LOCK_SH 处理锁定。否则，使用 LOCK_EX 用于更新数据库的排它访问。（同样，只有所有当事人都同意关注信号灯才有效。）

```
use Fcntl qw(:DEFAULT :flock);

use DB_File;    # 只是演示用途，任何 db 都可以
```

```
$DBNAME = "/path/to/database";

$LCK = $DBNAME. ".lockfile";


# 如果你想把数据写到锁文件里，使用  O_RDWR

sysopen(DBLOCK, $LCK, O_RDONLY| O_CREAT)

or die "can't open $LCK:$!";


# 在打开数据库之前必须锁住文件

flock(DBLOCK, LOCK_SH)

or die "can't LOCK_SH $LCK: $!";


tie(%hash, "DB_File", $DBNAME, O_RDWR | O_CREAT)

or die "can't tie $DBNAME: $!";
```

现在你可以安全地对捆绑了的 %hash 做任何你想做的处理了。如果你完成了对你的数据库的处理，那么确保你明确地释放了那些资源，并且是以你请求它们的相反的顺序：

```
untie %hash;    # 必须在锁定文件之前关闭数据库

close DBLOCK;    # 现在可以安全地释放锁了
```

如果你安装了 GNU DBM 库，你可以使用标准的 GDBM_File 模块的隐含锁定。除非最初的 tie 包含 GDBM_NOLOCK 标志，否则该库将保证任意时刻只有一个用户可以写入 GDBM 文件，而且读进程和写进程不能让数据库同时处于打开状态。

16.2.2 传递文件句柄

每当你用 fork 创建一个子进程，那个新的进程就从它的父进程继承所有打开了的文件句柄。用文件句柄做进程间通讯可以很容易先通过使用平面文件来演示。理解文件机制的原理对于理解本章后面的管道和套接字等更奇妙的机制有很大帮助。

下面这个最简单的例子打开一个文件然后开始一个子进程。然后子进程则使用已经为它打开了的文件句柄：

```
open(INPUT, "< /etc/motd")   or die "/etc/motd: $!";
if ($pid = fork) { waitpid($pid, 0);}
else {
        defined($pid)   or die "fork: $!";
        while (<INPUT>) { print "$.: $_" }
        exit;   # 不让子进程回到主代码
}
# INPUT 句柄现在在父进程里位于 EOF
```

一旦用 open 获取了文件的访问权，那么该文件就保持授权状态直到该文件句柄关闭；对文件的权限或者所有人的访问权限对文件的访问没有什么影响。即使该进程后面修改它自己的用户或者组 ID，或者该文件已经把自己的所有权赋予了一个不同的用户或者组，也不会影响已经打开的文件句柄。那些运行在提升权限级别的程序（比如 set-id （SID）程序或者系统守护进程）通常在它们提升以后的权限下打开一个文件，然后把文件句柄传递给一个不能自己打开文件的子进程。

尽管这个机制在有意识地使用的时候非常便利，但如果文件句柄碰巧从一个程序漏到了另外一个程序，那么它就有可能导致安全问题。为避免给所有可能的文件句柄赋予隐含的访问权限，当你明确地用 exec 生成一个新的程序或者通过调用一个透过管道的 open，system，或者 qx//（反钩号）隐含地执行一个新程序的时候，Perl 都会自动关闭任何他已经打开的文件句柄（包括管道和套接字） 。STDIN，STDOUT，和 STDERR 被排除在外，因为他们的主要目的是提供程序之间的联系。所以将文件句柄传递给一个新程序的方法之一是把该文件句柄拷贝到一个标准文件句柄里：

```
open(INPUT, "< /etc/motd")   or die "/etc/motd: $!";
if ($pid = fork) { wait }
else {
        defined($pid)   or die "fork:$!";
        open(STDIN, "<&INPUT")   or die "dup: $!";
        exec("cat", "-n")       or die "exec cat: $!";
}
```

如果你真的希望新程序能够获取除了上面三个之外的文件句柄的访问权限，你也能做到，不过你必须做两件事之一。当 Perl 打开一个新文件（或者管道和套接字）的时候，它检查变量 $^F($SYSTEM_FD_MAX) 的当前设置。如果新文件句柄用的数字文件描述符大于那个 $^F，该描述符就标记为一个要关闭的。否则，Perl 就把它放着，并且你 exec 出来的新的程序就会继承访问。

通常很难预料你新创建的文件句柄是什么，但是你可以在 open 期间暂时把你的最大系统文件描述符数设置的非常大：

```
# 打开文件并且把 INPUT 标记为在 exec 之间可以使用

{

        local $^F = 10_000;

        open(INPUT, "< /etc/motd") or die "/etc/motd: $!";

} # 在范围退出后，恢复旧的 $^F 值
```

现在你所要干的就是让新程序关照你刚刚打开的文件句柄的文件描述符。最干净的解决方法（在那些支持这些的系统上）就是传递一个文件名是刚创建的文件描述符的特殊文件。如果你的系统有一个目录叫 /dev/fd 或者 /proc/$$/fd，里面包含从了 0 到你的系统支持的最大文件描述符数字，那么你就可能可以使用这个方法。（许多 Linux 系统两个都有，但是好象只有 /proc 的版本是正确填充的。 BSD 和 Solaris 喜欢用 /dev/fd。你最好自己看看你的系统，检查一下你的系统是哪种情况。）首先，用我们前面显示的代码打开一个文件句柄并且把它标记成一个可以在 exec 之间传递的句柄，然后用下面的方法分裂进程：

```
if ($pid = fork) { wait }

else {

        defined($pid)       or die "fork: $!";

        $fdfile = "/dev/fd/" . fileno(INPUT);

        exec("cat", "-n", $fdfile) or die "exec cat: $!";

}
```

如果你的系统支持 fcntl 系统调用，你就可以手工骗过文件句柄在 exec 时候的关闭标志了。如果你创建了文件句柄而且还想与你的子进程共享，但是早些时候你还没有意识到想共享句柄的场合下，这个方法特别方便。

```
use Fcntl qw/F_SETFD/;
```

```
fcntl( INPUT, F_SETFD, 0)

or die "Can't clear close-on-exec flag on INPUT: $!\n";
```

你还可以强制一个文件句柄关闭：

```
fcntl(INPUT, F_SETFD, 1)

or die "Can't set close-on-exec flag on INPUT: $!\n";
```

你还可以查询当前状态：

```
use Fcntl qw/F_SETFD F_GETFD/;


printf("INPUT will be %s across exec\n",

fcntl(INPUT, F_GETFD, 1) ? "closed" : "left open");
```

如果你的系统不支持文件系统中的文件描述符名字，而你又不想通过 STDIN，STDOUT，或者 STDERR 传递文件句柄，你还是可以实现，但是你必须给那些程序做特殊的安排。常用的策略是用一个环境变量或者一个命令行选项传递这些描述符号。

如果被执行的程序是用 Perl 写的，你可以用 open 把一个文件描述符转成一个文件句柄。这次不是声明文件名，而是用 "&=" 后面跟着描述符号。

```
if (defined($ENV{input_fdno}) && $ENV{input_fdno}) =~ /^\d$/) {

        open(INPUT, "<&=$ENV{input_fdno}")

        or die "can't fdopen $ENV{input_fdno} for input: $!";

}
```

如果你准备运行的 Perl 子过程或者程序需要一个文件名参数，那么事情就更好办了。你可以利用 Perl 的普通 open 函数（不是 sysopen 或者三个参数的 open）的描述符打开打开特性来自动化这些动作。假如你有一个象下面这样的简单 Perl 程序：

```
#! /usr/bin/perl -p

# nl - 输入的行数

printf "%6d ", $.;
```

再假设你已经安排好让 INPUT 句柄在 exec 之间保持打开状态，你可以这样调用这个程序：

```
$fdspec = '<&=' . fileno(INPUT);

system("nl", $fdspec);
```

或者捕获输出：

```
@lines = `nl '$fdspec' 1;   # 单引号保护 spec 不被 shell 代换
```

不管你是否 exec 另外一个程序，如果你使用一个从 fork 继承过来的文件描述符，那么就有一个小收获。和 fork 拷贝的变量不同的是（那些变量总是复制为相同的是独立的变量），文件描述符在两个进程之间就是同一个。如果一个进程从该句柄中读取数据，那么另一个进程的文件指针（文件位置）也跟着前进，并且两个进程都不能再看到那些数据了。如果它们轮流读取，那么它们就会在文件里相互跳跃。这个特性对于附着在串行设备上的句柄，象管道或者套接字等而言非常直观，因为它们多数是只读设备，里面的数据也是短时存在的。但是磁盘文件的这个特性可能会让你觉得奇怪。如果这是一个问题，那么在进程分裂之后重新打开任何需要独立跟踪的文件。

fork 操作符是源自 Unix 的概念，这就意味着它可能不能在所有非 Unix/非 POSIX 平台上正确实现。尤其是，在 Windows 98（或者更新的版本）上，你只有运行 Perl 5.6 或者更新的版本才能在这些 Microsoft 系统上使用 fork。尽管 fork 在这些系统上是通过同一个程序里的多个并发的执行流实现的，但它也不是那些缺省时共享所有数据的线程；在 fork 里，只有文件描述符是共享的。又见第十七章，线程。

16.3 管道

管道是一个无方向性的 I/O 通道，它可以从一个程序向另外一个传递字节流。管道分命名管道和匿名管道两种。你可能对匿名管道更熟悉，所以我们先介绍它。

16.3.1 匿名管道

如果你给 open 的第二个参数后缀或者前缀一个管道符号，那么 Perl 会给你打开一个管道而不是一个文件。然后剩下的参数就成了一条命令，这条命令会被 Perl 解释成一个进程（或者一个进程集），而你则想从这条命令中输入或者取出数据流。下面就是如何启动一个你想给它写入的子进程的方法：

```
open SPOOLER, "| cat -v | lpr -h 2>/dev/null"
```

```
    or die "can't fork: $!";
local $SIG{PIPE} = sub {die "spooler pipe broke" };
print SPOOLER "stuff\n";
close SPOOLER or die "bad spool: $! $?";
```

这个例子实际上启动了两个进程，我们可以直接向第一个（运行 cat）打印。第二个进程（运行 lpr）则接收第一个进程的输出。在 shell 编程里，这样的技巧通常称为流水线。一个流水线在一行里可以有任意个进程，只要中间的那个明白如何表现得象过滤器；也就是说，它们从标准输入读取而写到标准输出。

如果一条管道命令包含 shell 照看的特殊字符，那么 Perl 使用你的缺省系统 shell（在 Unix 上 /bin/sh）。如果你只启动一条命令，而且你不需要--或者是不想--使用 shell，那么你就可以用打开管道的一个多参数的形式替代：

```
open SPOOLER, "|-", "lpr", "-h"   # 要求 5.6.1
    or die "can't run lpr: $!";
```

如果你重新打开你的程序的标准输出作为到另外一个程序的管道，那么你随后向 STDOUT print 的任何东西都将成为那个新程序的标准输入。因此如果你想给你的程序做成分页输出（注：也就是每次显示一频，而不是希里哗啦一堆），你可以：

```
if (-t STDOUT) {       # 只有标准输出是终端时
        my $pager = $ENV{PAGER}    || 'more';
open( STDOUT, "| $pager")    or die "can't fork a pager: $!";
}
END {
        close(STDOUT)          or die "can't close STDOUT: $!"
}
```

如果你向一个与管道连接的文件句柄写入数据，那么在你完成处理之后，你需要明确地 close 它。那样你的主程序才不会在它的后代之前退出。

下面就是如何启动一个你想读取数据的子进程的方法：

```
open STATUS, "netstat -an 2>/dev/null |"
```

```
or die "can't fork: $!";
while (<STATUS>) {
        next if /^(tcp|udp)/;
        print;
}
close STATUS or die "bad netstat: $! $?";
```

你同样也可以象用在输出里那样，打开一个多命令的输入管道。而且和以前一样，你可以通过使用一个可选的 open 形式避免 shell：

```
open STATUS, "-|", "netstat", "-an"   # 需要 5.6.1
or die "can't runnetstat: $!";
```

不过那样你就得不到 I/O 重定向，通配符扩展或者多命令管道，因为 Perl 得靠你的 shell 做这些事情。

你可能已经注意到你可以使用反钩号实现与打开一个管道读取数据一样的功能：

```
print grep { !/^(tcp|udp)/ } `netstat -an 2>&1`;
die "bad netstat" if $?;
```

尽管反钩号很方便，但是它们必须把所有东西都一次读进内存，所以，通常你打开自己的管道文件句柄然后每次一行或者一条记录地处理文件会更有效些。这样你就能对整个操作有更好的控制，让你可以提前杀死进程。你甚至还可以一边接收输入一边处理，这样效率更高，因为当有两个或者更多进程同时运行的时候，计算机可以插入许多操作。（即使在一台单 CPU 的机器上，输入和输出也可能在 CPU 处理其他什么事的时候发生。）

因为你正在并行运行两个或者更多进程，那么在 open 和 close 之间的任何时刻，子进程都有可能遭受灾难。这意味着父进程必须检查 open 和 close 两个的返回值。只检查 open 是不够安全的，因为它只告诉你进程分裂是否成功，以及（可能还有）随后的命令是否成功启动（只有在最近的版本的 Perl 中才能做到这一点，而且该命令还必须是通过直接分裂的子进程执行的，而不是通过 shell 执行的）。任何在那以后的灾难都是从子进程向父进程以非零退出状态返回的。当 close 函数看到这个返回值，那么它就知道要返回一个假值。表明实际的状态应该从 $?($CHILD_ERROR) 变量里读取。因此检查 close 的返回值和检查 open 的返回值一样重要，如果你往一个管道里写数据，那么你还应该准备处理

PIPE 信号，如果你还没有完成数据的发送，而另外一端的进程完蛋掉，那么系统就会给你发送这个信号。

16.3.2 自言自语

IPC 的另外一个用途就是让你的程序和自己讲话，就象自言自语一样。实际上，你的进程通过管道和一个它自己分裂的拷贝讲话时，它的工作方式和我们上一节里讲的用管道打开很类似，只不过是子进程继续执行你的脚本而不是其他命令。

要想把这个东西提交给 open 函数，你要使用一个包含负号的伪命令。所以 open 的第二个参数看起来就象 "-|" 或者 "|-"，取决于你是想从自己发出数据还是从自己接收数据。和一个普通的 fork 命令一样，open 函数在父进程里返回子进程的进程 ID，而在子进程里返回 0。另外一个不对称的方面是 open 命名的文件句柄名字只在父进程里使用。管道的子进程端根据实际情况要么是挂在 STDIN 上要么是 STDOUT 上。也就是说，如果你用 |- 打开一个"输出到"管道，那么你就可以向你打开的这个文件句柄写数据，而你的子进程将在它的 STDIN 里找到这些数据：

```
if (open(TO, "|-")) {
        print TO $fromparent;
}
else {
        $tochild = <STDIN>;
        exit;
}
```

如果你用 -| 打开一个"来自"管道，那么你可以从这个文件句柄读取数据，而那些数据就是你的子进程往 STDOUT 写的：

```
if (open(FROM, "-|" )) {
        $toparent = <FROM>;
}
else {
        print STDOUT $fromchild;
```

```
        exit
}
```

这个方法的一个常见的应用就是当你想从一个命令打开一个管道的时候绕开 shell。你想这么干的原因可能是你不希望 shell 代换任何你准备传递命令过去的文件名里的元字符吧。如果你运行 Perl 5.6.1 或者更新的版本，你可以利用 open 的多参数形式获取同样的结果。

使用分裂的文件打开的原因是为了在一个假想的 UID 或 GID 下也能打开一个文件或者命令。你 fork 出来的子进程会抛弃任何特殊的访问权限，然后安全地打开文件或者命令，然后充当一个中介者，在它的更强大的父进程和它打开的文件或命令之间传递数据。这样的例子可以在第二十三章的"在有限制的权限下访问命令和文件"节找到。

分裂的文件打开的一个创造性的用法是过滤你自己的输出。有些算法用两个独立的回合来实现要远比用一个回合实现来得简单。下面是一个简单的例子，我们通过把自己的正常输出放到一个管道里模拟 Unix 的 tee(1) 程序。在管道的另外一端的代理进程（我们自己的子过程之一）把我们的输出分发到所声明的所有文件中去。

```
tee("/tmp/foo", "/tmp/bar", "/tmp/glarch");

while(<>) {
        print "$ARGV at line $. => $_";
}

close(STDOUT)    or die "can't close STDOUT:$!";

sub tee {
        my @output = @_;
        my @handles = ();
        for my $path (@output) {
                my $fh;       # open 会填充这些
```

```
        unless (open ($fh, ">", $path)) {
                warn "cannot write to $path: $!";
                next;
        }
        push @handles, $fh;
    }

    # 在父进程的 STDOUT 里重新打开并且返回
    return if my $pid = open(STDOUT, "|-");
    die "cannot fork: $!" unless defined $pid;

    # 在子进程里处理 STDIN
    while(<STDIN>) {
        for my $fh (@handles) {
            print $fh $_ or die "tee output failed:$!";
        }
    }

    for my $fh (@handles) {
        close($fh) or die "tee closing failed: $!";
    }
    exit;       # 不让子进程返回到主循环!
}
```

你可以不停地重复使用这个技巧，而且在你的输出流上放你希望的任意多的过滤器。只要不停调用那个分裂打开 STDOUT 的函数，并且让子进程从它的父进程（它认为是 STDIN）里读取数据，然后把消息输出给流里面的下一个函数。

这种利用分裂后打开文件的自言自语的另外一个有趣的应用是从一个糟糕的函数中捕获输出，那些函数总是把它们的结果输出到 STDOUT。假设 Perl 只有 printf 但是没有 sprintf。那么你需要的就是类似反钩号那样的东西，但却是 Perl 的函数，而不是外部命令：

```
badfunc("arg");              # TMD，跑！

$string = forksub(\&badfunc, "arg");   # 把它当作字串捕获

@lines = forksub(\&badfunc, "arg");    # 当作独立的行


sub forksub {
        my $kidpid = open my $self, "-|";
        defined $kidpid    or die "cannot fork: $!";
        shift->(@_), exit    unless $kidpid;
        local $/       unless wantarray;
        return <$self>;          # 当退出范围的时候关闭
}
```

我们不能说这么做最好，一个捆绑的文件句柄可能更快一点。但是如果你比你的计算机更着急，那么这个方法更容易写代码。

16.3.3 双向通讯

尽管在单向通讯中，用 open 与另外一条命令通过管道运转得很好，但是双向通讯该怎么办？下面这种方法实际上行不通：

```
open(PROG_TO_READ_AND_WRITE, "| some program |")     # 错！
```

而且如果你忘记打开警告，那么你就会完全错过诊断信息：

```
Can't do bidirectional pipe at myprog line 3.
```

open 函数不允许你这么干，因为这种做法非常容易导致死锁，除非你非常小心。但是如果你决定了，那么你可以使用标准的 IPC::Open2 库模块，用这个模块给一个子过程的 STDIN 和 STDOUT 附着两个管道。还有一个 IPC::Open3 模块用于三通 I/O（还允许你捕获子进程的 STDERR），但这个模块需要一个笨拙的 select 循环或者更方便一些的 IO:Select 模块。不过那样你就得放弃 Perl 的缓冲的输入操作（比如 <>，读一行）。

下面是一个使用 open2 的例子：

```
use IPC::Open2;

local (*Reader, *Writer);

$pid = open2(\*Reader, \*Writer, "bc -l");

$sum = 2;

for (1 .. 5) {

        print Writer "$sum * $sum\n";

        chomp($sum = <Reader>);

}

close Writer;

close Reader;

waitpid($pid, 0);

print "sum is $sum\n";
```

你还可以自动激活词法句柄：

```
   my ($fhread, $fhwrite);

   $pid = open2($fhread, $fhwrite, "cat -u -n");
```

这个方法的普遍的问题就是标准 I/O 缓冲实在是会破坏你的好事。即使你的输出文件句柄是自动刷新的（库为你做这些事情），这样另一端的进程将能及时地收到你的数据，但是通常你没法强迫它也返回这样的风格。在一些特殊的情况下我们是很幸运的，bc 可以在管道的模式下运行，而且还会刷新每个输出行。但是只有很少的几条命令是这样设计的，因此这个方法很少管用，除非你自己就是双向管道的另外一端的程序的作者。甚至是简单而且明确

的 ftp 这样的交互式程序也会在这里失败，因为它们不会在管道上做行缓冲。它们只会在一个 tty 设备上这么干。

CPAN 上的 IO:Pty 和 Expect 模块可以在这方面做协助，因为它们提供真正的 tty（实际上是一个真正的伪 tty，不过看起来和真的一样）。它们让你可以获取其他进程的缓冲行而不用修改那些程序。

