

Perl 是一种能“干实事”的语言。它灵活、宽容、可塑。在一名编程能者的手中，它可以完成几乎所有的任务，从“一句话”的简单运算和自动完成任务到多人、多年的项目，外加夹在两者之间的部分。

Perl 功能强大，摩登 Perl——一个集最上乘知识和经验，以及来自全球 Perl 社区的可重用惯用语的 Perl——它可维护、快速、易用。也许最重要的是，它能够帮助你无挫折、无繁文缛节地做你需要做的事。

Perl 是一门实用主义语言。你，程序员，完全地掌控着自己编写的程序。相对于操控你的思想和你面对的问题来适应语言设计者认为你应该怎样写程序，Perl 允许你按你觉得合适的方式解决问题。

Perl 是一门伴随你成长的语言。你能够用你在一小时内阅读本书所学到的知识写出有用的程序。如果你还能花点时间理解语法、语义、语言设计背后的哲学，你会变得更有成效。

首先，你需要知道如何学到更多东西。

## Perldoc

Perl 最有用也是最不受感恩的功能之一就是 `perldoc` 实用工具。这个程序是每一个完整 Perl 5 安装的一部分。在免费的 GNU/Linux 的发行版或是其它类 Unix 系统下，你也许需要安装一项额外的软件包，Debian 和 Ubuntu 上为 `perl-doc`。它能显示系统上每一个已安装 Perl 模块的文档——无论是核心模块，或是那些来自 Comprehensive Perl Archive Network (Perl 综合典藏网, CPAN) 的部分——还包括上千页 Perl 核心文档。

如果你倾向于一份在线版本，<http://perldoc.perl.org/> 存有 Perl 文档的最近版本。<http://search.cpan.org/> 可显示 CPAN 上所有模块的文档。对于 Windows 用户，ActivePerl 和 Strawberry Perl 都在开始菜单提供了指向这份文档的链接。

`perldoc` 的默认行为是显示某模块的文档或是 Perl 核心文档的特定章节：

```
$ B<perldoc List::Util>
$ B<perldoc perltoc>
$ B<perldoc Moose::Manual>
```

第一个例子解开为 `List::Util` 模块所编写的文档并按适合你的显示屏的方式显示出来。CPAN 模块 (*cpan*) 的社区标准建议额外的库使用和核心模块一致的格式，使得阅读诸如 *Data::Dumper* 核心模块以及从 CPAN 安装的其他模块的文档并无差别。标准文档模板包含了一份对模块的描述，用法演示，和后续对模块的详细解释和接口介绍。虽然文档的长度因作者而不同，其形式却相当统一。

第二个例子显示一个纯粹的文档文件，此处是核心文档自己的目录。这个文件描述了核心文档的每一个独立的部分。浏览一下，简单地看看 Perl 的能力。

第三个例子是第二个例子的翻版。*Moose::Manual* 是 Moose CPAN 发行版 (*moose*) 的一部分。它同样也是一个纯文档，不包含任何代码。

类似的，`perldoc perlfaq` 将会显示 Perl 5 常见问题的目录。仅扫一眼这些问题也会使人获益匪浅。

`perldoc` 实用工具还有不少能力(参见 `perldoc perldoc`)。其中两个最有用的是 `-q` 以及 `-f` 参数。`-q` 参数读取一个或多个关键字，在 Perl FAQ 中查找，并显示所有结果。从而 `perldoc -q sort` 返回三个常见问题：我如何按（任何条件）排序数组？，我如何将一个哈希排序（可选的附加条件：通过值而非键）？，以及我如何使一个哈希总保持在有序状态？。

`-f` 参数显示核心文档中某 Perl 内置函数的相应部分。`perldoc -f sort` 解释了 `sort` 操作符的行为。如果你不知道你要查找函数的名称，`perldoc perlfunc` 可以给出一张函数列表。

`perldoc perlop` 和 `perldoc perlsyn` 记录了 Perl 中符号形式的操作符和语法结构。`perldoc perldiag`

解释了 *Perl* 警告消息的含义。

*Perl 5* 的文档系统是 *POD*，或作 *Plain Old Documentation*。*perldoc perlpod* 描述了 *POD* 的工作方式。*perldoc* 实用工具能显示你为你的项目创建和安装的所有 *Perl* 模块中的 *POD*，其他诸如 *podchecker* 等 *POD* 工具，可以验证你的 *POD* 的格式，*Pod::Webserver*，可以通过一个小型 *Web* 服务器以 *HTML* 的形式显示本地的 *POD*，同样也将验证其正确性。

