1 NAME

perlintro -- Perl简介和概览

2 DESCRIPTION

该文档将为您提供Perl编程语言的快速概览，并指导您阅读其他更深入的文档。对于刚刚接触Perl的人，它可以被当作一份"入门"向导。它提供刚刚好的信息，使你能够阅读别人的Perl代码并粗略知道这些代码在做什么，以及能够自己编写简单的脚本。

这份介绍性的文档并不打算覆盖全面。它甚至不打算写的非常精确。有时为了抓住重点，我们必须牺牲完美。强烈建议您看完本介绍以后继续阅读整个Perl手册获取更多的信息。Perl手册的目录参见perltoc。

在本文档中你会到处看到对Perl文档的其他部分的引用。你可以使用你现在正用来阅读本文档的方法或`perldoc`指令去阅读那些文档。

Perl是什么?

Perl是一门常规用途的语言。最初被开发出来用于文本处理（维护）。现在被用于系统管理、web开发、网络编程、图形用户界面(GUI)开发以及更多其他等各种任务。

这门语言被设计得实用(容易使用、高效、完整)胜于美观(微小、文雅、简化)。它的主要特点是容易使用，支持面向过程编程和面向对象编程，内建强大的文本处理能力，以及拥有世界上最著名的第三方模块集合。

perl，perlfaq1和其他地方给出了关于Perl的不同定义。从而我们可以看出Perl对于不同的人意味着不同的东西，但至少他们都愿意把它记述下来。

运行Perl程序

从Unix命令行运行Perl程序：

```
perl progname.pl
```

或者把下面的代码写到你脚本的第一行：

```
#!/usr/bin/env perl
```

... 然后运行`/path/to/script.pl`。当然，该脚本必须具有执行权限，使用 `chmod 755 script.pl`改变权限(Unix下)。

更多信息，包括其他平台(如Windows和Mac OS)的说明，请阅读perlrun。

基本语法概览

一个Perl脚本/程序由一条或多条语句组成。这些语句直接写在脚本里，而不是需要写在什么`main()`或者类似的东西里。

perl语句以分号(;)结尾：

```
print "Hello, world";
```

注释使用井号(#)开始，作用直到该行末尾

```
# 这是一个注释
```

空白是无关紧要的：

```
print
   "Hello, world"
   ;
```

... 除非是在被引起来的字符串里：

```
# 下面的打印中间会有一个换行符
print "Hello
world";
```

可以使用双引号或单引号包围文字串：

```
print "Hello, world";
print 'Hello, world';
```

然而，只有双引号可以"内插"变量和特殊字符，如换行(\n)：

```
print "Hello, $name\n";    # 内插变量和换行
print 'Hello, $name\n';    # 打印出字面的$name\n
```

数字不需要用引号引起：

```
print 42;
```

你可以根据你的喜好选择把函数参数用括号括起来，或者忽略括号。括号只有在偶尔用于 阐明优先级问题的时候才被需要。

```
print("Hello, world\n");
print "Hello, world\n";
```

关于Perl语法的详细信息请参阅perlsyn。

Perl变量类型

Perl有三种变量类型：标量(scalar)、数组(array)、散列(hash)。

标量(scalar)

一个标量表示一个单一的值：

```
my $animal = "camel";
my $answer = 42;
```

标量可以是字符串、整数或浮点数。在需要的时候，Perl可以自动对它们进行相互转换。 不需要预先声明变量类型。

标量的使用可以有几种方式：

```
print $animal;
print "The animal is $animal\n";
print "The square of $answer is ", $answer * $answer, "\n";
```

有许多看起来像标点符号或行噪音(line noise)的"魔力"标量。这些特殊的标量被用于 各种用途，参考perlvar。现在你唯一需要知道的是$_，即"缺省变量"。它被用 作Perl中许多函数的缺省参数，另外某些循环结构会隐含地设置它的值。

```
print;        # 缺省会打印出$_的内容
```

数组

一个数组表示一系列值(一些值的列表)：

```
my @animals = ("camel", "llama", "owl");
my @numbers = (23, 42, 69);
my @mixed   = ("camel", 42, 1.23);
```

数组下标从0开始。下面展示了如何从数组中获取元素：

```
print $animals[0];              # prints "camel"
print $animals[1];              # prints "llama"
```

特殊变量$#array能告诉你一个数组的最后一个元素的下标：

```
print $mixed[$#mixed];      # 最后一个元素，输出1.23
```

