

Perl 语言的语法由若干更小的部分组合而成。不像口语——说话的语气、语音、语调和直觉使得人们可以在略带误解或概念模糊的情况下继续沟通，计算机和源代码要求的是精确。不需了解语言每一项功能的每一处细节，你同样可以写出有效的 Perl 代码，但为了写好代码，你必须理解这些细节是如何在一起工作的。

## 名称

Perl 程序中，名称（或标识符）无处不在：变量、函数、包、类甚至是文件句柄也有名称。这些名称都以字母或下划线开头。他们可以选择性包含任何字母、数字和下划线的组合。当 utf8 编译命令（`pragmas、unicode`）生效时，你可以在标识符中使用任意合法的 UTF-8 字符。这些都是合法的 Perl 标识符：

```
my $name;
my @_private_names;
my %Names_to_Addresses;

sub anAwkwardName3;

# 启用 C<use utf8;> 时
package Ingy::DE<ouml>t::Net;
```

这些是不合法的 Perl 标识符：

```
my $invalid name;
my @3;
my %~flags;

package a-lisp-style-name;
```

这些规则仅适用于源代码中以字面形式出现的名称，就是说，直接键入 `sub fetch_pie` 或是 `my $waffleiron`。

Perl 动态的本质使得它可以按名称引用在运行时生成或者以输入的方式提供给程序的那些实体。这称为 符号查找。你可以通过这种方式获得更多的灵活性，但以牺牲安全性作为代价。特别地，间接调用函数或方法或是查找名称空间内的符号让你绕过 Perl 的语法分析器——Perl

中唯一强制执行语法规则的部分。请注意这样做可能会生成迷惑性的代码，一个哈希（`hashes`）或嵌套数据结构（`nested_data_structures`）（相比符号引用）会使代码更加清晰。

## 变量名和印记(sigil)

变量名 的开头总有一个标明其值类型的印记。标量变量（`scalars`）开头是美元符号（`$`）。数组变量（`arrays`）开头是“at”符号（`@`）。哈希变量（`hashes`）的开头则是一个百分号（`%`）。

```
my $scalar;
my @array;
my %hash;
```

这些印记多少为变量提供了一些名称空间，使得拥有同名不同型的变量成为可能（虽然通常具有迷惑性）：

```
my ($bad_name, @bad_name, %bad_name);
```

Perl 不会因此犯迷糊，但是阅读代码的人则会。

#### Perl 5 使用 变化印记

变量的印记可能会随使用情况的不同而不同。例如，访问数组或哈希中的（标量）元素，印记就变成了美元符号（\$）：

```
my $hash_element = $hash{ $key };
my $array_element = $array[ $index ]

$hash{ $key }      = 'value';
$array[ $index ]   = 'item';
```

在最后两行中，将集合类型的标量成员用作 左值（赋值的目标，位于 = 符号的左侧）会 向 右值（所赋之值，位于 = 符号的右侧）施加标量上下文（context\_philosophy）。

类似地，访问数组或哈希中的多个元素——一个被称为 分片的操作——使用“at”符号（@） 作为印记并施加列表上下文：

```
my @hash_elements = @hash{ @keys };
my @array_elements = @array[ @indexes ];

my %hash;
@hash{ @keys }      = @values;
```

决定一个变量——标量、数组或哈希——类型最可靠的方法是看对它进行何种操作。标量支持所

有基本的操作，诸如字符串、数值、布尔处理。数组通过中括号支持对元素的下标访问。哈希通过大括号支持对元素的按键访问。

## 包限定名称

你偶尔会需要引用其他名称空间中的函数或变量。通常你需要通过类的 完全限定名称 来引用它。这些名称由双冒号（::）分隔的包名组成。就是说，`My::Fine::Package` 指向一个逻辑上的函数以及变量的集合。

虽然标准命名规则也适用于包名，照惯例，用户定义的包的名称通常以大写字母开头。Perl 核心为内建编译命令（pragmas）保留了小写包名，如 `strict` 和 `warnings`。这是由社区指南 而非 Perl 自身强制的规矩。

Perl 5 没有嵌套名称空间。`Some::Package` 和 `Some::Package::Refinement` 的关系仅仅是 存储机制上的，并无第二重暗示指出在包关系上它们是父子还是兄弟。当 Perl 在 `Some::Package::Refinement` 中查找某一符号时，它向 `main::` 符号表查找代表 `Some::` 名称空间的符号，接着再在其中 查找 `Package::` 名称空间，如此等等。当你选择名称和组织代码时，使实体之间的 逻辑 关系明显是你的责任。

## 变量

Perl 中的 变量 是一个存放值（values）的地方。你可以直接处理一个值，但除了一些琐碎的代码之外，所有程序都在和变量打交道。变量是一个间接层，按照变量 `a`、`b` 和 `c` 来解释勾股定理较你能想象的直角三角形边长而言更为清楚。这乍看好像很理所当然，但是要写出健壮的、良好设计的、可测试、可组合的程序，你必须在任何可能的地方考虑广泛性。

## 变量作用域

变量同样有可见性，取决于它们的作用域（`scope`）。你能碰到的大多数变量拥有词法作用域（`lexical_scope`）。记住，文件也有自己的词法作用域，诸如文件内部的 `package` 声明并不创建新的作用域：

```
package Store::Toy;

our $discount = 0.10;

package Store::Music;

# $Store::Toy::discount 仍然可以通过 $discount 访问
say "Our current discount is $discount!";
```

## 变量印记

在 Perl 5

中，无论是标量、数组或是哈希，变量声明时的印记决定了它的类型。访问该变量所用的印记则决定了所访问的值的类型。变量上的印记因使用场合而不同，例如，你将 `@values` 声明为一个数组。你可以用 `$values[0]` 访问它的第一个元素——一个单值。你可以通过 `@values[ @indices ]` 来得到其中的一部分值的列表。参见“数组”（`arrays`）和“哈希”（`hashes`）这两个小节以获取更多信息。

## 匿名变量

Perl 5 中的变量不需要名称，Perl 能够另行分配存储空间而不必将它们存放在词法垫板（`lexical pad`）或是符号表中。它们被称为 匿名变量。访问到它们的唯一办法就是通过引用（`references`）。

## 变量、类型和强制转换

Perl 5

中的变量并不把类型强加给它们的值。你可以在程序的某行将某字符串存入一个变量，然后下一行再将一个数字追加到这个变量，第三行重新将一个函数分配给一个引用（`function references`）。这些变量的值是灵活的（或者说是动态的），但是变量的类型是静态的。标量变量就只能持有标量。数组就只能存放列表。哈希只能包含偶数个元素的键值对列表。

以两种不同的措辞谈论类型听起来很奇怪：某类型的变量和某类型的数据，但 Perl 确实如此进行区分。这两种类别的类型最明确的指代方式便是 容器类型 和 值类型。

对一个变量进行赋值可能引起强制转换（`coercion`）。文档中记载的获得某数组中元素个数的方式就是在标量上下文（`context philosophy`）中对数组进行求值。因为标量变量只能存放标量，将一个数组赋值给标量对此操作施加标量上下文并得出该数组内元素的个数：

```
my $count = @items;
```

变量类型、印记和上下文之间的关系对于正确理解 Perl 来说很重要。

## 值

高效的 Perl 程序依赖于对值的精确的表示方式及操作。

计算机程序包含 变量：持有 值 的容器。值是程序实际操作的数据。虽然很容易就能描述某项数据——你姑妈的名字和地址、你的办公室和月球上一堂高尔夫课程的距离，或者去年你吃的曲奇的重量——但是这些和数据格式相关的规则往往很严格。编写高效的程序通常意味着理解能表示该数据的最佳（最简单、最快、最紧凑或者最容易的）方式。

虽然程序的结构很大程度上依赖你用适合的变量为数据建立模型的方式，但如果不能准确地容下数据本身（即：值）这些变量将会是毫无意义的。

## 字符串

字符串 是无特定格式、无特定内容的一小片数据，对程序来说也没有其他特别的含义。它可能是你的名字，可能是从你的硬盘上读取的图像文件的内容，还可能是 Perl 程序本身。一个字符串在你赋予它意义之前对程序而言毫无意义。字符串是一块由某种形式的括号括起的固定长度的数据，Perl 字符串还能随着你的添加和抽减而变化。

最普通的字符串分隔符是单双引号：

```
my $name      = B<'Donner Odinson, Bringer of Despair'>;
my $address   = B<"Room 539, Bilskirnir, Valhalla">;
```

单引号字符串 中的字符代表它们自身的字面含义，但有两处例外。通过反斜杠转义，你可以在一个单引号字符串中内嵌另一个单引号字符串：

```
my $reminder = 'DonB<\> t forget to escape the single quote!';
```

如果反斜杠在字符串末尾，你可以用另一个反斜杠将其转义，否则 Perl 语法分析器将认为你是在转义结尾的单引号：

```
my $exception = 'This string ends with a backslash, not a quote:
B<\\>';
```

其他所有反斜杠都是作为字面值出现在字符串中，但如果出现两个相邻的反斜杠，则前者将后者转义。

```
is('Modern B<\\> Perl', 'Modern B<\\> Perl', 'single quotes backslash
escaping');
```

双引号字符串 有着更为复杂（通常也更有用）的行为。举例来说，你可以将非打印字符编码进一个字符串：

```
my $tab       = "B<\t>";
my $newline   = "B<\n>";
my $carriage  = "B<\r>";
my $formfeed  = "B<\f>";
my $backspace = "B<\b>";
```

字符串定义可以横跨多个逻辑行，如下列两个字符串是等价的：

```
my $escaped = "two\nlines";
```

```
my $literal = "two
lines";
is( $escaped, $literal, '\n and newline are equivalent' );
```

你 可以 将这些字符直接输入字符串，但在视觉上通常很难将一个制表符和四个（或者两个再或者八个）空格区分开来。

在双引号字符串内，你也可以将标量或数组的值 内插 入字符串，使得变量的值成为该字符串的一部分，就好像你直接对其进行拼接操作：

```
my $factoid = "Did you know that B<$name> lives at B<$address>?";

# 等同于

my $factoid = 'Did you know that ' . $name . ' lives at ' . $address .
'?';
```

在双引号字符串内，你可以通过 转义 插入一个字面双引号（即，在它前面加上一个反斜杠）：

```
my $quote = "\"Ouch,\"", he cried.  \"That I<hurt>!\";
```

如果你觉得这简直丑陋到令人发指，那你可以使用另外的 引号操作符。q 操作符进行单引号操作，而 qq

操作符是双引号。每种情况下，你都可以自行选择字符串分隔符。紧随操作符后的字符决定了该字符串的开头和结尾。如果该字符是诸如大括号之类标点对中的开标点，则对应的闭标点为字符串结尾的分隔符。除上述之外，字符本身将作为开头和结尾的分隔符。

```
my $quote      = B<qq{>"Ouch", he said.  "That I<hurt>!"B<}>;
my $reminder   = B<q^>Didn't need to escape the single quote!B<^>;
my $complaint  = B<q{>It's too early to be awake.B<}>;
```

即使你可以通过一系列内嵌的转义字符来声明一个复杂的字符串，有些时候跨行声明一个多行字符串更为方便。heredoc 的语法让你可以以另一种方式进行多行字符串赋值：

```
my $blurb =<<'END_BLURB';

He looked up.  "Time is never on our side, my child.  Do you see the
irony?
All they know is change.  Change is the constant on which they all can
agree.  Whereas we, born out of time to remain perfect and perfectly
self-aware, can only suffer change if we pursue it.  It is against our
nature.  We rebel against that change.  Shall we consider them greater
for it?"
END_BLURB
```

<<'END\_BLURB' 语法有三个部分。两重尖括号引入了 heredoc。引号决定此 heredoc 在处理变量内插和转义字符时遵循单引号还是双引号的规则。它们是可选的，默认按双引号规则处理。END\_BLURB 自身可以是任意标识符，由 Perl 5 语法分析器用作标示结束的分隔符。

注意，不管 heredoc 声明部分缩进多少，结束分隔符 必须 位于一行的开头：

```

sub some_function {
    my $ingredients =<<'END_INGREDIENTS';
    Two eggs
    One cup flour
    Two ounces butter
    One-quarter teaspoon salt
    One cup milk
    One drop vanilla
    Season to taste
END_INGREDIENTS
}

```

如果标识符以空白开始，则等数量的空白必须在结束分隔符中出现。即便你确实缩进了该标识符，Perl 5 也不会从 heredoc 主体部分的每行行首移除等量的空白。

你也可以在其他上下文中使用字符串，如布尔上下文和数值上下文，它的内容将决定结果的值（coercion）。

## Unicode and Strings

Unicode 一套用于表示世界手写语言字符的系统。虽然绝大部分英语文本使用一个仅含 127 个字符的字符集（该字符集要求7位的存储空间，并且对于8位的字节来说正合适），但举例来说，不相信需要变音的人是幼稚（“naïve”，注意第三个字符）的。