如果你把你的程序分裂成几个进程并且想让它们都能进行双向交流，那么你不能使用 Perl 的高端管道接口，因为那些接口都是单向通讯的。你需要使用两个低层的 pipe 函数调用，每个处理一个方向的交谈：

```
pipe(FROM_PARENT, TO_CHILD)    or die "pipe: $!";

pipe(FROM_CHILD, TO_PARENT)    or die "pipe:$!";

select((select(TO_CHILD), $| = 1))[0];       # 自动刷新

select((select(TO_PARENT), $| = 1))[0];    # 自动刷新


if ($pid = fork) {

        close FROM_PARENT; close TO_PARENT;

        print TO_CHILD "Parent Pid $$ is sending this\n";

        chomp($line = <FROM_CHILD>);

        print "Parent Pid $$ just read this: `$line'\n";

        close FROM_CHILD; close TO_CHILD;

        waitpid($pid, 0);

} else {

        die "cannot fork: $!" unless defined $pid;

        close FROM_CHILD; close TO_CHILD;

        chomp($line = <FROM_PARENT>);

        print "Child Pid $$ just read this: `$line'\n";

        print TO_PARENT "Child Pid $$ is sending this\n";
```

```
        close FROM_PARENT; close TO_PARENT;
        exit;
}
```

在许多 Unix 系统上，你实际上不必用两次独立的 pipe 调用来实现父子进程之间的全双工的通讯。socketpair 系统调用给在同一台机器上的相关进程提供了双向的联接。所以，除了用两个 pipe 以外，你还可以只用一个 socketpair。

```
use Socket;
socketpair(Child, Parent,AF_UNIX, SOCK_STREAM, PF_UNSPEC)
or die "socketpair: $!";

# 或者让 perl 给你选择文件句柄

my ($kidfh, $dadfh);
socketpair($kidfh, $dadfh, AF_UNIX, SOCK_STREAM, PF_UNSPEC)
or die "socketpair: $!";
```

在 fork 之后，父进程关闭 Parent 句柄，然后通过 Child 句柄读写。同时子进程关闭 Child 句柄，然后通过 Parent 句柄读写。

如果你正在寻找双向通讯的方法，而且是因为你准备交流的对方实现了标准的互联网服务，那么你通常应该先忽略这些中间人并且使用专为那些目的设计的 CPAN 模块。（参阅本章稍后"套接字"节获取它们的列表。）

16.3.4 命名管道

一个命名管道（通常称做 FIFO）是为同一台机器上不相关的进程之间建立交流的机制。"命名"管道的名字存在于文件系统中，实际上就是在文件系统的名字空间中放一个特殊的文件，而在文件背后不是磁盘，而是另外一个进程（注：你可以对 Unix 域套接字干一样的事情，不过你不能对它们用 open）。命名管道只不过是一个有趣的叫法而已。

如果你想把一个进程和一个不相干的进程联接起来，那么 FIFO 就很方便。如果你打开一个 FIFO，你的进程会阻塞住直到对端也有进程打开它为止。因此如果一个读进程先打开 FIFO，那么它会一直阻塞到写进程出现--反之亦然。

要创建一个命名管道，使用一个 POSIX mkfifo 函数--也就是说你用的必须是 POSIX 系统。在 Microsoft 系统上，你就要看看 Win32::Pipe 模块了，尽管看名字好象意思正相反，实际上它创建的是命名管道。(Win32 用户和我们其他人一样用 pipe 创建匿名管道。)

比如，假设你想把你的 .signature 文件在每次读取的时候都有不同的内容。那么只要把它作成一个命名管道，然后在另外一端应一个 Perl 程序守着，每次读取的时候都生成一个不同的数据就可以了。然后每当有任何程序（比如邮件程序，新闻阅读器，finger 程序等等）试图从那个文件中读取数据的时候，该程序都会与你的程序相联并且读取一个动态的签名。

在下面的例子里，我们使用很少见的 -p 文件测试器来判断某人（或某物）是否曾不小心删除了我们的 FIFO。（注：另外一个用途是看看一个文件句柄是否与一个命名的或者匿名的管道相联接，就象 -p STDIN。）如果 FIFO 被删除了，那么就没有理由做打开尝试，因此我们把这个看作退出请求。如果我们使用简单的用 ">$fpath" 模式的 open 函数，那么就存在一个微小的冲突条件：如果在 -p 测试和打开文件之间文件消失了，那么它会不小心创建成为一个普通平面文件。我们也不能使用 "+<$fpath" 模式，因为打开一个 FIFO 用于读写是一个非阻塞式的打开（只对 FIFO 为真）。通过使用 sysopen 并且忽略 O_CREAT 标志，我们可以通过坚决不创建文件来避免这个问题。

```
use Fcntl;      # 我们要 sysopen

chdir;          # 回到家目录

$fpath = '.signature';

$ENV{PATH} .= ":/usr/games";


unless (-p $fpath) {    # 不是一个管道
        if (-e _) {     # 而是其他东西
                die "$0: won't overwrite .signature\n";
        } else {
                require POSIX;
```

```
                POSIX::mkfifo($fpath, 0666) or die "can't mknod $fpath: $!";
                warn "$0: created $fpath as a named pipe\n";
        }
}

while (1) {
        # 如果签名文件被手工删除则退出
        die "Pipe file disappeared" unless -p $fpath;
        # 下一行阻塞住直到有读者出现
        sysopen(FIFO, $fpath, O_WRONLY)
        or die "can't write $fpath: $!";
        print FIFO "John Smith (smith\@host.org)\n", `fortune -s`;
        close FIFO;
        select(undef, undef, undef, 0.2);   #睡眠 1/5 秒
}
```

关闭之后的短暂的睡眠是为了给读者一个机会读取已经写了的数据。如果我们只是马上循环出去然后再次打开 FIFO，而读者还没有完成刚刚发送的数据读取，那么就不会发送 end-of-file，因为写入进程已经成为历史了。我们既要一圈一圈循环，也要在每次叙述过程中让写入者比读者略微慢一些，这样才能让读者最后看到难得的 end-of-file。（而且我们还担心冲突条件？）

16.4. System V IPC

每个人都讨厌 System V IPC。它比打孔纸带还慢，使用与文件系统完全无关少得可怜的名字空间，使用人类讨厌的数字给它的对象命名，并且还常常自己忘记自己的对象，你的系统管理员经常需要用 ipcs(1) 查找那些丢失了的对象并且用 ipcrm(1) 删除它们，还得求老天保佑不要在用光内存以后才发现问题。

尽管有这些痛苦，古老的 Sys IPC 仍然有好几种有效的用途。三种 IPC 对象是共享内存，信号灯和消息。对于传送消息，目前套接字是更好的机制，而且移植性也更好。对于简单的信号灯用途，用文件系统更好。对于共享内存--现在还有点问题。如果你的系统支持，用更现代的 mmap(2) 更好，（注：在 CPAN 上甚至有一个 Mmap 模块。）但是这些实现的质量因系统而异。而且它还需要更小心一些，以免让 Perl 从 mmap(2) 放你的字串的位置再次分配它们。但当程序员准备使用 mmap(2) 的时候，他们会听到那些常用这个的老鸟嘀咕，说自己是如何如何地绕开那些没有"统一缓冲缓存"（或者叫"统一总线捕蝇器"）的系统上的缓冲一致性问题，还有，他们讲他们知道的问题要比描述他们不懂的东西说得还要好， 然后新手就赶快退回 Sys V IPC 并且憎恨所有他们必须使用的共享内存。

下面是一个小程序，它演示了一窝兄妹进程对一个共享内存缓冲的有控制的访问。Sys V IPC 对象也可以在同一台计算机的不相关的进程之间共享，不过那样的话你就得想出它们相互之间找到对方的方法。为了保证安全的访问，我们将为每块内存创建一个信号灯。（注：可能给每块内存创建一对信号灯更现实：一个用于读而另外一个用于写，而且实际上，这就是在 CPAN 上的 IPC::Shareable 模块所用的方法。但是我们想在这里保持简单些。不过我们要承认，如果使用一对信号灯，你就可以利用好 SysV[2] IPC 唯一的优点：你可以在整个信号灯集上执行原子操作，就好象它们是一个单元一样，这一点有时候很有用。）

每当你想获取或者写入新值到共享内存里面，你就必须先通过信号灯这一关。这个过程可能非常乏味，所以我们将把访问封装在一个对象类里面。 IPC::Shareable 更进一步，把它的对象封装在一个 tie 接口里。

下面的程序会一直运行，直到你用一个 Control-C 或者相当的东西终止它：

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

use v5.6.0;    #或者更新

use strict;

use sigtrap wq(die INT TERM HUP QUIT);

my $PROGENY= shift(@ARGV) || 3;

eval { main() };    # 参阅下面的 DESTROY 找原因

die if $@ && $@ !~ /^Caught a SIG/;

print "\nDone.\n";

exit;
```

```
sub main{
        my $mem = ShMem->alloc("Original Creation at " . localtime);
        my (@kids, $child);
        $SIG{CHLD} = 'IGNORE';
        for (my $unborn = $PROGENY; $unborn > 0; $unborn--) {
                if ($child = fork) {
                        print "$$ begat $child\n";
                        next;
                }
                die "cannot fork: $!" unless defined $child;
                eval {
                        while (1) {
                                $mem->lock();
                                $mem->poke("$$ " . localtime)
                                unless $mem->peek =~ /^$$\b/o;
                                $mem->unlock();
                        }
                };
                die if $@ && $@ !~ /^CAught a SIG/;
                exit;   # 子进程退出
        }
        while (1) {
                print "Buffer is ", $mem->get, "\n";
                sleep 1;
```

```
        }
}
```

下面是 ShMem[2] 包，就是上面程序用的东西。你可以把它直接贴到程序的末尾，或者把它放到自己的文件里，（在结尾放一个"1;"）然后在主程序里 require 它。（用到的这两个 IPC 模块以后会在标准的 Perl 版本里找到。）

```
package ShMem;
use IPC::SysV qw(IPC_PRIVATE IPC_RMID IPC_CREAT S_IRWXU);
use IPC::Semaphore;
sub MAXBUF() { 2000 }

sub alloc {   # 构造方法
        my $class = shift;
        my $value = @_ ? shift : ' ';

        my $key = shmget(IPC_PRIVATE, MAXBUF, S_IRWXU) or die "shmget: $!";
        my $sem = IPC::Semaphore->new(IPC_PRIVATE, 1, S_IRWXU| IPC_CREAT)
        or die "IPC::Semaphore->new: $!";
        $sem->setval(0,1)    or die "sem setval: $!";

        my $self = bless {
                OWNER => $$,
                SHMKEY => $key,
                SEMA => $sem,
        } => $class;
```

```
        $self->put($value);
        return $self;
}
```

下面是抓取和存储方法。get 和 put 方法锁住缓冲区，但是 peek 和 poke 不会，因此后面两个只有在对象被手工锁住的时候才能用——当你想检索一个旧的数值并且存回一个修改过的数值，并且所有都处于同一把锁的时候你就必须手工上锁。演示程序在它的 while (1) 循环里做这些工作。整个事务必须在同一把锁里面发生，否则测试和设置就不可能是原子化的，并且可能爆炸。

```
sub get {
        my $self = shift;
        $self->lock;
        my $value = $self->peek(@_);
        $self->unlock;
        return $value;
}

sub peek {
        my $self = shift;
        shmread($self->{SHMKEY}, my $buff=' ', 0, MAXBUF) or die "shmread: $!";
        substr($buff, index($buff, "\0")) = ' ';
        return $buff;
}

sub put {
        my $self = shift;
        $self->lock;
```

```
        $self->poke(@_);
        $self->unlock;
}

sub poke {
        my($self, $msg) = @_;
        shmwrite($self->{SHMKEY}, $msg, 0, MAXBUF) or die "shmwrite: $!";
}

sub lock {
        my $self = shift;
        $self->{SEMA}->op(0,-1,0) or die "semop: $!";
}

sub unlock {
        my $self = shift;
        $self->{SEMA}->op(0,1,0) or die "semop: $!";
}
```

最后，此类需要一个析构器，这样当对象消失的时候，我们可以手工释放那些存放在对象内部的共享内存和信号灯。否则，它们会活得比它们的创造者长，因而你不得不用 ipcs 和 ipcrm （或者一个系统管理员）来删除它们。这也是为什么我们在主程序里精心设计了把信号转换成例外的封装的原因：在那里才能运行析构器，SysV IPC 对象才能被释放，并且我们才不需要系统管理员。

```
sub DESTROY {
        my $self = shift;
        return unless $self->{OWNER} == $$;   #避免复制释放
```

```
        shmctl ($self->{SHMKEY}, IPC_RMID, 0)      or warn "shmctl RMID: $!";
        $self->{SEMA}->remove()          or warn "sema->remove: $!";
}
```

16.5. 套接字

我们早先讨论的 IPC 机制都有一个非常严重的局限：它们是设计用来在运行在同一台计算机上的进程之间通讯用的。(即使有时候文件可以在机器之间通过象 NFS 这样的机制共享，但是在许多 NFS 实现中锁都会奇怪地失败，这样实际上就不可能对文件进行并发访问了。)对于通用目的的网络通讯，套接字是最好的办法。尽管套接字是在 BSD 里发明的，但它们很快就传播到其他类型的 Unix 里去了，并且现在你几乎可以在可以在任何能用的操作系统里找到它。如果你的机器上没有套接字，那么你想使用互联网的话就会碰到无数的麻烦。

利用套接字，你既可以使用虚电路（象 TCP 流）也可以使用数据报（象 UDP 包）。你甚至可以做得更多，取决于你的系统。不过最常见的套接字编程用的是基于互联网际的套接字 TCP，因此我们在这里介绍这种类型的套接字。这样的套接字提供可靠的联接，运行起来有点象双向管道，但是并不局限于本地机器。互联网的两个杀手级的应用，email 和 web 浏览，都几乎是完全依赖于 TCP 套接字的。

你还在不知情的情况下很频繁地使用了 UDP。每次你的机器试图访问互联网上的一台主机，它都向你的 DNS 服务器发送一个 UDP 包请求其真实的 IP 地址。如果你想发送和接收数据报，那么你也可以自己使用 UDP。数据报比 TCP 更经济是因为它们不是面向联接的；也就是说，它们不太象打电话倒是象发信件。但是 UDP 同样也缺乏 TCP 提供的可靠性，这样它就更适合那些你不在乎是否有一两个包丢掉，多出来，或者坏掉，或者是你知道更高层的协议将强制某种程度的冗余（DNS 就是这样）的场合。

还有其他选择，但是非常少见。你可以使用 Unix 域套接字，但是只能用于本地通讯。有许多其他系统支持各种其他的并非基于 IP 的协议。毫无疑问一些地方的一些人会对它们感兴趣，但是我们将只会略微提到它们。

Perl 里面处理套接字的函数和 C 里面的对应系统调用同名，不过参数有些区别，原因有二：首先，Perl 的文件句柄和 C 的文件描述符的工作机制不同；第二，Perl 已经知道它的字串的长度，所以你不需要传递这个信息。参阅第二十九章获取关于与套接字相关的系统调用的详细信息。

老的 Perl 的套接字代码有一个问题是人们会使用硬代码数值做常量传递到套接字函数里，那样就会破坏移植性。和大多数系统调用一样，与套接字相关的系统调用在失败的时候会礼貌而安静地返回 undef，而不是抛出一个例外。因此检查这些函数的返回值是很重要的，

因为如果你传一些垃圾给它们，它们也不会大声叫嚷的。如果你曾经看到任何明确地设置 $AF_INET = 2 的代码，你就知道你要面对大麻烦了。一个更好的方法（好得没法比）是使用 Socket 模块或者是更友善一些的 IO::Socket 模块，它们俩都是标准模块。这些模块给你提供了设置服务器和客户端所需要的各种常量和协助函数。为了最大可能地成功，你的套接字程序应该象下面这样开头（不要忘记给服务器的程序里加 -T 开关打开错误检查）：

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

use strict;

use sigtrap;

use Socket;   # 或者  IO:Socket
```

正如我们在其他地方说明的一样，Perl 依靠你的 C 库实现它的大部分系统特性，而且不是所有系统都支持所有的套接字类型。所以最安全的可能办法就是只做普通的 TCP 和 UDP 套接字操作。比如，如果你希望你的代码有机会能够移植到那些你还没有想过的系统上，千万别假设那台系统支持可靠的顺序报文协议。也别想在不相关的进程之间通过本地 Unix 域套接字传递打开的文件描述符。（的确，你可以在许多 Unix 机器上做那些事情--参阅你本机的 recvmsg(2) 手册页。）

如果你只是想使用一个标准的互联网服务，比如邮件，新闻组，域名服务，FTP，Telnet，Web 等等，那么你不用从零开始，先试者使用现有的 CPAN 模块。为这些目的设计的打了包的模块包括 Net::SMTP（或者 Mail::Mailer），Net::NNTP，Net::DNS，Net::FTP，Net::Telnet，和各种各样的 HTTP 相关的模块。有关 CPAN 上的模块，你可能要查看一下第五节的网络和 IPC，第十五节的 WWW 相关模块，以及第十六节的服务器和守护进程应用。

在随后的一节里，我们将讲述几个简单的客户和服务器的例子，但是我们不会对所使用的函数做过多的解释，因为那样会和我们将在第二十九章里的描述重合很大部分。

16.5.1 网络客户端程序

如果你需要在可能是不同机器之间的可靠的客户端-服务器通讯，那么请用互联网域套接字。

要在什么地方创建一个 TCP 与服务器联接的 TCP 客户端，最简单的方法通常是使用标准的 IO::Socket::INET 模块：

```
use IO::Socket::INET;
```

```
$socket = IO::Socket::INET->new(PeerAddr => $remote_host,

PeerPort  => $remote_port,

Proto => "tcp",

Type => SOCK_STREAM)

or die "Couldn't connect to $remote_host:$remote_port: $!\n";


# 通过套接字发送某些东西，

print $soket "Why don't you call me anymore?\n";


# 读取远端响应

$answer = <$socket>;


# 然后在结束之后终止联接。

close($socket);
```

如果你只有主机名和端口号准备联接，并且愿意在其他域里使用缺省的字段，那么调用这个函数的缩写形式也是很好的：

```
$socket = IO::Socket::INET->new("www.yahoo.com:80")

or die "Couldn't connect to port 80 of yahoo: $!";
```

如果使用基本的 Socket 模块联接：

```
use Socket;


# 创建一个套接字

socket(Server, PF_INET, SOCK_STREAM, getprotobyname('tcp'));
```

```
# 制作远端机器的地址
$internet_addr = inet_aton($remote_host)
or die "Couldn't convert $remote_host into  an Internet address: $!\n";
$paddr = sockaddr_in($remote_port, $internet_addr);

# 联接
connect(Server, $paddr)
or die "Couldn't connect to $remote_host:$remote_port: $!\n";

select((select(Server), $| = 1)[0]);   # 打开命令缓冲

# 通过套接字发送一些东西
print Server "Why don't you call me anymore?\n";

# 读取远端的响应
$answer = <Server>;

# 处理完之后终止联接
close(Server);
```

如果你想只关闭你方的联接，这样远端就可以得到一个 end-of-file，不过你还是能够读取来自该服务器的数据，使用 shutdown 系统调用进行半关闭：

```
# 不再向服务器写数据
shutdown(Server, 1);   # 在 v5.6 里的 Socket::SHUT_WR 常量
```

16.5.2 网络服务器

下面是和前面的客户端对应的服务器。如果用标准的 IO::Socket::INET 类来实现是相当简单的活：

```
use IO::Socket::INET;


$server = IO::Socket::INET->new(LocalPort => $server_port,

Type     => SOCK_STREAM,

Reuse    => 1,

Listen   => 10)   # 或者 SOMAXCONN

or die "Couldn't be a tcp server on port $server_port: $!\n";


while ($client = $server->accept()) {

        # $client 是新联接

}


close($server);
```

你还可以用低层 Socket 模块来写这些东西：

```
use Socket;


# 创建套接字

socket( Server, PF_INET, SOCK_STREAM, getprotobyname('tcp'));


# 为了我们可以快速重起服务器

setsockopt(Server, SOL_SOCKET, SO_REUSEADDR, 1);
```

```
# 制作自己的套接字地址

$my_addr = sockaddr_in($server_port, INADDR_ANY);

bind(Server, $my_addr)

or die "Couldn't bind to port $server_port: $!\n";


# 为来访联接建立一个队列

listen(Server, SOMAXCONN)

or die "Couldn't listen on port $server_port: $!\n";


# 接受并处理任何联接

while(accept(Client, Server)) {

        # 在新建的客户端联接上做些处理

}


close(Server);
```

