两个半小时学会Perl

By Sam Hughes

译者：胡瀚森（Sam Hu）（阅读原文）

Perl是一门动态的、动态类型的高阶脚本（解释型）语言，常常与PHP和Python相提并论。Perl的语法借鉴了许多古老的Shell脚本，并且因为过度使用难懂的符号而声名狼藉，这使得人们几乎无法从Google搜索到想要的语法知识。Perl作为一门脚本语言非常适合用作*胶水语言*，将其他脚本和程序连接到一起。Perl非常擅长处理和生成文本数据，同时也具有高度可移植性和良好的社区支持，使其成为一门应用广泛的脚本语言。Perl的设计哲学是"每个问题都有不止一种解决方法"（TMTOWTDI），而Python的设计哲学"每个问题最好只有唯一一个显而易见的解决方法"与之截然相反。

Perl有时令人厌恶，不过也有一些很棒的语言特性，就这一点上，Perl和其他编程语言是一样的。

这篇文章旨在提供知识，而不是为Perl做宣传，目标人群是（比如我自己）：

- 不喜欢Perl的官方文档（http://perl.org/），因为太过于学术性，并且花费太多的版面来讲述一些一辈子用不到的边缘问题
- 通过一些通用约定和示例程序能够快速学习编程语言
- 希望拉里·沃尔能够长话短说（译者注：拉里·沃尔是Perl语言的设计者）
- 已经知道怎么编程
- 除非对工作有用，否则不关心Perl的任何其他细节。

这篇文档会尽可能精简，而包含的每条信息都是必要的。

序言

- 文章中几乎所有的陈述语句几乎都是"严格说来不完全是真的，现实情况要复杂得多"，如果你看到一个严重的错误，请指出来，不过我保留不修正"善意的谎言"的权利。
- 在这片文章里我用`print`语句来输出数据，但是并不一定追加了换行或者空格，以免我因为过分注意输出样式而发疯。许多示例程序如果真的运行出来，现实结果可能像是"alotofwordsallsmusheduptogetherononeline"，但是，不要在意这些细节。

Hello world

Perl*脚本*是带有`.pl`后缀的文本文件。

下面是`helloworld.pl`的全部内容：

```
use strict;
use warnings;

print "Hello world";
```

Perl脚本由Perl解释器解释执行，`perl`或者`perl.exe`：

```
perl helloworld.pl [arg0 [arg1 [arg2 ...]]]
```

有几点要特别提一下：Perl的语法非常宽容，它允许你写出有歧义的或者有不可预期行为的代码。我不会去解释这些诡异的行为是什么样的，因为你最好避开它们。避免这种情况的方法是在你写的每个Perl脚本或者模块的开头加上`use strict; use warnings;`。`use foo;`这种语句叫做*编译指示（pragmas）*，编译指示是给`perl.exe`的一个提示，在程序开始执行之前的语法验证阶段会发挥作用，脚本语句实际执行的时候这些编译指示对于运行结果没有影响。

分号`;`是语句结束的标志，井号`#`表示注释的开始，注释直到这行的结尾结束。Perl没有块注释的语法。

变量

Perl的变量有三种类型：*标量（scalar）*、*数组（array）*和*哈希（hashes）*（译者注：下文会继续使用英文原文scalar、array和hash），每种类型都有属于自己的*符号*：分别是`$`、`@`和`%`。变量定义使用`my`关键字，生命期直到其所在的代码块结束或者文件的末尾。

Scalar变量

一个scalar变量能包含：

- `undef`（对应Python中的`None`、PHP中的`null`）
- 数值（Perl不区分整形和浮点类型）

- 字符串
- 其他变量的引用。

```
my $undef = undef;
print $undef; # 打印空字符串"", 并且抛出一个警告

# 隐式的undef（译者注：未初始化的变量初值默认为undef）：
my $undef2;
print $undef2; # 打印"", 并且抛出完全一样的警告

my $num = 4040.5;
print $num; # "4040.5"

my $string = "world";
print $string; # "world"
```

（稍后会详细说明“引用”。）

用.运算符进行字符串连接（与PHP一样）：

```
print "Hello ".$string; # "Hello world"
```

“布尔类型”（“Boolean”）

Perl没有内置的布尔类型。`if`语句中的scalar变量仅在以下情况下被认为是“false”：

- `undef`
- 数值`0`
- 字符串`""`
- 字符串`"0"`。

Perl的文档中*反复*强调函数在某些情况下返回“true”或者“false”。实际上，当一个函数声称它返回“true”，返回值往往是`1`，而当一个函数声称它返回“false”，返回值往往是一个空字符串`""`。

弱类型

无法判定一个scalar包含的是一个数值还是字符串。更准确的来说，我们没有必要知道这个信息。一个scalar按照数值还是字符串的方式参与运算，是完全取决于运算符的。因此，像字符串一样使用的时候，scalar就按字符串的方式参与运算，而像数值一样使用的时候，scalar就按照数值的方式参与运算（如果无法转换成数值则会抛出一个警告）：

```
my $str1 = "4G";
my $str2 = "4H";

print $str1 .  $str2; # "4G4H"
print $str1 +  $str2; # "8" 并且抛出两个警告
print $str1 eq $str2; # ""  （空字符串，也就是false）
print $str1 == $str2; # "1" 并且抛出两个警告

# 经典错误
print "yes" == "no"; # "1" 并且抛出两个警告，按数值方式参与运算，两边求值结果都是0
```

教训是应该总是在恰当的情况下使用正确的运算符，对于比较数值和字符串有两套不同的运算符：

```
# 数值运算符：   <,  >, <=, >=, ==, !=, <=>, +, *
# 字符串运算符：     lt, gt, le, ge, eq, ne, cmp, ., x
```

Array变量

Array变量是包含一个scalar列表的、由从0开始的整形数为下标存取的变量。在Python里被称为*list*，而在PHP里被称为*array*。数组可以用一个圆括号包围的scalar列表来声明（译者注：原文declaration，而这里实际表达的含义应为“初始化”，而不是对于变量标识符的声明，下同）：

```
my @array = (
        "print",
        "these",
        "strings",
        "out",
        "for",
        "me", # 末尾多余的逗号语法上是允许的
);
```

你必须要使用美元符号来存取array中的值，因为*取到*的值是一个scalar而非array：

```
print $array[0]; # "print"
print $array[1]; # "these"
print $array[2]; # "strings"
```

```
print $array[3]; # "out"
print $array[4]; # "for"
print $array[5]; # "me"
print $array[6]; # 返回undef，打印""并且抛出一个警告
```

你也可以使用负数作为下标，这样就可以从末尾开始往前取某个元素：

```
print $array[-1]; # "me"
print $array[-2]; # "for"
print $array[-3]; # "out"
print $array[-4]; # "strings"
print $array[-5]; # "these"
print $array[-6]; # "print"
print $array[-7]; # 返回undef，打印""并且抛出一个警告
```

同时存在scalar变量`$var`和包含scalar元素`$var[0]`的array变量`@var`是没有冲突的，不过会对代码的读者造成一些误导，所以请避免这种情况。

取得array的长度

```
print "This array has ".(scalar @array)."elements"; # "This array has 6 elements"
print "The last populated index is ".$#array;       # "The last populated index is 5"
```

调用Perl脚本时使用的参数列表被保存在内置的array变量`@ARGV`中。

变量可以被插入到字符串中被求值：

```
print "Hello $string"; # "Hello world"
print "@array";        # "print these strings out for me"
```

小心。也许有一点你会把某个人的email地址放在一个字符串里，比如`"jeff@gmail.com"`。Perl会去尝试找一个名叫`@gmail`的array变量，求值并插入到字符串中，如果没有找到这个变量，将会导致一个运行时错误。有两种方法可以避免对字符串中的变量名求值：用反斜杠对`@`进行转义，或者将双引号改为单引号。

```
print "Hello \$string"; # "Hello $string"
print 'Hello $string';  # "Hello $string"
print "\@array";        # "@array"
print '@array';         # "@array"
```