*perldoc* 还有其他用途。给出 *-l* 命令行参数，它将显示文档文件的路径而非文件内容。注意：一个模块除 *.pm* 文件外可能还有多个分离的 *.pod* 文件。给出 *-m* 参数，它将显示整个模块的内容，包括代码，并不对任何 *POD* 指令进行处理。

## 表达力

在 *Larry Wall* 创造 *Perl* 之前，他研究的是语言学和人类语言。他的经历持续地影响着 *Perl* 的设计。因项目的风格、编写程序的可用时间、所期望的维护代价，甚至是你个人的表达能力的不同，编写一个 *Perl* 程序的方法是多种多样的。你可以采用直截了当、自顶而下的风格。你可能编写很多小而独立的函数。你可能会用类和对象对你的问题建模。你也可以避让或拥抱 *Perl* 的高级功能。

*Perl* 黑客们对此有句口号，*TIMTOWTDI*，发音为“*Tim Toady*”，或作“*There's more than one way to do it!*”

这种表达力提供了一个大型调色盘，用它巨匠可以创造出惊人的、宏大的建筑物，仅将各式技巧不明智地堆砌起来只能阻碍代码的可维护性和易读性。你可以写出优秀或者一塌糊涂的代码。选择权在你这里。……但是如果你必须弄得一塌糊涂，那么对其他人好一点。

其他语言也许会建议说，按一条强制的指导思想走每一步路才是解决问题的正途。*Perl* 允许你针对最重要的标准进行优化。在问题的范围之内，你可以选择若干合适的方法——但是注意一下可读性和未来的可维护性。

作为一个 *Perl* 的新手，你可能会觉得某些结构难以理解。*Perl* 社区已经总结并推出了不少强大的惯用语 (*idioms*)。别指望一下就能理解它们。*Perl* 的一些特性以微妙的方式互相联系着。

学习 *Perl* 就好像在学习一门第二或第三口语。你会学到一些单词，然后将它们串起来构成句子，最终能够进行一小段简单的对话。熟能生巧，读也一样，写也一样。你不用一次性理解本章的所有细节就能够写出卓有成效的 *Perl* 程序。当你阅读本书余下部分时，请记住这些原则。

另一个 *Perl* 的设计目标是，尽可能不使有经验的 (*Perl*) 程序员吃惊。举例来说，用一个数值操作符将两个标量相加 (*\$first\_num + \$second\_num*) 很显然是一次数值的操作，该操作符必须按数值对待这两个标量来产生数值结果。无论 *\$first\_num* 和 *\$second\_num* 中的内容是什么，也不必用户或程序员手动操作，*Perl* 会将它们强制转为数值 (*numeric\_coercion*)。你已经通过选择数值操作符 (*numeric\_operators*) 表达了你想将它们用作数值的意图，因此 *Perl* 很高兴地替你处理后续工作。

一般而言，*Perl* 程序员可以期望 *Perl* 能完成他们想要 *Perl* 完成的事，这便是 *DWIM-- do what I mean* 的意思。你也能见到另一种提法 最小惊奇原则。对 *Perl* 有个粗略的了解之后（特别是它的 *context\_philosophy*），在读到一个不熟悉的 *Perl* 表达式时，就有可能猜出它的意图。

如果你刚接触 *Perl*，你将渐渐培养出这种技能。*Perl* 表达力的反面就是，新手在学全 *Perl* 强大功能之前就可以写出有用的程序。*Perl* 社区通常将此称作 *baby Perl*。虽然它可能听

上去有些轻蔑，但请千万别生气，每个人都是这样过来的。抓住向更有经验程序员学习的机会，并对你不理解的惯用语和结构（向他们）索求解释。

一个 *Perl* 新手可能会按下面的方法把列表中所有的元素乘以三：

```
my @tripled;  
my $count = @numbers;  
  
for (my $i = 0; $i < $count; $i++)  
{  
    $tripled[$i] = $numbers[$i] * 3;  
}
```

一个 *Perl* 内行可能会这样写：

```
my @tripled;  
  
for my $num (@numbers)  
{  
    push @tripled, $num * 3;  
}
```