Perl 5 可以按两种相关但不同的方式表示一个字符串：

### Unicode 字符序列

Unicode 字符集包含了绝大多数语言的书写字符及其他一些符号。每一个字符有一个对应的 代码点，一个在 Unicode 字符集中标识该字符的唯一数字。

### 八进制序列

二进制数据是一个 八进制 的序列——8位的数字，每一个都可以表示一个 0 到 255 直接的数字。

为什么称为 八进制数 而非 字节？将 Unicode 作为字符考虑，而不要考虑这些字符在内存中表示方式所占的特定长度。做出一个字符一个字节的假设只会带给你无尽的 Unicode 悲哀。

Unicode 字符串和二进制字符串看上去很像。它们都有 length() 函数，你可以对它们进行标准

字符串操作，如：拼接、分片和正则表达式处理。任何非纯二进制的字符串都是文本数据，它应是一个 Unicode 字符序列。

虽都是八进制序列，数据还是会因操作系统对磁盘数据的表示方式，以及其他用户、网络等原因而不同，因此 Perl

无从得知读取的某块数据究竟是图像文件、文本文档还是其他东西。默认地，Perl 将所有读入的数据按八进制序列处理。给字符串内容赋予额外的含义是你的责任。

## 字符编码

Unicode 字符串是一个表示一系列字符的八进制序列。Unicode 编码将八进制序列映射到字符上。一些编码方式，如 UTF-8，可以编码 Unicode 字符集中的所有字符。其他编码方式只能表示 Unicode 字符的一个子集。例如，ASCII 编码纯英语文本且不含重音字符，Latin-1 可以表示使用拉丁字母表的大部分语言的文本。

如果你总是以正确的方式编解码程序输入输出，你将避开许多麻烦。

## 文件句柄中的 Unicode

Unicode 输入的一个来源就是文件句柄 (files)。如果你告知 Perl 某特定的文件句柄对已编码的文本进行操作，Perl 能够将数据自动转换为 Unicode 字符串。为实现此功能，请向内置 `open` 操作符的模式添加一个 `IO` 层。`IO` 层包装了输入输出并对数据进行某种形式的转换。在此种情况下，`:utf8` 层将对 UTF-8 数据进行解码：

```
use autodie;

open my $fh, '<:utf8', $textfile;

my $unicode_string = <$fh>;
```

你也可以用 `binmode` 对已经打开的文件句柄进行修改，无论是输入还是输出：

```
binmode $fh, ':utf8';
my $unicode_string = <$fh>;

binmode STDOUT, ':utf8';
say $unicode_string;
```

不启用 `utf8` 模式时，向某文件句柄打印 Unicode 字符串会得到一个警告 (Wide character in %s)，因为 文件包含的是八进制数据而非 Unicode 字符。

## 数据中的 Unicode

Encode 核心模块提供了一个名为 `decode()` 的函数来将已知格式标量数据转换成一个 Unicode 字符串。例如，如果你有的是 UTF-8 数据：

```
my $string = decode('utf8', $data);
```

对应的 `encode()` 函数将 Perl 内部编码转换为所需输出编码：

```
my $latin1 = encode('iso-8859-1', $string);
```

## 程序中的 Unicode

在你的程序中包含 Unicode 字符有三种方式。最简易的方法是利用 `utf8` 编译命令 (pragmas)，它告诉 Perl 语法分析器将后续源代码编码解释为 UTF-8。这允许你在字符串和标识符中使用 Unicode 字符：

```
use utf8;

sub E<pound>_to_E<yen> { ... }

my $pounds = E<pound>_to_E<yen>('1000E<pound>');
```

要 编写 这些代码，你的文本编辑器必须理解 UTF-8 并且你必须按正确的编码保存该文件。

你也可以使用 Unicode 转义序列来表示字符编码。\`x{}` 语法代表一个单独的字符。将该字符的 Unicode 数字的十六进制表示放入大括号内：

```
my $escaped_thorn = "\x{00FE}";
```

注意这些转义序列仅在双引号字符串内内插。

一些 Unicode 字符有自己的名称。虽然这相当详细，但是比起 Unicode 数字来说更加易读。你必须使用 `charnames` 编译命令来启用它。用 \`N{}` 转义语法来指代它：

```
use charnames ':full';
use Test::More tests => 1;

my $escaped_thorn = "\x{00FE}";
my $named_thorn   = "\N{LATIN SMALL LETTER THORN}";

is( $escaped_thorn, $named_thorn, 'Thorn equivalence check' );
```

你可以在正则表达式内使用 \`x{}` 和 \`N{}` 形式，就像在他处合理地使用一个字符或字符串。

## 隐式转换

Perl 中的大多数 Unicode

问题因字符串既可以是八进制数据序列也可以是字符序列而起。Perl 允许你通过隐式转换结合使用这些类型。当这些转换出错时，它们不会错得那么明显。

当 Perl 拼接一个八进制序列和一个 Unicode 字符序列时，它隐式地将八进制序列按 Latin-1 编码方式 解码。结果字符串包含 Unicode 字符。当你打印 Unicode 字符时，Perl 用 UTF-8 编码该字符串，因为 Latin-1 无法表示完整的 Unicode 字符集。

这个不对称性可导致 Unicode 字符串在输出时编码为 UTF-8 以及在输入时候解码为 Latin-1。

更糟糕的是，当文本只包含除重音外的英文字符时，这个 bug 会隐藏起来——因为这两种编码对每一个 字符的表示方式一致。

```
my $hello      = "Hello, ";
my $greeting = $hello . $name;
```

如果 `$name` 包含例如 Alice 之类的英文名字，你决不会察觉任何问题，因为 Latin-1 表示和 UTF-8 表示是一样的。

又如果，`$name` 包含类似 Jos□ 的名字，则 `$name` 可能包含下列几种可能的值：

- `$name` 四个 Unicode 字符；
- `$name` 四个代表它们各自 Unicode 字符的八进制 Latin-1；
- `$name` 五个代表它们各自 Unicode 字符的八进制 UTF-8。

字符串字面值有如下可能的情形：

ASCII 字符串字面值

```
my $hello = "Hello, ";
```

此字符串字面值包含八进制数据。

Latin-1 字符串字面值，无明确编码，例如：

```
my $hello = "E<iexcl>Hola, ";
```

此字符串字面值包含八进制数据。



非 ASCII 字符串字面值，通过 `utf8` 或 `encoding` 编译命令指定：

```
use utf8;
my $hello = "E<#x5E9>E<#x5DC>E<#x5D5>E<#x5DD>, ";
```

此字符串字面值包含 Unicode 字符。

如果 `$hello` 和 `$name` 都是 Unicode 字符串，拼接操作会产生另一个 Unicode 字符串。

如果两个字符串都是八进制流，Perl 会将它们拼接为一个新的八进制字符串。如果两者的都为编码方式相同的八进制值——例如，全为

Latin-1，则拼接操作将正常执行。如果两者并不共用 编码方式，拼接操作将把 UTF-8 数据追加到 Latin-1 数据之后，生成两种编码方式 都无法

识别的八进制序列。这种情况可能在用户以 UTF-8 编码输入姓名而问候语是 Latin-1 字符串字面值、程序又不能按任一编码解码时发生。

如果仅有一值为 UTF-8 字符串，Perl 将其他部分按 Latin-1 解码。如果这不是正确的编码方式，作为结果的 UTF-8

字符串也会是错误的。例如，用户输入的是 UTF-8 数据并且此字符串字面值是一个 Unicode 字符串，那么这个姓名将被错误地解码成五个 Unicode 字符，Jos□□ 而非 Jos□，因为 UTF-8 数据在作为 Latin-1 解码时有着不同的含义。

参见 `perldoc perluniintro`

以获取一份有关如下话题更为详细的解释：Unicode、编码、以及如何在 Unicode 世界中正确处理输入输出数据。

## 数字

Perl 同样提供对数字的支持，无论是整数还是浮点数。它们可以按科学计数法、二进制、八进制和十六进制形式给出：

```
my $integer    = 42;
my $float      = 0.007;
my $sci_float  = 1.02e14;
my $binary     = B<0b>101010;
my $octal      = B<0>52;
my $hex        = B<0x>20;
```

加粗的字符是二进制、八进制、十六进制对应的数值前缀。注意开头的零总是意味着八进制模式，这偶尔会引发意想不到的迷惑。

即使你可以在 Perl 5 中明确且精准地表示一个浮点值，Perl 5 内部将它们存储为二进制格式。某些具体情况下无法精确地比较浮点值，详情请参考 `perldoc perlnumber`。

你无法用逗号按千分隔数值字面值，因为语法分析器会将逗号解释为逗号操作符。为此你 可以在数字内使用下划线。语法分析器会将其作为不可见的字符对待，但你程序的读者不会。下面这些是等价的：

```
my $billion = 1000000000;
my $billion = 1_000_000_000;
my $billion = 10_0_00_00_0_0_0;
```

请考虑使用最易读的方式。

由于强制转换 (coercion)，Perl 程序员几乎不用担心将外部读取的文本转换为数字一事。在 数值上下文中 Perl

会将所有看上去像数字的东西统一作为数字对待。虽然它几乎每都能正确处理此类事项，有时知道一下某物看上去是不是像数字也不赖。核心模块 `Scalar::Util` 包含一个名叫 `looks_like_number` 的函数，如果 Perl 将某物做数值考虑，它就会返回一个为真的值。

来自 CPAN 的 `Regexp::Common`

模块同样提供了若干经过良好测试的正则表达式来识别数值的有效类型（全数字、整数、浮点数值）。

## Undef

Perl 5 中由 `undef` 代表所有未赋值、未定义和未知的值。已声明但未定义的标量包含 `undef`：

```
my $name = undef;    # 多余的赋值
my $rank;            # 同样包含 undef
```

在布尔上下文中对 `undef` 求值得到假。将 `undef` 内插入一个字符串——或者对其在字符串上下文中求值——将产生一个 `uninitialized value` 的警告：

```
my $undefined;
my $defined = $undefined . '... and so forth';
```

.....产生：

```
Use of uninitialized value $undefined in concatenation (.) or string...
```

## 空列表

当用于一个赋值操作的右手边时，`()` 结构代表一个空列表。当在标量上下文中求值时得到 `undef`。在列表上下文中，它在效果上就是一个空列表。

当用于一个赋值操作的左手边时，`()` 规定了列表上下文。不用临时变量，计算一个表达式在列表上下文中返回结果的个数，你可以使用如下惯用语（idioms）：

```
my $count = B<()> = get_all_clown_hats();
```

由于赋值操作符的右结合性（*associativity*），Perl 先对第二个赋值操作通过在列表上下文中调用 `get_all_clown_hats()` 求值，这会产生一个列表。

对空列表的赋值丢弃了该列表全部的值，但这个赋值在标量上下文中发生，将求得赋值操作右手边的元素 的个数。结果就是 `$count` 包含了从 `get_all_clown_hats()` 返回的列表中元素的个数。

目前你不用理解这段代码中所有的隐含内容，但是它确实实地展示了 Perl 设计中区区几个基础功能的 组合就能产生如此有趣且有用的行为。

## 列表

列表是一个由逗号分隔、包含一个或多个表达式的组。

列表可能在源代码中逐字出现：

```
my @first_fibs = (1, 1, 2, 3, 5, 8, 13, 21);
```

.....或作为赋值的目标:

```
my ($package, $filename, $line) = caller();
```

.....或作为一系列表达式:

```
say name(), ' => ', age();
```

值得一提的是 创建

列表并不需要括号, 这些例子中 (括号) 出现的地方是为了让表达式成组出现, 以改变这些表达式的 优先级 (precedence)。

你可以用范围操作符以一种紧凑的方式创建一个字面值列表:

```
my @chars = 'a' .. 'z';
my @count = 13 .. 27;
```

.....同时, 你也可以用 `qw()` 操作符以空白符分隔一字符串字面值, 并创建一个字符串列表:

```
my @stooges = qw( Larry Curly Moe Shemp Joey Kenny );
```

如果 `qw()` 包含逗号或注释符 (`#`), Perl 会产生一条警告。不仅因为这些字符在 `qw()` 中出现的机会很少, 而且它们的出现往往意味着疏忽。

列表可以 (通常也) 作为一个表达式的结果, 但是这些列表不以字面形式在源代码中出现。

在 Perl

中, 列表和数组概念之间不可以交换。列表是值而数组是容器。你可以在一个数组中存放一个列表, 也可以将一个数组强转为列表, 但它们是不同的实体。举例来说, 按下标访问列表通常在列表上下文中出现。

按下标访问数组通常在标量上下文 (为获取单个元素) 或列表上下文中出现 (为数组分片):

```
# enable say and other features (see preface)
use Modern::Perl;

# you do not need to understand this all
sub context
{
    my $context = wantarray();

    say defined $context
        ? $context
        ? 'list'
        : 'scalar'
        : 'void';
    return 0;
}

my @list_slice = (1, 2, 3)[context()];
my @array_slice = @list_slice[context()];
my $array_index = $array_slice[context()];

# say imposes list context
say context();
```

```
# void context is obvious
context()
```

