

第一章 Perl 概述

- ↓ 第一章 Perl 概述
 - ↓ 1.1 从头开始
 - ↓ 1.2 自然语言与人工语言
 - ↓ 1.2.1 变量语法
 - ↓ 1.2.2 单数变量
 - ↓ 1.2.3 复数变量
 - ↓ 1.2.4 复杂数据结构
 - ↓ 1.2.5 简单数据结构
 - ↓ 1.2.6 动词
 - ↓ 1.3 一个平均值例子
 - ↓ 1.3.1 如何运行
 - ↓ 1.4 文件句柄
 - ↓ 1.5 操作符
 - ↓ 1.5.1 双目算术操作符
 - ↓ 1.5.2 字符串操作符
 - ↓ 1.5.3 赋值操作符
 - ↓ 1.5.4 单目算术操作符
 - ↓ 1.5.5 逻辑操作符
 - ↓ 1.5.6 比较操作符
 - ↓ 1.5.7 文件测试操作符
 - ↓ 1.6 流程控制
 - ↓ 1.6.1 什么是真
 - ↓ 1.6.2 If 和 unless 语句
 - ↓ 1.6.3 循环
 - ↓ 1.6.3.1 while 和 until 语句
 - ↓ 1.6.3.2 for 语句
 - ↓ 1.6.3.3 foreach 语句
 - ↓ 1.6.3.4 跳出控制结构: next 和 last
 - ↓ 1.7 正则表达式
 - ↓ 1.7.1 量词
 - ↓ 1.7.2 最小匹配
 - ↓ 1.7.3 把钉子敲牢
 - ↓ 1.7.4 反引用
 - ↓ 1.8 列表处理
 - ↓ 1.9 你不知道但不伤害你的东西(很多)

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 dogcatcher"（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从哪儿找到它。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(当我们接触到面向对象编程时，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	一系列数值，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”））和反斜杠代换（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

- `$pet = "Camel";` # 字符串
- `$sign = "I ove my $pet";` # 带代换的字符串
- `$cose = 'It cose $100';` # 不带代换的字符串
- `$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符串

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杠和 `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），要为一个数组赋值，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 **Jacob** 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 **Jacob** 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", "Z
    "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符串。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` 不加区别地去掉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
<code>\$a + \$b</code>	加法	将 <code>\$a</code> 和 <code>\$b</code> 相加
<code>\$a * \$b</code>	乘法	<code>\$a</code> 和 <code>\$b</code> 的积
<code>\$a % \$b</code>	模	<code>\$a</code> 被 <code>\$b</code> 除的余数
<code>\$a ** \$b</code>	幂	取 <code>\$a</code> 的 <code>\$b</code> 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 **lvalue** 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;         # 把字符串变成自填充 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
<code>\$a && \$b</code>	与	如果 <code>\$a</code> 为假则为 <code>\$a</code> ，否则为 <code>\$b</code>
<code>\$a</code>	<code>\$b</code>	或 如果 <code>\$a</code> 为真则为 <code>\$a</code> ，否则为 <code>\$b</code>
! <code>\$a</code> 非 如果 <code>\$a</code> 为假则为真		
<code>\$a and \$b</code>	与	如果 <code>\$a</code> 为假则为 <code>\$a</code> ，否则为 <code>\$b</code>
<code>\$a or \$b</code>	或	如果 <code>\$a</code> 为真则为 <code>\$a</code> ，否则为 <code>\$b</code>
<code>not \$a</code>	非	如果 <code>\$a</code> 为假则为真
<code>\$a xor \$b</code>	异或	如果 <code>\$a</code> 或 <code>\$b</code> 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: $!\n";
```

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 grades 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	字符串	返回值
等于	<code>==</code>	<code>eq</code>	如果 <code>\$a</code> 等于 <code>\$b</code> 返回真
不等于	<code>=</code>	<code>ne</code>	如果 <code>\$a</code> 不等于 <code>\$b</code> 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	<=	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,我们希望你能够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
- 所有未定义的值作假。

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!\n";  
    print "Debug: Answer was '54', expected '42'.\n";  
}
```

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

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.\n"; }
```

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.\n";
```

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 循环在圆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{$user}/.nextrc") {
        print "$user is cool&\\.\\.\\. they use a perl-aware vi!\n";
    }
}
```

不同于 **if** 和 **while** 语句，**foreach** 的条件表达式是在列表环境中，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：

```
foreach $user (@users) {
    if ($user eq "root" or $user eq "lp") {
        next;
    }
    if ($user eq "special") {
        print "Found the special account.\n";
        # 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（如果你认为英语是很容易读懂的）：

```
LINE: while ($line = <ARTICLE>) {
    last LINE if $line eq "\n"; # 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系列词。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	<code>[\t\n\r\f]</code>	<code>\s</code>
词	<code>[a-zA-Z_0-9]</code>	<code>\w</code>
数字	<code>[0-9]</code>	<code>\d</code>

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 xxxxxxxx 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符串。（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，因为我们在假设你的字符串不包含换行；`^` 和 `$` 实际上是用于行的开头和结尾，而不是用于字符串的。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。

-
- Set MYTITLE = Perl 编程:概述

Revision: r1.4 - 18 Aug 2005 - 08:24 - [TingYu](#)

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

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](#)