一个经验丰富的 *Perl* 黑客可能会这样写：

```
my @tripled = map { $_ * 3 } @numbers;
```

编写 *Perl* 程序的经验将会帮助你专注于要做什么而非怎么做。

### *Perl*

是一门有意随着你对编程理解程度的增长而成长的语言。它不会因写出简单的程序而惩罚你。它允许你为了直接明了、富表达力、代码重用和可维护性来完善、扩展你的程序。好好利用这条哲学。出色地完成你的任务比写出概念上纯美的程序更为重要。

本书余下部分展示如何按对你有利的方式使用 *Perl*。

## 上下文

[当阅读这个小节的时候，我意识到 *Perl* 中的上下文的基础是“用什么操作符”以及“在哪里用它”。在本小节和 *operator\_types.pod* 中，你决不会直接地说出来。但这是真的。我试图在这个小节的某处中直接地说出来，但是我无法决定在哪里已经怎样说，因此这只是一个提议。] 口语中有上下文

的概念，即某词汇或短语的正确用法和含义由语境所决定。理解一下在口语中的情况，“*Please give me one hamburgers!*”中不合适的复数用法 名词的复数形式和数量不符，听上去就不对，或者“*la gato*”中不正确的性别 冠词为阴性，但是名称为阳性 使得母语人士轻声窃笑。同样考虑代词“*you*”或者名词“*sheep*”，是单数是复数还得由句子的余下部分决定。

### *Perl*

中的上下文是类似的，这门语言理解对数据数量上的期望同时也知道应该提供什么样的数据。*Perl* 将高兴地尝试提供给你恰好合你心意的数据。

### *Perl*

中上下文的每一种类型，都和你所需某操作符结果的个数（零个、一个、或许多个）相对应，同一个操作符在（不同的上下文中）有着不同的行为。在 *Perl* 中，如果你要求：“给我零个结果，我不在乎有没有”或是“给我一个结果”再或是“给我多个结果”，那么一个特定的结构按这些不同的要求做不同的事情是完全可能的。

同样，特定上下文将你所需明朗化：是数值，是字符串值、或是一个为真或假的值。

如果你打算把 *Perl*

代码当成一系列独立于环境的单一表达式来读写，那么上下文将会变得十足狡猾。

在一次长时间的调试后，你也许会一拍脑门，发现你对程序上下文的假设是错误的。然而在对上下文了如指掌后，它们会使你的代码更清晰、更简洁、更灵活。

## 空、标量和列表上下文

上下文之一掌控着你期望事物的多少。这就是数量上下文。这个上下文和英语中“主谓一致”相当。即使尚未了解这条规则的正式定义，你很有可能理解句子 "*Perl are a fun language*" 中的错误。*Perl* 中的规则是，你所要求事物的数量决定了你得到的。

假设你有一个称为 *some\_expensive\_operation()* 的函数 (*functions*)，它进行一昂贵的计算并产生许许多多结果。如果你直接调用该函数且对返回值不加利用，那么就称你在空上下文中调用了这个函数：

```
some_expensive_operation();
```

将函数的返回值赋值给单个元素使得函数在标量上下文中求值：

```
my $single_result = some_expensive_operation();
```

将调用函数的结果赋值给一个数组 (*arrays*) 或是列表，或者在一个列表中使用该结果，使得函数在列表上下文中求值：

```
my @all_results      = some_expensive_operation();
my ($single_element) = some_expensive_operation();
process_list_of_results( some_expensive_operation() );
```

前例的第二行可能看上去有些迷惑，这里的括号给了编译器一点提示：尽管这里只有一个标量，该赋值应在列表上下文中发生。在语义上等同于将列表中的第一个元素赋值给一个标量，并将列表的其余部分赋值给一个临时数组，随即丢弃该数组。除非真的发生（类似于后例的）数组赋值：

```
my ($single_element, @rest) = some_expensive_operation();
```

为什么对函数来说上下文是有趣的？假如 *some\_expensive\_operation()* 计算的是按优先级排序过的家务事。如果只有做一件事的时间，你可以在标量上下文中调用它，获得一项有用的任务——也许并不一定是最重要的，但是不会是排在最底下的那一项。在列表上下文中，该函数能完成所有的排序、搜索和比较，可以给你一份顺序合适且详尽的列表。如果你想所有事情都做，但是只有那么些时间，你可以用一个一两个元素的列表（来接受该函数在列表上下文中的返回值）。

在列表上下文中对函数或表达式——除赋值外——求值，可能造成迷惑性结果。列表会把列表上下文散播到其所包含的表达式中。下列对 *some\_expensive\_operation()* 的调用都发生于列表上下文：

```
process_list_of_results( some_expensive_operation() );
```

```
my %results =
(
    cheap_operation    => $cheap_operation_results,
```

```
expensive_operation => some_expensive_operation(), # OOPS!
);
```