## 控制流程

### Perl 的基本 控制流程

相当直截了当。程序执行过程起始于程序开头（被执行文件的第一行） 然后一直到结尾：

```
say 'At start';
say 'In middle';
say 'At end';
```

大多数程序需要更为复杂的控制流程。Perl 的 控制流程语句改变了程序执行的顺序——程序  
中接下来要发生的——依赖于任意复杂的表达式的值。

### 分支语句

if 语句对一条件表达式求值并仅在此条件表达式的值为真时执行相关动作：

```
say 'Hello, Bob!' if $name eq 'Bob';
```

这种后缀形式在表达式较简单时很有用。代码块形式则将多个表达式组合成单一单元：

```
if ($name eq 'Bob')
{
    say 'Hello, Bob!';
    found_bob();
}
```

虽然代码块形式要求条件两边有括号，但后缀形式则相反。条件表达式也可以相对复杂：

```
if ($name eq 'Bob' && not greeted_bob())
{
    say 'Hello, Bob!';
    found_bob();
}
```

.....虽然在此种情况下，采用后缀形式的括号形式可以使其稍显清晰  
同时也会因此对是否使用后缀形式产生争论：

```
greet_bob() if ($name eq 'Bob' && not greeted_bob());
```

unless 语句是 if 的否定形式。Perl 在条件表达式的值为 假 时执行 所需操作：

```
say "You're no Bob!" unless $name eq 'Bob';
```

和 if 类似，unless 也有代码块形式。不同于 if，unless 的代码块形式相比其  
后缀形式来说很少见：

```
unless (is_leap_year() and is_full_moon())
{
```

```

        frolic();
        gambol();
    }

```

`unless` 很适合后缀条件，特别是函数中的参数验证 (`postfix_parameter_validation`) :

```

sub frolic
{
    return unless @_;

    for my $chant (@_)
    {
        ...
    }
}

```

条件一多 `unless` 就会变得难以阅读，这便是它很少以代码块形式出现的原因之一。

`if` 以及 `unless` 都可以搭配 `else` 语句，它提供了当条件表达式的值不为真 (`if`) 或假 (`unless`) 时运行的代码：

```

if ($name eq 'Bob')
{
    say 'Hi, Bob!';
    greet_user();
}
else
{
    say "I don't know you.";
    shun_user();
}

```

`else` 代码块允许你按不同于自身的方式重写 `if` 和 `unless` 条件语句：

```

B<unless> ($name eq 'Bob')
{
    say "I don't know you.";
    shun_user();
}
else
{
    say 'Hi, Bob!';
    greet_user();
}

```

如果你大声读出前面这个例子，你会发现一例不合适的伪代码用词：“除非该名字为 Bob，做这件事。否则 做那件。” 隐含的双重否定可能会很迷惑。Perl 提供了 `if` 和 `unless` 使得你可以按最为自然、最为上口的方式对条件语句进行组织。同样地，你可以根据比较操作符在肯定和否定断言之间进行选择：

```

if ($name B<ne> 'Bob')
{
    say "I don't know you.";
    shun_user();
}
else

```

```

{
    say 'Hi, Bob!';
    greet_user();
}

```

因 `else` 代码块的出现而隐含着的双重否定示意此代码组织方式是不合理的。

在单个 `else` 之前、`if` 代码块之后可以跟一个或多个 `elsif` 语句。`elsif` 代码块的使用数量不限，但你不可以改变块出现的顺序：

```

if ($name eq 'Bob')
{
    say 'Hi, Bob!';
    greet_user();
}
elsif ($name eq 'Jim')
{
    say 'Hi, Jim!';
    greet_user();
}
else
{
    say "You're not my uncle.";
    shun_user();
}

```

你也可以在 `unless` 链内使用 `elsif` 块，但是结果代码会不那么清晰。不存在 `elseunless` 的说法。

同样也没有 `else if` 这一语法结构 Larry 出于美学原因以及 Ada 编程语言的现有技术选择了 `elsif`，因此，这段代码含有语法错误：

```

if ($name eq 'Rick')
{
    say 'Hi, cousin!';
}

# warning; syntax error
else if ($name eq 'Kristen')
{
    say 'Hi, cousin-in-law!';
}

```

## 三元条件操作符

### 三元条件

操作符提供了另一种方法来控制流程。它先将条件表达式求值，并由此对两不同结果之一求值：

```
my $time_suffix = after_noon($time) ? 'morning' : 'afternoon';
```

条件表达式位于问号 (?) 之前，冒号 (:

) 分隔两种选择。这两个选择分支可以是字面值或者 (带括号)

任意复杂的表达式，包括其他的三元条件表达式 (以可读性为代价)。

一个有趣但晦涩的惯用语便是使用三元条件语句在两个候选 变量 之间做出选择，而非仅仅对值：

```
push @{ rand() > 0.5 ? \@red_team : \@blue_team }, Player->new();
```

再次提醒，请对清晰和简略各自带来的利益进行权衡。

## 短路求值

当遇到由多重待求值的表达式组成的复杂表达式时，Perl 将做出名为 短路求值 的行为。如果 Perl 可以决定一个复杂的表达式整体的值是真还是假，那么它不会对后续子表达式求值。用例子说明会更加明白：

```
# see preface
use Test::More 'no_plan';

say "Both true!" if ok(1, 'first subexpression')
                  && ok(1, 'second subexpression');

done_testing();
```

ok() 的返回值 (testing) 是对第一个参数求值得到的。

这个例子打印出：

```
ok 1 - first subexpression
ok 2 - second subexpression
Both true!
```

当第一个子表达式——对 ok 的第一次调用——求得真值时，Perl 必须对第二个子表达式求值。  
当第一个表达式求得假值时，整个表达式不为真，因此无需检查后续子表达式。

```
say "Both true!" if ok(0, 'first subexpression')
                  && ok(1, 'second subexpression');
```

这个例子打印出：

```
not ok 1 - first subexpression
```

虽然第二个子表达式显然为真，Perl 绝不对它求值。对于“一真即真”复杂条件表达式来说，其逻辑也是相似的：

```
say "Either true!" if ok(1, 'first subexpression')
                    || ok(1, 'second subexpression');
```

这个例子打印出：

```
ok 1 - first subexpression
Either true!
```

再次，第一个子表达式为真，Perl 可以避免对第二个子表达式求值。如果第一个子表达式为假，则对第二个子表达式求值的结果将决定整个表达式的真假。

除了可以让你避免潜在的昂贵计算，短路求值还可以帮助你避免错误和警告：

```
if (exists $barbeque{pork} and $barbeque{pork} eq 'shoulder') { ... }
```

## 条件语句相关的上下文

这些条件语句——`if`、`unless`，以及三元条件表达式——总是在布尔上下文（`context_philosophy`）中 对一个表达式进行求值。由于 `eq`、`==`、`ne` 和 `!=` 这类操作符在求值时总产生布尔结果，Perl 将其他表达式的求值结果——包括变量和值——强制转换为布尔形式。对空哈希和数组求值得假。

Perl 5 没有单一的真值、也没有单一的假值。任何求值为 0 的数字为假。包括 0、0.0、0e0、0x0 等等。空字符串（`''`）以及 `"0"` 求值得假，但是字符串如 `"0.0"`、`"0e0"` 等则不然。惯用语 `"0 but true"` 在数值上下文中求值得 0，但其值因字符串内容而为真。空列表和 `undef` 都为假。空数组和哈希在标量上下文中返回 0，因此它们在布尔上下文中为假。

带有单个元素的数组——即便该元素是 `undef`——在布尔上下文中求值得真，含任何元素的哈希——即使是一键一值两个 `undef`——在布尔上下文中也得真。

CPAN 模块 `Want` 允许你在你的函数内检测布尔上下文。核心编译命令 `overloading`（`overloading`、`pragmas`）允许你指定自己的数据类型在布尔上下文中求得的值。

## 循环语句

Perl 也提供了供循环和迭代使用的若干语句。

### `foreach`

风格的循环对一表达式求值而产生一个列表，接着执行某语句或代码块直到它消耗完该列表：

```
foreach (1 .. 10)
{
    say "$_ * $_ = ", $_ * $_;
}
```

这个例子使用范围操作符产生了一个包括一到十在内的整数列表。`foreach` 语句在其之上循环，依次设置话题变量（`$_`，参见 `default_scalar_variable`）。Perl 针对每个整数执行代码块并打印该整数的平方。

Perl 将 `foreach` 和 `for`

可互换地看待。循环余下部分的语句决定了循环的行为。尽管有经验的 Perl 程序员 倾向于用 `foreach` 循环来指代自动迭代循环，你可以在任何用到 `foreach` 的地方安全地用 `for` 替代。

如同 `if` 及 `unless`，`for` 循环也有一个后缀形式：

```
say "$_ * $_ = ", $_ * $_ for 1 .. 10;
```

有关清晰、简略的建议同样适用于此。

你可以提供一个用于赋值变量以代替话题变量：

```
for my $i (1 .. 10)
{
    say "$i * $i = ", $i * $i;
}
```

如果你这样做了，Perl 将不会将话题变量（`$_`）赋值为迭代值。注意变量 `$i` 的作用域仅限于循环 内部。如果你在循环外定义了词法变量 `$i`，它的值将不受循环内变量的影响：

```
my $i = 'cow';
```



```

for my $i (1 .. 10)
{
    say "$i * $i = ", $i * $i;
}

is( $i, 'cow', 'Lexical variable not overwritten in outer scope' );

```

甚至你不将该迭代变量重新声明为词法变量，它仍会被局部化：

```

my $i = 'horse';

for $i (1 .. 10)
{
    say "$i * $i = ", $i * $i;
}

is( $i, 'horse', 'Lexical variable still not overwritten in outer
scope' );

```

## 迭代和别名

`for` 循环将迭代变量 别名化 为迭代中的值，以便你可以直接在迭代过程中修改：

```

my @nums = 1 .. 10;

$_ **= 2 for @nums;

is( $nums[0], 1, '1 * 1 is 1' );
is( $nums[1], 4, '2 * 2 is 4' );

...

is( $nums[9], 100, '10 * 10 is 100' );

```

代码块形式的 `foreach` 循环同样也会别名化：

```

for my $num (@nums)
{
    $num **= 2;
}

```

.....用话题变量迭代时也会：

```

for (@nums)
{
    $_ **= 2;
}

```

你 cannot 通过别名修改 常量 值，然而：

```

for (qw( Huex Dewex Louie ))
{
    $_++;
    say;
}

```

.....会抛出有关修改只读值的异常。无论如何，这样并没有什么必要。

你偶尔会碰到 `for` 搭配别名化为 `$_` 的单个标量的用法：

```
for ($user_input)
{
    s/(\w)/\\$1/g; # 跳过非文字字符
    s/^\s*|\s$/g; # 修剪空白
}
```

## 迭代和作用范围

迭代器的作用范围连同话题变量一起是为常见的困惑之源。这种情况下，`some_function()` 有意修改 `$_`。如果 `some_function()` 调用了其他未经明确局部化 `$_` 而对其加以修改的代码，则在 `@values` 中的迭代值将改变。调试这样的问题很麻烦：

```
for (@values)
{
    some_function();
}

sub some_function
{
    s/foo/bar/;
}
```

如果你 必须 使用 `$_` 而非其他具名变量，请使用 `my $_` 将话题变量词法化：

```
sub some_function_called_later
{
    # 曾经是 $_ = shift;
    B<my> $_ = shift;

    s/foo/bar/;
    s/baz/quux/;

    return $_;
}
```

使用具名变量同时也避免了通过 `$_` 别名化的行为。

## C 语言风格的 For 循环

C 语言风格的 `for` 循环 允许程序员手动控制迭代：

```
for (my $i = 0; $i <= 10; $i += 2)
{
    say "$i * $i = ", $i * $i;
}
```

你必须手动给迭代变量赋值，因为现在已经不会自动给话题变量赋值。因此也没有别名化行为。虽然任何 已声明的词法变量的作用范围是代码块主体部分。一个 未 在迭代控制部分中明确声明的变量的内容 将 被覆盖：

```
my $i = 'pig';
```

```

for ($i = 0; $i <= 10; $i += 2)
{
    say "$i * $i = ", $i * $i;
}

isnt( $i, 'pig', '$i overwritten with a number' );

```

此循环的循环结构中有三个子表达式。第一个子表达式是初始化部分，它在第一次执行循环体前执行一次。第二个子表达式是条件比较子表达式。Perl 每次在循环体执行之前对其求值。当此子表达式值为真时循环继续。当此子表达式为假时，循环结束。最后一个子表达式在每次完成循环体时执行。