Hash变量

Hash变量是包含一个scalar列表的、由字符串为下标存取的变量。在Python中被称为*dictionary*，而在PHP中被称为*array*。

```
my %scientists = (
        "Newton"   => "Isaac",
        "Einstein" => "Albert",
        "Darwin"   => "Charles",
);
```

请注意这个声明与array何其相似。事实上，这个双箭头符号`=>`被称为"fat comma"（胖逗号），因为它与逗号完全等价。Hash变量由偶数个元素组成的列表来声明，其中偶数下标（0、2、……）的元素都被当做字符串使用。

与array一样，你也需要用美元符号来存取hash中的值，因为*取到*的值是scalar而非hash：

```
print $scientists{"Newton"};   # "Isaac"
print $scientists{"Einstein"}; # "Albert"
print $scientists{"Darwin"};   # "Charles"
print $scientists{"Dyson"};    # 返回undef，打印""并且抛出一个警告
```

注意在这里使用的花括号。同样的，同时存在scalar变量`$var`和包含scalar元素`$var{"foo"}`的hash变量`%var`是没有冲突的。

你可以将一个hash转换为两倍数量元素的array，原先hash中的键和值在转换后的array中交替出现（反向的转换也同样简单）：

```
my @scientists = %scientists;
```

然而有一点与array不同，hash中的键*没有特定的保存顺序*，而是以一种比较高效的方式进行存储。因此，需要注意转换后的array会将hash中的*键值对*重新排列*次序*：

```
print "@scientists"; # 输出可能是"Einstein Albert Darwin Charles Newton Isaac"
```

回顾一下，我们使用**方括号**来取array中的值，而使用**花括号**来取hash中的值。方括号是一个有效的数值运算符，而花括号是一个有效的字符串运算符，因此事实上，作为*下标*的值是数值还是字符串类型其实并不重要（译者注：正如前文所提到的，scalar以什么方式参与运算取决于运算符）：

```
my $data = "orange";
my @data = ("purple");
my %data = ( "0" => "blue");

print $data;      # "orange"
print $data[0];   # "purple"
```

```
print $data["0"]; # "purple"
print $data{0};   # "blue"
print $data{"0"}; # "blue"
```

列表（Lists）

Perl中的*列表*与array和hash都不一样。你已经见过一些列表了：

```
(
        "print",
        "these",
        "strings",
        "out",
        "for",
        "me",
)

(
        "Newton"   => "Isaac",
        "Einstein" => "Albert",
        "Darwin"   => "Charles",
)
```

列表不是一个变量列表是一个暂存的*值*，可以被*赋值*到一个array或者hash变量，这就是为什么声明array和hash的语法竟完全一样。在许多情况下"列表"和"array"这两个词可以混用，而在同样多的情况下，列表和array表现出微妙的区别，并且具有极其容易混淆的行为。

好的，回想一下`=>`只是`,`的一种伪装，然后看一下下面的例子：

```
("one", 1, "three", 3, "five", 5)
("one" => 1, "three" => 3, "five" => 5)
```

使用`=>`暗示了其中一个是hash的声明（译者注：第二个），而另一个是array的声明，但就这两个列表自身并没有声明任何东西，它们只是列表，而且是*完全相同的*列表。同样的：

```
()
```

这里甚至没有任何变量类型的提示，这个列表可以用来声明一个空array或者空hash，而作为`perl`解释器则完全无法知道将会是哪一种。一旦你理解了这一点，你也就能理解Perl的这个事实：**列表不能嵌套。**我们可以试一下：

```
my @array = (
        "apples",
        "bananas",
        (
                "inner",
                "list",
                "several",
                "entries",
        ),
        "cherries",
);
```

Perl无法知道`("inner", "list", "several", "entries")`应该是array还是hash，因此Perl假设两者都不是，而**将其扁平化为一个一维长列表**：

```
print $array[0]; # "apples"
print $array[1]; # "bananas"
print $array[2]; # "inner"
print $array[3]; # "list"
print $array[4]; # "several"
print $array[5]; # "entries"
print $array[6]; # "cherries"
```

即使使用fat comma也会是同样的情况：

```
my %hash = (
        "beer" => "good",
        "bananas" => (
                "green"  => "wait",
                "yellow" => "eat",
        ),
);

# 上面的代码会抛出一个警告，因为我们尝试用7个元素的列表来初始化这个hash

print $hash{"beer"};    # "good"
print $hash{"bananas"}; # "green"
print $hash{"wait"};    # "yellow";
print $hash{"eat"};     # undef，因此打印""并且抛出一个警告
```

当然，这倒让连接数组变得简单了：

```
my @bones   = ("humerus", ("jaw", "skull"), "tibia");
my @fingers = ("thumb", "index", "middle", "ring", "little");
```

```
my @parts   = (@bones, @fingers, ("foot", "toes"), "eyeball", "knuckle");
print @parts;
```

稍后会有更多关于这个问题的说明。

上下文

Perl最独特的特性在于它的代码*对于上下文是敏感的*。**每个Perl的表达式要么在scalar上下文中求值，要么在列表上下文中求值**，取决于此处期望产生一个scalar还是列表。许多Perl表达式和内置函数在不同的求值上下文中的行为大相径庭。

Scalar的赋值例如`$scalar =`会在scalar上下文求值，在这种例子中，表达式为`"Mendeleev"`，而返回的是同样的scalar值`"Mendeleev"`：

```
my $scalar = "Mendeleev";
```

Array或者hash的赋值例如`@array =`或者`%hash =`会在列表上下文求值，在列表上下文中求值的列表就会返回这个列表本身，然后被用于初始化这个array或者hash：

```
my @array = ("Alpha", "Beta", "Gamma", "Pie");
my %hash = ("Alpha" => "Beta", "Gamma" => "Pie");
```

到目前为止还没什么特别的。

在列表上下文中求值的scalar表达式会被转换成含有一个元素的列表：

```
my @array = "Mendeleev"; # 与'my @array = ("Mendeleev");'等价
```

在scalar上下文中求值的列表表达式会返回*列表中的最后一个scalar*：

```
my $scalar = ("Alpha", "Beta", "Gamma", "Pie"); # $scalar的值现在是"Pie"
```

在scalar上下文中求值的array（还记得array和列表不同吗？）表达式返回*该数组的长度*：

```
my @array = ("Alpha", "Beta", "Gamma", "Pie");
my $scalar = @array; # $scalar的值现在是4
```

`print`内置函数在列表上下文中求对所有的参数求值。事实上，`print`能够接受无限个参数的列表，并且一个接一个地打印它们，这就意味着我们可以直接用它来打印array：

```
my @array = ("Alpha", "Beta", "Goo");
my $scalar = "-X-";
print @array;              # "AlphaBetaGoo";
print $scalar, @array, 98; # "-X-AlphaBetaGoo98";
```

你可以用内置函数`scalar`强制让任何表达式在scalar上下文进行求值，这就是我们为什么用`scalar`来得到array的长度。

即使子过程要在scalar上下文进行求值，语法上也没有规定必须要返回一个scalar，就像我们看到的，Perl完全可以为你捏造一个需要的结果。

引用和嵌套数据结构

列表无法包含列表作为其元素，**array也同样无法包含其他array和hash作为其元素**，它们只能包含scalar。看看我们尝试下面的做法会发生什么：

```
my @outer = ("Sun", "Mercury", "Venus", undef, "Mars");
my @inner = ("Earth", "Moon");

$outer[3] = @inner;

print $outer[3]; # "2"
```

`$outer[3]`是个scalar，因此它需要一个scalar值。当你尝试将`@inner`这样的array值赋给它，`@inner`就会在scalar上下文中被求值，这就与将`scalar @inner`是同样的效果。这相当于求出了array `@inner`的长度，也就是2。