你可能会被引诱使用$#array + 1来获取数组中元素的个数。别烦恼。再有这种需要的时候，在Perl期待标量值的地方("在标量上下文中")使用@array就能得到数组中元 素的个数：

```
if (@animals < 5) { ... }
```

我们从数组中获取元素是以$打头，这是因为我们从数组中取出的仅仅是一个单一的值 -- 你请求一个标量，你得到一个标量。

要从数组中取得多个值：

```
@animals[0,1];                # 得到 ("camel", "llama");
@animals[0..2];               # 得到 ("camel", "llama", "owl");
@animals[1..$#animals];       # 得到除第一个之外的所有元素
```

这叫做"数组切片"。

你可以对列表做各种有用的事：

```
my @sorted    = sort @animals;
my @backwards = reverse @numbers;
```

同样，也存在也些特殊的数组，比如@ARGV(脚本的命令行参数)和@_(传递给子例程 的参数)。这些都在perlvar中有详细文档。

散列

一个散列表示一套"键/值"对：

```
my %fruit_color = ("apple", "red", "banana", "yellow");
```

你可以用空白和=>操作符把它们摆放的漂亮些：

```
my %fruit_color = (
   apple  => "red",
   banana => "yellow",
);
```

要获取一个散列元素：

```
$fruit_color{"apple"};          # 得到 "red"
```

你可以分别使用keys()和values()获取散列的键列表和值列表。

```
my @fruits = keys %fruit_colors;
my @colors = values %fruit_colors;
```

散列没有特定的内部顺序，尽管你可以排列所有的键并遍历它们。

就像特殊标量和特殊数组，Perl里也有特殊散列。这其中最著名的是%ENV。它包含着所有环境变量。关于这个(和其他特殊变量)请阅读perlvar。

变量、数组和散列的更多文档请见perldata。

更复杂的数据类型使用引用构造，可以使你创建列表和散列的列表和散列。

一个引用是一个标量值。它可以指向(引用)任何其他数据类型(的值)。这样，通过存储 一个对数组或散列的引用，你可以轻易地创建列表和散列的列表和散列。

```
my $variables = {
   scalar  =>  {
               description => "single item",
               sigil => '$',
              },
   array   =>  {
               description => "ordered list of items",
               sigil => '@',
              },
   hash    =>  {
               description => "key/value pairs",
               sigil => '%',
              },
};

print "Scalars begin with a $variables->{'scalar'}->{'sigil'}\n";
```

关于引用这个主题的详细信息可以参见perlreftut、perllol、perlref和 perldsc。

变量作用域

贯穿之前章节的所有例子，都使用了这样的语法：

```
my $var = "value";
```

my实际上不是必须的；你可以仅仅使用这样的语法：

```
$var = "value";
```

然后，上面这种用法会创建贯穿你程序的全局变量。这是一个不好的编程习惯。而my 创建的是词法变量。这种变量的作用域会局限于定义它的块(也就是一套包围在大括 号({})里的语句)里。

```
my $a = "foo";
if ($some_condition) {
    my $b = "bar";
    print $a;         # 输出 "foo"
    print $b;         # 输出 "bar"
}
print $a;             # 输出 "foo"
print $b;             # 什么都不输出；$b已经超出了作用域
```

配合使用my和Perl脚本顶部的`use strict;`，意味着解释器将检查某些常见的编程 错误。举例来说，在上面的例子里，最后的`print $b`会产生一个编译时错误并阻止你 运行程序。强烈推荐使用`strict`。

条件和循环结构

Perl拥有几乎所有常见的条件和循环结构，除了case/switch(但假如你实在想要， Perl 5.8及以上的版本或CPAN里有一个Switch模块。关于模块和CPAN，请看后面关于模 块的章节)。

条件可以是任何Perl表达式。参看下一节中关于比较和布尔逻辑操作符的列表。它们在条件语句里很常用。

if

```
if ( condition ) {
    ...
} elsif ( other condition ) {
    ...
} else {
    ...
}
```

if还有一个否定版本：

```
unless ( condition ) {
    ...
}
```

这等同于`if (!condition)`，但是一个更易读的版本。

注意在Perl里括号是必须的，就算块里只有一行。然而，有一个聪明的方法可以让你的单行语句块看起来更像英语：

```
# 传统方式
if ($zippy) {
    print "Yow!";
}

# Perlish前置方式
print "Yow!" if $zippy;
print "We have no bananas" unless $bananas;
```

while

```
while ( condition ) {
   ...
}
```

基于同样的原因，也有一个否定版本 -- `unless`：

```
until ( condition ) {
   ...
}
```

你也可以使用条件后置的`while`：

```
print "LA LA LA\n" while 1;      # 永远循环
```

for

跟C一模一样：

```
for ($i=0; $i <= $max; $i++) {
   ...
}
```

在Perl里很少需要用到C样式的循环，因为Perl提供了更友善的列表遍历方法 -- `foreach`循环。

foreach

```
foreach (@array) {
   print "This element is $_\n";
}

# 你不一定非得使用缺省的$_...
foreach my $key (keys %hash) {
   print "The value of $key is $hash{$key}\n";
}
```

