

第二章 集腋成裘

- ↓ 第二章 集腋成裘
 - ↓ 2.1 原子
 - ↓ 2.2 分子
 - ↓ 2.3 内置的数据类型
 - ↓ 2.4 变量
 - ↓ 2.5 名字
 - ↓ 2.5.1 名字查找
 - ↓ 2.6 标量值
 - ↓ 2.6.1 数字文本
 - ↓ 2.6.2 字符串文本
 - ↓ 2.6.3 选择自己的引号
 - ↓ 2.6.4 要么就完全不管引起
 - ↓ 2.6.5 代换数组数值
 - ↓ 2.6.6 “此处” 文档
 - ↓ 2.6.7 V-字符串文本
 - ↓ 2.6.8 其他文本记号
 - ↓ 2.7 环境
 - ↓ 2.7.1 标量和列表环境
 - ↓ 2.7.2 布尔环境
 - ↓ 2.7.3 空 (void) 环境
 - ↓ 2.7.4 代换环境
 - ↓ 2.8 列表值和数组
 - ↓ 2.8.1 列表赋值
 - ↓ 2.7.6 数组长度
 - ↓ 2.9 散列
 - ↓ 2.10 型团 (typeglob) 和文件句柄
 - ↓ 2.11 输入操作符
 - ↓ 2.11.1 命令输入 (反勾号) 操作符
 - ↓ 2.11.2 行输入 (尖角) 操作符
 - ↓ 2.11.3 文件名聚集操作符

我们准备从小处着手，因此本章介绍 Perl 的元素。

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够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。（这可不算是随处翻看。）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符串，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，**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
<code>\$days</code>	简单标量值 <code>\$days</code>
<code>\$days[28]</code>	数组 <code>@days</code> 的第二十九个元素
<code>\$days{'Feb'}</code>	散列 <code>%days</code> 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
<code>\${days}</code>	和 <code>\$days</code> 一样，不过在字母数字前面不易混淆

<code>\$Dog::days</code>	在 <code>Dog</code> 包里面的不同的 <code>\$days</code> 变量
<code>\$#days</code>	数组 <code>@days</code> 的最后一个索引
<code>\$days->[28]</code>	<code>\$days</code> 一个引用指向的数组的第二十九个元素
<code>\$days[0][2]</code>	多维数组
<code>\$days{200}{'Feb'}</code>	多维散列
<code>\$days{2000,'Feb'}</code>	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
<code>@days</code>	包含 (<code>\$days[0]</code> , <code>\$days[1]</code> , ... <code>\$days[n]</code>) 的数组
<code>@days[3,4,5]</code>	包含 (<code>\$days[3]</code> , <code>\$days[4]</code> , <code>\$days[5]</code>) 数组片段的数组
<code>@days[3..5]</code>	包含 (<code>\$days[3]</code> , <code>\$days[4]</code> , <code>\$days[5]</code>) 数组片段的数组
<code>@days{'Jan','Feb'}</code>	包含 (<code>\$days{'Jan'}</code> , <code>\$days{'Feb'}</code>) 片段的散列

每个散列都用 `%` 命名：

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

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母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 pragma`），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符串。）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符串，那事情就有趣了。一个当作运行时的 Perl 代码编译的字符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符串而并非任何真实的花括弧也如此。所以如果 Perl 没有在字符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符串的词法范围。

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。或者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

<code>\x{263a}</code>	Unicode (笑脸)
<code>\N{NAME}</code>	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
<code>\u</code>	强迫下一个字符为大写（Unicode里的“标题”）
<code>\l</code>	强制下一个字符小写
<code>\U</code>	强制后面所有字符大写
<code>\L</code>	强制后面所有字符小写
<code>\Q</code>	所有后面的非字母数字字符加反斜杠
<code>\E</code>	结束 <code>\U</code> ， <code>\L</code> ，或 <code>\Q</code> 。

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)
```

除了被替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//	文本字符串	否
" "	qq//	文本字符串	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”，以避免你在引起字符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#` 被分析为字符串 `'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符串看待。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::` 转换成字符串 `"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 `@foo` 不存在，它很显然是一个字符表。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-字符串文本

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；该字串的数字的自然数对应 **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符串环境，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** 在字符串环境中把数字转换成字符串，以及在数字环境中把字符串转换成数字。有些标量环境不关心返回的是字符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符串 "" 或者数字 0（或者它的等效字符串，"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符串还是数字。（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，一个数组名产生一系列值：

`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）一系列数值，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）。直到现在我们假设列表文本只是一个文本列表。不过，正如字符串文本可能代换其他子字符串一样，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，那么你可以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";
    }
}
```

```
    }
}
```

2.7.6 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 `()` 把它截断为什么都没有。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} = 0xffff0000;
$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符串。在列表环境里，返回一系列值，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>) { print; }                # 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 **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，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。

Revision: r1.1 - 14 Oct 2005 - 04:32 - [TingYu](#)

[Perl](#) > [PerlProgramming3](#) > [P3GoryDetail](#) > [PerlPerlBitsandPieces](#)

版权 © 1999-2006 归这里所有作者。 [PostgreSQL](#) 的中文文档版权归何伟平所有。
向为这里贡献想法,文章的人致敬 [PostgreSQL](#) 中文网
[反馈意见](#)