然而，scalar变量可以包含任何变量的*引用*，包括array和hash。在Perl中，复杂的数据结构就是这样被构造出来的。

我们用反斜杠来创建一个引用。

```
my $colour    = "Indigo";
my $scalarRef = \$colour;
```

如果你能够使用某个变量名，你可以加一些花括号，把一个变量的*引用*放进去。

```
print $colour;         # "Indigo"
print $scalarRef;      # 输出可能是 "SCALAR(0x182c180)"
print ${ $scalarRef }; # "Indigo"
```

如果结果没有歧义的话，你甚至可以直接省略掉花括号：

```
print $$scalarRef; # "Indigo"
```

如果是一个对array或者hash的引用，你可以用花括号或者更加风靡的箭头运算符`->`：

```
my @colours = ("Red", "Orange", "Yellow", "Green", "Blue");
my $arrayRef = \@colours;

print $colours[0];       # 直接访问array元素
print ${ $arrayRef }[0]; # 通过引用访问array元素
print $arrayRef->[0];    # 与上一句等价

my %atomicWeights = ("Hydrogen" => 1.008, "Helium" => 4.003, "Manganese" => 54.94);
my $hashRef = \%atomicWeights;

print $atomicWeights{"Helium"}; # 直接访问hash元素
print ${ $hashRef }{"Helium"};  # 通过引用访问hash元素
print $hashRef->{"Helium"};     # 与上一句等价 - 这种写法相当常见
```

声明数据结构

这里有4个例子，不过现实中最后一个最有用。

```
my %owner1 = (
        "name" => "Santa Claus",
        "DOB"  => "1882-12-25",
);

my $owner1Ref = \%owner1;

my %owner2 = (
        "name" => "Mickey Mouse",
        "DOB"  => "1928-11-18",
);

my $owner2Ref = \%owner2;

my @owners = ( $owner1Ref, $owner2Ref );

my $ownersRef = \@owners;

my %account = (
        "number" => "12345678",
        "opened" => "2000-01-01",
        "owners" => $ownersRef,
);
```

显然可以不用这么费劲，这段代码可以简化为：

```
my %owner1 = (
        "name" => "Santa Claus",
        "DOB"  => "1882-12-25",
);

my %owner2 = (
        "name" => "Mickey Mouse",
        "DOB"  => "1928-11-18",
);

my @owners = ( \%owner1, \%owner2 );

my %account = (
        "number" => "12345678",
        "opened" => "2000-01-01",
        "owners" => \@owners,
);
```

用不同的符号声明*匿名*的array和hash也是可行的。用方括号声明匿名array，而用花括号声明匿名hash，这两种方法返回的是声明的匿名数据结构的*引用*。看仔细了，下面的代码声明的`%account`和上面的完全等价：

```
# 花括号表示匿名hash
my $owner1Ref = {
        "name" => "Santa Claus",
        "DOB"  => "1882-12-25",
};

my $owner2Ref = {
        "name" => "Mickey Mouse",
        "DOB"  => "1928-11-18",
};

# 方括号表示匿名array
my $ownersRef = [ $owner1Ref, $owner2Ref ];
```

```
my %account = (
        "number" => "12345678",
        "opened" => "2000-01-01",
        "owners" => $ownersRef,
);
```

或者写得更加简短（这也是你*真正*应该用来声明复杂数据结构的方法）：

```
my %account = (
        "number" => "31415926",
        "opened" => "3000-01-01",
        "owners" => [
                {
                        "name" => "Philip Fry",
                        "DOB"  => "1974-08-06",
                },
                {
                        "name" => "Hubert Farnsworth",
                        "DOB"  => "2841-04-09",
                },
        ],
);
```

从数据结构中获取信息

现在我们假设你还在折腾那个`%account`，而且其他东西都不在作用域内（如果还有其他东西的话）。你可以逆向操作解引用，以取得每一项需要打印的信息。同样，这里有4个例子，其中最后一个最有用：

```
my $ownersRef = $account{"owners"};
my @owners    = @{ $ownersRef };
my $owner1Ref = $owners[0];
my %owner1    = %{ $owner1Ref };
my $owner2Ref = $owners[1];
my %owner2    = %{ $owner2Ref };
print "Account #", $account{"number"}, "\n";
print "Opened on ", $account{"opened"}, "\n";
print "Joint owners:\n";
print "\t", $owner1{"name"}, " (born ", $owner1{"DOB"}, ")\n";
print "\t", $owner2{"name"}, " (born ", $owner2{"DOB"}, ")\n";
```

或者写得更简短些：

```
my @owners = @{ $account{"owners"} };
my %owner1 = %{ $owners[0] };
my %owner2 = %{ $owners[1] };
print "Account #", $account{"number"}, "\n";
print "Opened on ", $account{"opened"}, "\n";
print "Joint owners:\n";
print "\t", $owner1{"name"}, " (born ", $owner1{"DOB"}, ")\n";
print "\t", $owner2{"name"}, " (born ", $owner2{"DOB"}, ")\n";
```

或者使用引用和`->`运算符：

```
my $ownersRef = $account{"owners"};
my $owner1Ref = $ownersRef->[0];
my $owner2Ref = $ownersRef->[1];
print "Account #", $account{"number"}, "\n";
print "Opened on ", $account{"opened"}, "\n";
print "Joint owners:\n";
print "\t", $owner1Ref->{"name"}, " (born ", $owner1Ref->{"DOB"}, ")\n";
print "\t", $owner2Ref->{"name"}, " (born ", $owner2Ref->{"DOB"}, ")\n";
```

如果我们完全跳过那些中间值，代码看起来就是这样：

```
print "Account #", $account{"number"}, "\n";
print "Opened on ", $account{"opened"}, "\n";
print "Joint owners:\n";
print "\t", $account{"owners"}->[0]->{"name"}, " (born ", $account{"owners"}->[0]->{"DOB"}, ")\n";
print "\t", $account{"owners"}->[1]->{"name"}, " (born ", $account{"owners"}->[1]->{"DOB"}, ")\n";
```

如何用array的引用作茧自缚

这个数组有5个元素：

```
my @array1 = (1, 2, 3, 4, 5);
print @array1; # "12345"
```

然而这个array只有*1*个元素（一个含有5个元素的匿名array的引用）：

```
my @array2 = [1, 2, 3, 4, 5];
print @array2; # e.g. "ARRAY(0x182c180)"
```

这个*scalar*是一个含有5个元素的匿名array的引用：

```
my $array3Ref = [1, 2, 3, 4, 5];
print $array3Ref;      # e.g. "ARRAY(0x22710c0)"
print @{ $array3Ref }; # "12345"
print @$array3Ref;     # "12345"
```

条件分支

if ... elsif ... else ...

这里没有什么特别之处，除了elsif的拼写：

```
my $word = "antidisestablishmentarianism";
my $strlen = length $word;

if($strlen >= 15) {
        print "'", $word, "' is a very long word";
} elsif(10 <= $strlen && $strlen < 15) {
        print "'", $word, "' is a medium-length word";
} else {
        print "'", $word, "' is a short word";
}
```

Perl提供了一种更简短的"*statement* if *condition*"语法，对于短的语句强烈推荐这种写法：

```
print "'", $word, "' is actually enormous" if $strlen >= 20;
```

unless ... else ...

```
my $temperature = 20;

unless($temperature > 30) {
        print $temperature, " degrees Celsius is not very hot";
} else {
        print $temperature, " degrees Celsius is actually pretty hot";
}
```

最好像避开瘟疫一样避开unless语句，因为这实在太容易把读者搞得晕头转向。"unless [... else]"语句块可以显而易见地通过对条件取反（或者保持条件不变调换语句块的位置）来重构成"if [... else]"。万幸的是，没有elsunless关键字。

而相比之下，这种写法就被强烈推荐，因为实在是太易于阅读了：

```
print "Oh no it's too cold" unless $temperature > 15;
```