一个例子会使问题更加清晰：

```

# 在外部声明以避免在条件语句中出现声明
my $i;

for (
    # 循环初始化子表达式
    say 'Initializing' and $i = 0;

    # 条件比较子表达式
    say "Iteration: $i" and $i < 10;

    # 迭代结尾子表达式
    say 'Incrementing $i' and $i++
)
{
    say "$i * $i = ", $i * $i;
}

```

请注意迭代结尾子表达式后分号的省略以及低优先级 `and` 的使用。这种语法真是令人惊讶地考究。当可能时，尽量使用 `foreach` 风格的循环来代替 `for` 循环。

所有这三个表达式都是可选的。你可以这样编写无限循环：

```

for (;;) { ... }

```

## While 和 Until

`while` 循环会一直执行直到循环条件得出布尔假值。一个无限循环可以按如下方式清晰地写出：

```

while (1) { ... }

```

这意味着 `while` 循环的迭代结束条件和 `foreach` 循环中的有所不同，即对表达式自身求值并不产生任何副作用。如果 `@values` 拥有一个或多个元素，下列代码也是一个无限循环：

```

while (@values)
{
    say $values[0];
}

```

为避免此类无限 `while` 循环，你必须通过每次迭代修改 `@values` 数组以对其进行析构更新：

```
while (my $value = shift @values)
{
    say $value;
}
```

`until` 循环刚好和 `while` 进行相反的测试。迭代会在循环条件表达式为假时继续：

```
until ($finished_running)
{
    ...
}
```

`while` 循环的典型用法是从一个文件句柄中迭代读取输入：

```
use autodie;

open my $fh, '<', $file;

while (<$fh>)
{
    ...
}
```

Perl 5 对此 `while` 循环进行解释时，就好像你编写了如下代码：

```
while (defined($_ = <$fh>))
{
    ...
}
```

不用明确写出 `defined`，任何从该文件句柄读出、且求值得假——空行或只包含字符 0——的行会结束整个循环。当完成从文件中读取行的任务后，`readline (< >)` 操作符才返回一个未定义的值。

一个常见的错误就是忘记从读入的每一行移除行结束符，使用 `chomp` 关键字可以完成这项工作。

`while` 和 `until` 都可以写成后缀形式。Perl 5 中最简单的无限循环是：

```
1 while 1;
```

任何单个表达式对后缀式的 `while` 和 `until` 来说都是合适的，例如来自八十年代早期 8 位计算机的经典 "Hello, world!":

```
print "Hello, world! " while 1;
```

无限循环可能看上去有点笨拙，但它们实际上很有用。一个简单的 GUI 程序或网络服务器事件循环可以是：

```
$server->dispatch_results() until $should_shutdown;
```

对于更加复杂的表达式，使用 `do` 代码块：

```
do
```

```

{
    say 'What is your name?';
    my $name = <>;
    chomp $name;
    say "Hello, $name!" if $name;
} until (eof);

```

出于语法分析的目的，虽然可以包含若干表达式，`do` 代码块本身却是一个单一的表达式。不像 `while` 循环 的代码块形式，搭配后缀式 `while` 或 `until` 的 `do` 代码块至少会执行它的主体一次。这个结构相比其他循环形式来说较为少见，但在功能上毫不逊色。

## 循环中的循环

你可以在循环中嵌套其他循环：

```

for my $suit (@suits)
{
    for my $values (@card_values)
    {
        ...
    }
}

```

在这种情况下，明确地声明具名变量对可维护性来说必不可少。就迭代变量的作用范围而言，当使用话题变量时，引发混淆的潜在可能太大了。

嵌套使用 `foreach` 和 `while` 时，一个常见的错误是：很容易就可以用 `while` 循环将某文件句柄整得筋疲力尽。

```

use autodie;

open my $fh, '<', $some_file;

for my $prefix (@prefixes)
{
    # 不要这样用，很可能是有问题的代码
    while (<$fh>)
    {
        say $prefix, $_;
    }
}

```

在 `for` 循环之外打开文件句柄使得 `for` 循环在两次迭代间放着文件位置不动。在第二次迭代中，`while` 循环无事可做且不会执行循环体。为解决此问题，你可以在 `for` 循环内重新打开文件（理解上很简单，但是是对系统资源的不恰当使用），将整个文件吸入内存（文件太大可能就不行），或者在每次迭代时用 `seek` 使文件句柄回到文件开头（通常被忽视的选择）：

```

use autodie;

open my $fh, '<', $some_file;

for my $prefix (@prefixes)
{
    while (<$fh>)
    {

```

```

        say $prefix, $_;
    }

    seek $fh, 0, 0;
}

```

## 循环控制

有时你需要在用尽迭代条件前跳出循环。Perl 5 的标准控制机制——异常和 `return`——可以实现这个目的，但 你也可以使用 循环控制 语句。

### next

语句在下一个迭代点重新开始循环。当你已经完成本次迭代的所有任务后可以使用它。要循环读取文件中的每一行并跳过所有看上去像注释的内容，即以 `#` 开始的行，你可能会这样写：

```

while (<$fh>)
{
    B<next> if /\A#/;
    ...
}

```

`last` 语句立即结束循环。想在遇到结束分隔符后结束文件处理，你可能会这样写：

```

while (<$fh>)
{
    next if /\A#/;
    B<last> if /\A__END__/;
    ...
}

```

### redo

语句不对条件语句再次求值并重新开始本次迭代。这在少数情况下很有用：比如你想当即修改你读到的行，接着从头开始处理而不想影响其他的行时。例如，你可以实现一个把所有以反斜杠结尾的行拼接起来的笨拙的文件分析器：

```

while (my $line = <$fh>)
{
    chomp $line;

    # 匹配行尾的反斜杠
    if ($line =~ s{\\$}{})
    {
        $line .= <$fh>;
        redo;
    }

    ...
}

```

.....虽然这是一个做作的例子。

嵌套循环可能会使这些循环控制语句的使用变得模棱两可。在这些情况下，循环标签可以消除歧义：

```
OUTER:
while (<$fh>)
{
    chomp;

    INNER:
    for my $prefix (@prefixes)
    {
        next OUTER unless $prefix;
        say "$prefix: $_";
    }
}
```

如果你发现你自己正编写需要标签来控制流程的嵌套循环，考虑简化你的代码：也许将内层循环压缩为 函数会更清晰。

## Continue

`continue` 语法结构的行为类似于 `for` 循环的第三个子表达式。Perl 会在循环的每一次迭代过程执行 该代码块，即便你使用 `next` 来跳出某次迭代。你可以把它和 `while`、`until`、`with` 或 `for` 循环 搭配使用。`continue` 的例子比较罕见，但在想保证每一次迭代某事都会发生一次并无需顾及循环如何结束时很有用：

```
while ($i < 10 )
{
    next unless $i % 2;
    say $i;
}
continue
{
    say 'Continuing...';
    $i++;
}
```

## Given/When

`given` 语法结构是 Perl 5.10 的新特性。它将某表达式的值赋给话题变量并引入一个代码块：

```
given ($name)
{
    ...
}
```

不像 `for`，它不对某集合类型进行迭代。它在标量上下文中求值，并总是赋值给话题变量：

```
given (my $username = find_user())
{
    is( $username, $_, 'topic assignment happens automatically' );
}
```

given 同时对话题变量进行局部化来防止无意的修改:

```
given ('mouse')
{
    say;
    mouse_to_man( $_ );
    say;
}

sub mouse_to_man
{
    $_ = shift;
    s/mouse/man/;
}
```

单独出现时, 这个功能看上去没什么用。但 `when` 和其他功能组合时候就会非常有用。使用 `given` 来 话题化 某个值。 在关联的代码块之内, 多个 `when` 语句使用 智能匹配 语义逐表达式匹配话题。因此你可以这样编写石头剪刀布游戏:

```
my @options = ( \&rock, \&paper, \&scissors );

do
{
    say "Rock, Paper, Scissors! Pick one: ";
    chomp( my $user = <STDIN> );
    my $computer_match = $options[ rand @options ];
    $computer_match->( lc( $user ) );
} until (eof);

sub rock
{
    print "I chose rock.  ";

    given (shift)
    {
        when (/paper/) { say 'You win!' };
        when (/rock/) { say 'We tie!' };
        when (/scissors/) { say 'I win!' };
        default { say "I don't understand your move" };
    }
}

sub paper
{
    print "I chose paper.  ";

    given (shift)
    {
        when (/paper/) { say 'We tie!' };
        when (/rock/) { say 'I win!' };
        when (/scissors/) { say 'You win!' };
        default { say "I don't understand your move" };
    }
}
```



```

    }

    sub scissors
    {
        print "I chose scissors.  ";

        given (shift)
        {
            when (/paper/)      { say 'I win!' };
            when (/rock/)       { say 'You win!' };
            when (/scissors/)   { say 'We tie!' };
            default              { say "I don't understand your move" };
        }
    }
}

```

当无一条件匹配时, Perl 执行 `default` 规则。

CPAN 模块 `MooseX::MultiMethods` 允许利用其他技术来减少这段代码。

`when` 语法结构甚至更为强大。它可以匹配 (`smart_match`) 其他表达式类型诸如标量、集合、引用、任意比较表达式甚至是代码引用。

## Tailcalls

某函数内最后一个表达式是对其他函数的调用时, 这种情况称为 尾部调用  
——外层函数的返回值 就是内层函数的返回值:

```

sub log_and_greet_person
{
    my $name = shift;
    log( "Greeting $name" );

    return greet_person( $name );
}

```

此例中, `greet_person()` 直接返回给 `log_and_greet_person()` 的调用者要比先返回 `log_and_greet_person()` 再立刻 从 `log_and_greet_person()` 中返回要来得高效。直接将 `greet_person()` 返回给 `log_and_greet_person()` 的调用者, 是一种叫做 尾部调用优化 的优化手段。

Perl 5 不会自动检测它是否应该进行这种优化。

深度递归的代码 (`recursion`) , 特别是互相递归的代码, 会迅速消耗大量内存。使用尾部调用会减少有关内部控制流程记录的内存消耗, 这使得某些昂贵的算法变得可行。

## 标量

Perl 5 的基本数据类型是 标量 , 它表示单个、离散的值。这个值可以是字符串、整数、浮点数、文件句柄或者引用——但它总是单个值。标量值和标量上下文有着深层次的联系, 向标量赋值提供了 标量上下文。

标量可以是词法、包或全局 (`globals`)

) 变量。你可能只希望声明词法或包变量。标量变量的名称 必须符合 `names` 中提及的指导条款。标量变量开头总是使用美元符号 (\$) 作为印记 (sigils)。

它的反面不 全

正确, 标量印记可以应用对集合变量的操作以决定通过此操作可以访问到的数量型。 (arrays、hashes)

## 标量和类型

Perl 5

标量并非静态类型。一个标量可以包含任何标量类型的值而不必进行特殊的转换, 并且变量内值的类型可以改变。下列代码是合法的:

```
my $value;
$value = 123.456;
$value = 77;
$value = "I am Chuck's big toe.";
$value = Store::IceCream->new();
```

虽然这是 合法的

, 它可能会导致混乱。请给你的变量选择具有描述性、唯一的名字以避免这种迷惑。

标量求值的类型上下文可能会使 Perl 对该标量的值进行强制转换 (coercion)。例如, 你可以把一个

标量的内容作为字符串对待, 即使你不明确地将其赋值给一个字符串。

```
my $zip_code      = 97006;
my $city_state_zip = 'Beaverton, Oregon' . ' ' . $zip_code;
```

你也可以对字符串进行数学操作:

```
my $call_sign = 'KBMIU';
my $next_sign = $call_sign++;

# 也可以这样
$next_sign    = ++$call_sign;

# 但是 I<不可以> 这样写:
$next_sign    = $call_sign + 1;
```

这个神奇的字符串自增操作并没有对应的神奇字符串自减操作。你不能通过编写 `$call_sign--` 得到之前的字符串。

字符串自增操作针对字符集和大小写将 `a` 变为 `b` 将 `z` 变为 `aa`。同样的, `zz9` 变为 `AA0`, 但是 `zz09` 就会成为 `zz10`——数字会照常进位, 但不会延续到字母部分。

在字符串上下文中对引用 (references

) 求值会得到一个字符串, 而在数值上下文对引用求值会

得到一个数字。两个操作都不对引用做出修改, 但你不能从得到的字符串或数字中重新创建该引用。

```
my $authors      = [qw( Pratchett Vinge Conway )];
my $stringy_ref = '' . $authors;
my $numeric_ref  = 0 + $authors;
```

`$authors` 仍旧可以作为引用使用, 但是 `$stringy_ref` 是一个和该引用没有关联的字符串, `$numeric_ref` 是一个和该引用没有关联的数字。