上例 `expensive_operation` 位于列表上下文，因为它被赋值到一个哈希中，而哈希赋值需要一  
键值对列表，导致在哈希赋值内的所有表达式在列表上下文中求值。

后一个例子通常使期望该调用为标量上下文的新手程序员吃惊。相反，这是列表上下文，因为该上  
下文为哈希赋值所强制。使用 `scalar` 操作符可以迫使其在标量上下文求值：

```
my %results =
(
    cheap_operation    => $cheap_operation_results,
    expensive_operation => scalar some_expensive_operation(),
);
```

## 数值、字符串及布尔上下文

另一类型的上下文决定了 *Perl* 如何理解某块数据——不是你要多少数据，而是数据的意义。  
你也许早已觉察到 *Perl*  
可以灵活地指出你所有的是数字还是字符串并在按需在两者之间转换。这就是 值上下文，有助解释  
*Perl* 是如何做这些事情的。不必明确声明（至少是跟踪）某变量包含（或产生自某函数）的数据的  
类型，作为交换，*Perl* 提供了特定类型的上下文，由它们告知编译器在某项  
操作期间如何对待一个给定的值。

假设你想比较两个字符串的内容。`eq` 操作符能告诉你这些字符串中是否包含相同的信息：

```
say "Catastrophic crypto fail!" if $alice eq $bob;
```

当 明明知道 字符串内容不同，但是比较结果却是相同时，你可能会感到莫名其妙：

```
my $alice = 'alice';
say "Catastrophic crypto fail!" if $alice == 'Bob'; # 啊呀
```

`eq` 操作符通过强制 字符串上下文，按字符串对待它的操作数。`==` 操作符则强制 数值上下文。  
示例代码出错的原因在于，两个字符串在用作数字时候的值是 0 (*numeric\_coercion*)。

布尔上下文 发生在当你在条件语句中使用某值时。在前面的例子中，`if` 语句在布尔上下文中求出 `eq` 和  
`==` 操作的结果。

*Perl* 会尽最大努力将值强制转换成正确的类型 (*coercion*)，并依赖于所用的操作符。一定要针对你所需的 上下文使用正确的操作符。

在极少数情况下，没有合适类型的操作符存在，你也许需要明确地强制上下文。强制数值上  
下文，在变量前加零。

强制字符串上下文，将变量和空字符串拼接起来。强制布尔上下文，使用双重否定操作符。

```
my $numeric_x = 0 + $x; # 强制数值上下文
my $stringy_x = ". $x; # 强制字符串上下文
my $boolean_x = !!$x; # 强制布尔上下文
```

大体上说，相比数量上下文，类型上下文较易理解和识别。一旦你理解它们的存在，并知道  
什么操作符提供什 么上下文 (*operator\_types*)，你很少会因为它们而犯错。

## 隐式理念

像不少口语一样，*Perl* 提供了语言学捷径。上下文即是这样一个特性。阅读代码的无论是编译器还是程序员，都可以通过现有信息了解到所期望结果的数量和操作符的类型，而不必额外添加明确信息来消歧。*Perl* 中也存在其他类似的特性，包括本质上是代词的默认变量。

### 默认标量变量

默认标量变量（也称为话题变量）`$_`，是 *Perl* 中语言学捷径的最佳例证。它最显眼的地方 就是它的“缺席”：当缺少一明确变量时，*Perl* 中许多内置操作是针对 `$_` 的内容进行的。你仍可以将 `$_` 填入所缺变量的位置，但是这通常是多此一举。

举例来说，`chomp` 操作符移去任何尾随字符串的换行符序列：

```
my $uncle = "Bob\n";
say "$uncle";
chomp $uncle;
say "$uncle";
```

在没有指明变量时，`chomp` 移去 `$_` 尾部的换行符，因此下列两行代码是等价的：

```
chomp $_;
chomp;
```

`$_` 在 *Perl* 中的功能等同于汉语中的代词“它”（英语中的代词 *it*）。第一行读作“`chomp` 它”，第二行则读作“`chomp`”。当你不指明对什么做 `chomp` 操作时，*Perl* 理解你的意思，*Perl* 总是 `chomp` 它。