关于循环结构(以及一些这篇概览里没有提到的结构)的更多细节，请参见perlsyn。

内建操作符和函数

Perl自带了一大套内建函数。有一些我们已经见识过了：`print`、`sort`和`reverse`。perlfunc的开始处有内建函数的完整列表。另外你可以很方便地使用 `perldoc -f` *`functionname`*查看某个给定函数。

Perl操作符详细记述于perlop。不过这里我们先看看一些最常用的：

算术

```
+  加
-  减
*  乘
/  除
```

数字比较

```
== 相等
!= 不等
<  小于
>  大于
<= 小等于
>= 大等于
```

字符串比较

```
eq 相等
ne 不等
lt 小于
gt 大于
le 小等于
ge 大等于
```

(为什么我们要分离数字和字符串比较？因为我们没有指定的变量类型，而perl需要知道 我们是要按数字排序(99应该小于100)，还是按字符排序(100应该在99前面)。

布尔逻辑

```
&& and
|| or
!  not
```

(and、or和not不仅仅是在上表中作为操作符的描述 -- 它们也是享有Perl支 持的操作符。它们比C样式的操作符更易读，但是与&&及其他友符具有不同的有限级。 查看perlop以获取更多细节。)

其他

```
=  赋值
.  字符串比较
x  字符串乘
.. 范围操作符(创建数字的列表)
```

许多操作符可以和=结合使用，像下面这样：

```
$a += 1;      # same as $a = $a + 1
$a -= 1;      # same as $a = $a - 1
$a .= "\n";   # same as $a = $a . "\n";
```

文件和I/O

你可以使用open()函数打开一个文件用于输入或输出。在perlfunc和 perlopentut中有详细得近乎奢侈的文档。简而言之：

```
open(INFILE,  "input.txt")   or die "Can't open input.txt: $!";
open(OUTFILE, ">output.txt") or die "Can't open output.txt: $!";
open(LOGFILE, ">>my.log")    or die "Can't open logfile: $!";
```

使用<>操作符，你可以从一个已打开的文件句柄中读取数据。在标量上下文中，它每次从文件句柄中读取一行。在列表上下文中，它会一次读入整个文件，并把每一行赋给列表的一个元素：

```
my $line  = <INFILE>;
my @lines = <INFILE>;
```

一次读入整个文件也叫做啜食(sluping)。有时候会有用，不过很耗内存。多数的文本处理工作可以使用Perl的循环结构一次一行地进行。

<>操作符最常见于`while`循环：

```
while (<INFILE>) {     # 每行轮流赋给$_
   print "Just read in this line: $_";
}
```

我们已经见过如何向标准输出打印 -- 使用`print()`。然而，`print()`还有一个可选的第一个参数，用于指定输出用的文件句柄：

```
print STDERR "This is your final warning.\n";
print OUTFILE $record;
print LOGFILE $logmessage;
```

当你进行完所有对文件句柄的操作后，你应该`close()`它们(虽然老实地说，如果你忘了，Perl会替你清理)：

```
close INFILE;
```

正则表达式

Perl对正则表达式的支持是广泛而深入的。perlrequick、perlretut及其他地方的冗长文档都是关于这个主题的。尽管如此，简而言之：

简单匹配

```
if (/foo/)      { ... } # 当$_包含"foo"时为真
if ($a =~ /foo/) { ... } # 当$a包含"foo"时为真
```

匹配操作符`//`在perlop中有文档记述。它缺省对`$_`进行操作，或者可以通过使用绑定操作符`=~`(也在perlop中有记述)绑定到其他变量。

简单替换

```
s/foo/bar/;              # 把$_中的foo替换成bar
$a =~ s/foo/bar/;        # 把$a中的foo替换成bar
$a =~ s/foo/bar/g;       # 把$a中的所有foo都替换成bar
```