客户端不需要和任何地址 bind （绑定），但是服务器需要这么做。我们把它的地址声明为 INADDR_ANY，意味着客户端可以从任意可用的网络接口上来，如果你想固定使用某个接口（比如一个网关或者防火墙机器），那么使用该机器的真实地址。（客户端也可以这么干，不过很少必须这么干。）

如果你想知道是哪个机器和你相联，你可以在客户联接上调用 getpeername。这样返回一个 IP 地址，你可以自己把它转换成一个名字（如果你能）：

```
use Socket;

$other_end = getpeername(Client)

or die "Couldn't identify other end: $!\n";
```

```
($port, $iaddr) = unpack_sockaddr_in($other_end);

$actual_ip = inet_ntoa($iaddr);

$claimed_hostname = gethostbyaddr($iaddr, AF_INET);
```

这个方法一般都可以欺骗过去，因为该 IP 地址的所有者可以把它们的反向域名表设置为任何它们想要的东西。为了略微增强信心，从另外一个方向再转换一次：

```
@name_lookup = gethostbyname($claimed_hostname)

or die "Could not reverse $claimed_hostname: $!\n";

@resolved_ips = map { inet_ntoa($_) } @name_lookup[ 4 .. $#name_lookup ];

$might_spoof = !grep { $actual_ip eq $_ } @resolved_ips;
```

一旦一个客户端与你的服务器相联，你的服务器就可以对那个客户端句柄进行 I/O。不过因为你的服务器忙着做这件事，所以它不能在给来自其他客户端的请求提供服务了。为了避免每次只能给一个客户端服务，许多服务器都是马上 fork 一个自己的克隆来处理每次到来的联接。（其他的服务器预先 fork，或者使用 select 系统调用在多个客户端之间复用 I/O。）

```
REQUEST:

while (accept(Client, Server)) {

        if ($kidpid = fork) {

                close Client;       # 父进程关闭不用的句柄

                next REQUEST;

        }

        defined($kidpid)    or die "cannot fork: $!";



        close Server;           # 子进程关闭无用的句柄


        select(Client);       # 打印用的新的缺省句柄
```

```
        $| = 1;          # 自动刷新

        # 预联接的子进程代码与客户端句柄进行 I/O
        $inpu t= <Client>;
        print Client "output\n";   # 或者 STDOUT，一样的东西

        open(STDIN, "<&Client")   or die "can't dup client: $!";
        open(STDOUT, ">&Client")   or die "can't dup client: $!";
        open(STDERR, ">&Client")   or die "can't dup client: $!";

        # 运行计算器，就当一个例子
        system("bc -l");       # 或者任何你喜欢的东西，只要它不会逃逸出 shell

        print "done\n";       # 仍然是给客户端

        clase Client;
        exit;          # 不让子进程回到 accept!
}
```

这个服务器为每个到来的请求用 fork 克隆一个子进程。那样它就可以一次处理许多请求——只要你创建更多的进程。（你可能想限制这些数目。）即使你不 fork，listen 也会允许最多有 SOMAXCONN （通常是五个或更多个）挂起的联接。每个联接使用一些资源，不过不象一个进程用的那么多。分裂出的服务器进程必须仔细清理它们运行结束了的子进程（在 Unix 语言里叫 "zombies" 僵死进程），否则它们很快就会填满你的进程表。在“信号”一节里的讨论的 REAPER 代码会为你做这些事情，或者你可以赋值 $SIG{CHLD} = 'IGNORE'。

在运行其他命令之前，我们把标准输入和输出（以及错误）联接到客户端联接中。这样任何从 STDIN 读取输入并且写出到 STDOUT 的命令都可以与远端的机器交谈——如果没有

重新赋值，这条命令就无法找到客户端句柄——因为客户端句柄缺省的时候在 exec 范围外自动关闭。

如果你写一个网络服务器，我们强烈建议你使用 -T 开关以打开污染检查，即使它们没有运行 setuid 或者 setgid 也这样干。这个主意对那些服务器和任何代表其他进程运行的程序（比如所有 CGI 脚本）都是好主意，因为它减少了来自外部的人威胁你的系统的机会。参阅第二十三章里"操作不安全的数据"节获取更多相关信息。

当写互联网程序的时候有一个额外的考虑：许多协议声明行结束符应该是 CRLF，它可以用好多种方法来声明："\015\12"，或者 "\xd\xa"，或者还可以是 chr(13).chr(10)。对于版本 5.6 的 Perl，说 v13.10 也生成一样的字串。（在许多机器上，你还可以用 "\r\n" 表示 CRLF，不过如果你想向 Mac 上移植，不要用"\r\n"，因为在 Mac 上 "\r\n" 的含义正相反！）许多互联网程序实际上会把只有 "\012" 的字串当行结束符，但那是因为互联网程序总是对它们接收的东西很宽容而对它们发出的东西很严格。（现在只有我们能让人们这么做了...）

16.5.3 消息传递

我们早先提到过，UDP 通讯更少过热但是可靠性也更低，因为它不保证消息会按照恰当的顺序到达——或者甚至不能保证消息能够到达。人们常把 UDP 称做不可靠的数据报协议（Unreliable Datagram Protocol）。

不过，UDP 提供了一些 TCP 所没有的优点，包括同时向一堆目的主机广播或者多点广播的能力（通常在你的本地子网）。如果你觉得自己太关心可靠性并且开始给你的消息系统做检查子系统，那么你可能应该使用 TCP。的确，建立或者解除一个 TCP 联接开销更大，但是如果你可以用许多消息补偿（或者一条长消息）这些，那么也是值得的。

不管怎么说，下面是一个 UDP 程序的例子。它与在命令行声明的机器的 UDP 时间端口联接，如果命令行上没有声明机器名，它与它能找到的任何使用统一广播地址的机器联接。（注：如果还不行，那么运行 ifconfig -a 找出合适的本地广播地址。）不是所有机器都打开了时间服务，尤其是跨防火墙边界的时候，但是那些给你发包的机器会给你按照网络字节序打包发送一个 4 字节的整数，表明它认为的时间值。返回的这个时间值是自 1900 年以来的秒数。你必须减去 1970 年 和 1900 年之间的描述然后才能传给 localtime 或者 gmtime 转换函数。

```
#! /usr/bin/perl

# clockdrift - 与其他系统的时钟进行比较

#           如果没有参数，广播给其他人听
```

```
#         等待 1.5 秒听取回答

use v5.6.0;   # 或者更高版本
use warnings;
use strict;
use Socket;

unshift(@ARGV, inet_ntoa(INADDR_BROCAST))
unless @ARGV;

socket(my $msgsock, PF_INET, SOCK_DGRAM, getprotobyname("udp"))
or die "socket: $!";

# 有些有毛病的机器需要这个。不会伤害任何其他的机器。
setsockopt($msgsock, SOL_SOCKET, SO_BROADCAST, 1)
or die "setsockopt: $!";

my $portno = getservbyname("time", "udp")
or die "no udp time port";

for my $target (@ARGV) {
        print "Sending to $target: $portno\n";
        my $destpaddr = sockaddr_in($portno, inet_aton($target));
        send($msgsock, "x", 0, $destpaddr)
```

```
        or die "send: $!";
}

# 时间服务返回自 1900 年以来以秒计的 32 位时间
my $FROM_1900_TO_EPOCH = 2_208_988_800;
my $time_fmt = "N";       # 并且它以这种二进制格式处理。
my $time_len = length(pack($time_fmt, 1));   # 任何数字都可以

my $inmask = '';   # 存储用于 select 文件号位的字串，
vec($inmask, fileno($msgsock), 1) = 1;

# 等待半秒钟等待输入出现
while (select (my $outmask = $inmask, undef, undef, 0.5)) {
        defined(my $srcpaddr = recv($msgsock, my $bintime, $time_len, 0))
        or die "recv: $!";
        my($port, $ipaddr) = sockaddr_in($srcpaddr);
        my $sendhost = sprintf "%s [%s]",
        gethostbyaddr($ipaddr, AF_INET) || 'UNKNOWN',
        inet_ntoa($ipaddr);
        my $delta = unpack($time_fmt, $bintime) -
        $FROM_1900_TO_EPOCH - time();
        print "Clock on $sendhost is $delta seconds ahead of this one.\n";
}
```

第十七章 线程

并行编程要比看上去要难得多。假设我们从一个烹饪书拿出一条菜谱，然后把它转换成某种几十个厨师可以同时工作的东西。那么你有两个实现方法。

一个方法是给每个厨师一个专用的厨房，给它装备原料和器具。对于那些可以很容易分解的菜谱，以及那些可以很容易从一个厨房转到另外一个厨房的食物而言，这个方法很好用，因为它把不同厨师分隔开，互不影响。

另外，你也可以把所有厨师都放在一个厨房里，然后让他们把菜烧出来，让他们混合使用那些东西。这样可能会很乱，尤其是切肉机开始飞转的时候。

这两个方法对应计算机的两种并行编程方法。第一种是 Unix 系统里典型的多进程模型，这种模型里每个控制线索都有自己的一套资源，我们把它们放在一起叫进程。第二种模型是多线程模型，这种模型里每个控制线索和其他控制线索共享资源。或者有些场合可能（或者必须）不共享（资源）。

我们都知道厨师喜欢掌勺；这一点我们明白，因为只有让厨师掌好勺才能实现我们想让他们干的事。但是厨师也需要有组织，不管用什么方法。

Perl 支持上面两种模式的组织形式。本章我们将把它们称为进程模型和线程模型。

17.1 进程模型

我们不会在这里太多地讨论进程模型，原因很简单：它遍及本书的所有其他部分。Perl 起源于 Unix 系统，所以它浸满了每个进程处理自己的事情的概念。如果一个进程想并行处理某些事情，那么逻辑上它必须启动一个并行的进程；也就是说，它必须分裂一个重量级的新进程，它和父进程共享很少东西，除了一些文件描述符以外。（有时候看起来父进程和子进程共享很多东西，但大多数都只是在子进程中复制父进程并且在逻辑概念上并没有真正共享什么东西。操作系统为了偷懒也会强制那种逻辑分离的，这种情况下我们叫它写时拷贝（copy-on-write）语意，但是如果我们不首先逻辑上分开，我们实际上就根本不能做拷贝。）

由于历史原因，这种工业级的多进程观点在 Microsoft 系统上引起一些问题，因为 Windows 没有完善的多进程模型（并且老实说，它并不常依靠并发编程技术）。而且它通常采用一种多线程的方法。

不过，通过不懈的努力，Perl 5.6 现在在 Windows 里实现了 fork 操作，方法是在同一个进程里克隆一个新的解释器对象。这意味着本书其余部分使用 fork 的例子现在在 Windows 里都可以使用了。克隆出来的解释器与其他解释器共享不可改变的代码段但是有自己的数据段。（当然，对那些可能不理解线程的 C 库可能仍然有问题。）

这种多进程的方法被命名为 ithread，是"interpreter threads"（解释器线程）的缩写。实现 ithread 的最初的动力就是在 Microsoft 系统上模拟 fork。不过，我们很快就意识到，尽管其他解释器是作为独立的线程运行的，它们仍然在同一个进程里运行，因此，要让这些独立的解释器共享数据是相当容易的，尽管缺省时它们并不共享数据。

这种做法和典型的线程模型是相对的，那种模型里所有东西都是共享的，而且你还得花点力气才能不共享一些东西。但是你不能把这两种模型看作是完全分离的两个模式，因为它们都试图在同一条河上架桥通路；只不过它们是从相对的河两岸分别开工的。任何解决并发进程问题的方法实际上最后都是某种程度的共享和某种程度的自私。

所以从长远来看，ithread 的目的是允许你需要或者想要的尽可能多的共享。不过，到我们写这些为止，ithread 唯一的用户可见的接口是在 Perl 的 Microsoft 版本里面的 fork 调用。我们最终认为，这个方法能比标准的线程方法提供更干净的程序。原则上来说，如果你假设每个人都拥有他该拥有的东西，那么这样的系统更容易运转，相比之下每个人都拥有所有的东西的系统要难于运转得多。不象资本主义经济那样没有共享的东西，也不是共产主义经济那样所有东西都共享。这些东西有点象中庸主义。类似社会主义。不过对于一大堆人而言，只有在拥有一个很大的绞肉机，而且"厨师长"认为它拥有所有东西的时候共享全部东西才可行。

当然，任何计算机的实际控制都是由那个被称做操作系统的独裁者控制的。不过是聪明的独裁者知道什么时候让人们认为它们是资本主义--以及什么时候让他们觉得是共产主义。

17.2 线程模型

并发处理的线程模模型是在版本 5.005 里第一次以一种试验特性介绍到 Perl 里来的。（这里"线程模型"的意思是那些缺省时共享数据资源的线程，不是版本 5.6 里的 ithreads。）从某种角度来说，这个线程模型即使在 5.6 里也仍然是一个实验的模型，因为 Perl 是一门复杂的语言，而多线程即使在最简单的语言里也可能搞得一团糟。在 Perl 的语意里仍然有隐蔽的地方和小漏洞，这些地方不能和共享所有东西的概念完全融合在一起。新的 ithread 模型是一个绕开这些问题的尝试，并且在将来的某个时间，现在的线程模型可能会被包容在 ithread 模型里（那时我们就可以有一个"缺省时共享所有你能共享的东西"的 ithread 接口）。不过，尽管有瑕疵，目前的"试验性"的线程模型仍然在现实世界中的许多方面用得上，那种情况下你只能换一种你不想要的更笨拙的方法。你可以用线

程化的 Perl 写出足够坚固的应用，不过你必须非常小心。如果你能想出一种用管道而不是用共享数据结构的方法解决你的问题的话，那么你至少应该考虑使用 fork。

不过如果多个任务能够容易且有效地访问同一组数据池（注：上一章讨论的 System V 里的共享内存的模型并不算"容易且有效"），那么有些算法可以更容易地表达。这样就可以把代码写的更少和更简单。并且因为在创建线程的时候内核并不必为数据拷贝内存页表（甚至写时拷贝（copy-on-write）都不用），那么用这种方法启动一个任务就应该更快些。类似，因为内核不必交换内存页表，环境切换也应该更快一些。（实际上，对于用户级线程而言，内核根本不用参与——当然，用户级的线程有一些内核级的线程没有的问题。）

这些可是好消息。那么现在我们要做些弃权声明。我们已经说过线程在 Perl 里是某种试验特性，而且即使它们不再是试验特性了，那么线程编程也是非常危险的。一个执行流能够把另外一个执行流的数据区捅得乱七八糟的能力可以暴露出比你能想象得更多的导致灾难的机会。你可能会对自己说，"那很容易修理，我只需要在任何共享的数据上加锁就可以了。"不错，共享数据的锁是不可缺少的，不过设计正确的锁协议是臭名昭著地难，协议的错误会导致死锁或者不可预料的结果。如果你在程序里有定时问题，那么使用线程不仅会恶化它们，而且还让他们难于跟踪。

你不仅要确保你自己的共享数据的安全，而且你还要保证这些数据在所有你调用的 Perl 模块和 C 库里安全。你的 Perl 代码可以是 100% 线程安全的，但是如果你调用了一个线程不安全的模块或者 C 的子过程，而又没有提供你自己的信号灯保护，那么你完了。在你能证明之前，你应该假设任何模块都是现成不安全的。那些甚至包括一些标准的模块。甚至是它们的大多数。

我们有没有让你泄气？没有？然后我们还要指出，如果事情到了调度和优先级策略的份上，你还很大程度上依赖你的操作系统的线程库的慈悲。有些线程库在阻塞的系统调用的时候只做线程切换。有些线程库在某个线程做阻塞的系统调用的时候阻塞住整个进程。有些库只在时间量超时（quantum expiration）的时候才切换线程（线程或者进程）。有些库只能明确地切换线程。

哦，对了，如果你的进程接收到一个信号，那么信号发送给哪个线程完全是由系统决定的。

如果想在 Perl 里写线程程序，你必须制作一个特殊的 Perl 的版本，遵照 Perl 源程序目录里的 README.threads 文件的指示就可以了。这个特殊的 Perl 版本几乎是可以肯定要比标准的版本慢一些的。

请不要觉得你只要知道了其他线程模型（POSIX，DEC，Microsoft，等等）的编程特点就认为自己认识了 Perl 的线程的运转模式。就象 Perl 里的其他东西一样， Perl 就是 Perl，它不是 C++ 或者 Java 或者其他什么东西。比如，Perl 里没有实时线程优先级（也

没有实现它们的方法）。也没有互斥线程。你只能用锁定或者是 Thread::Semaphore 模块或者 cond_wait 设施。

还没泄气？好，因为线程实在是酷。你准备迎接一些乐趣吧。

17.2.1 线程模块

目前的 Perl 的线程接口是由 Thread 模块定义的。另外还增加了一个新的 Perl 关键字，lock 操作符。我们将在本章稍后描述 lock 操作符。其他标准的线程模块都是在这个基本接口的基础上制作的。

Thread 模块提供了下列类模块：

模块	用途
new	构造一个新的 Thread。
self	返回我的当前 Thread 对象。
list	返回一个 Thread 列表。

并且，对于 Thread 对象，它还提供了这些对象方法：

模块	用途
join	结束一个线程（传播错误）
eval	结束一个线程（捕获错误）
equal	比较两个线程是否相同
tid	返回内部线程 ID

另外，Thread 模块还提供了这些重要的函数：

函数	用途
yield	告诉调度器返回一个不同的线程
async	通过闭合域构造一个 Thread
cond_signal	只唤醒一个正在 cond_wait() 一个变量的线程
cond_braodcast	唤醒所有可能在 cond_wait() 一个变量的线程
cond_wait	等待一个变量，直到被一个 cond_signal() 打断或变量上有 cond_broadcast()。

17.2.1.1 创建线程

你可以用两种方法之一派生线程，要么是用 Thread->new 类方法或者使用 async 函数。不管那种方法，返回值都是一个 Thread 对象。Thread->new 接收一个要运行的表示某函数的代码引用以及传给那个函数的参数：

```
use Thread;

...

$t = Thread->new(\&func, $arg1, $arg2);
```

你通常会想传递一个闭合域做第一个参数而省略其他的参数：

```
my $something;

$t = Thread->new( sub {say{$something} } );
```

对这种特殊的例子，async 函数提供了一些符号上的解放（就是语法糖）：

```
use Thread qw(async);

...

my $something;

$t = async {

   say($something);

};
```

你会注意到我们明确地输入了 async 函数。你当然可以用全称的 Thread::async 代换，不过那样你的语法糖就不够甜了。因为 async 只包含一个闭合域，所以你想放进去的任何东西都必须是一个在传入是的范围的一个词法变量。

17.2.1.2 线程删除

一旦开始——并且开始遭受你的线程库的反复无常——该线程将保持运行直到它的顶层函数（就是你传给构造器的函数）返回。如果你想提前终止一个线程，只需要从那个顶层函数中 return 就行了。（注：不要调用 exit！那样就试图停止你的整个进程，并且可能会成功。但实际上该进程直到所有线程都退出之后才能退出，而且有些线程在 exit 的时候可能拒绝退出。我们稍后有更多内容。）

现在是你的顶层子过程返回的好时机，但是它返回给谁呢？派生这个线程的那个线程可能已经转去做别的事情，并且不再等待一个方法调用的响应了。答案很简单：该线程等待某个过程发出一个方法调用并真正等待一个返回值。那个调用的方法叫 join，因为它概念上是把两个线程连接回一个：

```
$retval = $t->join();       # 结束线程 $t
```

join 的操作是对子进程的 waitpid 的回忆。如果线程已经停止，join 方法马上返回该线程的顶层子过程的返回值。如果该线程还没有完蛋，join 的动作就象一个阻塞调用，把调用线程不确定地挂起。（这儿没有超时机制。）当该线程最终结束时，join 返回。

不过，和 waitpid 不一样，waitpid 只能终止该进程自己的子过程，一个线程可以 join 任何同一个进程内的其他线程。也就是说，与主线程或者父线程连接并不是必要的。唯一的限制是线程不能 join 它自己（那样的话就好象安排你自己的葬礼。），而且一个线程不能 join 一个已经连接过的线程（那样就好象两个葬礼主持在尸体上扭打）。如果你试图做这两件事之一那么就会抛出一个例外。

join 的返回值不一定是标量值——它可以是一个列表：

```
use Thread 'async';


$t1 = async {
        my Sstuff = getpwuid($>);
        return @stuff;
};


$t2 = async {
        my $motd = `cat /etc/modt`;
        return $motd;
};


@retlist = $t1->join();
```

```
$retval = $t2->join();


print "1st kid returned @retlist\n";

print "2nd kid returned $retval\n";
```