使得所有这些强制转换和操作成为可能的是，因为 Perl 5 标量可以包含数值部分以及字符串部分。Perl 5 中表示标量的内部数据结构有一个数值槽和一个字符串槽。在数值上下文中访问一个字符串最终产生一个拥有字符串和数值槽两者的标量。核心模块 `Scalar::Util` 中的 `dualvar()` 函数允许你对一个标量的这两个值进行改动。相似地，模块中的 `looks_like_number()` 函数在提供标量值被 Perl 5 认作数字时返回真。

标量没有单独针对布尔值的槽。在布尔上下文中，空字符串（''）和 '0' 为假。其他所有字符串为真。在布尔上下文中，求值得零的数字（0、0.0 和 0e0）为假，其他所有数字为真。

注意 字符串 '0.0' 和 '0e0' 是真值，这是 Perl 5 将看上去像数字的事物和数字区别对待的地方之一。

另有一个总是为假的值：`undef`。这是未定义变量和它本身的值。

## 数组

Perl 5 数组 是存放零个或多个标量的数据结构。它们是一等 数据结构，意味着 Perl 5 在语言级别提供了单独的数据类型。数组支持下标访问，即，你可以按整数下标访问某数组中每一个独立的成员。

@ 印记是数组的标识。要想声明一个数组：

```
my @items;
```

## 数组元素

在 Perl 5 中 访问 数组中独立的元素要求标量印记。Perl 5（和你）可以识别出 `$cats[0]` 指的是 `@cats` 数组 而不必理会印记的改动，因为中括号（[]）始终标明对集合变量的下标访问。用简单的话来说，意思就是“通过一个整数在一组东西里查找某件东西”。

数组中第一个元素的下标为零：

```
# @cats 包含一系列 Cat（猫）对象
my $first_cat = $cats[0];
```

数组中最后一个元素的下标取决于数组中元素的个数。标量上下文中的数组（因标量赋值、字符串拼接、加法或布尔上下文）求值得出数组中所含元素的个数：

```
# 标量赋值
my $num_cats = @cats;

# 字符串拼接
say 'I have ' . @cats . ' cats!';

# 加法
my $num_animals = @cats + @dogs + @fish;

# 布尔上下文
say 'Yep, a cat owner!' if @cats;
```

如果你需要得到最后一个元素指定的下标，将数组元素的个数减去一即可（因为数组下标从零开始）：

```
my $first_index = 0;
my $last_index  = @cats - 1;

say    'My first cat has an index of $first_index, '
      . 'and my last cat has an index of $last_index.'
```

你也可以使用数组的特殊变量形式来找出最后一个下标，将 @ 数组印记替换为更为笨拙的 \$#：

```
my $first_index = 0;
B<my $last_index = $#cats;>

say    'My first cat has an index of $first_index, '
      . 'and my last cat has an index of $last_index.'
```

然而，读上去可能不那么好。大多数时间你不会需要那种用法，因为你还可以使用负偏移量从末尾而不是 开头访问一个数组。数组的最后一个元素可以由下标 -1 取到。倒数第二个元素可以用下标 -2，等等。 例如：

```
my $last_cat      = $cats[-1];
my $second_to_last_cat = $cats[-2];
```

你可以通过对 \$# 赋值来调整数组大小。如果你对数组进行收缩，Perl 将丢弃所有不符合调整后尺寸的值。 如果你扩展一个数组，Perl 将把 undef 填入多出来的位置。

## 数组赋值

你可以通过对某下标位置直接对数组赋值：

```
my @cats;
$cats[0] = 'Daisy';
$cats[1] = 'Petunia';
$cats[2] = 'Tuxedo';
$cats[3] = 'Jack';
$cats[4] = 'Brad';
```

Perl 5 的数组是可变的。它没有静态尺寸，它会按需扩展和收缩。

你不也必按顺序赋值。如果你对超出范围的位置赋值的话，Perl 会将数组扩展到合适的大小，并向夹在中间的所有空槽都填入 undef。

多行赋值太过冗长。你可以用列表对一个数组进行一步到位的初始化：

```
my @cats = ( 'Daisy', 'Petunia', 'Tuxedo', 'Jack', 'Brad' );
```

注意括号 并不 创建一个列表。没有了括号，根据操作符优先级（precedence），Daisy 将赋值成为数组的第一个也是唯一一个元素。

任何在列表上下文中产生列表的表达式可以对数组进行赋值：

```
my @cats      = get_cat_list();
my @timeinfo  = localtime();
my @nums      = 1 .. 10;
```

对数组的标量元素赋值将施加标量上下文，对数组整体进行赋值则施加列表上下文。

要清空一个数组，用空列表对其赋值：

```
my @dates = ( 1969, 2001, 2010, 2051, 1787 );
...
@dates    = ();
```

由于新声明的数组起始为空, `my @items = ();` 便成为了 `my @items` 的加长版。请使用后者。

## 数组分片

你也可以通过一个名为 数组分片 的语法结构在列表上下文中访问一个数组。不同于对某数组元素进行标量访问, 此操作使用一系列下标以及数组印记 (@) :

```
my @youngest_cats = @cats[-1, -2];
my @oldest_cats   = @cats[0 .. 2];
my @selected_cats = @cats[ @indexes ];
```

你也可以对一个数组分片进行赋值:

```
@users[ @replace_indices ] = @replace_users;
```

一个分片可以包含零个或多个元素——包括一个:

```
# 单元素数组分片, I<列表> 上下文中的函数调用
@cats[-1] = get_more_cats();

# 单元素数组访问, I<标量> 上下文中的函数调用
$cats[-1] = get_more_cats();
```

数组分片和单元素访问之间唯一的语法区别就是开头的印记。语义 区别就比较大了: 一个数组分片总是强制列表上下文。任何在标量上下文中求值的数组分片都会引发错误:

```
Scalar value @cats[1] better written as $cats[1] at...
```

数组分片对用于下标的表达式强制列表上下文 (`context_philosophy`) :

```
# 函数调用于列表上下文
my @cats = @cats[ get_cat_indices() ];
```

## 数组操作

管理数组下标会是个麻烦。因为 Perl 5 可以按需扩展或收缩数组, 语言同时提供了若干操作以将数组作为栈、队列, 等对待。

`push` 和 `pop` 操作符各自在数组尾部添加和删除元素:

```
my @meals;

# 这里有什么可以吃的?
push @meals, qw( hamburgers pizza lasagna turnip );

# .....但是侄儿不喜欢蔬菜
pop @meals;
```

你可以用 `push` 向数组添加任何数量的元素。它的第二个参数是一个值列表。但你只能用 `pop` 一次删除一个元素。`push` 返回的是更新后数组中元素的个数。`pop` 返回删除的元素。

类似的, `unshift` 和 `shift` 向数组开头添加和删除元素:

```
# 扩展我们的烹饪视野
unshift @meals, qw( tofu curry spanakopita taquitos );

# 对豆类重新考虑
shift @meals;
```

`unshift` 将一个含零个或多个元素的列表前置某数组开头并返回数组中元素最新个数。`shift` 删除并返回数组的第一个元素。

很少有程序使用 `push` 和 `unshift` 的返回值。编写本章催生了一个优化 `push` 在空上下文中使用的 Perl 5 补丁。

### `splice`

是另一个重要的——可能较少使用——数组操作符。它按给出偏移量、列表分片长度以及替代物删除并替换数组元素。替换和删除都是可选的, 你可以忽略这些行为。`perlfunc` 中对 `splice` 的描述演示了它和 `push`、`pop`、`shift` 及 `unshift` 的等价性。

数组元素通常在循环中处理。有关 Perl 5 控制流程和数组处理的更多信息, 请参见 `looping_directives`。

在 Perl 5.12 中, 你可以使用 `each` 来将某数组迭代为键值对:

```
while (my ($index, $value) = each @bookshelf)
{
    say "#$index: $value";
    ...
}
```

## 数组和上下文

在列表上下文中, 数组平整为列表。如果你将多个数组传递给一个常规 Perl 5 函数, 它们会平整为 单个列表:

```
my @cats = qw( Daisy Petunia Tuxedo Brad Jack );
my @dogs = qw( Rodney Lucky );

take_pets_to_vet( @cats, @dogs );

sub take_pets_to_vet
{
    # 不要这样用!
    my (@cats, @dogs) = @_;
    ...
}
```

在此函数内, `@_` 会包含七个元素而非二。与此类似, 数组的列表赋值是 贪心的。一个数组会从列表 中消耗尽可能多的元素。赋值之后, `@cats` 将包含传递给函数的 每一个参数。而 `@dogs` 为空。

平整行为有时会给想要在 Perl 5 中创建嵌套数组的新手带来疑惑:

```
# 创建单个数组，不是数组的数组
my @array_of_arrays = ( 1 .. 10, ( 11 .. 20, ( 21 .. 30 ) ) );
```

有些人起初会期望这段代码产生一个数组，它的头十个元素是一到十的数字，第十一个元素是一个数组，包含数字十一到二十以及内嵌另一个包含二十一到三十的数组。注意在这种情况下下括号并不 创建 列表——它只用来给表达式分组。

解决平整问题的方法和给函数传递参数、创建嵌套数组的一样（references）。

## 数组内插

数组会作为字符串化的列表内插入双引号字符串，它的每一个元素由神奇的全局变量 `$"` 的当前值分隔。此变量的默认值是一个空格。它的 `English.pm` 助记形式是 `$LIST_SEPARATOR`。因此：

```
my @alphabet = 'a' .. 'z';
say "[@alphabet]";
B<[a b c d e f g h i j k l m n o p q r s t u v w x y z]>
```

临时局部化并将另外值赋给 `$"` 是一个很好的调试用法。致谢：Mark-Jason Dominus 在几年前演示了本例中的用法：

```
# 这个数组中有什么？
{
    local $" = ')((';
    say "(@sweet_treats)";
}
```

.....这个例子产生如下结果：

```
(pie)(cake)(doughnuts)(cookies)(raisin bread)
```

## 哈希

哈希 是 Perl 数据结构中的一等公民，它将字符串键和标量值之间一一联系起来。在其他编程语言中，它们可能被称为 表格、关联数组、字典 或是 映射。正如变量名和一个存储位置相对应，哈希中的一个键对应一个值。

一个备受推崇但老旧的说法，便是将哈希比方成电话簿：你可以按朋友的名字来查找她的电话号码。

哈希有两个重要的属性。第一，它们将一个标量以唯一的键存储。第二，它们不提供特定的键顺序。哈希就是一个大尺寸键值对容器。

## 声明哈希

哈希使用 `%` 印记。按如下方式声明一个词法哈希：

```
my %favorite_flavors;
```

哈希初始为空，没有键也没有值。在布尔上下文中，不含键的哈希求值得假。除此之外，它返回一个求值为真的字符串。

你可以对哈希的每一个独立的元素进行赋值和访问：

```
my %favorite_flavors;
```

```
$favorite_flavors{Gabi} = 'Mint chocolate chip';
$favorite_flavors{Annette} = 'French vanilla';
```

当访问独立元素时，哈希将使用标量印记 `$` 且将大括号 `{ }` 用于字符串键。

你可以在单一的表达式内将一个键值对列表赋值给一个哈希：

```
my %favorite_flavors = (
    'Gabi',      'Mint chocolate chip',
    'Annette',  'French vanilla',
);
```

如果你将奇数个元素赋值给一个哈希，你将收到警告说得不到预想的结果。通常用胖逗号操作符来关联键和值会更加明显，因为它能使“成对”的需求更加突出。请比较：

```
my %favorite_flavors = (
    Gabi      B<< => >> 'Mint chocolate chip',
    Annette B<< => >> 'French vanilla',
);
```

.....和：

```
my %favorite_flavors = (
    'Jacob', 'anything',
    'Floyd', 'Pistachio',
);
```

胖逗号操作符表现得和常规逗号一样，但是它使得 Perl 词法分析器将其前的裸字 (barewords) 作加了引号般对待。编译命令 `strict` 不会对这些裸字发出警告，并且如果你有一个和此哈希键同名的函数，胖逗号不会调用此函数：

```
sub name { 'Leonardo' }

my %address =
(
    name => '1123 Fib Place',
);
```

哈希的键是 `name` 而不是 `Leonardo`。如果你想调用该函数以得到键，请明确地使用函数调用来消除歧义：

```
my %address =
(
    B<name()> => '1123 Fib Place',
);
```

哈希赋值发生于列表上下文中，如果你在这样的赋值中调用函数，你可以使用 `scalar()` 对该上下文消歧。

要清空一个哈希，可以将空列表赋值给它 一元 `undef` 也可以，但相对比较少用：

```
%favorite_flavors = ();
```

## 哈希下标

由于哈希是一个集合，你可以通过下标操作访问其中独立的值。把键用作下标（按键访问 操作）



可以从一个哈希中取得对应的值：`my $address = $addresses{$name};`

在这个例子中，`$name` 包含用作哈希键的字符串。和访问数组中的单个元素一样，按照键访问一个标量值时哈希的印记也会从 `%` 改为 `$`。

你也可以将字符串面值用作哈希的键。Perl 会按胖逗号的规则自动为裸字加上引号：

```
# 自动加引号
my $address = $addresses{Victor};

# 需要加引号，不是一个合法的裸字
my $address = $addresses{B<'>Sue-LinnB<'>};

# 函数调用需要消歧
my $address = $addresses{get_nameB<(>>};
```