三目运算符

三目运算符?:使得简单的if语句可以嵌入到其他语句内。一种常规的用法就是用来处理单复数形式：

```
my $gain = 48;
print "You gained ", $gain, " ", ($gain == 1 ? "experience point" : "experience points"), "!";
```

题外话：单复数形式最好都写出完整的拼写，不要自作聪明地写成下面这种样子，要不然别人永远也无法在代码中查找、替换到"tooth"或者"teeth"了：

```
my $lost = 1;
print "You lost ", $lost, " t", ($lost == 1 ? "oo" : "ee"), "th!";
```

三目运算符可以嵌套：

```
my $eggs = 5;
print "You have ", $eggs == 0 ? "no eggs" :
                   $eggs == 1 ? "an egg"  :
                   "some eggs";
```

if语句在scalar上下文中进行求值。举例说明，if(@array)当且仅当@array包含大于等于1个元素的时候返回true，而元素的内容则无关紧要（也许包括undef或者其他我们真正关心的代表false的值）。

循环

"每个问题都有不止一种解决方法"

Perl支持传统的while循环：

```
my $i = 0;
while($i < scalar @array) {
        print $i, ": ", $array[$i];
        $i++;
}
```

Perl也提供了until关键字：

```
my $i = 0;
until($i >= scalar @array) {
        print $i, ": ", $array[$i];
        $i++;
}
```

这些do循环*几乎*和上面的循环等价（如果@array为空会抛出一个警告）：

```
my $i = 0;
do {
        print $i, ": ", $array[$i];
        $i++;
} while ($i < scalar @array);
```

and

```
my $i = 0;
do {
        print $i, ": ", $array[$i];
        $i++;
} until ($i >= scalar @array);
```

基本的C风格for循环也可以使用。注意我们怎么将my放到for语句内部，这样声明的$i的作用于就仅限于循环内部：

```
for(my $i = 0; $i < scalar @array; $i++) {
        print $i, ": ", $array[$i];
}
# $i在这里就不存在了，代码显得更加整洁。
```

这种for循环被视为过时的东西，应该尽量避免使用，使用原生的array迭代语法看起来更漂亮。注意：与PHP不同，for和foreach关键字是等价的，选可读性比较好的那个来用就可以了：

```
foreach my $string ( @array ) {
        print $string;
}
```

如果你需要使用下标，range运算符..会创建一个匿名的整形数列表：

```
foreach my $i ( 0 .. $#array ) {
        print $i, ": ", $array[$i];
}
```

你无法迭代一个hash，而你可以迭代它所有的键。使用内置函数keys来取得包含这个hash所有键的array，然后使用foreach来遍历它：

```
foreach my $key (keys %scientists) {
        print $key, ": ", $scientists{$key};
}
```

因为hash没有既定的次序，键可能以任何次序返回，使用内置函数sort事先对包含键的array进行字母序排序：

```
foreach my $key (sort keys %scientists) {
        print $key, ": ", $scientists{$key};
}
```

如果你没有显示指定迭代器，Perl将使用默认迭代器$_。$_是第一个也是最友好的一个内置变量：

```
foreach ( @array ) {
        print $_;
}
```

如果使用默认迭代器，并且你希望在循环里只放一句语句，你可以使用下面这种超级简洁的语法：

```
print $_ foreach @array;
```

循环控制

next和last可以用来控制循环过程，在其他大部分编程语言中分别相当于continue和break。一般约定，行标写成**全部大写**。在循环里加上行标以后，next和last可以选择指定跳转到某个行标。下面的示例程序能够找出100以内的素数：

```
CANDIDATE: for my $candidate ( 2 .. 100 ) {
        for my $divisor ( 2 .. sqrt $candidate ) {
                next CANDIDATE if $candidate % $divisor == 0;
        }
        print $candidate." is prime\n";
}
```

Array函数

原地（In-place）array修改函数

我们用@stack来演示这些函数：

```
my @stack = ("Fred", "Eileen", "Denise", "Charlie");
print @stack; # "FredEileenDeniseCharlie"
```

pop抽取并返回array的最后一个元素，可以认为是栈顶的元素：

```
print pop @stack; # "Charlie"
print @stack;     # "FredEileenDenise"
```

push向array末尾添加一个元素：

```
push @stack, "Bob", "Alice";
print @stack; # "FredEileenDeniseBobAlice"
```

shift抽取并返回array的第一个元素：

```
print shift @stack; # "Fred"
print @stack;       # "EileenDeniseBobAlice"
```

unshift向array的头部插入一个元素：

```
unshift @stack, "Hank", "Grace";
print @stack; # "HankGraceEileenDeniseBobAlice"
```

pop、push、shift和unshift都是splice的特例。splice返回删除的一个array的切片，并且用另一个array的切片在原array中替换之：

```
print splice(@stack, 1, 4, "<<<", ">>>"); # "GraceEileenDeniseBob"
print @stack;                             # "Hank<<<>>>Alice"
```

从现有的array创建新的array

Perl提供下面这些函数，可以操作现有的array产生新的array。

join函数把多个字符串连接成一个字符串：

```
my @elements = ("Antimony", "Arsenic", "Aluminum", "Selenium");
print @elements;             # "AntimonyArsenicAluminumSelenium"
print "@elements";           # "Antimony Arsenic Aluminum Selenium"
print join(", ", @elements); # "Antimony, Arsenic, Aluminum, Selenium"
```

在列表上下文，reverse函数把传入的列表逆序返回，在scalar上下文，reverse先把字符串列表连接起来，再将这个字符串反转。

```
print reverse("Hello", "World");        # "WorldHello"
print reverse("HelloWorld");            # "HelloWorld"
print scalar reverse("HelloWorld");     # "dlroWolleH"
print scalar reverse("Hello", "World"); # "dlroWolleH"
```

map函数接受一个array，并将一个操作应用于这个array中的每一个scalar $_，然后返回用这些scalar创建的array。这个操作用在花括号中的一个表达式来表示：

```
my @capitals = ("Baton Rouge", "Indianapolis", "Columbus", "Montgomery", "Helena", "Denver", "Boise");

print join ", ", map { uc $_ } @capitals;
# "BATON ROUGE, INDIANAPOLIS, COLUMBUS, MONTGOMERY, HELENA, DENVER, BOISE"
```

grep函数接受一个array，并返回一个经过筛选的array。语法与map类似，而第二个参数会对array中的每个scalar $_求值，如果返回true，这个scalar就会被放到输出array中，否则就不会。

```
print join ", ", grep { length $_ == 6 } @capitals;
# "Helena, Denver"
```

显然，返回的array长度是*满足条件的元素个数*，这就意味着你可以用grep检查array中是否包含某个元素：

```
print scalar grep { $_ eq "Columbus" } @capitals; # "1"
```

grep和map的组合形成了*list comprehensions*这种许多其他语言中欠缺的非常强大特性。（译者注：list comprehensions大致的意思是利用map和filter从现有的列表构造新的列表，表达的含义是对一个列表中满足某个条件的所有元素上应用某个操作，而形成一个新的列表。）

默认情况下，sort函数对输入的array按字母序进行排序：

```
my @elevations = (19, 1, 2, 100, 3, 98, 100, 1056);

print join ", ", sort @elevations;
# "1, 100, 100, 1056, 19, 2, 3, 98"
```

然而，与grep和map类似，排序总是通过一系列元素的两两比较来进行的。你的代码块接受$a和$b作为输入，如果$a"小于"$b则返回-1，如果"相等"则返回0，而如果$a"大于"$b则返回1。

cmp运算符适用于字符串（译者注：按字母序比较）：

```
print join ", ", sort { $a cmp $b } @elevations;
# "1, 100, 100, 1056, 19, 2, 3, 98"
```

这个"宇宙飞船运算符"`<=>`适用于数值：

```
print join ", ", sort { $a <=> $b } @elevations;
# "1, 2, 3, 19, 98, 100, 100, 1056"
```