实际上，一个线程的返回表达式总是在列表环境里计算的，即使 jion 是在一个标量环境里调用的也如此，那种情况下返回列表的最后一个值。

17.2.1.3 捕获来自 join 的例外

如果一个线程带着一个未捕获的例外终止，不会立即杀死整个程序。这就有点头疼了。相比之下，如果一个 join 在那个线程上运行，那么 join 本身会抛出例外。在一个线程上使用了 join 意味着愿意传播该线程抛出的例外。如果你宁可到处捕获这些例外，那么使用 eval 方法，它就象一个内建的配对儿，导致例外被放进 $@：

```
$retval = $t->eval(); # 捕获 join 错误 if ($@) { warn "thread failed: $@"; }
else { print "thread returned $retval\n"; }
```

你在实际中可能还是只想在创建被连接的线程的那个线程里连接该线程——尽管我们没有这个影响的规则。也就是说，你只从父线程里终止子线程。这样可以比较方便地跟踪你应该在那里操作哪个例外。

17.2.1.4 detach 方法

这是另外一个终止线程的方法，如果你不准备稍后 join 一个线程以获取它的返回值，那么你可以对它调用 detach 方法，这样 Perl 会为你清理干净。然后该线程就不能再被连接了。它有点象 Unix 里的一个进程继承给 init，不过在 Unix 里这么做唯一的方法是父进程退出。

detach 方法并不把该线程"放到后台"；如果你试图退出主程序并且一个已发配的线程仍在运行，那么退出过程将挂起，直到该线程自己退出。更准确一点说，detach 只是替你做清理工作。它只是告诉 Perl 在该线程退出之后不必再保留它的返回值和退出状态。从某种意义上来说，detach 告诉 Perl 当该线程结束后做一个隐含的 join 并且丢掉结果。这一点很重要，如果你既不 join 也不 detach 一个返回巨大列表的线程，那么那部分存储空间将直到结束时都不能使用，因为 Perl 将不得不为以后（在我们的例子里是非常以后）可能会出现的某个家伙想 join 该线程的机会挂起。

在一个发配了的子线程里抛出的例外也不再通过 join 传播，因为它们将不再被使用。在顶层函数里合理使用 eval {}，你可能会找到其他汇报错误的方法。

17.2.1.5 标识线程

每个 Perl 线程都有一个唯一的线程标识，由 tid 对象方法返回：

```
$his_tidno = $t1->tid();
```

一个线程可以通过 Thread->self 调用访问它自己的线程对象。不要把这个和线程 ID 混淆：要获得自身的线程 ID，一个线程可以这样：

```
$mytid = Thread->self->tid();   #$$ 是线程，和以前一样
```

要拿一个线程对象和另外一个做比较，用下列之一：

```
Thread::equal($t1, $t2)

$t1->equal($t2)

$t1->tid() == $td->tid()
```

17.2.1.6 列出当前线程

你可以用 Thread->list 类方法调用在当前进程获取一个当前线程对象的列表。该列表包括运行着的现成和已经退出但还未连接的线程。你可以在任何线程里做这个工作：

```
for my $t (Thread->list()) { printf "$t has tid = %d\n", $t->tid(); }
```

17.2.1.7 交出处理器

Thread 模块支持一个重要的函数，叫 yield。它的工作是令调用它的线程放弃处理器。不幸的是，这个函数具体干的事情完全依赖于你所的线程的实现方式。不管怎样，我们还是认为它是一个偶然放弃 CPU 的控制的很好的手势。

```
use Thread 'yield';

yield();
```

你不必使用圆括号。从语法上来讲，这样可能更安全，因为这样能捕获看起来无法避免的“yeild”的错误输入：

```
use strict;

use Thread 'yield';

yeild;          # 编译器出错，然后才过去

yield;          # 正确
```

17.2.2 数据访问

到目前为止我们看到的东西都不算太难，不过很快就不是那样了。我们到目前为止所做的任何事情实际上都没有面对线程的并行本性。访问共享数据就会结束上面的状况了。

Perl 里的线程代码在面对数据可视性问题的时候和任何其他的 Perl 代码都要经受一样的约束。全局量仍然是通过全局符号表来访问的，而词汇变量仍然是通过某些包含它的词法范围（便签本）访问的。

不过，因为程序里存在多个线程的控制，所以带来一些新的问题。Perl 不允许两个线程同时访问同一个全局变量，否则它们就会踩着对方的脚。（踩得严重与否取决于访问的性质）。相似的情况还有两个线程不能同时访问同一个词法变量，因为如果词法范围在线程使用的闭合域的外面声明，那么它的性质和全局量类似。通过用子过程引用来启动线程（用 Thread->new）而不是用闭合域启动（用 async），可以限制对词法变量的访问，这样也许能满足你的要求。（不过有时候也不行。）

Perl 给一些内建特殊变量解决了这个问题，比如 $! 和 $_ 和 @_ 等等，方法是把它们标记为线程相关的数据。糟糕的是所有你日常使用的基础包变量都没有受到保护。

好消息是通常你完全不必为你的词法变量担心——只要它们是在当前线程内部声明的；因为每个线程在启动的时候都会生成自己的词法范围的实例，这是与任何其他的线程隔离的。只有在词法变量在线程之间共享的时候你才需要担心，（比如，四处传递引用，或者在多线程里运行的闭合域里引用词法变量）。

17.2.2.1 用 lock 进行访问同步

如果同一时间有多个用户能够访问同一条目，那么就会发生冲突，就象十字路口一样。你唯一的武器就是仔细地锁定。

内建的 lock 函数是 Perl 用于访问控制的的红绿灯机制。尽管 lock 是各种关键字之一，但它是那种层次比较低的，因为如果编译器已经发现用户代码中有一个 sub lock {} 定义存在，那么就不会使用内建的 lock。这是为了向下兼容，不过， CORE::LOCK 总是内建

的函数。（在不是为线程使用制作的 perl 版本里调用 lock 不是个错误，只是一个无害的“无动作”，至少在最近的版本里如此。）

就好象 flock 操作符只是阻塞其它的 flock 的实例，而不是实际 I/O 一样，lock 也只是阻塞其它 lock 的实例，而不是普通的数据访问。实际上，它们也是劝告性锁定。就象交通灯一样。（注：有些铁路十字路口是强制性锁（那些有门的），有些家伙认为 lock 也应该是强制性的。不过想象一下，如果现实世界中的每个十字路口都有升降杆是多么可怕。）

你可以锁住独立的标量变量，整个数组和整个哈希。

```
lock $var;

lock @values;

lock %table;
```

不过，在一个聚集上使用 lock 并非隐含的对该聚集的每一个标量元素都锁定：

```
    lock @values;       # 在线程 1

    ...

    lock $values[23];     # 在线程 2 -- 不会阻塞!
```

如果你锁定一个引用，那么也自动锁住了对引用的访问。也就是说，你获得一个可以释放的析引用。这个技巧很有用，因为对象总是隐藏在一个引用后面，并且你经常想锁住对象。（并且你几乎从来不会想锁住引用。）

当然，交通灯的问题是它们有一半时间是红灯，这时候你只能等待。同样，lock 也是阻塞性调用——你的线程会挂起，直到获得了锁。这个过程中没有超时机制。也没有解锁设施，因为锁是动态范围对象。它们持续到它们的语句块，文件或者 eval 的结束。如果它们超出了范围，那么它们被自动释放。

锁还是递归的。这意味着如果你在一个函数里锁住了一个变量，而且该函数在持有锁的时候递归，那么同一个线程可以再次成功地锁住该变量。当所有拥有锁的框架都退出以后，锁才最终被删除。

下面是一个简单的演示程序，看看如果没有锁，世界将会怎样。我们将用 yield 强制一次环境切换以显示在优先级调度的时候也可能偶然发生的这类问题：

```
use Thread qw/async yield/;

my $var = 0;
```

```
sub abump {
        if ($var == 0) {
                yield;
                $var++;
        }
}

my $t1 = new Thread \&abump;
my $t2 = new Thread \&abump;

for my $t ($t1, $t2) { $t->join}
print "var is $var\n";
```

这段代码总是打印 2（某种意义上的"总是"），因为我们在看到数值为 0 后决定增加数值，但在我们增加之前，另外一个线程也在做一样的事情。

我们可以在检查 $var 之前用一个微乎其微的锁来修补这个冲突。下面的代码总是打印 1：

```
sub bump {
        lock $var;
        if ($var == 0) {
                yield;
                $var++;
        }
}
```

请记住我们没有明确的 unlock 函数。要控制解锁，只需要增加另外一个嵌套的范围层次就行了，这样锁就会在范围结束后释放。

```
sub abump {
```

```
        {
                lock $var;
                if ($var == 0) {
                        yield;
                        $var++;
                }
        } # 锁在这里释放
        # 其他不用锁定 $var 的代码
}
```

17.2.2.2 死锁

死锁是线程程序员的毒药，因为很容易偶然地就死锁了，但即使你努力做好却很难避免。下面是一个死锁的简单的例子：

```
my $t1 = async { lock $a; yield; lock $b; $a++; $b++ };
my $t2 = async { lock $b; yield; lock $a; $b++; $a++ };
```

解决方法是对于所有需要某个锁集合的当事方，都必须按照相同的顺序获取锁。

把你持有锁的时间最小化也是很好的做法。（至少出于性能的考虑也是好的。但是如果你只是为了减少死锁的风险，那么你所做的只是让复现问题和诊断问题变得更难。）

17.2.2.3 锁定子过程

你可以在一个子过程上加一把锁：

```
lock &func;
```

和数据锁不一样，数据锁只有劝告性锁，而子过程锁是强制性的。除了拥有锁的线程以外其它线程都不能进入子过程。

考虑一下下面的代码，它包含一个涉及 $done 变量的冲突条件。（yield 只是用于演示）。

```
use Thread qw/async yield/;
```

```
my $done = 0;
sub frob {
        my $arg = shift;
        my $tid = Thread->self->tid;
        print "thread $tid: frob $arg\n";
        yield;
        unless ($done) {
                yield;
                $done++;
                frob($arg + 10);
        }
}
```

如果你这样运行：

```
my @t;
for my $i (1..3) {
        push @t, Thread->new(\&frob, $i);
}
for (@t) { $_->join}
print "done is $done\n";
```

下面是输出（哦，有时候是这样的——输出是不可预料的）：

```
thread 1: frob 1
thread 2: frob 2
thread 3: frob 3
thread 1: frob 11
```

```
thread 2: frob 12

thread 3: frob 13

done is 3
```

不过如果你这么运行：

```
for my $i (1..3) {
        push @t, async {
                lock &frob;
                frob($i);
        };
}
for (@t) { $_->join }
print "done is $done\n";
```

输出是下面的东西：

```
thread 1: frob 1

thread 1: frob 11

thread 2: frob 2

thread 3: frob 3

done is 1
```

17.2.2.4 locked 属性

尽管你必须遵守子过程锁，但是没有什么东西让你一开始就锁住他们。你可以说锁的位置是劝告性的。不过有些子过程确实需要在调用之前把它们锁住。

子过程的 locked 属性就是干这个的。它比调用 lock &sub 快，因为它在编译时就知道了，而不只是在运行时。但是其性质和我们提前明确地锁住它是一样的。语法如下：

```
   sub frob : locked {
```

```
    # 和以前一样
}
```

如果你有函数原形，它放在名字和任意属性之间：

```
sub frob ($) : locked {
    # 和以前一样
}
```

17.2.2.5. 锁定方法

在子过程上自动加锁的特性是非常棒的，但有时候杀伤力太大。通常来说，当你调用一个对象方法时，是否有多个方法同时运行并没有什么关系，因为它们都代表不同的对象运行。因此你真正想锁住的是其方法正在被调用的那个对象。向该子过程里增加一个 method 属性可以实现这个目的：

```
sub frob : locked method {
    # 和以前一样
}
```

如果它被当作一个方法调用，那么正在调用的对象被锁住，这样就可以对该对象进行串行访问，但是允许在其他对象上调用该方法。如果该方法不是在对象上调用的，该属性仍然力图做正确的事情：如果你把一个锁住的方法当作一个类方法调用（Package->new 而不是 $obj->new），那么包的符号表被锁住。如果你把一个锁住的方法当作普通子过程调用，Perl 会抛出一个错误。

17.2.2.6 条件变量

条件变量允许一个线程放弃处理器，直到某些条件得到满足。当你需要比锁能提供的更多控制机制的时候，条件变量是在线程之间提供协调的点。另一方面，你并不需要比锁有更多过荷的东西，而条件变量就是带着这些思想设计的。你只是用普通锁加上普通条件。如果条件失败，那么你必须通过 cond_wait 函数采取特殊的措施；但是我们很有可能能成功，因为在一个设计良好的应用里，我们不应该在当前的条件上设置瓶颈。

除了锁和测试，对条件变量的基本操作是由发送或者接收一个“信号”事件（不是 %SIG 意义上的真正的信号）组成的。你要么推迟你自己的执行以等待一个事件的到来，要么发送一

条事件以唤醒其他正在等待事件到来的线程。Thread 模块提供了三个不可移植的函数做这些事情： cond_wait，cond_signal，和 cond_broadcast。这些都是比较原始的机制，在它们的基础上构造了更抽象的模块，比如 Thread::Queue 和 Thread::Semaphore。如果可能的话，使用那些抽象可能更方便些。

cond_wait 函数接受一个已经被当前的线程锁住的变量，给那个变量解锁，然后阻塞住直到另外一个线程对同一个锁住了的变量做了一次 cond_signal 或者 cond_broadcast。

被 cond_wait 阻塞住的变量在 cond_wait 返回以后重新锁住。如果有多个线程在 cond_wait 这个变量，那么只有一个线程重新阻塞，因为它们无法重新获得变量的锁。因此，如果你只使用 cond_wait 进行同步工作，那么应该尽快放弃变量锁。

cond_signal 函数接受一个已经被当前线程锁住的变量，然后解除一个当前正在 cond_wait 该变量的线程的阻塞。如果不止一个线程阻塞在对该变量的 cond_wait 上，只有解除一个的阻塞，而且你无法预料是哪个。如果没有线程阻塞在对该变量的 cond_wait 上，该事件被丢弃。

cond_broadcast 函数运行得象 cond_signal，但是解除所有在锁住的变量的 cond_wait 的线程的阻塞，而不只是一个。（当然，仍然是某一时刻只有一个线程可以拥有锁住的变量。）

cond_wait 应该是一个线程在条件没有得到满足后的最后的手段。cond_signal 和 cond_broadcast 表明条件已经改变了。我们假设各个事件的安排是这样的：锁定，然后检查一下看看是否满足你需要的条件；如果满足，很好，如果不满足， cond_wait 直到满足。这里的重点是放在尽可能避免阻塞这方面的。（在对付线程的时候通常是个好建议。）

下面是一个在两个线程之间来回传递控制的一个例子。千万不要因为看到实际条件都在语句修饰词的右边而被欺骗；除非我们等待的条件是假的，否则决不会调用 cond_wait。

```
use Thread qw(async cond_wait cond_signal);

my $wait_var = 0;

async {

        lock $wait_var;

        $wait_var = 1;

        cond_wait $wait_var until $wait_var == 2;

        cond_signal($wait_var);
```

```
        $wait_var = 1;
        cond_wait $wait_var until $wait_var == 2;
        cond_signal($wait_var);
};

async {
        lock $wait_var;
        cond_wait $wait_var   until $wait_var == 1;
        $wait_var = 2;
        cond_signal($wait_var);
        cond_wait $wait_var   until $wait_var == 1;
        $wait_var = 2;
        cond_signal($wait_var);
        cond_wait $wait_var    until $wait_var == 1;
};
```

17.2.3 其他线程模块

有几个模块是在基本的 cond_wait 上构造的。

17.2.3.1 队列（queue）

标准的 Thread::Queue 模块提供了一个在线程之间传递对象而又不用担心锁定和同步问题的方法。它的接口更简单：

方法	用途
new	构造一个 Thread::Queue
equeue	向队列结尾压入一个或更多标量
dequeue	把队列头的第一个标量移出。如果队列里没有内容了，那么 dequeue 阻塞。

请注意，队列和普通管道非常相似，只不过不是发送字节而是传递整个标量，包括引用和赐福过的对象而已！

下面是一个从 perlthrtut 手册页来的一个例子：

```
use Thread qw/async/;

use Thread::Queue;


my $Q = Thread::Queue->new();

async {

        while (defined($datum = $Q->dequeue)) {

                print "Pulled $datum from queue\n";

        }

};


$Q->enqueue(12);

$Q->enqueue("A", "B", "C");

$Q->enqueue($thr);

sleep 3;

$Q->enqueue(\%ENV);

$Q->enqueue(undef);
```

下面是你获得的输出：

```
Pulled 12 from queue

Pulled A from queue

Pulled B from queue

Pulled C from queue
```

```
Pulled Thread=SCALAR(0x8117200) from queue

Pulled HASH(0x80dfd8c) from queue
```

请注意当我们通过一个 async 闭合域启动一个异步线程的时候 $Q 在范围里是怎样的。线程和 Perl 里的其他东西一样遵守同样的范围规则。如果 $Q 在 async 调用之后才声明，那么上面的例子就不能运行了。

17.2.3.2. 信号灯

Thread::Semaphre 给你提供了线程安全的计数信号灯对象，你可以用它来实现你自己的 p() 和 v() 操作。因为我们大部分人都不把这些操作和荷兰语的 passeer （"回合"）和 verlaat（"树叶"）联系在一起，所以此模块把这些操作相应称做 "向下"和"向上"。（在有些文化里，它们叫"等"和"信号"。）此模块支持下面的方法：

方法	用途
new	构造一个新的 Thread::Semaphore。
down	分配一个或更多项目。
up	析构一个或者更多项目。

new 方法创建一个新的信号灯并且把它初始化为声明的初始计数。如果没有声明初始数值，则该信号灯的初始值设置为 1。（数字代表某些条目的"池"，如果所有数字都分配完了则它们会"用光"。）

```
use Thread::Semaphore;

$mutex = Thread::Semaphore->new($MAX);
```

down 方法把信号灯的计数值减去所声明的数值，如果没有给出此数值则为 1。你可以认为它是一个分配某些或者所有资源的动作。如果信号灯计数减到零以下，这个方法会阻塞住直到信号灯计数等于或者大于你要求的数量。用下面的方法调用它：

```
$mutex->down();
```

up 方法给该信号灯的计数值加指定的数值，如果没有给出此数值则为 1。你可以认为这是一个释放原先分配的资源的动作。这样的操作至少要解除一个因为试图 down 这个信号等而阻塞住的线程。用下面这样的方法调用：

```
$mutex->up();
```

17.2.3.3 其他标准线程模块

Thread::Signal 允许你启动一个线程用于接收你的进程的 %SIG 信号。这就解决了目前仍然让人头疼的问题：目前的 Perl 实现里信号是不可靠的，如果轻率使用可能会偶而导致内核倾倒。

这些模块仍然在开发中，并且也可能无法在你的系统上提供你需要的结果。但，它们也可能可以用。如果不能，那么就是因为某些象你一样的人还没有修补它们。可能你或者某个人就可以参与进来帮助解决问题。

第十八章 编译

如果你到这里来是为了找一个 Perl 的编译器，你可能很奇怪地发现你已经有一个了——你的 perl 程序（通常是 /usr/bin/perl）已经包含一个 Perl 编译器。这个东西可能不是你想要的，如果不是你想象的东西，你可能会很开心地得知我们还提供代码生成器（也就是那些要求意义严格的人所谓的"编译器"），我们将在本章讨论那些东西。但是首先我们想讲讲我们眼中的编译器是什么。本章不可避免地要讲一些底层的细节，而有些人会喜欢这些内容，有些人则不。如果你发现自己并不喜欢这些内容，那就把它当作一个提高你阅读速度的练习吧。（呵呵，不能不看，但是可以不用全明白。）

假设你是一个指挥家，任务是给一个大管弦乐队排练乐谱。当乐谱到货的时候，你会发现有一些小册子，管弦乐队成员人手一册，每人只有自己需要演奏的部分乐章。但奇怪的是，你的主乐谱里面什么东西也没有。而更奇怪的是，那些有部分乐章的乐谱都是用纯英语写的，而不是用音乐符号写的。在你开始准备一个演奏的节目之前，或者甚至把乐谱给你的乐队演奏之前，你首先得把这样的散文翻译成普通的音符和小节线。然后你要把所有独立的部分编辑成一个完整的乐谱，这样你才能对整个节目有一个完整的印象。