你也许会发现加上引号的哈希字符串面值键会比较清晰，但是自动加引号的行为已经在 Perl 5 文化中根深蒂固，因此最好将引号留给一些“词不达意”的特殊情况。

甚至连 Perl 5 的关键字也会做自动加引号的处理：

```
my %addresses =
(
    Leonardo => '1123 Fib Place',
    Utako    => 'Cantor Hotel, Room 1',
);

sub get_address_from_name
{
    return $addresses{B<+>shift};
}
```

一元加号 (`unary_coercions`、`numeric_operators`) 使得原本将成为裸字的 `shift` 跳过自动加引号

行为而变成表达式。隐含的意思是，你可以使用任意表达式——不仅是函数调用——作为哈希的键：

```
# 虽然可以，不要真的这样 I<做>
my $address = $addresses{reverse 'odranoel'};

# 可以使用字符串内插
my $address = $addresses{"$first_name $last_name"};

# 方法调用也可以
my $address = $addresses{ $user->name() };
```

任何求值为字符串的事物都是一个可接受的哈希键。当然，哈希键只能是字符串，如果你使用某对象作为哈希键，你将得到对象字符串化后的版本而非对象本身：

```
for my $isbn (@isbns)
{
    my $book = Book->fetch_by_isbn( $isbn );

    # 不太会是你想要的
    $books{$book} = $book->price;
}
```

## 哈希键的存在性

`exists` 操作符返回布尔值，指示某哈希是否含有给定的键：

```
my %addresses =
(
    Leonardo => '1123 Fib Place',
    Utako    => 'Cantor Hotel, Room 1',
);

say "Have Leonardo's address" if exists $addresses{Leonardo};
say "Have Warnie's address"   if exists $addresses{Warnie};
```

不直接访问哈希键而使用 `exists` 避免了两个问题。第一，它不对哈希值的布尔本质做检查：一个哈希键可能对应到一个求值得布尔假的值（包括 `undef`）：

```
my %false_key_value = ( 0 => '' );
ok( %false_key_value,
    'hash containing false key & value should evaluate to a true
value' );
```

第二，在处理嵌套数据结构时，`exists` 避免值的自生现象（autovivification）。

哈希键对应的操作是 `defined`。如果一个哈希键存在，它对应是值可能是 `undef`。你可以用 `defined` 检查这个值：

```
$addresses{Leibniz} = undef;

say "Gottfried lives at $addresses{Leibniz}"
    if exists $addresses{Leibniz}
    && defined $addresses{Leibniz};
```

## 访问哈希的键和值

哈希是集合变量，但是它的行为和数组不太一样。尤其是，你可以迭代一个哈希的所有键和值或者是键值对。`keys` 操作符返回由哈希键组成的列表：

```
for my $addressee (keys %addresses)
{
    say "Found an address for $addressee!";
}
```

`values` 操作符返回由哈希的值组成的列表：

```
for my $address (values %addresses)
{
    say "Someone lives at $address";
}
```

`each` 操作符返回一个列表，由一个个键值二元列表组成：

```

while (my ($addressee, $address) = each %addresses)
{
    say "$addressee lives at $address";
}

```

不像数组，哈希没有明确的键值列表排序方式。其顺序依赖于哈希的内部实现，实现又依赖于你使用的特定 Perl 版本、哈希的尺寸以及一个随机的因素。遵循上述条件，哈希中元素的顺序对于 `keys`、`values` 和 `each` 来说是一致的。修改哈希可能改变这个顺序，但只要哈希不改变，你可以依赖此顺序。

每一个哈希对于 `each` 操作符来说只有 单一 的迭代器。通过 `each` 对哈希进行多次迭代是不可靠的，如果你在一次迭代过程中开始另一次迭代，前者将过早地结束而后者将从半路开始。

在空上下文中使用 `keys` 或 `values` 可以重置一个哈希的迭代器：

```

# 重置哈希迭代器
keys %addresses;

while (my ($addressee, $address) = each %addresses)
{
    ...
}

```

你也应该确保不调用尝试使用 `each` 来迭代哈希的函数。

单个哈希迭代器是一个众所周知的注意点，但是它不如你想象的那么常见。小心一些，但在你需要时，也请大胆地使用 `each`。

## 哈希分片

和数组一样，你也可以在一个操作内访问一系列哈希元素。哈希分片 就是一个由哈希键值对组成的列表。最简单的解释就是使用无序列表初始化多个哈希元素：

```

my %cats;
@cats{qw( Jack Brad Mars Grumpy )} = (1) x 4;

```

这和下列初始化是等价的：

```

my %cats = map { $_ => 1 } qw( Jack Brad Mars Grumpy );

```

.....除了上例中的哈希分片初始化不会 替换 哈希的原有内容。

你可以用分片一次性从哈希中取出多个值：

```

my @buyer_addresses = @addresses{ @buyers };

```

正如数组分片，哈希印记的改变反映了列表上下文。通过使用大括号进行按键访问，你仍然可以得知 `%addresses` 的哈希本质没有变。

哈希分片可以使合并两个哈希变得容易：

```

my %addresses      = ( ... );
my %canada_addresses = ( ... );

@addresses{ keys %canada_addresses } = values %canada_addresses;

```

这样做和手动对 `%canada_addresses` 的内容进行循环等价，但是更为短小。

对两种方式的选择取决于你的合并策略。如果两个哈希中出现相同的键怎么办？哈希分片方式总是覆盖已存在于 `%addresses` 中的键值对。

## 空哈希

一个空哈希不包含键和值。它在布尔上下文中得假。含有至少一个键值对的哈希在布尔上下文得真，哪怕其所有的键或值或两者同时在布尔上下文中求值得假。

```
use Test::More;

my %empty;
ok( ! %empty, 'empty hash should evaluate to false' );

my %false_key = ( 0 => 'true value' );
ok( %false_key, 'hash containing false key should evaluate to true'
);

my %false_value = ( 'true key' => 0 );
ok( %false_value, 'hash containing false value should evaluate to
true' );

...

done_testing();
```

在标量上下文中，对哈希求值得到的是一个字符串，表示已用哈希桶比上已分配哈希桶。这个字符串并不怎么有用，因为它仅仅表示有关哈希的内部细节且对于 Perl 程序来说基本没有意义。你可以安全地忽略它。

在列表上下文中，对哈希求值得到类似从 `each` 操作符取得的键值对列表。然而，你 不能按迭代产生自 `each` 的列表的方式迭代此列表，因为这将无限循环，除非此 哈希为空。

## 哈希惯用语

哈希有若干用途，诸如查找列表或数组中唯一的元素。因为每个键只在哈希中存在一份，对哈希中相同的键赋值多次仅存储最近的值：

```
my %uniq;
undef @uniq{ @items };
my @uniques = keys %uniq;
```

对哈希分片使用 `undef` 操作符可以将哈希的值设置为 `undef`。这是判定某元素是否存在于集合中成本最低的方法。

在对元素计数时哈希也很有用，比如在日志文件中的一列 IP 地址：

```
my %ip_addresses;

while (my $line = <$logfile>)
{
    my ($ip, $resource) = analyze_line( $line );
    $ip_addresses{$ip}++;
    ...
}
```

哈希值初始为 `undef`。后缀自增操作符 (`++`) 将其作为零对待。这个对值即时修改

增加某个键对应的值。如果该键对应的值不存在，它创建一个值（undef）并立刻将其加一，因为数值化的 undef 产生值 0。

此策略的一个变种很适合缓存，即你愿意付出一点存取代价来存放某昂贵计算的结果：

```
{
    my %user_cache;

    sub fetch_user
    {
        my $id = shift;
        $user_cache{$id} ||= create_user($id);
        return $user_cache{$id};
    }
}
```

如果已经存在，这个 Orcish Maneuver “Or-cache”，如果你喜欢双关语的话 会从哈希中返回值。否则，计算该值，存入缓存，再返回它。注意布尔或赋值操作符（||=）作用于布尔值之上，如果你的缓存值在布尔上下文中求值得假，则可以使用“已定义-或”赋值操作符（//=）来代替：

```
sub fetch_user
{
    my $id = shift;
    $user_cache{$id} B<//=> create_user($id);
    return $user_cache{$id};
}
```

这个惰性 Orcish Maneuver

检查缓存值是否已被定义，而非其布尔真假。“已定义-或”赋值 操作符是 Perl 5.10 中的新功能。

哈希也可以收集传递给函数的具名参数。如果你的函数接受若干参数，你可以使用吸入式哈希（parameter\_slurping）来把键值对收集在单个哈希中：

```
sub make_sundae
{
    my %parameters = @_;
    ...
}

make_sundae( flavor => 'Lemon Burst', topping => 'cookie bits' );
```

你甚至用如下方式设置默认参数：

```
sub make_sundae
{
    my %parameters = @_;
    B<$parameters{flavor}    //= 'Vanilla';>
    B<$parameters{topping}   //= 'fudge';>
    B<$parameters{sprinkles} //= 100;>
    ...
}
```

.....或者将它们包含在最初的声明和赋值中:

```
sub make_sundae
{
    my %parameters =
    (
        B<< flavor      => 'Vanilla', >>
        B<< topping     => 'fudge', >>
        B<< sprinkles   => 100, >>
        @_,
    );
    ...
}
```

.....因为后续对同一键的不同值声明会覆盖前面的值。

## 哈希上锁

哈希的一个缺点就是它们的键是几乎不提供打字错误保护的裸字（特别是将其和受 `strict` 编译命令保护的函数、变量名相比）。核心模块 `Hash::Util` 提供了一些机制来对哈希的修改和允许的键做出限制。

为避免他人向哈希添加你不想要的键（假设一个打字错误或是不受信任的输入），你可以使用 `lock_keys()` 函数将哈希键限制在当前集合中。任何添加不被允许的键值对的意图将引发一个异常。

当然，其他想达到此目的的人总是可以使用 `unlock_keys()` 函数来去掉保护，因此不要将此作为防止其他程序员误用的安全保障来信任。

类似的你可以对哈希中给定的键对应已存在的值进行上锁和解锁（`lock_value()` 和 `unlock_value()`）以及利用 `lock_hash()` 和 `unlock_hash()` 使得或不使整个哈希变为只读。

## 强制转换

不像其他语言中一个变量只能存放一个特定类型的值（字符串、浮点数、对象），Perl 依赖操作符的上下文来决定如何对值进行解释（`value_contexts`）。如果你将数字作为字符串，Perl 将尽最大努力将该数字转换为字符串（反之亦然）。这个过程就是 强制转换。

植根于设计，Perl 就试图按你的意思办事（DWIM 代表 `do what I mean`），虽然你必须对你的意图进行较为具体的描述。

## 布尔强制转换

布尔强制转换发生于测试某值 为真性 的时候 为真性和真假性有些类似，设想你斜着眼睛说“啊，那是真的，但是.....”，就是 `if` 或 `while` 语句中的测试条件。数值 `0` 是假。未定义值是假。空字符串和字符串 `'0'` 是假。然而字符串 `'0.0'` 和 `'0e'` 是真。

所有其他的值是真的，包括惯用语字符串 `'0 but true'`。对于诸如有着字符串和数值两面的标量这种情况（`dualvars`），Perl 5 选择字符串部分作为检查布尔真假的依据。`'0 but true'` 在数值上下文中求值得零，但不是空字符串，因此它在布尔上下文中求值得真。

## 字符串强制转换

字符串强制转换发生于使用字符串操作符之时，例如比较（`eq` 和 `cmp`，比方说）、拼接、`split`、`substr` 以及正则表达式。它也发生在将某值作为哈希键时。未定义值字符串化得到空字符串，但会引发 “`use of uninitialized value`（使用未初始化值）” 的警告。数值字符串化得到包含其值的字符串，就是说，值 `10` 字符串化得到字符串 `10`，如此你便可以用 `split` 将某数值分割成组成它的各个数字：

```
my @digits = split '', 1234567890;
```

## 数值强制转换

数值强制转换发生于使用数值比较操作符（诸如 `==` 和 `<=>`）、进行数学计算操作以及将某值作为数组或列表的下标时。未定义值数值化得零，虽然这会产生一个 “`Use of uninitialized value`（使用未初始化的值）” 的警告。不以数值部分开头的字符串数值化为零，并产生 “`Argument isn't numeric`（参数不是数值）” 的警告。以数值面值中允许出现的字符开头的字符串将数值化为对应的值，就是说 `10 leptons leaping` 数值化为 `10`，同样地，`6.022e23 moles marauding` 数值化为 `6.022e23`。