`$a`和`$b`总是scalar，但是它们也许是某个复杂对象的引用，那样就很难直接进行比较。如果你需要更多篇幅来描述这种比较，你可以单独创建一个子程序来描述它，并在用到它的地方提供这个子程序的名字：

```
sub comparator {
        # lots of code...
        # return -1, 0 or 1
}

print join ", ", sort comparator @elevations;
```

不过你不能对`grep`或`map`这样做。

请注意，我们从来没有显式提供`$a`和`$b`给子程序和语句块。就像`$_`一样，`$a`和`$b`实际上是当一对值需要比较时被*填入*的全局变量。

内置函数

截止目前你已经看到过不少内置函数了：`print`、`sort`、`map`、`grep`、`keys`、`scalar`等等。内置函数是Perl的一大优势，它们：

- 数不胜数
- 非常实用
- 有全面的文档支持
- 语法上差异很大，因此使用前请先查文档
- 有时接受正则表达式作为参数
- 有时接受一整块代码作为参数
- 有时参数之间不需要逗号分隔
- 有时消耗任意数量由逗号分隔的参数，有时则不时
- 有时在提供的参数不足的情况下会填入默认值
- 通常不要求参数列表用括号包围，除非会产生歧义

关于内置函数最好的建议是**知道它们的存在**，浏览一下文档以供将来参考。如果你在完成某个任务并且发现那工作太底层也太常用了，以至于你觉得别人肯定已经做过多次了，那么事实往往的确如此。

用户自定义的子程序

子程序用`sub`关键字来声明。相比内置函数，自定义子程序总是接受一种输入：一个scalar的列表。当然这个列表可以只包含一个元素，甚至为空。一个scalar会被转换成包含一个scalar的列表来处理，而一个有*N*个元素的hash会被转换成包含2*N*个元素的列表来处理。

尽管括号可以省略，我们还是应该总是在调用子程序的时候加上括号，即使不提供任何参数，读者就能更容易发现子程序的调用。

在子程序中，参数被保存在内置array变量`@_`中。例如：

```
sub hyphenate {

  # 从array中取出第一个参数，忽略其他
  my $word = shift @_;

  # 聪明过头的list comprehension
  $word = join "-", map { substr $word, $_, 1 } (0 .. (length $word) - 1);
  return $word;
}

print hyphenate("exterminate"); # "e-x-t-e-r-m-i-n-a-t-e"
```

Perl以引用方式调用

不像其他主流编程语言，Perl以引用方式调用子程序（译者注：以引用方式传递参数）。这意味着子程序中用到的变量或值不是实参的副本，它们*本身就是*实参。

```
my $x = 7;

sub reassign {
  $_[0] = 42;
}

reassign($x);
print $x; # "42"
```

如果你尝试这样做：

```
reassign(8);
```

程序就会因为错误而终止运行，因为`reassign()`的第一行就相当于

```
8 = 42;
```

这显然是非常荒谬的。

这边可以学到的经验教训是，在子程序中你总是应该在使用参数之前将它们提取出来。

提取参数

我们有不止一种方法来提取`@_`中的参数，但总有一些方法比其他方法更好。

下面的示例子程序`left_pad`在字符串左边填充某个字符直到达到需要的长度。（`x`函数将同一个字符串的多个副本连接起来。）（注意：为了简化问题，这些子程序都缺乏必要的错误检查，比如确保填充字符串长度为1，检查要求的宽度是否大于等于字符串的长度，需要的参数是否都提供了。）

`left_pad`通常就像下面这样调用：

```
print left_pad("hello", 10, "+"); # "+++++hello"
```

1. 逐个抽取`@_`中的参数很有效，但也并不是那么地美观：

   ```
   sub left_pad {
           my $oldString = $_[0];
           my $width     = $_[1];
           my $padChar   = $_[2];
           my $newString = ($padChar x ($width - length $oldString)) . $oldString;
           return $newString;
   }
   ```

2. 对于不超过4个参数的情况推荐用`shift`通过移出元素的方法来提取`@_`中的参数：

   ```
   sub left_pad {
           my $oldString = shift @_;
           my $width     = shift @_;
           my $padChar   = shift @_;
           my $newString = ($padChar x ($width - length $oldString)) . $oldString;
           return $newString;
   }
   ```

   如果没有给`shift`函数提供array参数，它就会默认对`@_`进行操作。这种用法很常见：

   ```
   sub left_pad {
           my $oldString = shift;
           my $width     = shift;
           my $padChar   = shift;
           my $newString = ($padChar x ($width - length $oldString)) . $oldString;
           return $newString;
   }
   ```

   超过4个参数以后就很难搞清楚参数的哪部分被赋值给谁了。

3. 你也可以一次性把所有`@_`中的参数提取出来。仍然是适用于少于4个参数的情形：

   ```
   sub left_pad {
           my ($oldString, $width, $padChar) = @_;
           my $newString = ($padChar x ($width - length $oldString)) . $oldString;
           return $newString;
   }
   ```

4. 对于有大量参数的子程序，或者有些参数可选或无法和其他参数组合使用的子程序，最佳实践是要求用户构造参数的hash来调用这个子程序，然后将整个`@_`放回到一个hash中。用这种方法，我们子程序的调用会看起来会有点不一样：

   ```
   print left_pad("oldString" => "pod", "width" => 10, "padChar" => "+");
   ```

   而子程序自身就变成这样：

   ```
   sub left_pad {
           my %args = @_;
           my $newString = ($args{"padChar"} x ($args{"width"} - length $args{"oldString"})) . $args{"oldString"};
           return $newString;
   }
   ```

返回值

就像其他Perl表达式一样，子程序调用也会根据上下文表现出不同的行为。你可以用`wantarray`函数（也许我们应该叫它`wantlist`（译者注：上下文可以是scalar或者列表，不是一个array或者hash），不过不要在意这些细节）来检测子程序是在什么上

下文中被调用的，这样就可以返回恰当类型的结果：

```
sub contextualSubroutine {
        # 调用这里需要一个列表，那么就返回一个列表
        return ("Everest", "K2", "Etna") if wantarray;

        # 调用者需要一个scalar，那么就返回一个scalar
        return 3;
}

my @array = contextualSubroutine();
print @array; # "EverestK2Etna"

my $scalar = contextualSubroutine();
print $scalar; # "3"
```

系统调用

如果你已经知道下面说的这些和Perl无关的事实，那抱歉我还是要多说几句。每当一个进程在Windows或Linux系统（以及其他大部分的系统）中结束，它将产生一个16位的*状态字*，高8位表示*返回码*，值落在0到255之间，其中0约定俗成地表示无条件的成功，而其他值则表示不同程度的失败，另外8位则少有人关心，它们"表示了错误的原因，比如因为收到了信号或者产生core dump信息"。

你可以调用**exit**，用你选择的返回码（0到255之间）退出Perl脚本。

Perl提供了不止一种方法通过一句调用语句来启动一个子进程、等待子进程执行结束、然后继续解释执行当前的脚本。无论用那种方法，你会发现紧接着，子进程结束时返回的状态字已经被填入了内置scalar变量`$?`中。你可以通过取出16位中的高8位来得到返回码：`$? >> 8`。

我们可以用**system**函数调用另一个程序，并且提供一个参数列表，`system`的返回值与填入`$?`的值一致：

```
my $rc = system "perl", "anotherscript.pl", "foo", "bar", "baz";
$rc >>= 8;
print $rc; # "37"
```

另一种选择，我们也可以用反引号``在命令行中运行一条真正的命令，并且捕获它的标准输出。在scalar上下文中，整个输出被当做一整个字符串返回返回，而在列表上下文中，整个输出按一个字符串的array返回，其中每个字符串是输出中的一行。

```
my $text = `perl anotherscript.pl foo bar baz`;
print $text; # "foobarbaz"
```