与之类似，当你把你的 Perl 脚本的源程序交给 perl 执行的时候，对计算机而言，它就象用英语写的交响乐对音乐家一样毫无用处。在你的程序开始运行之前，Perl 需要把这些看着象英文似的东西编译（注：或曰翻译，或转换，或改变或变形）为一种特殊符号表现形式。不过你的程序仍然不会运行，因为编译器只是编译。就象指挥的乐谱一样，就算你的程序已经转换成一种容易解释的指令格式，它仍然需要一个活跃的对象来解释那些指令。

18.1. Perl 程序的生命周期

你可以把一个 Perl 程序分裂为四个独立的阶段，每个阶段都有自己的独立的子阶段。第一个和最后一个是最让人感兴趣的两个，而中间的两个是可选的。这些阶段在图 18-1 里描绘。

1. 编译阶段

在第一阶段：编译阶段里，Perl 编译器把你的程序转换成一个叫分析树的 数据结构。除了使用标准的分析技术以外，Perl 还使用了一种更强大的分析技术：它使用 BEGIN 块引导编译进行得更深入。BEGIN 块一完成分析就转交给解释器运行，结果是它们以 FIFO （先进先出）顺序运行。这样处理的包括 use 和 no 声明；它们实际上是伪装的 BEGIN 块。任何 CHECK，INIT，和 END 块都由编译器安排推迟的执行。

词法声明也做上记号，但是还不执行给它们的赋值。所有 eval BLOCKS，s///e 构造，和非代换的规则表达式都在这里编译，并且常量表达式都预先计算。现在编译器完成工作，除非稍后它又被再次调用进行服务。在这个阶段的结尾，再次调用解释器，以 LIFO 顺序（后进先出）执行任何安排好了的 CHECK 块。是否有 CHECK 块出现决定了我们下一步是进入第二阶段还是直接进入第四阶段。

2. 代码生成阶段（可选）

CHECK 块是由代码生成器装配的，因此这个阶段是在你明确地使用一个代码生成器（稍后在"代码生成器"里描述）的时候发生的。这时候把编译完成的（但还没运行的）程序转换成 C 源程序或者串行的 Perl 字节码——一个代表内部 Perl 指令的数值序列。如果你选择生成 C 源程序，它最后可以生成一个称做可执行影象的本机机器语言。（注： 你最初的脚本也是一个可执行文件，但它不是机器语言，因此我们不把它称做影象。之所以称其为影象文件是因为它是一个你的 CPU 用于直接执行的机器编码的逐字拷贝。）

这时候，你的程序暂时休眠。如果你制作了一个可执行影象，那么你可以直接进入阶段 4；否则，你需要在阶段三里重新组成冻干的字节码。

3. 分析树重构阶段（可选）

要复活你的程序，它的分析树必须重新构造。这个阶段只有在发生了代码生成并且你选择生成字节码的情况下存在。Perl 必须首先从字节码序列中重新构造它的分析树，然后才能运行程序。Perl 并不直接从字节码运行程序， 因为那样会比较慢。

4. 执行阶段

最后，就是你等待的时刻：运行你的程序。因此，这个阶段也称做运行阶段。解释器拿来分析树（可能是直接从编译器来的或者间接从代码生成阶段以及随后的分析树重构阶段过来的）并且运行之。（或者，如果你生成了一个可执行影象文件，它可以当作一个独立的程序运行，因为它包含一个内嵌的 Perl 解释器。）

这个阶段的开始，在你的主程序运行之前，所有安排好了的 INIT 块以 FIFO 顺序执行。然后你的主程序运行。解释器在碰到一个 eval STRING，一个 do FILE 或者 require 语句，一个 s///ee 构造，或者一个代换变量里含有合法代码断言的模式匹配的时候会根据需要回过头调用编译器。

当你的主程序结束之后，最后执行任何推迟的 END 块，这回是以 LIFO 顺序。最早的一个最后执行，然后你的程序结束。（END 块只有在你的 exec 或者你的进程被一个未捕获的灾难性错误摧毁后才能忽略。普通的例外都不认为是灾难性的。）

下面我们将以非常详细的方式讨论这些阶段，而且是以不同的顺序。

18.2 编译你的代码

Perl 总是处于两种操作模式之一：要么它在编译你的程序，要么是在执行它——从来不会同时处于两种模式。在我们这本书里，我们把某些事件称做在编译时发生，或者我们说"Perl 编译器做这做那"。在其他地方，我们说某些东西在运行时发生，或者说"Perl 的解释器做这做那"。尽管你可以认为"Perl"就是编译器和解释器的合体，但是把这 Perl 在不同场合所扮演的两个角色之一区分清楚还是非常重要的，这样你才能理解许多事情发生的原因。（也可能有其他角色；perl 还是一个优化器和代码生成器。有时候，它还是一个骗子——不过都是善意的玩笑。）

同时，区分编译阶段和编译时，以及运行阶段和运行时之间的区别也非常重要。一个典型的 Perl 程序只有一个编译阶段，然后只有一个运行阶段。"阶段"是一个大范围的概念。但是编译时和运行时是小范围的概念。一个给定的编译阶段大多数时间做的是编译时工作，但是也通过 BEGIN 块做一些运行时工作。一个给定的运行阶段大多数时间做的是运行时工作，但是它也可能通过类似 eval STRING 这样的操作符做编译时任务。

在典型的事件过程中，Perl 先阅读你的整个程序然后才开始执行。这就是 Perl 分析声明，语句和表达式，确保它们语法上是正确的时候。（注：不，这个过程中没有正式的象 BNF 那样的语法图表，不过我们欢迎你细读 Perl 源程序树里的 perly.y 文件，里面包含 Perl 用的 yacc(1) 语法。我们建议你离词法远一些，因为它让小白鼠产生进食紊乱的症状了。译

注：呵呵，象大话里的唐僧。）如果它发现语法错误，编译器会试图从错误中恢复过来，这样它才能汇报任何后面的源程序的错误。有时候可以这样恢复，但是有时候不行；语法错误有一个很讨厌的问题是会出发一系列错误警告。Perl 在汇报近 10 个错误后暂时退出。

除了处理 BEGIN 块的解释器以外，编译器默许三个概念上的过程处理你的程序。词法分析器（lexer）扫描你的程序里的每一个最小的单元。这些东西有时候称为"词位"（lexemes），但你在讲述编程语言的文章里看到的可能更多的是"记号"（token）。词法分析器有时候被称做标记器或扫描器，而它干的工作有时候被称做是词法分析或记号分析。然后分析器（parser）以 Perl 语言的语法为基础，试图通过把这些记号组合成更大的构造，比如表达式和语句，来获取合适的意义，优化器（optimizer）对这些词法的组合进行重新排列并且把它们归减成更有效的序列。优化器仔细地选择最优的方法，它不会在边缘优化上花费时间，因为 Perl 编译器用做即时编译器时必须运行得极快。

这些过程并不是在相互独立的阶段进行的，而是同时发生，并且相互之间有大量交互。词法分析器偶尔需要来自 parser 的提示，这样它才能够知道它需要注意哪几种记号类型。（很奇怪的是，词法范围就是词法分析器不能理解的事物之一，因为那是"词法"的其他含义。）优化器同时还需要跟踪分析器的处理，因为有些优化在分析器到达某一点之前是无法进行的，比如完成一个表达式，语句，块，或者子过程。

你可能会奇怪，为什么 Perl 编译器同时做这些事情，而不是一件一件做呢？因为这个混乱的过程就是当你在听取或者读取自然语言的时候，你即时地理解它们的过程。你用不着直到读到本章的结束才理解第一句话的含义。你可以想象下面的对应关系：

计算机语言	自然语言
字符	字母
记号	词素
术语	词
表达式	短语
语句	句子
块	段落
文件	章节
程序	故事

如果分析过程进展顺利，编译器就认为你输入了一则合法的故事，哦，是程序。如果你运行程序的时候使用了 -c 开关，那么编译器会打印一条"syntax OK"消息然后退出。否则，编译器会把它自己的成果转交给其他过程。这些"成果"是以分析树的形式表示的。在分析树上的每个"果实"——或者称做节点——代表一个 Perl 内部的操作码，而树上的分支代表树

的历史增长模式。最后，这些节点都会线性地一个接一个地串在一起，以表示运行时系统的访问这些节点的执行顺序。

每个操作码都是 Perl 认为的最小的可执行单元。你可能见过一条象 $a = -($b + $c) 这样的语句，但是 Perl 认为它是六个独立的操作码。如果让我们用一种简单的格式来表示，上面那个表达式的分析树看起来象图 18-2。黑圆点里的数字代表 Perl 运行时系统将遵循的访问顺序。

Perl 不是有些人想象的那种单回合编译器。（单回合编译器是那种把事情做得对计算机简单而对程序员复杂的东西。）Perl 编译器是那种多回合的，优化的编译器，它是由至少三种不同的，相互交错的逻辑回合组成的。回合 1 和 2 在编译器在分析树上上窜下跳构造执行流的时候轮流运行，而回合 3 在一个子过程或者文件完全分析完的时候运行。下面是那些回合：

- 回合 1：自底向上分析

  在这个回合里，分析树是由 yacc(1) 分析器建造的，用的记号是从下层的词法分析器处理出来的（那个过程也可以认为是另外一个逻辑回合）。自底向上只是意味着该分析器先知道树叶后知道树支和树根。它并不是象在图 18-2 里那样从底向上处理各种元素，因为我们是在顶部画树根的，这是

  计算机科学家（和语言学家）的特殊的传统。

  在构造每个操作码节点的时候同时对每个操作码进行完成性检查，以校验语意是否正确，比如是否使用了正确数量和类型的参数来调用内建函数。在完成一条分析树的分支以后，优化器就参与进来看看现在它能否对整个子树做某些转换。比如，一旦它知道我们给某个接收一定数量参数的函数一列数值，那么它就可以把记录可变长参数函数的参数个数的操作码仍掉。还有一个更重要的优化，我们称之为常量消除，将在本节稍后讲述。

  这个回合同时还构造稍后执行的时候要用的节点访问顺序，这个处理也几乎完全是戏法，因为第一个 要访问的地方几乎从来都不是顶端节点。编译器把操作数作成一个临时的循环，把顶端节点指向第一个要访问的操作码。如果顶端操作码无法和更大的操作码匹配，那么这个操作码环就破裂了，于是更大的操作码就成为新的顶端节点。最后是这个环因为进入其他结构里（比如子过程描述符）而合理破裂。尽管其分析树会一直延

伸到树的底部（象图 18-2 里那样）子过程调用者仍然可以找到第一个操作码。而且这里不需要解释器回朔到分析树里寻找从哪里开始。

- 回合 2： 自顶向下优化

如果你在阅读一小段 Perl 代码（或者英文文章），那么如果你不检查上下文的词法元素的话，你就无法判断环境。有时候你在获取更多的消息之前无法判断真正将要发生什么事情。不过，不必害怕，因为你并不孤独：编译器也

一样。在这个回合里，在它刚创建的子树上向回退，以便进行局部优化，最需要注意的事情是环境传播。编译器用当前节点产生的恰当的环境（空，标量，列表，引用或者左值等）标记到相邻的下层节点上。不需要的操作码被清空但并不删除，因为现在重新构造执行顺序已经太晚了。我们将依赖第三回合把他们从第一回合决定了的临时执行顺序中删除。

- 回合 3：窥探孔优化器

有些代码单元有自己的存储空间，它们在里面保存词法范围的变量。（在 Perl 的说法里，这样的空间称为便条簿（scratchpad））。这样的单元包括 eval STRING，子过程和整个文件。从优化器的角度来说，更重要的是它们 1. 有自己的进入点，这就意味着尽管我们知道从这里开始的执行顺序，我们也不知道以前发生过什么，因为这个构造可能是从其他什么地方调用的。因此如果一个这样的单元被分析器分析完成，Perl 就在那段代码上运行一个窥探孔优化器。和前面两个回合不同的是，前面两个回合在分析树的各个分支上运行，而这个回合是以线性执行顺序横跨代码，因为这里基本上是我们采取这个步骤的最后的机会了，然后我们就要从分析器上砍断操作码列表了。大多数优化已经在头两个回合里完成了，但是有些不行。

最后的分类优化在这个阶段发生，包括把最后的执行顺序缝合在一起，忽略清空了的操作码，以及识别什么时候可以把各种操作码缩减成更简单的东西。识别字串的链接就是一个非常重要的优化，因为你肯定不想每次向字串结尾加一点东西的时候就要到处拷贝字串。这个回合不仅仅做优化；它还做大量"实际"的工作：捕获光字，在有问题的构造上生成警告信息，检查那些可能无法抵达的代码，分析伪哈希键字，以及在子过程的原型被编译前寻找它们。

- 回合 4：代码生成

这个回合是可选的；在普通的情况下并不出现这个回合。但是，如果调用了任何三个代码生成器之一——B::Bytecode，B::C，和 B::CC，那么就会最后再访问一次分析树。代码生成器要么发出用于稍后重新构造分析树的串行的 Perl 字节码，要么是代表编译时分析树状态的文本 C 代码。

C 代码的生成源自两种不同的风格。B::C 简单地重新构造分析树，然后用 Perl 在执行的时候自己用的普通的 runops() 循环运行之。B::CC 生成一个线性化了并且优化过的运行时代码路径（组成一个巨大的跳转表）的 C 等效物，并且执行之。

在编译的时候，Perl 用许多方法优化你的代码。它重新排列代码好让它在运行时更有效。它删除那些在执行时可能永远不会到达的代码，比如一个 if(0) 块，或者在if (1) 块里的 elsif 和 else。如果你使用用 my ClassName[2] $var 或 our ClassName[2] $var 声明的词法类型，而且 ClassName[2] 包是用 use fields 用法设置的，那么对下层的伪哈希的常量域的访问会在编译时进行拼写检查并且转换成一个数组访问。如果你给 sort 操作符一个足够简单的比较路径，比如{$a <=> $b} 或者 {$b cmp $a}，那么它会被一个编译好的 C 代码代替。

Perl 里最富戏剧性的优化可能就是它尽可能快地解析常量表达式的方法。比如，让我们看看图 18-2 的分析树。如果节点 1 和 2 都有文本和常量函数，节点 1 到 4 将已经被那些计算代替了，就象图 18-3 的分析树：

图 18-3 （略。。。）

这就叫常量消除。常量消除并不仅限于象把 2**10 转成 1024 这么简单的场合。它还解析函数调用——包括内建的和用户定义的子过程，只要它们符合第六章，子过程，的"内联常量函数"的标准。回想一下 FORTRAN 编译器对它们内在函数臭名昭著的知识，Perl 在编译的过程中也知道要调用它的哪些内建函数。这就是为什么如果你试着做 log(0.0) 或者 sqrt （求平方根）一个负数的时候，会导致一个编译错误，而不是一个运行时错误，并且解释器根本没有运行。（注：实际上，我们在这里实在是简化的太厉害了。解释器实际上是运行了，因为那就是常量消除器实现的方法。不过它是在编译时立即运行的，类似 BEGIN 块执行的方式。）

甚至任意复杂的表达式都是提前解析的，有时候导致整个块的删除，象下面这个：

```
if (2* sin(1)/cos(1) < 3 && somefn() ) { whatever() }
```

那些永不计算的东西不会生成任何代码。因为第一部分总是假，所以 somefn 和 whatever 都不会调用。（所以不必期待那个语句块里会有 goto 标签，因为它甚至都不

会在运行时出现。）如果 somefn 是一个可以内联的常量函数，那么即使你把上面的顺序换成下面这样：

```
if ( somefn() &&  2*sin(1)/cos(1) <3 )) { whatever() }
```

也不会改变输出，因为整个表达式仍然在编译时解析。如果 whatever 可以内联，那么它在运行时不会被调用，甚至在编译的时候也不会；它的值会被当作一个文本常量那样嵌入程序中。然后你会收到一个警告 “Useless use of a constant in void context”。如果你没有意识到它是常量，你可能会觉得奇怪。不过，如果 whatever 是一个在非空的环境中（就象由优化器决定的那样）计算的最后一条语句，那么你就看不到警告。

你可以用 perl -Dx 看到在所有优化阶段完成之后构造的分析树的最终结果。（-D 要求你用的是特殊的，制作时打开调试的 Perl）。还可以看看后面描述的 B::Deparse 节。

总而言之，Perl 编译器为优化代码工作得很努力（不过不是特别努力），然后来的就是运行时整体的执行速度提高了。现在就是让你的程序运行的时候了，所以让我们讨论它吧。

18.3 执行你的代码

打个比方，Sparc 程序只能运行在 Sparc 机器上，Intel 程序只能运行在 Intel 的机器上，而 Perl 程序只能运行在 Perl 机器上。Perl 机器处理那些 Perl 程序认为在一台计算机里完美的属性：内存是自动分配和释放的，基本数据类型是动态的字串，数组和哈希，而且没有尺寸限制，并且系统表现得都非常相象。Perl 解释器的工作就是把它所运行的任何计算机都搞得象那种理想的 Perl 机器一样。

这样的假想的机器就好象是一种特别设计来运行 Perl 程序的机器一样。编译器生成的每个操作码都是这种假想的指令集中的一个基本命令。Perl 里没有使用硬件程序计数器，而是由解释器跟踪当前要执行的操作数。Perl 里也没有硬件堆栈指针，解释器有它自己的虚拟堆栈。这个堆栈非常重要，因为 Perl 虚拟机（我们拒绝称之为 PVM ）是一个基于堆栈的机器。Perl 操作码在内部称做 PP 代码（“压栈-出栈代码”("push-pop codes")）因为它们操作解释器的虚拟堆栈以寻找所有操作数，处理临时数值，还有存储所有结果。

如果你曾经用 Forth 或者 PostScript[2] 写过程序，或者用 RPN（“反转润色符号” "Reverse Polish Notation"）一个 HP 的科学计算器记录，你就知道堆栈机器是如何运行的了。甚至如果你没有用过这些东西，它的概念也是简单的：要把 3 和 4 相加，你按照 3 4 + 这样的顺序做处理而不是习惯的 3 + 4。从堆栈的角度来看，这里的意思是你把 3 然后是 4，压入堆栈，而 + 则把两个参数弹出堆栈，把它们相加，然后把 7 压回堆栈，然后 7 就会留在那里直到你对它进行其他处理。

与 Perl 的编译器相比，Perl 的解释器是非常直接的，直接得几乎让人厌倦的程序。它所做的一切就是走过那些编译出来的操作码，每次一个，然后把它们发配给 Perl 运行时环境，也就是 Perl 虚拟机。它只不过是一小堆 C 代码，对吧？

实际上，它一点也不乏味。Perl 的虚拟机替你跟踪一堆动态环境，这样你就不用跟踪了。Perl 维护不少堆栈，你用不着理解它们，但是我们会在这里列出来好加深你的印象：

- 操作数堆栈（operand stack）

  这个堆栈我们已经讲过了。

- 保存堆栈（save stack）

  在这里存放等待恢复的局部数值。许多内部过程也有许多你不知道的局部值。

- 范围堆栈（scope stack）

  轻量的动态环境，它控制何时保存堆栈应该"弹出"。

- 环境堆栈（context stack）

  重量级的动态环境；是谁调用了谁最后把你放到了你现在所处的位置。caller 函数遍历这个堆栈。循环控制函数扫描这个堆栈以找出需要控制哪个循环。如果你从环境堆栈剥出，那么范围堆栈也相应剥出，这样就从你的保存堆栈里恢复所有局部变量，甚至你用一些极端的方法，比如抛出例外或者 longjmp(3) 出去等也是如此。

- 环境跳转堆栈（jumpenv stack）

  longjmp(3) 环境的堆栈，它允许你抛出错误或者迅速退出。

- 返回堆栈（return stack）

  我们进入这个子过程时的来路。

- 标记堆栈（mark stack）

  在操作数堆栈里的列出的当前的杂参数的起点。

- 递归词法填充堆栈（recursive lexical pad stacks）

当子过程被递归地调用时词法变量和其他"原始寄存器"存放的地方。

当然，还有存放所有 C 变量的 C 堆栈。Perl 实际上力图避免依赖 C 的堆栈存储保存的数值，因为 longjmp(3) 忽略了这样的数值的合理的恢复。

所有的这些就是说，我们对解释器的通常的看法：一个解释另外一个程序的程序，是非常不足以描述其内部的真正情况的。的确，它的内部是一些 C 的代码实现了一些操作码，但当我们提到"解释器"的时候，我们所说的含义要比上面这句话更多些，就好象我们谈到"音乐家"时，我们说的含义可不仅仅是一个可以把符号转换成声音的 DNA 指令集。音乐家是活生生的"有状态"的有机组织。解释器也一样。