核心模块 `Scalar::Util` 包含一个名为 `looks_like_number()` 的函数，它使用和 Perl 5 语法一样的语法分析规则从字符串中提取数字。

字符串 `Inf` 和 `Infinity` 表示无穷值，并且出于在数值化时不会产生 “`Argument isn't numeric`（参数不是数值）” 的警告，它们行为和数字一致。字符串 `NaN` 表示 “不是一个数字（英语：not a number）” 的概念。除非你是数学家，否则一般不会关心这些。

## 引用强制转换

在某些的情况下，将一个值作为引用对待将使该值转变为一个引用。这个自生（`autovivification`）的过程对于嵌套数据结构来说比较有用。它发生在当你对非引用进行解引用操作时：

```
my %users;

$users{Bradley}{id} = 228;
$users{Jack}{id}    = 229;
```

虽然上例中的哈希从未包含对应 `Bradley` 和 `Jack` 的值，Perl 5 热心地为这些值创建了哈希引用，接着将以 `id` 为键的键值对赋值给它们。

## 强制转换的缓存

Perl 5 对值的内部存储机制允许每个值拥有字符串化和数值化的结果 这是一种简化，但残酷的现实真的很残酷。字符串化一个数值并不将数值替换为字符串。取而代之的是，它将字符串化后的值作为对数值的补充附着到该值上。类似的操作还发生于数值化一个字符串值时。

你几乎不必知道发生了这种转换——如果传闻证据可信的话，也许十年内会有个一两次。

Perl 5 也许会偏向于使用某种形式的转换。如果一个值拥有一个缓存了的、但不是你期望的表示方式，依赖于隐式的转换可能会产生令人惊讶的结果。你几乎不必明确提出你的期望，但是明白这种缓存确实存在，你可以对某些奇怪的情况做出诊断。

## 双重变量

对数值和字符串值的缓存允许你使用一个“罕见但有用”的特性，称为 双重变量，或者说一个同时拥有数值和字符串表示的值。核心模块 `Scalar::Util` 提供了一个名为 `dualvar()` 的函数，它允许你创建一个拥有两种不同形式的值：

```
use Scalar::Util 'dualvar';
my $false_name = dualvar 0, 'Sparkles & Blue';

say 'Boolean true!' if      !! $false_name;
say 'Numeric false!' unless 0 + $false_name;
say 'String true!'  if      '' . $false_name;
```

## 包

Perl 中的 名称空间 是一种机制，它将若干具名实体关联并封装于某具名分类之下。它就像你的姓或是某种品牌，只能反映出命名归类上的关系而非其他。（这类关系可以存在，但不是必须的。）

Perl 5 中的 包 是单一名称空间下代码的集合。在某种意义上，包和名称空间是等价的；包代表源代码而名称空间代表当 Perl 分析这段代码时创建的实体。这是个微妙的区别

`package` 关键字声明一个包和一个名称空间：

```
package MyCode;

our @boxes;

sub add_box { ... }
```

所有在包声明语句之后的对全局变量和函数的声明或引用，都指向位于 `MyCode` 名称空间内的符号。如果代码是这样写的，那么你只能通过 完全限定 名称 `@MyCode::boxes` 来从 `main` 名称空间访问 `@boxes` 变量。相似地，你仅可以通过 `MyCode::add_box()` 来调用 `add_box()` 函数。一个完全限定名称包括了完整包名。

默认包是 `main` 包。如果你不明确声明一个包，无论是在命令行 `one-liner` 或在独立的 Perl 程序甚至是磁盘上的 `.pm` 文件，那么当前包就是 `main` 包。

除包名（`main` 或是 `MyCode` 或其他任何允许的标识符）外，一个包还拥有版本以及三个隐含的方法，分别是 `VERSION()`、`import()`（`importing`）和 `unimport()`。`VERSION()` 返回一个包的版本。

包版本是包含在名为 `$VERSION` 的包全局变量中的一系列数字。按照惯例，版本号倾向于写成一系列由点分隔整数的形式，例如 `1.23` 和 `1.1.10`，其中每一段是一个整数，但通常就这样写。

Perl 5.12 引入了一种简化版本号的新语法。如果你编写的代码不会在早先版本的 Perl 5 上运行，你可以避免不少不必要的复杂性：

```
package MyCode 1.2.1;
```

在 5.10 以及早期的版本中，声明包版本最简单的方式是：

```
package MyCode;
```



```
our $VERSION = 1.21;
```

每个包都有 `VERSION()` 方法；它们继承自 `UNIVERSAL` 基类。它返回 `$VERSION` 中的值。虽然没有什么理由这样做，但你还是可以按需重写此方法。使用 `VERSION()` 方法是获取一个包版本号最简便的办法：

```
my $version = Some::Plugin->VERSION();

die "Your plugin $version is too old"
    unless $version > 2;
```

## 包和名称空间

每一句 `package` 声明都会使 Perl 完成两件任务。如果名称空间不存在则创建它。它还告诉语法分析器将后续的包全局符号（全局变量和函数）放入该名称空间下。

Perl 有 开放式名称空间。通过使用包声明语句，你在任何时候向一个名称空间添加函数和变量：

```
package Pack;

sub first_sub { ... }

package main;

Pack::first_sub();

package Pack;

sub second_sub { ... }

package main;

Pack::second_sub();
```

.....或者在声明时使用完全限定的函数名称：

```
# 隐式
package main;

sub Pack::third_sub { ... }
```

Perl 5 是那么的开放以至于你可以在编译期、运行时的任何时刻或从其他文件向其中添加内容。当然，这样做使人迷惑，因此应尽量避免。

名称空间可以按组织需要分为多个级别。这并不意味着继承关系，包与包之间也没有什么技术上的联系———仅对于这段代码的 阅读者 来说有语义上的关系罢了。

常见的做法是为业务或项目创建一个顶层名称空间。这对于阅读代码和发现组件间关系来说变得方便，同时也为代码和包在磁盘上的组织提供便利。因此：

- \* `StrangeMonkey` 是项目名称;
- \* `StrangeMonkey::UI` 包含顶层用户接口的代码;
- \* `StrangeMonkey::Persistence` 包含顶层数据管理代码;
- \* `StrangeMonkey::Test` 包含为项目编写的顶层测试代码;

.....等等。

## 引用

即便你的期望相当微妙，Perl 通常也能实现它。考虑当你将值传递给函数时候会发生什么：

```
sub reverse_greeting
{
    my $name = reverse shift;
    return "Hello, $name!";
}

my $name = 'Chuck';
say reverse_greeting( $name );
say $name;
```

你也许会期望，虽然该值曾传递给函数并且逆序为 `kcuhC`，但在函数之外，`$name` 仍包含值 `Chuck`——确实，这就是所发生的事。在函数之外的 `$name` 和函数内部的 `$name` 是两个分离的标量，各有一份独立的字符串拷贝。修改一个不会影响另一个。

这是有用且合意的默认行为。如果你每次执行可能引发修改的操作之前都必须明确地复制这些值，将导致编写大量多余的、不必要的代码来抵抗善意而错误的修改。

有些时候，直接对值进行修改也有不少用处。试想仅为更新某值或删除一个键值对就将一个装满数据的哈希传递给函数，导致每次更改都要创建和返回新哈希——这是一个麻烦的过程（毫无效率可言）。

Perl 5 提供了一种机制，通过它你可以间接使用某值而不必为此创建一份拷贝。任何对该引用做出的修改将就地对值进行更新，如此，所有对该值的引用都将见到最新的值。引用是 Perl 5 中的一等公民，是一种内置的标量数据类型。它不是字符串、不是数组、也不是哈希。它就是一个引用其他第一等数据类型的标量。

## 标量引用

引用操作符是反斜杠 (`\`)。在标量上下文中，它创建对另一个值的单一引用。在列表上下文中，它创建一个引用列表。因此，你可以按如下方式引用前例中的 `$name`：

```
my $name      = 'Larry';
my $name_ref = B<\>$name;
```

要访问引用指向的值，你必须对其解引用。解引用需要你在解开每一重引用时加上额外的印记：

```
sub reverse_in_place
{
    my $name_ref = shift;
    B<$$name_ref> = reverse B<$$name_ref>;
}
```

```
my $name = 'Blabby';
reverse_in_place( B<\>$name );
say $name;
```

双标量印记对一个标量引用进行解引用。

这个例子看上去不怎么有用，为什么不让函数直接返回修改后的值？标量引用在处理 大型标量时很有用——复制这些标量的内容会花去大量的时间和内存。

复杂的引用需要加上一对大括号以消除表达式断句上的歧义。这对于简单解引用来说是可选的，加上它会变得很臃肿：

```
sub reverse_in_place
{
    my $name_ref = shift;
    B<${ $name_ref }> = reverse B<${ $name_ref }>;
}
```

如果你忘记对一个标量引用解引用，它将会字符串化或数值化。字符串值将拥有 `SCALAR(0x93339e8)` 的形式，数值则为 `0x93339e8` 这部分。该值将引用的类型（此例为标量，即 `SCALAR`）以及引用在 内存中的位置编码在一起。

#### Perl

不提供对内存位置的原生访问。因为引用不一定有名称，引用的地址是一个可用作唯一标识符的值。不像一些语言（如：C）中的指针，你不能修改该地址或将其作为对应于内存的地址。

这些地址仅在 大致上 唯一，因为在垃圾回收器回收某未引用的引用后，Perl 可能重用此存储位置。

## 数组引用

你也可以创建对数组的引用，或称 数组引用。说它有用有如下几个理由：

- \* 不加平整地向函数传递及从函数中返回数组；
- \* 创建多维数据结构；
- \* 避免不必要的数组复制；
- \* 持有匿名数据结构。

对已声明的数组创建引用，使用引用操作符：

```
my @cards = qw( K Q J 10 9 8 7 6 5 4 3 2 A );
my $cards_ref = B<\>@cards;
```

现在 `$cards_ref` 包含了对该数组的引用。所有通过 `$cards_ref` 做出的修改 将同样影响 `@cards`，反之亦然。

你可以通过 `@` 已经对数组进行整体访问，可能会使数组平整为列表，或者得到其中包含的元素个数：

```
my $card_count = B<@ $cards_ref>;
my @card_copy = B<@ $cards_ref>;
```

你也可以通过解引用箭头 (`->`) 访问数组中的独立元素：

```
my $first_card = B<< $cards_ref->[0] >>;
my $last_card  = B<< $cards_ref->[-1] >>;
```

访问单个元素时候，为区分名为 `$cards_ref` 的标量和名为 `@cards_ref` 数组，箭头是必需的。

还有另外一种写法，你可以在数组引用之前加上标量印记。它更简短，但是可读性较差：`my $first_card = $$cards_ref[0];`

通过大括号解引用和分组语法，可以通过引用对数组分片：

```
my @high_cards = B<@{ $cards_ref }>[0 .. 2, -1];
```

这种情况下，你 可以忽略大括号，但是它（和空白）带来的视觉上的分组有助于可读性的提高。

你可以不用命名就地创建匿名数组。在一个值或表达式列表的周围加上中括号：

```
my $suits_ref = [qw( Monkeys Robots Dinosaurs Cheese )];
```

这个数组引用的行为和具名数组引用一样，除了匿名数组引用 总是 创建新引用，而具名数组的引用总是在作用域内指向 同一个 数组。就是说：

```
my @meals      = qw( waffles sandwiches pizza );
my $sunday_ref = \@meals;
my $monday_ref = \@meals;
```

```
push @meals, 'ice cream sundae';
```

.....`$sunday_ref` 和 `$monday_ref` 现在包含了一份甜点，但是：

```
my @meals      = qw( waffles sandwiches pizza );
my $sunday_ref = [ @meals ];
my $monday_ref = [ @meals ];
```

```
push @meals, 'berry pie';
```

.....无论 `$sunday_ref` 还是 `$monday_ref` 都不包含甜点。在用于创建匿名数组引用的中括号内，`@meals` 数组在列表上下文中被展开。

## 哈希引用

要创建一个 哈希引用，可以在具名哈希上使用引用操作符：

```
my %colors = (
    black => 'negro',
    blue  => 'azul',
    gold  => 'dorado',
    red   => 'rojo',
    yellow => 'amarillo',
    purple => 'morado',
);

my $colors_ref = B<\%>colors;
```

按如下方法在引用前加上哈希印记 `%` 可以访问其中的键和值：

```
my @english_colors = keys B<%$colors_ref>;
my @spanish_colors = values B<%$colors_ref>;
```

你可以通过解引用箭头访问哈希中独立的值（存储、删除、检查存在性或取值）：

```
sub translate_to_spanish
{
    my $color = shift;
    return B<< $colors_ref->{$color} >>;
}
```

虽然箭头有时更为清晰，和数组引用一样，你可以避开解引用箭头转用前缀标量印记：  
`$$colors_ref{$color}`。

你也可以通过引用对哈希分片：

```
my @colors = qw( red blue green );
my @colores = B<@{ $colors_ref }{@colors}>;
```