如果`anotherscript.pl`包含形如下面这样的代码，你就能看到上面这种结果：

```
use strict;
use warnings;

print @ARGV;
exit 37;
```

文件和文件句柄

Scalar变量除了能够包含数值、字符串、引用或者`undef`，还能包含一个*文件句柄*。文件句柄本质上就是对于某个文件中某个位置的引用。

用**open**可以把一个scalar变量编程文件句柄。我们必须给`open`提供一个*打开模式*。模式`<`表示我们想要读取这个文件：

```
my $f = "text.txt";
my $result = open my $fh, "<", $f;

if(!$result) {
        die "Couldn't open '".$f."' for reading because: ".$!;
}
```

如果成功，`open`返回true，否则返回false，并且错误消息会被填入内置变量`$!`。就像你在上面的代码里看到的，你总是应该检查`open`操作是否成功完成了，不过像那样检查真是冗长乏味，更常见的写法是：

```
open(my $fh, "<", $f) || die "Couldn't open '".$f."' for reading because: ".$!;
```

注意，你需要在`open`的参数列表两边加上括号。

要从文件句柄中读取一行，可以用内置函数**readline**，`readline`返回一整行文本，并且结尾有一个换行符（除了文件末尾的那行可能例外），如果已经读到文件末尾则返回`undef`。

```
while(1) {
        my $line = readline $fh;
        last unless defined $line;
        # 处理$line...
}
```

可以用chomp移除末尾可能存在的换行符：

```
chomp $line;
```

请注意，chomp直接作用于$line上，因此$line = chomp $line可能不会得到你想要的东西。

你也可以用eof来检测是否已经读到文件末尾：

```
while(!eof $fh) {
        my $line = readline $fh;
        # 处理$line...
}
```

不过使用while(my $line = readline $fh)的时候要小心了，因为如果$line的内容恰好是"0"，循环可能过早结束。如果你想要这样写，Perl提供了<>功能上更安全的运算符，你可以用它包围readline。这种写法很常见而且也非常安全：

```
while(my $line = <$fh>) {
        # 处理$line...
}
```

甚至：

```
while(<$fh>) {
        # 处理$_...
}
```

如果要写一个文件，首先你需要另一种打开模式。模式>表示我们想要写入这个文件。（如果目标文件存在的话，>会清空它，如果你只是想附加在文件的原有内容后面，你应该用模式>>。）然后，将文件句柄作为print方法的第0个参数提供就行了。

```
open(my $fh2, ">", $f) || die "Couldn't open '".$f."' for writing because: ".$!;
print $fh2 "The eagles have left the nest";
```

请注意在$fh2和后面的参数之间没有逗号。

文件句柄在超出它们的作用域以后会自动关闭，如果你想主动关闭：

```
close $fh2;
close $fh;
```

有三个文件句柄以全局常量形式存在：STDIN、STDOUT和STDERR，它们在脚本开始时就被自动打开。要读取一行用户的输入：

```
my $line = <STDIN>;
```

如果只是等待用户按回车：

```
<STDIN>;
```

调用<>而不提供文件句柄参数，表示从STDIN或者在Perl脚本启动时指定的参数指向的文件中读取。

你可能已经知道了，如果不提供文件句柄，print默认会打印到STDOUT。

文件检测

内置函数-e用于测试文件是否存在。

```
print "what" unless -e "/usr/bin/perl";
```

内置函数-d用于测试文件是否是目录。

内置函数-f用于测试文件是否是普通文件。

这只是一大波形如-X的函数中的三个，其中X是某些小写或大写字母。这类函数被称作*文件检测函数*。请注意字母前面的减号，用Google搜索的时候，减号表示从搜索结果中排除包含这个词的结果，这样就导致很难用Google搜索文件检测函数了！用"Perl file test"来搜索就好。

正则表达式

除了Perl以外，正则表达式也被应用在许多其他的语言和工具中。Perl的核心正则表达式语法基本上和其他地方别无二致，不过Perl完整的正则表达式功能复杂到令人发指，并且难以理解。我能给的最好的建议就是尽可能避免引入不必要的复杂性。

用=~ m//运算符进行正则表达式匹配。在scalar上下文中，=~ m//在成功时返回true，而失败是返回false。

```
my $string = "Hello world";
if($string =~ m/(\w+)\s+(\w+)/) {
        print "success";
}
```

圆括号表示匹配组，匹配成功以后，匹配组被填入内置变量$1、$2、$3.....：

```
print $1; # "Hello"
print $2; # "world"
```

在列表上下文中，`=~ m//`返回`$1`、`$2`.....组成的列表。

```
my $string = "colourless green ideas sleep furiously";
my @matches = $string =~ m/(\w+)\s+((\w+)\s+(\w+))\s+(\w+)\s+(\w+)/;

print join ", ", map { "'".$_."'" } @matches;
# prints "'colourless', 'green ideas', 'green', 'ideas', 'sleep', 'furiously'"
```

用`=~ s///`运算符进行正则表达式替换。

```
my $string = "Good morning world";
$string =~ s/world/Vietnam/;
print $string; # "Good morning Vietnam"
```

请注意`$string`的内容发生了怎样的改变。你必须在`=~ s///`运算符左边提供一个scalar变量，如果你提供了字面字符串，会返回一个错误。

`/g`标志表示"全局匹配"（译者注：原文"group match"，应为"global match"更为确切）。

在scalar上下文中，每次`=~ m//g`调用都会返回下一个匹配项，成功是返回true，而失败时返回false。然后你还是可以通过`$1`等等来得到匹配的组。例如：

```
my $string = "a tonne of feathers or a tonne of bricks";
while($string =~ m/(\w+)/g) {
  print "'".$1."'\n";
}
```

在列表上下文中，`=~ m//g`一次性返回所有匹配的结果。

```
my @matches = $string =~ m/(\w+)/g;
print join ", ", map { "'".$_."'" } @matches;
```

每次`=~ s///g`调用会进行一次全局的查找/替换，并且返回匹配的次数。在这里，我们把所有元音字母用字母"r"替代。

```
# 先不用/g进行一次替换
$string =~ s/[aeiou]/r/;
print $string; # "r tonne of feathers or a tonne of bricks"

# 再替换一次
$string =~ s/[aeiou]/r/;
print $string; # "r trnne of feathers or a tonne of bricks"

# 用/g全部替换
$string =~ s/[aeiou]/r/g;
print $string, "\n"; # "r trnnr rf frrthrrs rr r trnnr rf brrcks"
```

`/i`标志表示查找替换对于大小写不敏感。

`/x`标志允许正则表达式中包含空白符（例如换行符）和注释。

```
"Hello world" =~ m/
  (\w+) # one or more word characters
  [ ]   # single literal space, stored inside a character class
  world # literal "world"
/x;

# returns true
```

模块和包

在Perl中，模块（module）和包（package）是不同的东西。

模块

*模块*是你可以*包含*在另一个Perl文件（脚本或模块）中的一个`.pm`文件，是与`.pl`Perl脚本语法完全相同的文本文件。一个示例模块文件可能位于`C:\foo\bar\baz\Demo\StringUtils.pm`或者`/foo/bar/baz/Demo/StringUtils.pm`，并且有如下内容：

```
use strict;
use warnings;

sub zombify {
        my $word = shift @_;
        $word =~ s/[aeiou]/r/g;
        return $word;
}

return 1;
```

因为模块在被加载时会自顶向下执行，你需要在结尾处返回一个true表示加载成功。

为了让Perl解释器能够找到这些Perl模块文件，调用`perl`程序前，包含它们的目录名需要被添加到环境变量`PERL5LIB`中。列出包含这些模块的根目录，而不是其中的某些子目录或者模块本身：

```
set PERL5LIB=C:\foo\bar\baz;%PERL5LIB%
```