具体来说，所有这些动态和词法范围，加上全局符号表，带上分析树，最后加上一个执行的线索，就是我们所谓的一个解释器。从执行的环境来看，解释器实际上在编译器开始运行之前就存在了，并且甚至是可以在编译器正在制作它的环境的时候解释器就开始进行初步的运行了。实际上，这就是当编译器调用解释器执行 BEGIN 块等的时候发生的事情。而解释器可以回过头来使用编译器进一步制作自己的运行环境。每次你定义另外一个子过程或者装载另外一个模块，那么我们称之为解释器的特定的虚拟 Perl 机器实际上就在重新定义自身。你实际上不能说是编译器还是解释器在控制这一切，因为它们合作控制我们通常称之为"运行 Perl 脚本"的引导过程。就象启动一个孩子的大脑。是 DNA 还是蛋白质在做处理？我们认为两个都在起作用，并且还有一些来自外部程序员的输入。

我们可以在一个进程里运行多个解释器；它们可以或者不可以共享分析树，取决于它们是通过克隆一个现有解释器生成的还是通过从头开始制作一个新的解释器生成的。我们也可能在一个解释器里运行多个线程，这种情况下我们不仅共享分析树而且还共享全局符号——参阅第十七章，线程。

不过大多数 Perl 程序只使用一个 Perl 解释器执行它们编译好了的代码。并且尽管你可以在一个进程里运行多个独立的 Perl 解释器，目前可以实现这个目的的 API 只能从 C 里访问。（注：到目前为止只有一个例外：Perl 5.6.0 可以在 Microsoft Windows 的仿真 fork 的支持下实现克隆解释器。到你阅读到此处时，可能也有一个实现 "ithread" 的 Perl API 。）每个独立的 Perl 解释器起到一个完全独立的进程的作用，但是并不象创建一个完全新的进程那样开销巨大。这就是为什么 Apache 的 mod_perl 扩展的性能如此突出的原因：当你在 mod_perl 里启动一个 CGI 脚本时，那个脚本已经被编译成 Perl 的操作码了，消除了重新编译的需要—— 但是更重要的是，消除了启动一个新进程的需要，这才是真正的瓶颈。Apache 在一个现存的进程里初始化一个新的 Perl 解释器然后把前面编译完了的代码交给它执行。当然，这里头的东西要远比我们说的多——一直是这样的。更

多关于 mod_perl 的东西，请参考 Writing Apache Modules with Perl and C（O'Reilly，1999）。

许多其他应用都可以内嵌 Perl 解释器，比如 nvi，vim 和 innd；我们可不指望在这里把它们都列出来。而且还有许多甚至都不敢宣传它们有内嵌的 Perl 引擎的商业产品。它们只是在内部使用它，因为它能按照他们的风格实现他们的程序。

18.4 编译器后端

所以，如果 Apache 可以现在编译一个 Perl 程序而稍后才执行它，你为什么不行？Apache 和其他包含内嵌 Perl 解释器的程序做得非常简单——它们从来不把分析树存到一个外部文件中。如果你对这样的做法表示满意，而且不介意使用 C API 获得这样的特性，那么你可以做一样的事情。参阅第二十一章，内部和外部，里的“嵌入 Perl”一节，获取如何从一个闭合的 C 框架里访问 Perl 的信息。

如果你不想走这条路或者有其他需要，那么还有几个选择可用。你可以不让来自 Perl 编译器的输出立即输入 Perl 解释器，而是调用任意可用的后端。这些后端可以把编译好的操作码串行化和存储到任何外部文件中，甚至可以把它们转换成几种不同风格的 C 代码。

请注意那些代码生成器都是非常试验性的工具，在生产环境中不可靠。实际上，你甚至都不能指望它们在非生产环境里面能用——除了极为稀有的情况以外。现在我们已经把你的期望值降得足够低了，这样任何成功都可以比较容易超过它们，这时候我们才能放心地告诉你后端是如何运行的。

有些后端模块是代码生成器，比如 B::Bytecode，B::C，和 B::CC。其他的实际上都是代码分析和调试工具，比如 B::Deparse，B::Lint，和 B::Xref。除了这些后端以外，标准版还包括几种其他的底层模块，那些潜在的 Perl 代码开发工具的作者可能对它们感兴趣。其他的后端模块可以在 CPAN 找到，包括（到我们写这些为止）B::Fathom，B::Graph，B::JVM::Jasmin，和 B::Size。

如果你除了给解释器提供输入以外还有其他地方使用 Perl 编译器，那么 O 模块（也就是 O.pm 文件）位于编译器和你分配的后端模块之间。你并不直接调用该后端；相反，你调用中间层，然后由它调用你指定的后端。因此如果你有一个模块调用 B::Backend，你可以在一个脚本里这样来调用：

```
%perl -MO=Backend SCRIPTNAME
```

有些后端需要选项，用下面的方法声明：

```
%perl -MO=Backend, OPTS SCRIPTNAME
```

有些后端已经有调用它们的中间层的前端了，所以你不必费心记忆它们的 M.O。尤其是 perlcc(1) 调用那个代码生成器，而代码生成器启动起来可能比较麻烦。

18.5 代码生成器

目前的三种把 Perl 操作码转换成其他格式的后端都是处于实验阶段的。（没错，我们前面说过这些，但是我们不想你忘记这点。）甚至就算它门生成的输出碰巧能正确运行，生成的程序也可能比平常需要更多的磁盘空间，更多的存储器，和更多的 CPU 时间。这是一个正在进行的研究可开发领域。不过一切都会慢慢好起来的。

18.5.1 字节码生成器

B::Bytecode 模块将分析树的操作码以平台无关的编码写出。你可以把一个 Perl 脚本编译成字节码然后把它们拷贝到另外一台安装了 Perl 的机器上跑。

perlcc 命令知道怎么把一个 Perl 源程序转换成一个编译成字节码的 Perl 程序。这个命令是标准的，不过仍然处于实验阶段。你要做的事情只是：

```
%perlcc -b -o pbyscript srcscript
```

然后你就应该能直接"执行"所生成的 pbyscript。该文件的开头看起来象下面这样：

```
#!/usr/bin/perl

use ByteLoader 0.03;

^C^@^E^A^C^@^@^@^A^F^@^C^@^@^@^B^F^@^C^@^@^@^C^F^@^C^@^@^@

B^@^@^@H9^A8M-^?M-^?M-^?M-^?7M-^?M-^?M-^?M-^?6^@^@^@^A6^@

^G^D^D^@^@^@^KR^@^@^@^HS^@^@^@^HV^@M-2W<^FU^@^@^@^@X^Y@Z^@

...
```

你会看到一小段脚本头后面跟着一堆纯二进制数据。这些东西看起来非常神秘，不过其实不过是一个技巧而已。ByteLoader 模块使用一种叫做"源码过滤器"的技巧在 Perl 能够看到源程序之前修改它们。源码过滤器是一种预处理器，它接收当前文件中在它后面的所有内容。与类似 cpp(1) 和 m4(1) 这样的宏预处理器不同，它们只能做简单的转换，而源码过滤器没有限制。源码过滤器已经用于检查 Perl 的语法，用于压缩或加密源代码，甚至用 Latin. E perlibus unicode； cogito，ergo substr；挑剔的 dbm，等写 Perl 程序；

ByteLoader[2] 模块是源码过滤器，它知道如何把 B::Bytecode 生成的串行的字节码分解并重新组合成原来的分析树。这样重新组合的 Perl 代码被接合进入当前分析树，不需要通过编译器。当解释器拿到这些操作码，它就开始执行它们，就好象它们早就在那里一样。

18.5.2. C 代码生成器

剩下的代码生成器，B::C 和 B::C 都生成 C 代码，而不是串行化的 Perl 操作码。它们生成的代码非常难读，如果你想试着读他们那你就傻了。它可不是那种转换好了的 Perl 到 C 的代码片段，可以插入到一个更大的 C 程序里。关于那方面的内容，请参阅第二十一章。

B::C 模块只是把创建整个 Perl 运行时环境所需要的 C 数据结构写出来。你得到一个专用的解释器，它的所有由编译器制作的数据结构都已经初始化好了。从某种意义上来说，所生成的代码类似 B::Bytecode 生成的东西。它们都是编译器制作的操作码树的直接翻译，只不过 B::Bytecodes 把他们以符号的形式输出，这些符号稍后可以重建操作码树并且插入到一个运行着的 Perl 解释器，而 B::C 把那些操作码输出为 C 代码。当你用你的 C 编译器编译这些 C 程序并且把它们和 Perl 库链接，生成的程序就不需要在目标系统安装 Perl 解释器就可以运行。（不过，它可能需要一些共享库——如果你没有把所有的东西都静态链接的话。）不过，这个程序和那些运行你的脚本的普通的 Perl 解释器没有什么根本的区别。它只不过是预先编译成一个独立的可执行影象而已。

不过，B::CC 模块试图做得更多。它生成的 C 源文件的开头看起来很象 B::C 生成的东西，（不过，当你犯傻的时候当然什么东西看起来都一样。我们难道没有告诉你别看吗？）不过，最终所有相似都会消失。在 B::C 生成的代码里，有一个 C 程序的很大的操作码表，它的作用就象解释器自己做的处理，而 B::CC 生成的 C 代码是以你的程序对应的运行时顺序为布局输出的。它甚至还有 C 函数对应你的程序的每个函数。它做了一些基于变量类型的优化；有些速度测试可以比在标准的解释器里快一倍。这是目前的代码生成器中最具野心的一个，也是对未来做出了最多承诺的一个。不过同时也是最不稳定的一个。

那些为毕业设计找主题的计算机科学系的学生可以在这里仔细找找。这里有大量还未琢磨的钻石等待你们发掘。

18.6 代码开发工具

O 模块里有许多很有趣的操作数方法（Modi Operandi），可以用来给易怒的实验性代码生成器做输入用。这个模块给你提供了相对而言痛苦少些的访问 Perl 编译器输出的方法，这样你就可以比较容易地制作那些认识 Perl 程序所有内涵所需要的其他工具。

B::Lint 模块是参考 lint(1) 命名的，lint(1) 是 C 程序的校验器。它检查那些常常拌倒初学者但又不会触发警告的有问题的构造。直接调用这个模块：

```
%perl -MO=Lint, all myprog
```

目前只定义了几个检查，象在一个隐含的标量环境里使用数组啦，依赖缺省变量啦，以及访问另外一个包（通常是私有包）中以_开头的标识啦。参阅 B::lint(3) 获取细节。

B::Xref 模块生成一个交叉引用，里面列出一个程序里的声明以及所有变量（包括全局和词法范围）的使用，子过程，和格式，用文件和子过程分隔。用下面方法调用这个模块：

```
%perl -MO=Xref myprog > myprof.pxref
```

举例来说，下面就是一部分输出；

```
Subroutine parse_argv

  Package (lexical)

    $on               i113, 114

    $opt              i113, 114

    %getopt_cfg       i107, 113

    @cfg_args         i112, 114, 116, 116

  Package Getopt::Long

    $ignorecase       101

    &GetOptions       &124

  Package main

    $Options          123, 124, 141, 150, 165, 169

    %$Options         141, 150, 165, 169

    &check_read       &167

    @ARGV             121, 157, 157, 162, 166, 166
```

这里显示出 parse_argv 子过程自己有四个词法变量；它还通过 main 包和 Getopt::Long 包访问全局标识符。列表中的数字是使用这些项的行号：前导的 i 表明该项在随后的行号首次引入，而一个前导 & 意味着在这里有一个子过程调用。析引用分别列出，于是 $Options 和 %$Options 都会显示出来。

B::Deparse 是一个很好的打印机，它可以揭开 Perl 代码神秘的面纱，帮助你理解优化器为你的代码做了那些转换。比如，下面的东西显示了 Perl 给各种构造使用了什么缺省：

```
% perl -MO=Deparse -ne 'for (1 .. 10) { print if -t }'
LINE: while (defined($_ = )) {
    foreach $_ (1 .. 10) {
        print $_ if -t STDIN;
    }
}
```

-p 开关给你的程序加圆括号，这样你就可以看到 Perl 对优先级的看法：

```
%perl -MO=Deparse, -p -e 'print $a ** 3 + sqrt(2) /10 ** -2 ** $c'
print((($a ** 3) + (1.4142135623731 / (10 ** (-(2 ** %c))))));
```

你可以使用 -p 看看哪个优先代换的字串编译到代码里：

```
%perl -MO=Deparse, -q -e '"A $name and some @ARGV\n"'
'A ' . $name . ' and some ' . join($", @ARGV) . "\n";
```

下面的例子显示了 Perl 是怎样把一个三部分的 for 循环变成一个 while 循环：

```
%perl -MO=Deparse -e 'for ($i=0;$i<0;$i++) { $x++ }'
$i = 0;
while ( $i < 10 ) {
    ++$x;
}
continue {
    ++$i
}
```

你甚至可以在一个 perlcc -b 生成的 Perl 字节码上调用 B::Deparse，让它为你反编译那段二进制文件。串行化的 Perl 操作码可能有点难读，但并不是强加密的东西。

18.6 提前编译，回头解释

做事情的时候总有考虑所有事情的合适时机；有时候是在做事之前，有时候是做事之后。有时候是做事情的过程中间。Perl 并不假定何时是适合考虑的好时机，所以它给程序员许多选项，好让程序员告诉它什么时候思考。其他时间里它知道有些东西是必要的，但它不知道应该考虑哪个方案，因此它需要一些手段来询问你的程序。你的程序通过定义一些子过程来回答这些问题，这些子过程名字与 Perl 试图找出来的答案相对应。

不仅编译器可以在需要提前思考的时候调用解释器，而且解释器也可以在想修改历史的时候回过头来调用编译器。你的程序可以使用好几个操作符回过头来调用编译器。和编译器类似，解释器也可以在需要的时候调用命名子过程。因为所有这些来来回回都是在编译器，解释器，和你的程序之间进行的，所以你需要清楚何时发生何事。首先我们谈谈这些命名子过程何时被触发。

在第十章，包，里，我们讲了如果该包里的一个未定义函数被调用的时候，包的 AUTOLOAD 子过程是如何触发的。在第十二章，对象，里我们提到了 DESTROY 方法，它是在对象的内存要自动被 Perl 回收的时候调用的。以及在第十四章，捆绑变量，里，我们碰到了许多访问一个捆绑了的变量是要隐含地调用的函数。

这些子过程都遵循一个传统：如果一个子过程会被编译器或者解释器自动触发，那么我们用大写字符为之命名。与你的程序的生命期的不同阶段相联的是另外四个子过程，分别是 BEGIN，CHECK，INIT，和 END。它们前面的 sub 关键字是可选的。可能最好叫它们"语句块"，因为它们从某种程度上来说更象命名语句块而不象真的子过程。

比如，和普通的子过程不同的是，你多次定义这些块不会有任何问题，因为 Perl 会跟踪何时调用它们，因此你不用通过名字调用它们。（它们还和普通子过程不同的是 shift 和 pop 表现得象在主程序里面，因此它们缺省时对 @ARGV 进行操作，而不是 @_。）

这四种块类型以下面顺序运行：

- BEGIN

  如果在编译过程中碰到则在编译其他文件之前尽可能快地运行。

- CHECK

  当编译完成之后，但在程序开始之前运行。

（CHECK 可以理解为“检查点”或者“仔细检查”或者就是“停止”。）

- INIT

  在你的程序的主流程开始执行之前运行。

- END

  在程序执行结束之后运行。

如果你声明了多于一个这样的同名语句块，即使它们在不同的模块里，BEGIN 也都是在 CHECK 前面运行的，而 CHECK 也都是在 INIT 前面运行，以及 INIT 都在 END 前面——END 都是在最后，你的主程序退出的时候运行，多个 BEGIN 和 INIT 以声明的顺序运行（FIFO），而 CHECK 和 END 以相反的顺序运行（LIFO）。

下面的可能是最容易演示的例子：

```
#! /usr/bin/perl -l

print   "start main running here";

die   "main now dying here\n";

die    "XXX: not reached\n";

END   { print "1st END: done running" }

CHECK    { print "1st CHECK : done compiling" }

INIT   { print "1st INIT: started running" }

END    { print "2nd END: done running"   }

BEGIN    { print "1st BEGIN: still compiling"}

INIT   { print "2nd INIT: started running" }

BEGIN   { print "2nd CHECK: done compiling"}

END   { print "3rd END:   done running"}
```

如果运行它，这个演示程序输出下面的结果：

```
1st BEGIN: still compiling

2nd BEGIN: still compiling

2nd CHECK: done compiling

1st CHECK: done compiling

1st INIT: started running

2nd INIT: started running

start main running here

main now dying here

3rd END: done running

2nd END: done running

1st END: done running
```

因为一个 BEGIN 块立即就执行了，所以它甚至可以在其他文件编译前把子过程声明，定义以及输入等抓过来。这些动作可能改变编译器对当前文件其他部分分析的结果，特别是在你输入了子过程定义的情况下。至少，声明一个子过程就把它当作一个列表操作符使用，这样就令圆括号是可选的。如果输入的子过程定义了原型，那么调用它的时候就会当作内建函数分析，甚至覆盖同名的内建函数，这样就可以给它们不同的语意。use 声明就是一个带有目的的 BEGIN 块声明。

相比之下，END 块是尽可能晚地执行：在你的程序退出 Perl 解释器的时候，甚至是因为一个没有捕获的 die 或者其他致命错误。有两种情况下会忽略 END 块（或者一个 DESTROY 方法）。如果一个程序不是退出，而是用 exec 从一个程序变形到另外一个程序，那么 END 就不会运行。一个进程被一个未捕获的信号杀死的时候也不会执行 END 过程。（参阅在第三十一章，用法模块，里描述的 use sigtrap 用法。那里面有将可捕获信号转换成例外的一个比较容易的方法。关于信号操作的通用信息，请参考第十六章，进程间通讯，里的“信号”。）想要绕开所有 END 处理，你可以调用 POSIX::exit，也就是 kill -9，$$，或者就是 exec 任何无伤大雅的程序，比如 Unix 系统里的 /bin/true。

在一个 END 块里面，$? 包含程序 exit 时准备的状态。你可以在 END 块里修改 $? 以修改程序的退出值。要小心不要碰巧用 system 或者反勾号运行了其它程序而改变了 $?。

如果你在一个文件里有好几个 END 块，那么它们以定义它们的相反顺序执行。也就是说，你的程序结束的时候定义在最后的 END 块首先执行。如果你把 BEGIN 和 END 成对使

用的话，这样的反序允许相关的 BEGIN 和 END 块按照你预期的方法嵌套，比如，如果如果主程序和它装载的模块都有自己的成对的 BEGIN 和 END 子过程，象下面这样：

```
BEGIN { parint "main begun" }

END { print "main ended" }

use Module;
```

并且在那个模块里，定义了下面的声明：

```
BEGIN { print "module begun" }

END { print "module ended" }
```

那么主程序就知道它的 BEGIN 总是首先发生，而它的 END 总是最后使用。（不错，BEGIN 实际上是一个编译时的块，但类似的现象对运行时的 INIT 和 END 对身上也会发生。）如果一个文件包含另外一个文件，而且它们都有类似这样的声明的时候，这个原则是递归地正确的。这样的嵌套属性令这些块可以很好地当作包构造器和析构器来使用。每个模块都可以有它们自己的安装和删除函数，而 Perl 可以自动地调用它们。这样，程序员就不用总是记住是否用了某个库，是否在某时需要调用特殊的初始化或者清理代码。这些事件都由模块的声明来保证。

如果你把 eval STRING 当作一个从解释器到编译器的回调函数，那么你可以把 BEGIN 看作从编译器到解释器的前进函数。它们两个都是暂时把当前正在处理的事情挂起来然后切换操作的模式。如果我们说一个 BEGIN 块是尽可能早地执行，我们的意思就是说它在完成定义以后马上就执行，甚至是在包含它的文件的其他部分分析之前。因此 BEGIN 块是在编译时执行的，而不是在运行时。一旦一个 BEGIN 块开始运行，那么它马上就取消定义并且它使用的任何代码都返回到 Perl 的内存池中。你不能把 BEGIN 当作一个子过程调用，（你试了也没有用。）因为当它存在那里的时候，它就已经运行（消失）了。