注意大括号标示了哈希下标操作，数组印记标示了对引用进行的列表操作。

你可以通过大括号就地创建匿名哈希：

```
my $food_ref = B<{>
    'birthday cake' => 'la torta de cumpleaE<ntilde>os',
    candy           => 'dulces',
    cupcake         => 'pancucitos',
    'ice cream'     => 'helado',
B<}>;
```

和匿名数组一样，每次执行都会创建一个匿名哈希。

一个常见的新手错误就是将匿名哈希赋值给标准哈希。这会产生一个有关哈希元素个数为奇数的警告。请对具名哈希使用小括号，对匿名哈希使用大括号。

## 函数引用

**Perl 5** 支持 第一级函数。至少在你使用 函数引用 时，函数和数组、哈希一样，是一种数据类型。这个特性启用了许多高级功能（**closures**）。和其他数据类型一样，你可以通过在函数名称上使用引用操作符来创建函数引用：

```
sub bake_cake { say 'Baking a wonderful cake!' };

my $cake_ref = B<\&>bake_cake;
```

没有了 函数印记（&），你将得到的是函数返回值的引用。

你也可以创建匿名函数：

```
my $pie_ref = B<sub { say 'Making a delicious pie!' }>;
```

使用 **sub** 关键字 不用 函数名称也可以使得函数正常编译，但是它不会被安装到当前的名称空间中。访问此函数的唯一方法就是通过引用。

你可以通过解引用箭头调用引用指向的函数：

```
$cake_ref->();
$pie_ref->();
```

将空小括号的解引用作用想成和中括号对数组进行下标查找、大括号进行哈希查找一样。你可以在小括号内将函数的参数传递进去：

```
$bake_something_ref->( 'cupcakes' );
```

你也可以将函数引用用作对象的方法（**moose**），当你已经完成方法查找时它最有用：

```
my $clean = $robot_maid->can( 'cleanup' );
$robot_maid->$clean( $kitchen );
```

你会看到另外一种调用函数引用的语法，它使用函数印记（**&**）而非解引用箭头。请避免使用这种语法； 它暗示隐式参数传递。

## 文件句柄引用

文件句柄也可以是引用。当你使用 **open** 的（以及 **opendir** 的）词法文件句柄形式，你就在处理文件句柄引用。对此文件句柄进行字符串化可以得到类似 **GLOB(0x8bda880)** 形式的东西。

说到内部机制，这些文件句柄都是 **IO::Handle** 类的对象。当你加载这个模块时，你可以调用文件句柄上的方法：

```
use IO::Handle;
use autodie;

open my $out_fh, '>', 'output_file.txt';
$out_fh->say( 'Have some text!' );
```

你会碰到使用型团（**typeglob**）引用的陈旧代码，例如：

```
my $fh = do {
    local *FH;
    open FH, "> $file" or die "Can't write to '$file': $!\n";
    B<\*FH>;
};
```

这个惯用语出现于词法文件句柄之前，是在 2000 年 3 月作为 **Perl 5.6.0** 的一部分引入的……你现在知道那段代码有多老了。你仍可以在型团上使用引用操作符得到包全局文件句柄 如 **STDIN**、**STDOUT**、**STDERR** 和 **DATA** 的引用——无论如何，这些仅代表全局数据，对于其他情况，请使用词法文件句柄。

除了使用词法作用域代替包或全局作用域所带来的好处，词法文件句柄允许你管理文件句柄的生存期限。这是一个 **Perl 5** 如何管理内存和作用域的优秀特性。

## 引用计数

**Perl** 怎么知道它何时可以安全地释放一个变量所占内存以及何时需要继续保留？**Perl** 怎么知道它何时可以安全地关闭在内层作用域内打开的文件：

```
use autodie;
use IO::Handle;
```

```

sub show_off_scope
{
    say 'file not open';

    {
        open my $fh, '>', 'inner_scope.txt';
        $fh->say( 'file open here' );
    }

    say 'file closed here';
}

```

**Perl 5** 使用一种名为 引用计数 的内存管理技术。程序中的每一个值都有附加的计数器。每次被其他东西引用时，**Perl** 增加计数器的值，无论隐式还是显式。每次引用消失后，**Perl** 将减少计数器的值。当计数器减至零，**Perl** 就可以安全地回收这个值。

上例中，在内层代码块内，有一个 `$fh`。（源代码中有多行提及它，但只有一处 引用 它，即 `$fh` 自己。）`$fh` 的作用域仅限于此代码块，并且没有被赋值到代码块外，因此，当此代码块结束时候，它的引用计数减为零。对 `$fh` 的回收隐式地调用了该文件句柄 上的 `close()` 方法，使得文件最终被关闭。

你不需要了解全部这些是如何工作的。你只需理解你对值的引用和传递将影响 **Perl** 如何管理内存。（请参考 `circular_references` 获得有关循环引用的告诫。）

## 引用和函数

当你使用引用作为函数的参数时，请小心地记录你的意图。在函数内部修改引用指向的值可能会是调用它的代码感到意外，因为调用者其实并不期望这种修改。

如果你需要对引用的内容做出破坏性改动，请将其所含的值复制到一个新的变量中：

```

my @new_array = @{ $array_ref };
my %new_hash = %{ $hash_ref };

```

这仅在少数情况是必须的，同时为了避免调用者感到诧异，在这些情况下最好明确地复制。如果引用更加复杂（`nested_data_structures`），请考虑使用核心模块 `Storable` 和它的 `dclone`（`deep cloning`）函数。

## 嵌套数据结构

### **Perl**

的集合数据类型——数组和哈希——允许你按整数下标或字符串键存储标量。**Perl 5** 的引用（`references`）则允许你通过特殊标量间接访问集合数据类型。**Perl** 中的嵌套数据结构，例如数组的数组、哈希的哈希，是通过引用机制来实现的。

## 声明嵌套数据结构

一个对数组的数组简单声明可能是：

```

my @famous_triplets = (
    [qw( eenie miney moe )],
    [qw( huey dewey louie )],
    [qw( duck duck goose )],
);

```

.....一个对哈希的哈希简单声明可能是：

```
my %meals = (
    breakfast => { entree => 'eggs',    side => 'hash browns' },
    lunch      => { entree => 'panini',  side => 'apple' },
    dinner     => { entree => 'steak',   side => 'avocado salad' },
);
```

**Perl** 允许在结尾添加逗号，但并非必须，这样做只是为了方便以后添加元素。

## 访问嵌套数据结构

访问嵌套数据结构中的元素需要用到 **Perl** 的引用语法。印记标示了欲取得数据的数量，解引用箭头表明数据结构中的这部分值是一个引用：

```
my $last_nephew = $famous_triplets[1]->[2];
my $breaky_side = $meals{breakfast}->{side};
```

对于嵌套数据结构这种情况，嵌套一个数据结构的唯一方法就是通过引用，因此箭头是多余的。下面的代码和前面的等价，并且更清楚：

```
my $last_nephew = $famous_triplets[1][2];
my $breaky_side = $meals{breakfast}{side};
```

调用存放于嵌套数据结构内的函数引用时，使用箭头调用语法是最清晰的，除此之外，你可以避开箭头的使用。

将嵌套数据结构作为第一等数组或哈希访问时，需要消歧代码块：

```
my $nephew_count = @{ $famous_triplets[1] };
my $dinner_courses = keys %{ $meals{dinner} };
```

类似的，对嵌套数据结构分片也需要额外的标点：

```
my ($entree, $side) = @{ $meals{breakfast} }{qw( entree side )};
```

空白的使用有助于，但不能完全消除这个语法结构的噪音。一些时候，使用临时变量会 更清晰：

```
my $breakfast_ref = $meals{breakfast};
my ($entree, $side) = @$breakfast_ref{qw( entree side )};
```

**perldoc perldsc**，数据结构的“食谱”，给出了有关如何使用 **Perl** 中各式数据结构丰富的实例。

## 自生

**Perl** 的表达力同样也扩展到了嵌套数据结构。当你试图编写一个嵌套数据结构组件时，如果不存在，**Perl** 会创建通向这部分数据结构的路径：

```
my @aoaoaoa;

$aaoaoaoa[0][0][0][0] = 'nested deeply';
```

第二行代码之后，这个数组的数组的数组的数组包含了对数组的引用的引用的引用的引用。每一个引用包含一个元素。类似的，在嵌套数据结构中将未定义值作为哈希引用会创建以合适的值作为键的中间哈希。

```
my %hohoh;

$hohoh{Robot}{Santa}{Claus} = 'mostly harmful';
```



这个行为称为 自生，并且很有用。它的好处是减少嵌套数据结构的初始化代码。它的坏处是无法区分创建嵌套数据结构中所缺元素究竟是有意还是无意。

**CPAN** 上的 **autovivification** 编译命令 (**pragmas**) 让你可以在词法作用域内对某特定类型操作禁用自生行为。在多人参与的大型项目中很值得考虑这些问题。

你也可以在对复杂的数据结构进行层层解引用前检查特定的哈希键是否存在以及获取数组中元素的个数，但是这会导致许多程序员都不愿碰的冗长代码。

你也许会考虑利用自生和对代码启用 **strictures** 这对矛盾。这是一个权衡问题。以禁用针对几个严实封装的符号引用错误检查为代价来捕获错误是不是更方便？让数据结构自行增长是不是比指定它们的大小和允许的键来得方便？

后一个问题的答案取决于特定的项目。刚开始开发时，你可能想要严格要求编码以防止意外的副作用。好在有了 **strict** 和 **autovivification** 编译命令的词法作用域，你可以按需要启用禁用它们。

## 调试嵌套数据结构

### Perl 5

的解引用语法的复杂结合多级引用潜在的迷惑性，使得调试嵌套数据结构变得困难。所幸有两种可视化它们的好选择。

核心模块 **Data::Dumper** 可以将任意复杂的数据结构的值字符串化为 **Perl 5** 代码：

```
use Data::Dumper;

print Dumper( $my_complex_structure );
```

在识别数据结构所含内容以及找出应该访问到和实际访问到什么时很有用。**Data::Dumper** 可以转储对象和函数引用（如果你将 **\$Data::Dumper::Deparse** 设置为真）。

**Data::Dumper** 是核心模块，并且打印出 **Perl 5** 代码，但它也给出详细的输出。一些开发人员更愿意使用 **YAML::XS** 和 **JSON** 来调试程序。为理解它们的输出，你必须学习不同的格式，但它们的输出更易阅读也更易理解。

## 循环引用

**Perl 5** 的引用计数 (**reference\_counts**) 内存管理系统对于用户代码来说有一个明星的坏处。两个互指的引用最终形成了一 循环引用，**Perl** 无法自行销毁它。考虑生物模型，每一个实体有父方母方，并可以有子代：

```
my $alice = { mother => '',      father => '',      children => [] };
my $robert = { mother => '',      father => '',      children => [] };
my $cianne = { mother => $alice, father => $robert, children => [] };

push @{$alice->{children} }, $cianne;
push @{$robert->{children} }, $cianne;
```

因为 **\$alice** 和 **\$robert** 都包含了一个指向 **\$cianne** 数组引用，并且由于 **\$cianne** 是一个包含 **\$alice** 和 **\$robert** 的哈希引用，**Perl** 始终无法将这三者的引用计数减为零。它无法认识到循环引用的存在，并且无法管理这些实体的生存 期限。

你必须手动打断引用计数（通过清除 **\$alice** 和 **\$robert** 的子代或 **\$cianne** 的亲代），或者利用一个名为 弱引用 的特性。弱引用是一个不增加被引用者引用计数的

引用。弱引用可以通过核心模块 `Scalar::Util` 来使用。导出 `weaken()` 函数并对某引用使用它可以防止引用计数的增加：

```
use Scalar::Util 'weaken';

my $alice = { mother => '',      father => '',      children => [] };
my $robert = { mother => '',      father => '',      children => [] };
my $cianne = { mother => $alice, father => $robert, children => [] };

push @{$alice->{children}}, $cianne;
push @{$robert->{children}}, $cianne;

B<< weaken( $cianne->{mother} ); >>
B<< weaken( $cianne->{father} ); >>
```

完成之后，`$cianne` 仍持有对 `$alice` 和 `$robert` 的引用，但是这些引用不会主动 阻拦 Perl 的垃圾回收器回收这些数据结构。经过正确设计的数据结构一般不会用到弱引用，但在极少数情况下它们仍可能被用到。

### 嵌套数据结构的替代选择

不管数据结构嵌套得多深，Perl 都愿意处理，但是理解这些数据结构、理顺边边角角关系所花的人力代价，别提访问数据结构各个部分所用的代码，就已经够高了。除了两三层嵌套的数据结构，其他情况就应该考虑是不是应该用类和对象（`moose`）来对系统的各个组件进行建模，它会更清楚地表达你的数据。

有的时候，将数据和合适的行为绑定在一起会使代码更清晰。