或者

```
export PERL5LIB=/foo/bar/baz:$PERL5LIB
```

一旦Perl模块被创建并且`perl`知道如何找到它以后，你就可以使用内置函数`require`在Perl脚本中查找并执行它。比如，调用`require Demo::StringUtils`使Perl解释器去逐个查找所有列在`PERL5LIB`中的目录，看是否有叫做`Demo/StringUtils.pm`的文件。我们的示例脚本可以叫做`main.pl`，并且包含以下内容：

```
use strict;
use warnings;

require Demo::StringUtils;

print zombify("i want brains"); # "r wrnt brrrns"
```

注意，在这里我们用双冒号`::`作为目录的分隔符。

现在问题来了：如果`main.pl`包含很多`require`调用，而且每个被加载的模块又包含更多`require`调用，那我们要找到`zombify()`子程序最初的定义就太困难了。解决方案是使用包。

包

*包*是用来声明子程序的命名空间。所有的子程序默认都被声明在当前包中，而程序开始执行的时候，你位于`main`包中，不过你可以用内置函数`package`来切换包：

```
use strict;
use warnings;

sub subroutine {
        print "universe";
}

package Food::Potatoes;

# 没有冲突:
sub subroutine {
        print "kingedward";
}
```

注意，我们这里使用双冒号`::`作为命名空间的分隔符。

当你调用一个子程序的时候，你默认会调用当前包中的子程序。你也可以显示指定包的名字，我们继续上面的脚本，看看会发生什么：

```
subroutine();                # "kingedward"
main::subroutine();          # "universe"
Food::Potatoes::subroutine(); # "kingedward"
```

所以对上面描述的问题的一个符合逻辑的解决方案就是把`C:\foo\bar\baz\Demo\StringUtils.pm`或者`/foo/bar/baz/Demo/StringUtils.pm`改为：

```
use strict;
use warnings;

package Demo::StringUtils;

sub zombify {
        my $word = shift @_;
        $word =~ s/[aeiou]/r/g;
        return $word;
}

return 1;
```

然后把`main.pl`改为：

```
use strict;
use warnings;

require Demo::StringUtils;

print Demo::StringUtils::zombify("i want brains"); # "r wrnt brrrns"
```

下面这些内容可要仔细阅读了。

在Perl语言中包和模块是彼此独立完全不同的两个功能，它们恰好都是用双冒号作为分隔符根本就是个掩人耳目的把戏。在一个脚本或者模块中多次切换包是可行的，在不用位置的多个文件中使用同一个包名也是可行的。调用`require Foo::Bar`*并不会*去查找并且加载一个有`package Foo::Bar`的文件，也不一定会加载定义在`Foo::Bar`命名空间里的子程序。调用`require Foo::Bar`仅仅表示加载一个名为`Foo/Bar.pm`的问题，与其中有什么包的声明没有*任何*关系，也许那个文件中声明了`package Baz::Qux`和其他乱七八糟的内容。

同样的，调用`Baz::Qux::processThis()`子程序并不一定要声明在名叫`Baz/Qux.pm`的文件里，它可能被定义在*任何地方*。

分离这两种功能可能是Perl中最糟糕的一个设计，而如果把它们视作分开的功能，将带来混乱，以及让人抓狂的代码。值得庆幸的是，主流的Perl程序员总是遵循下面两个规则：

1. **Perl脚本（`.pl`文件）不应该包含`package`声明。**
2. **Perl模块（`.pm`文件）必须包含且仅包含一个`package`声明，且包名与它的文件名、所在的位置一致。**例如，模块`Demo/StringUtils.pm`必须由`package Demo::StringUtils`开头。

因此，你会发现实际工作中，绝大部分由可靠的第三方提供的"包"和"模块"的概念是*可以*交换混用的。然而，很重要的是，你千万不能把这个当做承诺，因为将来有一天你一*定会*碰上一个疯子写的代码。

Perl的面向对象

Perl不是面向对象编程的最佳选择，Perl的面向对象机制是后来嫁接进去的，下面我们就看看是怎么回事。

- *对象*只是一个引用（也就是一个scalar变量），它恰好知道自己属于哪个类。要告诉一个引用它所指向的内容属于哪个类，使用**`bless`**。要知道引用所指向的内容属于哪个类（如果有的话），使用**`ref`**。
- *方法*只是一个子程序，接受对象（或者对于类的方法，就是包名）作为第一个参数。使用`$obj->method()`可以调用对象的方法，用`Package::Name->method()`可以调用类的方法。（译者注：所谓类的方法，在其他语言里就相当于类的静态方法。）
- *类*就是包含一组方法的包。

下面有个简短的例子来帮助我们弄清楚这些概念。示例模块`Animal.pm`包含`Animal`类，内容如下：

```
use strict;
use warnings;

package Animal;

sub eat {
        # 第一个参数总是操作所基于的对象
        my $self = shift @_;

        foreach my $food ( @_ ) {
                if($self->can_eat($food)) {
                        print "Eating ", $food;
                } else {
                        print "Can't eat ", $food;
                }
        }
}

# 就这个参数来说，假设动物可以吃任何东西
sub can_eat {
        return 1;
}

return 1;
```

然后我们可以这样使用这个类：

```
require Animal;

my $animal = {
        "legs"   => 4,
        "colour" => "brown",
};                      # $animal是一个普通的hash的引用
print ref $animal;      # "HASH"
bless $animal, "Animal"; # 现在它是"Animal"类的对象
print ref $animal;      # "Animal"
```

注意：任何引用都可以被转换（bless）成任何类的对象。需要由你来保证（1）这个引用指向的内容可以被当做这个类的对象来使用，并且（2）被转换成的这个类存在，并且已经被加载了。

你仍然可以按以前的方式操作这个hash：

```
print "Animal has ", $animal->{"legs"}, " leg(s)";
```

但你也可以同样用`->`运算符调用这个对象的方法，就像这样：

```
$animal->eat("insects", "curry", "eucalyptus");
```

最后那句调用等价于`Animal::eat($animal, "insects", "curry", "eucalyptus")`。

构造函数

构造函数是这个类返回新对象的方法。如果你需要，声明一个就是了，用你喜欢的任何名字都可以。对于类的方法，第一个参数是类名而不是一个对象，在这个例子里就是`"Animal"`：

```
use strict;
use warnings;

package Animal;

sub new {
        my $class = shift @_;
        return bless { "legs" => 4, "colour" => "brown" }, $class;
}

# ...etc.
```

然后像下面这样使用：

```
my $animal = Animal->new();
```

继承

要创建一个类继承自基类，用`use parent`，假设我们给`Animal`创建一个子类叫`Koala`，位于`Koala.pm`：

```
use strict;
use warnings;

package Koala;

# 继承自Animal
use parent ("Animal");

# 重载一个方法
sub can_eat {
        my $self = shift @_; # 没有使用，你也可以直接在这里写"shift @_;"
        my $food = shift @_;
        return $food eq "eucalyptus";
}

return 1;
```

下面是一些示例程序：

```
use strict;
use warnings;

require Koala;

my $koala = Koala->new();

$koala->eat("insects", "curry", "eucalyptus"); # 只吃eucalyptus
```

最后那个方法调用尝试执行`Koala::eat($koala, "insects", "curry", "eucalyptus")`，但子程序`eat()`并没有在`Koala`包里定义。然而，因为`Koala`有父类`Animal`，Perl解释器会再尝试调用`Animal::eat($koala, "insects", "curry", "eucalyptus")`，这回没问题。请注意`Animal`类是如何自动被`Koala.pm`加载的。

因为`use parent`接受一组父类的名字，所以Perl支持多重继承，当然也就包含了它所带来的所有好处和噩梦。

BEGIN块

`BEGIN`块在`perl`解释完这个代码块以后就立即被执行，甚至在文件剩下的部分被解释之前，而这个代码块在运行时则被忽略：

```
use strict;
use warnings;

print "This gets printed second";

BEGIN {
        print "This gets printed first";
}

print "This gets printed third";
```