类似的，内置函数 `say` 和 `print` 在缺少参数时作用于 `$_` 之上：

```
print; # 将 $_ 打印到当前所选文件句柄
say;   # 将 $_ 打印到当前所选文件句柄
       # 外加结尾换行符
```

*Perl* 的这套正则表达式装备 (*regular\_expressions*) 同样可以在 `$_` 上进行匹配、替换和转译操作：

```
$_ = 'My name is Paquito';
say if /My name is/;

s/Paquito/Paquita/;

tr/A-Z/a-z/;
say;
```

*Perl* 中许多标量操作符（包括 `chr`、`ord`、`lc`、`length`、`reverse` 及 `uc`）在你不提供替代选项时，作用于默认标量变量上。

*Perl* 的循环指令 (*looping\_directives*) 同样设置 `$_` 变量，比如用 `for` 遍历一个列表：

```
say "#B<$_>" for 1 .. 10;

for (1 .. 10)
{
    say "#B<$_>";
}
```

```
}
```

.....或者是 *while*:

```
while (<STDIN>)
{
    chomp;
    say scalar reverse;
}
```

.....再或者是用 *map* 转换列表:

```
my @squares = map { B<$_> * B<$_> } 1 .. 10;
say for @squares;
```

.....又或者是用 *grep* 过滤列表:

```
say 'Brunch time!' if grep { /pancake mix/ } @pantry;
```

如果你在用到 `$_` 的代码内调用函数，无论是隐式还是显式，可能导致 `$_` 的值被覆盖。相似地，如果你编写了一个使用 `$_` 的函数，就有可能搅乱调用者对 `$_` 的利用。*Perl 5.10* 允许你用 *my* 将 `$_` 作为词法变量来声明，这样便可以避免上述行为。明智一点。

```
while (<STDIN>)
{
    chomp;

    # 一例反面教材
    my $munged = calculate_value( $_ );
    say "Original: $_";
    say "Munged : $munged";
}
```

在本例子中，若 `calculate_value()` 或它偶然调用了其他函数导致 `$_` 的值改变，则在 整一次 *while* 循环中，此值将保持被改后的状态。用 *my* 来声明可以避免此类情况:

```
while (my $_ = <STDIN>)
{
    ...
}
```

当然，使用带有具体名字变量会比较清晰:

```
while (my $line = <STDIN>)
{
    ...
}
```

你可以在书面写作中用到“它”（英语：“*it*”）的地方使用 `$_`：适度地、在小且精心圈定的范围内。

## 默认数组变量

在 *Perl* 拥有一个隐式的标量变量的同时，它也有两个隐式数组变量。*Perl* 通过一个名为 `@_` 的数组向函数传递参数。函数内部处理数组的操作符 (*arrays*) 默认影响这个数组。因此，以下二例代码是等价的:

```

sub foo
{
    my $arg = shift;
    ...
}

sub foo_explicit
{
    my $arg = shift @_;
    ...
}

```

正如 `$_` 对应代词 它，`@_` 对应代词 它们。不同于 `$_`，当你调用其他函数时，*Perl* 自动地为你局部化 `@_`。数组操作符 *shift* 和 *pop* 在没有提供其他操作数时作用于 `@_`。

在所有函数之外，默认数组变量 `@ARGV` 存有传递给程序的命令行参数。在函数内隐式用到 `@_` 的同一批数组操作符，在函数外隐式地使用 `@ARGV`。你不可以将 `@_` 用作 `@ARGV`。

`ARGV` 有一个特殊的用法。如果你从空文件句柄 `<>` 读入，则 *Perl* 会将 `@ARGV` 中的每一个元素当作文件的 名字 而打开。（如果 `@ARGV` 为空，*Perl* 会从标准输入读取。）这个隐含的 `@ARGV` 行为在编写短小的程序（例如将输入逆序输出的命令行过滤器）时很有用：

```

while (<>)
{
    chomp;
    say scalar reverse;
}

```

为什么用到 *scalar*? *say* 对其操作数施加列表上下文。而 *reverse* 又将它的上下文传递给自己的操作数，在列表上下文中它将它们作为列表对待，在标量上下文中则看作拼接字符串。这听上去有点迷糊，因为的确如此。*Perl 5* 应将不同的操作作用不同的操作符处理。

如果你用一系列文件作参数运行这个程序：

```
$ B<perl reverse_lines.pl encrypted/*.txt>
```

.....结果应该会是一系列冗长的输出。不带参数运行时，你可以提供自己的标准输入：通过管道的方式接通其他程序，或者直接从键盘打字。