和 BEGIN 块类似，INIT 块都是在 Perl 运行时开始执行之前运行的，顺序是"先进先出"（FIFO）。比如，在 perlcc 里讲到的代码生成器使用 INIT 块初始化并解析指向 XSUB 的指针。 INIT 块实际上和 BEGIN 块一样，只不过是它们让程序员分开了必须在编译阶段发生的构造和必须在运行阶段发生的构造。如果你直接运行一个脚本，这两者没什么大区别，因为每次运行编译器都要运行；但是如果编译和执行是分开的，那么这样的区别就可能是关键的。编译器可能只调用一次，而生成的可执行文件可以运行多次。

和 END 块类似，CHECK 块在 Perl 编译阶段完成之后而在开始运行阶段之前运行。顺序是 LIFO。CHECK 块可以用于"退出"编译器，就好象 END 块可以用于退出你的程序一样。特别是后端都把 CHECK 块当作挂钩使用，这样它们可以调用相应的代码生成器。它们需

要做的只是把一个 CHECK 块放到它们自己的模块里，而这个 CHECK 块在合适的时刻就会运行，这样你就不用把 CHECK 写到你的程序里。因此，你很少需要写自己的 CHECK 块，除非你正在写这样的模块。

把上面的内容都放到一起，表 18-1 列出了各种构造，列出了它们何时编译或者运行“...”代表的代码。

表 18-1 何时发生何事

Block or Expression	Compiles During Phase	Traps Compile Errors	Runs During Phase	Traps Run Errors	Call Trigger Policy
use ...	C	No	C	No	Now
no ...	C	No	C	No	Now
BEGIN {...}	C	No	C	No	Now
CHECK {...}	C	No	C	No	Late
INIT {...}	C	No	R	No	Early
END {...}	C	No	R	No	Late
eval {...}	C	No	R	Yes	Inline
eval "..."	R	Yes	R	Yes	Inline
foo(...)	C	No	R	No	Inline
sub foo {...}	C	No	R	No	Call anytime
eval "sub {...}"	R	Yes	R	No	Call later
s/pat/.../e	C	No	R	No	Inline
s/pat/"..."/ee	R	Yes	R	Yes	Inline

现在你知道结果了，我们希望你能更有信心的编辑和使用的 Perl 程序。

第十九章 命令行接口

19.1 命令行处理

很幸运的是 Perl 是在 Unix 世界里成长起来的，因为那就意味着它的调用语法在其他操作系统的命令行解释器里也能运行得相当好。大多数命令行解释器知道如何把一列单词当作参数处理，而用不着关心某个参数是否以一个负号开头。当然，如果你从一个系统转到另外一个系统，也有一些乱七八糟的地方会把事情搞糟。比如，你不能在 MS-DOS 里象在 Unix 里那样使用单引号。而且对于象 VMS 这样的系统来说，有些封装代码必须经过一些处理才能模拟 Unix I/O 的重定向。而通配符就解释成通配符。一旦你搞定这些问题，那么 Perl 就能在任何操作系统上非常一致地处理它的开关和参数。

即使你自己并没有一个命令行解释器，你也很容易从用其它语言写的另外一个程序里执行 Perl 程序。调用程序不仅能够用常用的方法传递参数，而且它还可以通过环境变量传递信息，还有，如果你的操作系统支持的话，你还可以通过继承文件描述符来传递信息（参阅第十六章，进程间通讯，里的"传递文件句柄"一节。）甚至一些外来的参数传递机制都可以很容易地在一个模块里封装，然后通过简单的 use 指示引入的你的 Perl 程序里来。

Perl 以标准的风格传递命令行参数。（注：假设你认为 Unix 是标准风格。）也就是说，它预料在命令行上先出现开关（以负号开头的字）。在那之后通常出现脚本的名字，后面跟着任何附加的传递到脚本里的参数。有些附加的参数本身看起来就象开关，不过如果是这样的话，它们必须由脚本本身处理，因为 Perl 一旦看到一个非开关参数，或者特殊的"--"开关（意思是说："我是最后一个开关"），它就停止分析开关。

Perl 对你放源代码的地方提供了一些灵活性。对于短小的快速使用的工作，你可以把 Perl 程序全部放在命令行上。对于大型的，比较永久的工作，你可以把 Perl 脚本当作一个独立的文件使用。Perl 按照下面三种方式之一寻找一个脚本编译和运行：

- 通过在命令行上的 -e 开关一行一行地声明。比如：

```
%perl -e "print 'Hello, World.'"

Hello, World.
```

- 包含在命令行声明的第一个文件名的文件里面。系统支持可执行脚本第一行的 #! 符号为你调用解释器的用法。

1. 通过标准输入隐含地传递。这个方法只有在没有文件名参数的情况下才能用；要给一个标准输入脚本传递参数，你必须使用方法 2，为脚本名字明确地声明一个"-"。比如：

```
%echo "print qq(Hello, @ARGV.)"| perl - World

Hello, World.
```

对于方法 2 和 3， Perl 从文件开头开始分析——除非你声明了 -x 开关，那样它会寻找第一个以 #! 开头并且包含 "perl" 的行，然后从那里开始分析。这个开关对于在一个更大的消息里运行一段嵌入的脚本很有用。如果是这样，你可以用 __END__ 记号指明脚本的结尾。

不管你是否使用了 -x，分析 #! 行的时候总是要检查是否有开关。这样，如果你所在的平台只允许在 #! 行里有一个参数，或者更惨，就根本不把 #! 行当作一个特殊的行，你仍然能够获得一致的开关特性，而不用管是如何调用 Perl 的，即使你是用 -x 来寻找脚本的开头。

警告：因为老版本的 Unix 不声不响地把内核对 #! 行分析时超过 32 个字符的部分截去，最后可能是有些开关原封不动地传给你的程序而其他的参数就没了；如果你不小心，你甚至有可能收到一个 "-" 而没有它的字母。你可能会想确保所有你的开关要么在 32 字符之前，要么在其后。大多数开关并不关心它们是否被多次处理，但如果拿到一个“-”而不是整个开关，就会导致 Perl 试图从标准输入读取它的源代码，而不是从你的脚本里。而且一个 -I 开关的片段也会导致很奇怪的结果。不过，有些开关的确关心它们是否被处理了两次，比如 -l 和 -0 的组合。你要么把所有开关放到 32 字符范围之后（如果可行），要么把 -0DIGITS 换成 BEGIN{ $/ = "\0DIGITS"; }。当然，如果你用的不是 Unix 平台，那么我们保证不会有这种问题发生。

对 #! 行的开关的分析从该行中首次出现 "perl" 的地方开始。为了 emacs 用户的习惯，"-*" 和 "-" 组成的序列特别被忽略掉，因此，如果你有意使用，你可以说：

```
#! /bin/sh -- # -*- perl -*- -p
eval 'exec perl -S $0 ${1+"$@"}'
    if 0;
```

于是 Perl 就只看见 -p 开关。奇妙的小发明"-*- perl -*-" 告诉 emacs 以 Perl 模式启动；如果你不使用 emacs，那么你用不着它。我们稍后在描述 -S 的时候解释它那堆东西。

如果你有 env(1) 程序，也可以用类似的技巧：

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

前面的例子使用 Perl 解释器的相对路径，把用户路径里出现的第一个 perl 拿过来。如果你想要特定版本的 Perl，比如，perl5.6.1，那么把它直接放进 #! 行的路径里，要么是和 env 程序一起，要么是 -S 那堆东西在一起，或者在普通的 #! 里。

如果 #! 行不包含单词 “perl”，那么在 #! 后面的程序代替 Perl 解释器执行。比如，假设你有一个普通的 Bourne shell 脚本，内容是：

```
#! /bin/sh
echo "I am a shell script"
```

如果你把这个文件给 Perl ，那么 Perl 会为你运行 /bin/sh。这个举止可能有些怪异，但是它可以帮助那些不识别 #! 的机器的用户。因为这些用户可以通过设置 SHELL 环境变量

告诉一个程序（比如一个邮件程序）说，它们的 shell 是/usr/bin/perl，然后 Perl 就帮他们把该程序发配给正确的解释器，就算他们的内核傻得不会干这事也没关系。不过还是让我们回到真正的 Perl 脚本里头来。在完成你的脚本的定位之后，Perl 把整个程序编译成一种内部格式（参阅第十八章，编译）。如果发生任何编译错误，脚本的执行甚至都不能开始。（这一点和典型的 shell 脚本或者命令文件不同，它们在发现一个语法错误之前可能先跑上一段。）如果脚本语法正确，那么就开始执行。如果脚本运行到最后也没有发现一个 exit 或者 die 操作符，那么 Perl 隐含地提供一个 exit(0)，为你的脚本的调用者标识一个成功的结束状态。（这一点和典型的 C 程序也不一样，在 C 里面，如果你的程序只是按照通常的方法结束，那么你的退出状态是随机的。）

**在非 Unix 系统上的 #! 和引号

Unix 的 #! 技巧可以在其他系统上仿真：

Macintosh

在 Macintosh 上的 Perl 程序有合适的创建者和类型，所以双击它们就会

调用 Perl 应用。

MS-DOS

创建一个批处理文件运行你的程序，并且把它在 ALTERNATIVE_SHEBANG 里成文。 参阅 Perl 源程序发布的顶级目录里的 dosish.h 文件获取更多这方面的信息。

OS/2

把下面这行：

```
extproc perl -S -your_siwtches
```

放在 *.cmd 文件的第一行里（-S 绕开了一个在 cmd.exe 里的“extproc”

处理的臭虫。）

VMS

把下面几行：

```
% perl -mysw 'f$env("procedure")' 'p1' 'p2' 'p3' 'p4' 'p5' 'p6' 'p7' 'p8' !
$ exit++ + ++$status != 0 and $exit = $status = undef;
```

放在你的程序的顶端，这里的 -mysw 是任何你想传递给 Perl 的命令行　开关。现在你可以直接通过键入 perl program 调用你的程序，或者说　@program　把它当作一个 DCL 过程调用，或者使用程序名通过隐含地 DCL$PATH 调用。这些方法记起来有点困难，不过如果你在 perl 里键入　"-V:startperl"，那么 Perl 会给你显示出来。如果你记不住这个用法——　很好，那就是你买这本书的原因。

Win??

如果在一些 Microsoft Windows 系列操作系统里（也就是 Win95，Win98，　Win00(注：请原谅，我们只用两位数表示年代)，WinNT，不过不包括 Win31。)使用 Perl 的 ActiveState 版本。Perl 的安装过程修改了 Windows 的注册表，把 .pl 扩展名和 Perl 解释器关联起来。

如果你安装了另外一个移植的 Perl，包括那个在 Perl 版本里 Win32 目录里的那个，那么你就必须　自己修改 Windows 注册表。

请注意如果你使用 .pl 扩展名就意味着你再也不能区分一个可执行 Perl 程序和一个“perl 库”文件了。　你可以用 .plx 做 Perl 程序的扩展名以避免这个问题。现在这个问题已经不明显了，因为大多数 Perl 模块在 .pm 文件里。

在非 Unix 系统上的命令行解释器通常和 Unix shell 有不同的引号的用法。你必须了解你的命令行解释器里的特殊字符（*，\，和 " 是比较常见的）以及如何保护通过 -e 开关运

行的一行程序里的空白和这些特殊字符。如果 % 是你的 shell 的特殊字符，你还可以把单个 % 改成 %%，或者把它逃逸。

在一些系统上，你可能还要把单引号改成双引号。但是不要在 Unix 或者 Plan9 系统，或者任何运行 Unix 风格的 shell 上这么干，比如从 MKS 工具箱或者来自 Cygnus 的哥几个（现在在 Redhat）的 Cygwin 包。呃，Microsoft 的叫 Interix 的新的 Unix 仿真器也开始看到了，也要注意。

比如，在 Unix 和 Mac OS X 里，用：

```
%perl -e 'print "Hello world\n"'
```

在 Macintosh （Mac OS X 的前身）里，用：

```
print "Hello world\n"
```

然后运行 "Myscript" 或者 Shift-Command-R。

在 VMS 上，使用：

```
$perl -e "print ""Hello world\n"""
```

或者再次使用 qq//：

```
$perl -e "print qq(Hello world\n)"
```

在 MS-DOS 等等里，用：

```
A:> perl -e "print \"Hello world\n\"
```

或者用 qq// 使用自己的引号：

```
A:> perl -e "print qq(Hello world\n)"
```

问题是这些方法都是不可靠的：它依赖于你使用的命令行解释器。如果 4DOS 是命令行 shell，下面这样的命令可能跑得好些：

```
perl  -e "print "Hello world\n""
```

Windows NT 上的 CMD.EXE 程序好象在没有人注意的情况下偷偷加入了许多 Unix shell 的功能，但你需要看看它的文档，查找它的引号规则。

在 Macintosh 上，（注：至少在 Mac OS X 发布之前，我们可以很开心地说它是源于 BSD 的系统。）所有这些取决于你用的是哪个环境。MacPerl shell，或者 MPW，都很象 Unix shell，支持几个引号变种，只不过它把 Macintosh 的非 ASCII 字符自由地用做控制字符。

对这些问题我们没有通用的解决方法。它们就这么乱。如果你用的不是 Unix 系统，但是想做命令行类的处理，最好的解决方法是找一个比你的供应商提供的更好的命令行解释器，这个不会太难。或者把所有的东西都在 Perl 里写，完全忘掉那些单行命令。

**Perl 的位置

尽管这个问题非常明显，但是 Perl 只有在用户很容易找到它的时候才有用。如果可能，最好 /usr/bin/perl 和 /usr/local/bin/perl 都是指向真正二进制文件的符号链接。如果无法做到这一点，我们强烈建议系统管理员把 Perl 和其相关工具都放到一个用户的标准 PATH 里面去，或者其他的什么明显而又方便的位置。

在本书中，我们在程序的第一行使用标准的 #!/usr/bin/perl 符号来表示在你的系统上能用的任何相应机制。如果你想运行特定版本的 Perl，那么使用声明的位置：

```
#! /usr/local/bin/perl5.6.0
```

如果你只是想运行至少某个版本的 Perl ，而不在乎运行更高版本，那么在你的程序顶端附近放下面这样的语句：

```
use v5.6.0
```

（请注意：更早的 Perl 版本使用象 5.005 或者 5.004_05 这样的数字。现在我们会把它们看作 5.5.0 和 5.4.5，不过比 5.6.0 早的 Perl 版本不能理解那些符号。）

**开关

单字符并且没有自己的参数的命令行开关可以与其他跟在后面的开关组合（捆绑）在一起。

```
#! /usr/bin/perl -spi.bak    # 和  -s -p -i.bak 一样
```

开关，有时候也叫选项或者标志。不管你叫他们什么，下面的是 Perl 识别的一些：

-- 结束开关处理，就算下一个参数以一个负号开头也要结束。它没有其他作用。

-0OCTNUM

-0 把记录分隔符（$/）声明为一个八进制数字。如果没有提供 OCTNUM，那么 NUL 字符（也就是 ASCII 字符 0，Perl 的 "\0"）就是分隔符。其他开关可以超越或者遵循这个八进制数字。比如，如果你有一个可以打印文件名是空字符结尾的文件的 find(1) 版本，那么你可以这么说：

```
% find . -name '*.bak' -print0 | perl -n0e unlink
```

特殊数值 00 令 Perl 以段落模式读取文件，等效于把 $/ 变量设置为 ""。数值 0777 令 Perl 立即把整个文件都吃掉。这么做等效于解除 $/ 变量的定义。我们使用 0777 是因为没有 ASCII 字符是那个数值。（不幸的是，有一个 Unicode 字符是那个数值，\N{LATIN SMALL LETTER O WITH STROKE AND ACUTE}，不过有人说你不会用那个字符分隔你的记录。）

-a 打开自动分割模式，不过只有在和 -n 或 -p 一起使用时才有效。在 -n 和 -p 开关生成的 while 循环里首先对 @F 数组进行一次隐含的 split 命令调用。因此：

```
% perl -ane 'print pop(@F), "\n";'
```

等效于：

```
LINE: while (<>) {
    @F = split(' ');
    print pop(@F), "\n";
}
```

你可以通过给 split 的 -F 开关传递一个正则表达式声明另外一个域分隔符。比如，下面两个调用是等效的：

```
% awk -F: '$7 && $7 !~ /^\/bin/' /etc/passwd

% perl -F: -lane 'print if $F[6] && $F[6] !~ m(^/bin)' /etc/passwd
```

-c 令 Perl 检查脚本的语法然后不执行刚编译的程序退出。从技术角度来讲，它比那做得更多一些：它会执行任何 BEGIN 或 CHECK 块以及任何 use 指令，因为这些都是在执行你的程序之前要发生的事情。不过它不会再执行任何 INIT 或者 END 块了。你仍然通过在你的主脚本的末尾包括下面的行获得老一些的但很少用到的性质：

```
BEGIN { $^C = 0; exit; }
```

-C 如果目标系统支持本机宽字符，则允许 Perl 在目标系统上使用本机宽字符 API 对于版本 5.6.0 而言，它只能用于 Microsoft 平台）。特殊变量${^WIDE_SYSTEM_CALLS} 反映这个开关的状态。

-d 在 Perl 调试器里运行脚本。参阅第二十章，Perl 调试器。

-dMODULE

在调试和跟踪模块的控制下运行该脚本，该模块以 Devel::MODULE 形式安装在 Perl 库里。比如， -d:Dprof 使用 Devel::Dprof 调节器执行该脚本。参阅第二十章的调试节。

-DLETTERS

-DNUMBER

设置调试标志。（这个开关只有在你的 Perl 版本里编译了调试特性（下面描述）之后才能用。）你可以声明一个 NUMBER，它是你想要的位的总和，或者一个 LETTER 的列表。比如，想看看 Perl 是如何执行你的脚本的，用 -D14 或者 -Dslt。另外一个有用的值是 -D1024 或 -Dx，它会列出你编译好的语法树。而 -D512 或 -Dr 显示编译好的正则表达式。数字值在内部可以作为特殊的变量 $^D 获得。表 19-1 列出了赋了值的位。

表 19-1 -D 选项

位	字母	含义
1	p	记号分解和分析
2	s	堆栈快照
4	l	标签堆栈处理
8	t	跟踪执行
16	o	方法和重载解析
32	c	字串/数字转换
64	P	为 -P 打印预处理器命令
128	m	存储器分配
256	f	格式化处理
512	r	正则分析和执行
1024	x	语法树倾倒
2048	u	污染检查
4096	L	内存泄露（需要在编译 Perl 时使用 -DLEAKTEST）
8192	H	哈希倾倒--侵占 values()
16384	X	便签簿分配
32768	D	清理
65536	S	线程同步

所有这些标志都需要 Perl 的可执行文件是为调试目的特殊制作的。不过，因为调试制作不是缺省，所以除非你的系统管理员制作了这个特殊的 Perl 的调试版本，否则你根本别想用 -D 开关。参阅 Perl 源文件目录里面的 INSTALL 文件获取细节，短

一些的说法是你在编译 Perl 本身的时候需要给你的 C 编译器传递 -DDEBUGGING 编译选项。在 Configure 问你优化选项和调试选项的时候，如果你包括了 -g 选项，那么这个编译选项会自动加上。

如果你只是想在你的每行 Perl 代码执行的时候获取一个打印输出（象 sh -x 为 shell 脚本做的那样），那你就不能用 -D 开关。你应该用：

```
#  Bourne shell 语法
$PERLDB_OPTS="NonStop=1 AutoTrace=1 frame=2" perl -dS program

# csh 语法
% (setenv PERLDB_OPTS "NonStop=1 AutoTrace=1 frame=2"; perl -dS program)
```

见第二十章获取细节和变种。

-e PERLCODE

可以用于输入一行或多行脚本。如果使用了 -e ，Perl 将不在参数列表中寻找程序的文件名。Perl 把 PERLCODE 参数当作有新行结尾看待，所以可以给出多个 -e 命令形成一个多行的程序。（一定要使用分号，就象你在文件里的程序一样。） -e 给每个参数提供新行并不意味着你必须使用多个 -e 开关；如果你的 shell 支持多行的引用，比如 sh，ksh，或 bash，你可以把多行脚本当作一个 -e 参数传递：

```
$perl -e 'print "Howdy, ";
    print "@ARGV!\n";' world
```

```
Howdy, world!
```

对于 csh 而言，可能最好还是使用多个 -e 开关：

```
%perl -e 'print "Howdy, ";'             -e 'print "@ARGV!\n";' world
Howdy, world!
```

在行计数的时候，隐含的和明确给出的新行都算数，所以两个程序中的第二个 print 都是在 -e 脚本的第 2 行。

-F PATTERN

声明当通过 -a 开关（否则没有作用）自动分裂是要 split 的模式。该模式可以由斜杠（//），双引号（""），或者单引号（''）包围。否则，他会自动放到单引号里。请注意如果要通过 shell 传递引号，你必须把你的引号引起来，具体怎么做取决于你的系统。