`BEGIN`块总是首先执行。如果你创建了多个`BEGIN`块（别这么做），它们将按照解释器解释它们的顺序自上而下执行。`BEGIN`即使出现在脚本中间（别这么做）或者脚本最后（也别这么做），它也会首先被执行。**不要搞乱自然的代码执行顺序，总是把`BEGIN`块放在开**

头！

`BEGIN`块在解释完后立即被执行，执行完毕以后将从这个`BEGIN`块结束处继续*解释*剩下的代码。如果`BEGIN`块以外的任何代码被执行了，那么整个脚本或者模块就已经被解释了一遍，且仅有一遍。

```
use strict;
use warnings;

print "This 'print' statement gets parsed successfully but never executed";

BEGIN {
        print "This gets printed first";
}

print "This, also, is parsed successfully but never executed";

...because e4h8v3oitv8h4o8gch3o84c3 there is a huge parsing error down here.
```

（译者注：上面程序的最后一行不是注释，作者写最后一行是构造一个语法错误，因而造成BEGIN块在解释到这里之前就已经被执行，而BEGIN块执行完毕以后继续恢复解释，一旦遇上语法错误，脚本其他部分将不会再被执行。）

因为它们在脚本编译时就执行，`BEGIN`块即使在条件分支中也*仍然*会在编译时就运行，哪怕条件将被判定为false，因为在那时条件*还根本没有被求值*，甚至*可能永远不会被求值*。

```
if(0) {
        BEGIN {
                print "This will definitely get printed";
        }
        print "Even though this won't";
}
```

不要把`BEGIN`块放在条件分支里！如果你要在编译时做一些条件判断，把这个条件判断放在`BEGIN`块*里面*：

```
BEGIN {
        if($condition) {
                # etc.
        }
}
```

use

好，现在让我们来理解一下包、模块、类的方法和`BEGIN`块那模棱两可的行为以及语义，我会来解释一下超级常见的`use`函数。

下面三条语句：

```
use Caterpillar ("crawl", "pupate");
use Caterpillar ();
use Caterpillar;
```

分别和下面的三段等价：

```
BEGIN {
        require Caterpillar;
        Caterpillar->import("crawl", "pupate");
}
BEGIN {
        require Caterpillar;
}
BEGIN {
        require Caterpillar;
        Caterpillar->import();
}
```

- 不，这三个例子并没有放错顺序，只是Perl比较笨罢了。
- `use`只是`BEGIN`块的伪装，同样的警告对此也适用。`use` 语句必须总是放在文件开头，并且**永远不要放在条件分支里**。
- `import()`并不是Perl的内置函数，它只是一个**用户自定义的类方法**。定义或者继承`import()`函数的重任就落在写`Caterpillar`这个包的程序员身上了。这个方法理论上可以接受任何东西作为参数，也可以对参数做任何操作。`use Caterpillar;`可以做任何事情，你需要查询`Caterpillar.pm`的文档来判断到底会发生什么。
- 请注意`require Caterpillar`是如何加载一个名为`Caterpillar.pm`的**模块**的，而`Caterpillar->import()`则调用定义在`Caterpillar`**包**里的子程序`import()`。我们只能一起期待这里的模块和包是一致的！

Exporter

定义一个`import()`方法最常见的办法是从Exporter模块继承下来。Exporter是一个核心模块，也是Perl语言中成为*事实标准*的核心功能。在Exporter的`import()`实现中，你传入的参数列表将被认为是子程序名字的列表，当一个子程序被`import()`，它在当前包和原来所在的包里就都可以被使用了。

用一个例子最能帮助理解这个概念。`Caterpillar.pm`的内容如下：

```
use strict;
use warnings;

package Caterpillar;

# 继承自Exporter
use parent ("Exporter");

sub crawl  { print "inch inch";   }
sub eat    { print "chomp chomp"; }
sub pupate { print "bloop bloop"; }

our @EXPORT_OK = ("crawl", "eat");

return 1;
```

包变量`@EXPORT_OK`应该包含子程序名字的列表。

另一块代码就可以通过名字来`import()`这些子程序，一般使用`use`语句：

```
use strict;
use warnings;
use Caterpillar ("crawl");

crawl(); # "inch inch"
```

在这种情况下，当前包是`main`所以`crawl()`实际上是调用了`main::crawl()`，（因为被导入了）映射到`Caterpillar::crawl()`。

注意：不管`@EXPORT_OK`的内容是什么，通过"常规写法"使用这些函数总是可以的：

```
use strict;
use warnings;
use Caterpillar (); # 没有提供任何子程序名，import()不会被调用

# 然而……
Caterpillar::crawl();  # "inch inch"
Caterpillar::eat();    # "chomp chomp"
Caterpillar::pupate(); # "bloop bloop"
```

Perl没有私有方法，习惯上在希望私有的方法名前面有一个或者两个下划线。

`@EXPORT`

Exporter模块还定义了一个包变量叫`@EXPORT`，也包含一组子程序名。

```
use strict;
use warnings;

package Caterpillar;

# 继承自Exporter
use parent ("Exporter");

sub crawl  { print "inch inch";   }
sub eat    { print "chomp chomp"; }
sub pupate { print "bloop bloop"; }

our @EXPORT = ("crawl", "eat", "pupate");

return 1;
```

如果没有给`import()`传入任何参数，`@EXPORT`中写出的子程序将全部被导出，就像这样：

```
use strict;
use warnings;
use Caterpillar; # 调用import()但不提供参数

crawl();  # "inch inch"
eat();    # "chomp chomp"
pupate(); # "bloop bloop"
```

不过我们又回到了那种情况，没有其他提示的话，我们很难知道`crawl()`原先是在哪儿定义的。这件事情有两个寓意：

1. 当我们用Exporter创建模块的时候，不要用`@EXPORT`来导出子程序，总是让调用者以"常规方法"调用子程序，或者显式地`import()`它们（使用比如：`use Caterpillar ("crawl")`提供了一条很强的线索，告诉我们可以从`Caterpillar.pm`中找到`crawl()`的定义）。
2. 当`use`一个使用Exporter的模块时，总是显式写明你希望`import()`的子程序，如果你不想`import()`任何子程序，而是用常规方法引用它们，你必须显式提供一个空的列表：`use Caterpillar ()`。

杂项

- 核心模块Data::Dumper可以被用于输出任意scalar到屏幕上，这是非常有用的调试工具。
- 还有另一种语法`qw{ }`可以用来声明array，常常在`use`语句用到它：

    ```
    use Account qw{create open close suspend delete};
    ```

    有许多引号一样的运算符。
- 在`=~ m//`和`=~ s///`运算符中，你可以用花括号代替斜杠作为正则表达式的分隔符，当你的正则表达式中包含很多斜杠时候就很有用了，要不然你就得使用很多反斜杠来进行跳脱。例如，`=~ m{///}`将匹配三个斜杠而`=~ s{^https?://}{}`会移除URL的协议部分。
- Perl没有`CONSTANTS`。现在不鼓励使用它们，不过以前不一定。常量实际上就是省略括号的子程序调用。
- 有时候人们省略hash键两旁的引号，写成`$hash{key}`而非`$hash{"key"}`。 当这个孤零零的`key`恰好表示字符串`"key"`而不是子程序调用`key()`的时候，它们才能侥幸成功。
- 如果你看到一块由两个左尖括号作为分隔符包围起来的没有格式化的代码，就像`<<EOF`，可以通过在Google中搜索"here-doc"找到它的解释。（译者注：这是再一次吐槽因为Perl滥用符号导致难以搜索。）
- 警告！许多内置函数调用时都可以不给参数，**那样它们就会使用`$_`代替**，希望这可以帮助你理解下面这种写法：

    ```
    print foreach @array;
    ```

    还有

    ```
    foreach ( @array ) {
            next unless defined;
    }
    ```

    我不喜欢这种写法，因为在代码重构时将会遇到麻烦。

两个半小时这就到了。

Back to Things Of Interest