替换操作符`s///`记述于perlop中。

更复杂的正则表达式

你不必仅仅匹配固定的字符串。实际上，通过使用复杂的正则表达式，你可以匹配梦想的任何东西。这些都记述在超长的perlre文档里。但是趁这会儿，先看看快速作弊卡：

```
.               单个字符
\s              空白字符(空格、制表符(tab)、换行)
\S              非空白字符
\d              一个阿拉伯数字(0-9)
\D              一个非(阿拉伯)数字
\w              一个单词(word)字符(a-z、A-Z、0-9、_)
\W              一个非单词字符
[aeiou]         匹配给定集合中的单个字符
[^aeiou]        匹配给定集合之外的单个字符
(foo|bar|baz)     匹配指定选择中的任何一个

^               字符串开始
$               字符串结尾
```

限量符(quantifier)可以用来指定它前面的东西匹配的次数。这里"东西"指的可以是 一个文字的字符，或者上面列出的这些元字符中的一个，也可以是括在括号中的一组 字符和元字符。

```
*               0次或多次
+               1次或多次
?               0次或1次
{3}             匹配3次
{3,6}           匹配3到6次
{3,}            匹配3次或更多
```

一些简要的例子：

```
/^\d+/          以一个或多个阿拉伯数字开头的字符串
/^$/            空字符串(开始位置和结束位置连在一起)
/(\d\s){3}/       三个阿拉伯数字，每个数字后面有个空白字符(如："3 4 5 ")
/(a.)+/         匹配一个字符串，它的每个第奇数个字符都是a
                (如："abacadaf")

# 下面这个循环从标准输入(STDIN)读入，然后输出每一个非空的行：
while (<>) {
   print;
}
```

用于捕捉的括号

除了分组，括号还有第二个作用。它们可以被用来捕捉正则匹配的某些部分的结果以备 后用。结果被保存在$1、$2依此类推等变量里。

```
# 一个简单而肮脏的方式来把email地址拆成几部分

if ($email =~ /([^@]+)@(.+)/) {
   print "Username is $1\n";
   print "Hostname is $2\n";
}
```

其他正则表达式特性

Perl正则还支持向后引用(backreference)、前瞻(lookahead)以及各种其他复杂的细节。 这些全都能从perlrequick、perlretut和perlre中读到。

编写子例程

编写子例程很容易：

```
sub log {
    my $logmessage = shift;
    print LOGFILE $logmessage;
}
```

那个shift是什么？好的，传递给子例程的参数都存储在一个叫@_的特殊数组里(参阅perlvar)。而shift函数的缺省参数正好是@_。因而 my $logmessage = shift;会移出参数列表的第一个元素并把它赋给$logmessage。

我们也可以用其他方式来操作@_：

```
my ($logmessage, $priority) = @_;      # 常用
my $logmessage = $_[0];                # 不常用，而且丑陋
```

子例程可以返回值：

```
sub square {
    my $num = shift;
    my $result = $num * $num;
    return $result;
}
```

关于编写子例程的更多信息，参见perlsub。

面向对象(OO)Perl

面向对象Perl相对简单，是用引用实现的。这些引用知道它们自己是哪种对象(基于Perl 中的包的概念)。然后，面向对象Perl严重超出了本文档的讨论范围。请阅读perlboot、 perltoot、perltooc和perlobj。

作为Perl初级程序员，你用到面向对象Perl最多是在使用第三方模块的时候，下面的文档就会讲到。

使用Perl模块

Perl模块提供一系列的特性以使你避免重复发明轮子。Perl模块可以从CPAN (http:www.cpan.org/)下载。大量的常用的模块被直接包含在Perl发行版里。

模块类别从文本维护到网络协议到数据库集成，再到图形图像，应有尽有。模块的分类列表也可以在CPAN上找到。

要学习如何安装你从CPAN上下载的模块，阅读perlmodinstall。

要学习如何使用一个特定的模块，使用perldoc *Module::Name*。一般来说，你会需要use *Module::Name*。这会使你随后能够访问导出的函数或者该模块的一个面向对象 接口。

perlfaq里有关于各种常见任务的提问和答案，而且常常会建议你使用一些优秀CPAN模块。

perlmod提供Perl模块的全面概要。perlmodlib列出了所有你安装的Perl自带的 模块。

如果你有编写Perl模块的冲动，perlnewmod会给你很好的建议。

3 AUTHOR

Kirrily "Skud" Robert <skud@cpan.org>

Translated by Achilles Xu <formalin14@gmail.com>