-h 打印一个 Perl 命令行选项的概要

-i EXTENSION

-i 声明那些由 <> 构造处理的文件将被现场处理。Perl 是通过这样的办法实现的：先重命名输入文件，然后用原文件名打开输出文件，并且把该输出文件选为调用 print，printf，和 write 的缺省。（注：通常，这并不是真的“现场”。它是相同的文件名，但是不同的物理文件。）

EXTENSION 用于修改旧文件名然后做一个备份的拷贝。如果没有提供 EXTENSION，那么不做备份并且当前文件被覆盖。如果 EXTENSION 不包含一个 *，那么该字串被附加到当前文件名的后面。如果 EXTENSION 包含一个或多个 * 字符，那么每个 * 都被当前正在处理的文件名替换。用 Perl 的话来说，你可以认为事情是这样的：

```
($backup = $extension) =~ s/\*/$file_name/g;
```

这样就允许你把一个前缀用于备份文件，而不是--或者可以说是除了后缀以外：

```
%perl -pi'orig_*' -e 's/foo/bar/' xyx   # 备份到 'orig_xyx'
```

你甚至可以可以把原来文件的备份放到另外一个目录里（只要该目录已经存在）：

```
%perl -pi'old/*.orig' -e 's/foo/bar/' xyx # 备份到 'old/xyx.orig'
```

这些一行程序对都是相等的：

```
%perl -pi -e 's/foo/bar/' xyx       # 覆盖当前文件
%perl -pi'*' -e 's/foo/bar/' xyx       # 覆盖当前文件

%perl -pi'.orig' -e 's/foo/bar/' xyx       #备份到 'xyx.orig'
%perl -pi'*.orig' -e 's/foo/bar/' xyx       #备份到 'xyx.orig'
```

从 shell 上，你说：

```
%perl -p -i.oirg -e "s/foo/bar/;"
```

等效于使用下面的程序：

```
#! /usr/bin/perl   -pi.orig
s/foo/bar/;
```

而上面的又是下面的程序的便利缩写：

```
#! /usr/bin/perl
$extension = '.orig';
LINE: while(<>){
   if ($ARGV ne $oldargv) {
      if ($extension !~ /\*/) {
         $backup = $ARGV . $extension;
      }
      else {
         ($backup = $extension) =~ s/\*/$ARGV/g;
      }
      unless (rename($ARGV, $bckup)) {
         warn "cannot rename $ARGV to $backup: $! \n";
         close ARGV;
```

```
            next;
        }
        open(ARGVOUT, ">$ARGV");
        select(ARGVOUT);
        $oldargv = $ARGV;
    }
    s/foo/bar/;
}
continue {
    print;    # 这一步打印到原来的文件名
}
select(STDOUT);
```

这一段长代码实际上相当于那条简单的单行带 -i 开关的命令，只不过 -i 的形式不需要拿 $ARGV 和 $oldargv 进行比较以判断文件名是否改变。不过，它的确使用 ARGVOUT 作为选出的文件句柄并且在循环结束以后把原来的 STDOUT 恢复为缺省文件句柄。象上面的代码那样，Perl 创建备份文件时并不考虑任何输出是否真的被修改了。如果你想附加到每个文件背后，或者重置行号，那么请参阅 eof 函数的描述，获取关于如何使用不带圆括号的 eof 定位每个输入文件的结尾的例子。

如果对于某个文件来说，Perl 不能创建象 EXTENSION 里声明的那样的备份文件，它会为之发出一个警告然后继续处理列出的其他文件。

你不能用 -i 创建目录或者剥离文件的扩展名。你也不能用一个 ~ 来表示家目录 --

因为有些家伙喜欢使用这个字符来表示他们的备份文件：

```
%perl -pi~ -e 's/foo/bar' file1 file2 file3...
```

最后，如果命令行上没有给出文件名，那么 -i 开关不会停止 Perl 的运行。如果发生这种事情，则不会做任何备份，因为不能判断原始文件，而可能会发生从 STDIN 到 STDOUT 的处理。

-IDIRECTORY

-I 声明的目录比 @INC 优先考虑，它包含搜索模块的目录。-I 还告诉 C 预处理器到那里寻找包含文件。C 预处理器是用 -P 调用的；缺省时它搜索 /usr/include 和 /usr/lib/perl。除非你打算使用 C 预处理器（而实际上几乎没人再干这事了），否则你最好在你的脚本里使用 use lib 指示器。不过，-I 和 use lib 类似，它隐含地增加平台相关的目录。参阅第三十一章，实用模块，里的 use lib 获取细节。

-lOCTNUM

-l 打开自动行结束处理。它有两个效果：首先，如果它和 -n 或者 -p 一起使用，它自动 chomp 行终止符，其次，它把 $\ 设置为 OCTNUM 的数值，这样任何打印语句将有一个值为 OCTNUM 的 ASCII 字符追加在结尾代替行终止符。如果省略了 OCTNUM，-l 把 $\ 设置为 $/ 的当前值，通常是新行。因此要把行截断为 80 列，你这么说：

```
%perl -lpe 'substr($_, 80) = ""'
```

请注意在处理这个开关的时候完成了 $\ = $/ 的赋值，因此如果 -l 开关后面跟着 -0 开关，那么这个输入记录分隔符可以与输出记录分隔符不同：

```
%gnufind / -print0 | perl -ln0e 'print "found $_" if -p'
```

这条命令把 $\ 设置为新行而稍后把 $/ 设置为空字符。（请注意如果 0 直接跟在 -l 后面，它将被当作 -l 开关的一部分。这就是为什么我们在它们之间绑上了 -n 开关。）

-m 和 -M

这些开关装载一个 MODULE，就象你执行了一个 use 那样，如果你声明的是 -MODULE，而不是 MODULE，那么它将调用 no。比如，-Mstrict 类似 use strict，而 -M-strict 类似 -no strict。

-mMODULE

在执行你的脚本之前执行 use MODULE()。

-MMODULE

-M'MODULE ...'

在执行你的脚本之前执行 use MODULE。这条命令是通过简单的解析 -M 后面剩下的参数形成的，因此你可以用引号在该模块名后面加额外的代码，比如，-M'MODULE qw(foo bar)'。

-MMODULE=arg1,arg2...

一块小小的语法糖，意思是你还可以把 -Mmodule=foo,bar 当作

-M'module qw(foo bar)' 的一个缩写来用。这样当输入符号的时候就避免了引号的使用。-Mmodule=foo,bar 生成的实际的代码是：

```
use module split(/,/, q{foo,bar})
```

请注意 = 的形式删除了 -m 和 -M 之间的区别，但是最好还是使用大写的形式以避免混淆。

你可能只会在真正的 Perl 命令行调用的时候使用 -M 和 -m 开关，而不会在 #! 封装的选项行上用。（如果你准备把它放在文件里，为什么不用一个等效的 use 或者 no 代替呢？）

-n 令 Perl 认为在你的脚本周围围绕着下面的循环，这样就让你的脚本遍历文件名参数，就象 sed -n 或者 awk 做的那样：

```
LINE:
while(<>) {
    ...       # 你的脚本在这里
}
```

你可以在你的脚本里把 LINE 当作一个循环标记来用，即使你在你的文件里看不到实际的标记也如此。

请注意，那些行缺省的时候并不打印。参阅 -p 选项看看如何打印。下面是一个删除旧于一周的文件的有效方法：

```
find . -mtime +7 -print | perl -nle unlink
```

这样做比用 find(1) 的 -exec 开关要快，因为你不需要对每个找到的文件启动一个进程。一个很有趣的一致性是，你可以用 BEGIN 和 END 块捕获这个隐含的循环之前或之后的控制，就象 awk 一样。

-p 令 Perl 认为在你的脚本周围围绕着下面的循环，这样就让你的脚本遍历文件名参数，象 sed 那样：

```
LINE:
while(<>) {
   ...      # 你的脚本在这里
}
continue {
   print or die "-p destination: $!\n";
}
```

你可以在你的脚本里把 LINE 当作一个循环标记来用，即使你在你的文件里看不到实际的标记也如此。

如果由于某种原因一个参数命名的文件无法打开，Perl 会警告你，然后继续下一个文件。请注意这些行都自动打印出来。在打印的过程中如果发生错误则认为是致命错误。同样也是一个很有趣的一致性是，你可以用 BEGIN 和 END 块捕获这个

隐含的循环之前或之后的控制，就象 awk 一样。

-P 令你的脚本先由 C 预处理器运行，然后才由 Perl 编译。（因为注释和 cpp(1) 指示都是由 # 字符开头，所以你应该避免注释任何 C 预处理器可以识别的词，比如 "if"，"else"，或者"define"。）不过你用不用 -P 开关，Perl 都会注意 #line 指示以控制行号和文件名，这样任何预处理器都可以通知 Perl 这些事情。参阅第二十四章，普通实践，里面的“在其他语言里生成 Perl”。

-s 打开命令行上脚本名之后，但在任何文件名或者一个“--”开关处理终止符之前的基本的开关分析。找到的任何开关都从 @ARGV 里删除，并且在 Perl 里设置一个同名的开关变量。这里不允许开关捆绑，因为这里允许使用多字符开关。

下面的脚本只有在你带着 -foo 开关调用脚本的时候才打印“true”。

```
#! /sr/bin/perl -s
if  ($foo) {print "true\n"}
```

如果该开关形如 -xxx=yyy，那么 $xxx 变量的值设置为跟在这个参数的等号后面的值（本例中是”yyy“）。下面的脚本只有在你带着 -foo=bar 开关调用的时候才打印“true”。

```
#! /usr/bin/perl -s
if ($foo eq 'bar') { print "true\n" }
```

-S 让 Perl 使用 PATH 环境变量搜索该脚本（除非脚本的名字包含目录分隔符）。

通常，这个开关用于帮助在那些不支持 #! 的平台上仿真 #!。在许多有兼容 Bourne 或者 C shell 的平台上，你可以用下面这些：

```
#! /usr/bin/perl
eval "exec /usr/bin/perl -S $0 $*"
   if $running_under_some_shell;
```

系统忽略第一行然后把这个脚本交给 /bin/sh，然后它继续运行然后试图把 Perl 脚本当作一个 shell 脚本运行。该 shelll 把第二行当作一个普通的 shell 命令执行，因此启动 Perl 解释器。在一些系统上，$0 并不总是包含路径全名，因此 -S 告诉 Perl 在必要的时候搜索该脚本。在 Perl 找到该脚本以后，它分析该行并且忽略它们，因为变量 $running_under_some_shell 总是为假。一个更好的构造是 $* 应该是 ${1+"$@"}，它可以处理文件名中嵌入的空白这样的东西，但如果该脚本被 csh 解释将不能运行，为了用 sh 代替 csh 启动，有些系统必须用一个只有一个冒号的行替代 #! 行，Perl 会礼貌地忽略那样的行。其他不能支持这些的系统必须用一种完全迂回的构造，这种构造可以在任何 csh，sh 或者 perl 里运行，这种构造是这样的：

```
eval '(exit $?0)' && eval 'exec /usr/bin/perl -S $0 ${1+"$@"}'
   & eval 'exec /usr/bin/perl -S $0 $argv:q'
      if -;
```

的确，这个东西很难看，不过那些系统也一样难看（注：我们故意用这个词）。

在一些平台上，-S 开关同样也令 Perl 在搜索的时候给文件名附加后缀。比如，在 Win32 平台上，如果对最初的文件名查找失败而且这个文件名原来没有用 .bat 或者 .com，则会给该文件名后缀上 .bat 或者 .com 进行查找。如果你的 Perl 是打开调试编译的，那么你可以使用 Perl 的 -Dp 开关来观察搜索过程。

如果你提供的文件名包含目录分隔符（即使只是一个相对路径名，而不是绝对路径名），而且如果没有找到该文件，那么在那些会隐含附加文件扩展名的平台上（不是 Unix）就会做这件事，然后一个接一个的找那些带这些扩展名的文件。

在类似 DOS 的平台上，如果脚本不包含目录分隔符，它首先会在当前目录搜索，然后再寻找 PATH。在 Unix 平台上，出于安全性考虑，为了避免未经明确请求，偶然执行了当前工作目录里面的东西，将严格地在 PATH 里搜索，

-T 强制打开“感染”检查，这样你就可以检查它们了。通常，这些检查只有在运行 setuid 或者 setgid 的时候才进行。把它们明确地打开，让程序的作者自己控制是一个不错的主意，比如在 CGI 程序上。参阅第二十三章，安全。

-u 令 Perl 在编译完你的脚本以后倾倒核心。然后从理论上来讲你可以用 undump 程序（未提供）把它转成一个可执行文件。这样就以一定的磁盘空间为代价（你可以通过删除可执行文件来最小化这个代价）换来了速度的提升。如果你想在输出之前执行一部分你的脚本，那么使用 Perl 的 dump 操作符。注意：undump 的可用性是平台相关的；可能在某些 Perl 的移植版本里不能用。它已经被新的

Perl 到 C 的代码生成器替换掉了，由这个代码生成器生成的东西更具移植性（不过仍然处于实验期）。

-U 允许 Perl 进行不安全的操作。目前，唯一的“不安全”的操作是以超级用户身份运行是删除目录，以及在把致命污染检查转换成警告的情况下运行 setuid 程序。请注意如果要真的生成污染检查警告，你必须打开警告。

-v 打印你的 Perl 的版本和补丁级别，以及另外一些信息。

-V 打印 Perl 的主要配置值的概要以及 @INC 的当前值。

-V:NAME

向 STDOUT 打印命名配置变量的值。NAME 可以包括正则字符，比如用“.”匹配任何字符， 或者 ".*" 匹配任何可选的字符序列。

```
%perl -V:man.dir
man1dir='/usr/local/man/man1'
man3dir='/usr/local/man/man3'

%perl -V:'.*threads'
d_oldpathreads='undef'
...（略）
```

如果你要求的环境变量不存在，它的值将输出为 "UNKNOWN"。在程序里可以使用

Config 模块获取配置信息，不过在哈希脚标里不支持模式：

```
%perl -MConfig -le 'print $Config{man1dir}'
/usr/local/man/man1
```

参阅第三十二章，标准模块，里的 Config 模块。

-w 打印关于只提到一次的变量警告，以及在设置之前就使用了的标量的警告。同时还警告子过程重定义，以及未定义的文件句柄的引用或者文件句柄是只读方式打开的而你却试图写它们这样的信息。如果你用的数值看上去不象数字，而你却把它们当作数字来使用；如果你把一个数组当作标量使用；如果你的子过程递归深度超过 100 层；以及无数其他的东西时也会警告。参阅第三十三章，诊断信息，里的每条标记着“(W)”的记录。

这个开关只是设置全局的 $^W 变量。它对词法范围的警告没有作用--那方面的请参阅 -W 和 -X 开关。你可以通过使用 use warning 用法打开或者关闭特定的警告，这些在第三十一章描述。

-W 无条件地永久打开程序中的所有警告，即使你用 no warnings 或者 $^W = 0 局部关闭了警告也没用。它的作用包括所有通过 use，require，或者 do 包括进来的文件。把它当作 Perl 的等效 lint(1) 命令看待。

-XDIRECTORY

-x 告诉 Perl 抽取一个嵌入在一条信息里面的脚本。前导的垃圾会被丢弃，直到以

#! 开头并包括字串“perl”的第一行出现。任何在该行的单词“perl”之后的有意义的开关都会被 Perl 使用。如果声明了一个目录名，Perl 在运行该脚本之前将切换到该目录。-x 开关只控制前导的垃圾，而不关心尾随的垃圾。如果该脚本有需要忽略的尾随的垃圾，那它就必须以 __END__ 或者 __DATA__ 结束，（如果需要，该脚本可以通过 DATA 文件句柄处理任何部分或者全部尾随的垃圾。它在理论上甚至可以 seek 到文件的开头并且处理前导的垃圾。）

-X 无条件及永久地关闭所有警告，做的正好和 -W 完全相反。

*环境变量

除了各种明确修改 Perl 行为的开关以外，你还可以设置各种环境变量以影响各种潜在的性质。怎样设置环境变量是系统相关的，不过，如果你用 sh，ksh 或者 bash，有一个技巧就是你可以为单条命令临时地设置一个环境变量，就好象它是一个有趣的开关一样。你必须在命令前面设置它：

```
$PATH='/bin:/usr/bin' perl myproggie
```

你可以在 csh 或者 tcsh 里用一个子 shell 干类似的事：

```
%(setenv PATH "/bin:/usr/bin"; perl myproggie)
```

否则，你通常就要在你的家目录里的一些名字象 .chsrc 或者 .profile 这样的文件里设置环境变量。在 csh 或者 tcsh 里，你说：

```
%setenv PATH '/bin:/usr/bin'
```

而在 sh，ksh，和 bash 里，你说：

```
$PATH='/bin:/usr/bin'; export PATH
```

其他系统里有其他的半永久的设置方法。下面是 Perl 会注意的环境变量：

HOME

如果不带参数调用 chdir 时要用到。

LC_ALL, LC_CTYPE, LC_COLLATE, LC_NUMERIC, PERL_BADLAND

控制 Perl 操作某种自然语言的方法的环境变量。参阅 perllocale 的联机文档。

LOGDIR

如果没有给 chdir 参数，而且没有设置 HOME。

PATH

用于执行子进程，以及使用了 -S 开关的时候寻找程序。

PERL5LIB

一个用冒号分隔的目录的列表，用于指明在搜索标准库和当前目录之前搜索

Perl 库文件的目录。如果存在有任何声明的路径下的体系相关的目录，则都回自动包括进来。如果没有定义 PERL5LIB，则测试 PERLLIB，以保持与老版本的向下兼容。如果运行了污染检查（要么是因为该程序正在运行 setuid 或者 setgid，要么是因为使用了 -T 开关。），则两个库变量都不使用。这样的程序必须用 use lib 用法来实现这些目的。

PERL5OPT

缺省命令行开关。在这个变量里的开关会赋予每个 Perl 命令行。只允许 -[DIMUdmw]。如果你运行污染检查（因为该程序正在运行 setuid 或者 setgid，或者使用了 -T 开关），则忽略这个变量。如果 PERL5OPT 是以 -T 开头，那么将打开污染检查，导致任何后继选项都被忽略。

PERL5DB

用于装载调试器代码的命令。缺省的是：

```
BEGIN { require 'perl5db.pl' }
```

参阅第二十章获取这个变量的更多用法。

PERL5SHELL（仅用于 Microsoft 移植）

可以设置一个侯选 shell，这个 shell 是 Perl 在通过反勾号或者 system 执行命令的时候必须的。缺省时在 WinNT 上是 cmd.exe /x/c 以及在 Win95 上是 command.com /c。Perl 认为该值是空白分隔的。在

任何需要保护的字符（比如空白和反斜杠）之前用反斜杠保护。

请注意 Perl 并不将 COMSPEC 用于这个目的，因为 COMSPEC 在用户中有更高的可变性，容易导致移植问题。另外，Perl 可以使用一个不适合交互使用的 shell，而把 COMSPEC 设置为这样的 shell 可能会干涉其他程序的正常功能（那些程序通常检查 COMSPEC 以寻找一个适于交互使用的 shell）。

PERLLIB

一个冒号分隔的目录列表，在到标准库和当前目录查找库文件之前先到这些目录搜索。如果定义了 PERL5LIB，那么就不会使用 PERLLIB。

PERL_DEBUG_MSTATS

只有在编译时带上了与 Perl 发布带在一起的 malloc 函数时才有效（也就是说，如果 perl -V:d_mymalloc 生成“define”）。如果设置，这就会导致在执行之后显示存储器统计信息。如果设置为一个大于一的整数，那么也会导致在编译以后的存储器统计的显示。

PERL_DESTRUCTI_LEVEL

只有在 Perl 的可执行文件是打开了调试编译的时候才相关，它控制对对象和其他引用的全局删除的行为。

除了这些以外，Perl 本身不再使用其他环境变量，只是让它执行的程序以及任何该程序运行的子程序可以使用他们。有些模块，标准的或者非标准的，可能会关心其他的环境变量。比如，use re 用法使用 PERL_RE_TC 和 PERL_RE_COLORS，Cwd

模块使用 PWD，而 CGI 模块使用许多你的 HTTP 守护进程（也就是你的 web 服务器）设置的环境变量向 CGI 脚本传递信息。

运行 setuid 的程序在做任何事情之前执行下面几行将会运行得更好，它只是保持人们的诚实：

```
$ENV{PATH} = '/bin:/usr/bin';      # 或者任何你需要的
$ENV{SHELL} = '/bin/sh' if exists $ENV{SHELL};
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};
```

参阅第二十三章获取细节。